FaunaDB 2.5.4

Kyle Kingsbury
2019-03-05

FaunaDB is a distributed, indexed document store based on the Calvin transaction protocol. We found that basic key-value operations in FaunaDB 2.5.4 appeared to provide snapshot isolation up to strict serializability, depending on workload. However, queries involving indices, temporal queries, or event streams failed to live up to claimed guarantees. We found 19 issues in FaunaDB, including nontransactional schema changes, lockups removing nodes from clusters, unavailability in response to clock skew and reboots, indices which failed to return negative integer values or skipped records at the end of pages, and multiple snapshot isolation violations in temporal and indexed queries. By 2.6.0-rc10, Fauna had addressed almost all issues we identified; some minor work around availability and schema changes is still in progress. Fauna has written a companion blog post to this piece, which is available here. This work was funded by Fauna, and conducted in accordance with the Jepsen ethics policy.

1 Background

In 2012, Thomson, Diamond, Weng, et al. published Calvin: Fast Distributed Transactions for Partitioned Database Systems: a transactional protocol optimized for geographic replication. Each Calvin cluster is comprised of multiple *replicas*, where a replica is a collection of nodes which store a complete copy of the dataset. Communication within replicas is assumed to be relatively fast, whereas communication between replicas (e.g. those situated in different datacenters) may incur high latency costs.

Calvin's key insight is that *ordering* transactions, and actually *executing* those transactions, are separable problems.[1] Traditional databases lock objects, or use multi-version concurrency control over read and write sets to provide an implicit transaction order. Calvin, by contrast, establishes a total order of transactions *up front*, then executes those transactions in parallel across all replicas.

In order to do this, Calvin transactions must be submitted in a single request, rather than the interactive sessions provided by many traditional databases. An ordering system, called the *sequencer*, accepts transactions, batches them up into time windows, and appends those batches to a sharded, totally ordered log. In FaunaDB, this process requires a round trip to a majority of log replicas in order to obtain consensus.

With consensus on log entries, each Calvin replica can read the log and execute the transactions in it *independently*: no coordination with other replicas is required. This is possible so long as transactions are pure—i.e., they do not perform external IO, and execute the same way given the same state at each replica. Purity also allows Calvin-based systems to batch and pipeline transactions before consensus, improving throughput.

Because replicas execute transactions independently, the only time replicas must communicate is when the sequencer is constructing the log. Rather than the multiple rounds required by two-phase commit, Calvin transactions require only a single round trip between replicas (as well as a few short message delays *within* each replica).

In addition, Calvin avoids a common problem with distributed transaction protocols: there is no single point of coordination for cross-shard transactions. While many databases can execute operations on each shard independently, they may fail to support transactions across shards (e.g. MongoDB, Cassandra), or introduce a global coordinator (e.g. Zookeeper, VoltDB). Calvin has no single coordinator in the transaction path–sequencers can be made up of independent shards,

[1] Subject to some constraints. For instance, Calvin transactions must know their read and write sets in advance, which may require some special pre-processing.

each backed by a consensus system like Raft or Paxos. The total order is derived by deterministically combining short windows of transactions from each shard. This introduces a fixed latency floor, since executors must wait for each window to complete before they can begin executing that window's transactions, but this floor can be tuned to be small, relative to the inter-replica latency within a shard's consensus group.

Some systems, like Spanner and CockroachDB, avoid the need for coordination between shards by relying on semi-synchronized wall clocks. If clock skew exceeds a certain critical threshold, such systems can exhibit transactional anomalies. In Calvin, clock skew has no impact on correctness.

1.1 FaunaDB

FaunaDB adapts the Calvin protocol for use in a modern, indexed, and temporal document store. FaunaDB bypasses Calvin's requirement that transactions know their read & write sets before execution; instead, snapshot-isolated reads are executed by a coordinator *before* sequencing the transaction. An optimistic concurrency control protocol includes read timestamps in the transaction sent to the sequencer, allowing executors to identify whether objects have been modified since they were last read. Since transactions are pure, these conflicts can be transparently retried.

FaunaDB's records are JSON-style objects, called instances; each instance is identified by a primary key called a *ref*. Instances belong to a collection, called a *class*, which defines a namespace for keys as well as an optional, partial schema. An index system extracts data from instances and maintains maps of terms to values, providing uniqueness constraints, secondary indices, and materialized views. Indices and classes are namespaced inside of logical *databases*, which can be nested.[2]

To ensure purity, each transaction is a single query expression; there is no concept of an interactive transaction.[3] To accommodate this, FaunaDB's query language is richer than most databases', embedding a full functional programming language based on the lambda calculus. FaunaDB queries include composite datatypes like vectors and maps, first-order anonymous functions, `let` bindings, `do` notation for executing multiple side effects like writes, and higher-order functional constructs like `join`, `map` and `filter`. For instance, to extract names and ages from a collection of cats, indexed by type:

```
(q/map (q/paginate (q/match cats "tabby"))
  (q/fn [cat-ref]
    (q/let [cat (q/get cat-ref)]
      [(q/select ["data" "name"] cat)
       (q/select ["data" "age"] cat)])))
```

We've written this query in a Clojure DSL, but the JSON AST it constructs is essentially the same—just a little more verbose. We take our index `cats`, and ask for every value matching the term `"tabby"`, paginating those results. Each result is a reference to a cat instance, which we transform using an anonymous function. That function looks up the value of each reference, binds that value to a variable `cat`, and returns an array with two elements: the cat's name and age. This query might return results like:

```
[["Professor Tiddlywinks", 11]
 ["Little Miss Snookums", 3]
 ...]
```

This expression-oriented syntax makes for easily composable queries that lend themselves well to programmatic construction.

1.2 Consistency

Fauna's home page advertises "strong consistency". Its datasheet claims FaunaDB is "100% ACID", providing "data accuracy and transactional correctness without compromise[4]", thanks to "global strongly consistent replication".

In the October 2018 blog post Consistency Without Clocks, Fauna claimed their consistency protocol, unlike many competitors, prevents stale reads:

> … once a transaction commits, it is guaranteed that any subsequent transaction—no matter which replica is processing it—will read all data that was written by the earlier transaction. Other NoSQL systems, and even most SQL systems, cannot guarantee global replica consistency.

And goes on to say:

> FaunaDB is an elegant, software-only solution for achieving global ACID transac-

[2] Fauna plans to rename instances to documents, and classes to collections.

[3] However, FaunaDB offers a form of session consistency. Each FaunaDB client maintains a local index of the last timestamp it interacted with, which should ensure that successive queries on the same client must take effect at logically higher times.

[4] There are, as usual, some compromises.

> tions, with complete guarantees of serializability and consistency.

The official documentation has little to say about consistency invariants. However, Fauna wrote their own Jepsen tests, and published a report which made more specific claims:

> FaunaDB provides strict serializability—or linearizability—for transactions that write, and serializability for transactions that only read data.

This is, however, not entirely correct. A more nuanced story may be found in an architecture blog post from 2017, which lays out FaunaDB's replication algorithm and guarantees in detail:

> Read-write transactions in FaunaDB where all reads opt in to optimistic locking as described above are strictly serializable.

The key detail here is "where all reads opt in": indices in Fauna do not participate in optimistic locking by default, and only guarantee snapshot isolation. However, by enabling an index's `serialized` mode, we can recover serializability for indices. Unique indices imply `serialized` as well.

Furthermore, read-only transactions, for performance reasons, also execute at snapshot isolation, and may return stale data:

> Since read-only transactions in FaunaDB always have a specific snapshot time but are not sequenced via the transaction log, they run at snapshot isolation, which for read-only transactions is equivalent to serializable.

After careful consultation with Fauna's engineers, we believe FaunaDB's intended consistency levels fall between snapshot isolation and strict serializability, depending on whether the transaction is read-only, whether indices are used, and whether those indices are flagged as serializable.

	Read-Write	Read-Only
No indices	Strict-1SR	Serializable
Serializable indices	Strict-1SR	Serializable
Indices	SI	SI

Although snapshot isolation allows anomalies like stale reads and write skew, it's still a relatively strong consistency model. We expect to observe snapshot isolation at a minimum, and where desired, we can promote SI or serializable transactions to strict serializability: the gold standard for concurrent systems.

2 Test Design

Fauna wrote their own Jepsen tests, which we refined and expanded throughout our collaboration. We evaluated FaunaDB 2.5.4 and 2.5.5, as well as several development builds up to 2.6.0-rc10. Our tests used three replicas, and 5–10 nodes, striped across replicas evenly. Log node topologies in 2.5.4 and 2.5.5 were explicitly partitioned, with a copy in every replica. We waited for data movement to complete, and for all indices to signal readiness, before beginning testing.

Starting a FaunaDB cluster in 2.5.5 and 2.5.6 was a slow process, requiring ~10 minutes to stabilize. While FaunaDB will service requests during this time, latencies are highly variable. Moreover, queries against newly created indices can return inconsistent data until data movement completes. To speed up testing, we shut down every node and cache their data files after completing the initial join process, and begin subsequent tests by resetting the cluster to that saved state. Fauna reports that cluster bootstrap time has been improved in version 2.6.1.

We evaluated a variety of network failure modes, including partitions isolating a single node, majority/minority partitions within a single replica, and partitions isolating a single replica from the others. We checked FaunaDB's behavior through process crashes and restarts, small and large jumps in clock skew, and rapidly strobing clocks. We also tested with rolling restarts, while changing FaunaDB's log node configuration, and removing and adding nodes to the cluster.

We designed a family of workloads for FaunaDB using the Jepsen library, designed to stress inserts, single-key linearizable transactions, multi-key snapshot isolation, pagination, phantoms, monotonicity, temporal queries, and internal consistency within transactions.

2.1 Sets

Our set test inserts a series of unique numbers as separate instances, one per transaction, and attempts to read them back through an index. Previous Jepsen analyses relied on a single final read to determine the fate of each element, but for FaunaDB, we designed a more thorough, quantitative analyzer: we read throughout the test, and measure whether successfully inserted instances eventually disappeared, or

were visible to all reads after some time t. We also compute latency distributions for both lost and stable reads, e.g. how long must one wait to ensure that a successful write is visible to all future reads.

Because FaunaDB treats read-only and update transactions differently, a variant of the set test uses a "strong read" transaction, which includes a spurious write to an unrelated class in order to force the transaction to go through the full commit path.

2.2 Registers

Strict serializability implies linearizability, so we evaluated whether FaunaDB supports linearizable operations on single instances, using an undocumented `/linearized` endpoint in the FaunaDB API. We generate randomized reads, writes, and compare-and-set operations, and measure whether the resulting history is linearizable using the Knossos linearizability checker.

2.3 Bank

As with other transactional systems like CockroachDB and Dgraph, we stressed FaunaDB's snapshot isolation in a simulated bank-account system. We model each account as a single FaunaDB instance, and transfer money between accounts in transactions. Because transfers write every value they read, transfer transactions are serializable under snapshot isolation. Snapshot reads should therefore observe a constant total balance, across all accounts.

To stress reads of recently created and deleted instances, one variant of this test deletes accounts when their balance falls to zero, and creates new ones when necessary. To explore both instance and index reads, we perform reads by directly querying all n accounts, or by reading their balances from an index. We can also test temporal queries by reading at a particular snapshot time, vs reading the current value.

2.4 Pagination

Exploratory testing suggested that reading multiple pages of records (e.g. from an index) could result in inconsistent results. To quantify this behavior, we designed a test specifically to stress FaunaDB's query pagination mechanism. We insert groups of n instances in a single transaction, and, concurrently, paginate through all records in the index. We expect that if any instance from an insert transaction is present in a read, then that read should also contain all other instances inserted in the same transaction; violations imply read skew.

2.5 G2

The bank test verifies snapshot isolation, but serializability implies more restrictive invariants—for instance, the absence of Adya's phenomenon G2: anti-dependency cycles. To stress FaunaDB's support for serializable indices, we execute pairs of transactions, each of which performs an index read looking for a specific write which would have been performed by the other transaction. If the other transaction's write is detected, we abort the transaction. In a serializable system, at most one of these transactions may commit, but in a weaker isolation model, like snapshot isolation or repeatable read, these anti-dependency cycles may be allowed.

2.6 Internal

Many of our tests measure the isolation boundaries between different transactions; ensuring, for instance, that a transaction's effects become visible all at once, or that there exists an apparent total order of transactions. However, transactions should also exhibit *internal* consistency: changes made within a transaction should be visible to later reads within that same transaction.

We explore internal ordering effects by creating objects matching a predicate, and within the same transaction, querying that predicate to check whether or not they appear. We also alter those objects, changing what predicate they fall under, and confirm that old and new predicates reflect that change.

2.7 Monotonic

FaunaDB clients keep track of the highest timestamp they've interacted with, and provide it with each request to ensure that they always read successive states of the system; although reads may observe stale states, they should never observe an older state than one that client previously observed. To verify this, we set up a counter which increments over time. Since the counter value always increases, successive reads of that value by any single client should observe monotonically increasing transaction timestamps and values.

We also perform temporal queries, reading from a timestamp a few seconds in the past or future. Of course the resulting timestamps and values are non-monotonic, since we are reading from random times. However, the *relationship* between timestamps and values should be globally monotonic.

3 Multi-monotonic

We observed sporadic failures in the monotonic test, and designed a variant optimized for read and write throughput, while removing write contention. Instead of performing increment transactions (which read a register's value, add one, and write the resulting value back) we perform blind writes of sequential values from a single process. This still guarantees that the values in the database should increase monotonically, but reduces transaction retries (and potential race conditions) due to increment contention. Unlike the monotonic test, which observes only a single register, we work with several registers concurrently, giving us more chances to observe non-monotonic behavior.

Read transactions fetch the current values of a randomly selected subset of recently written keys, and record not only the time that the read transaction executed, but also the modification timestamps from each observed register. We then order reads by transaction timestamp, and verify that the values observed for each key monotonically increase.

4 Results

4.1 Performance Limitations

FaunaDB's recovery from a network partition was relatively slow: in 2.5.4 and 2.5.5, it took 20–80 seconds, depending on cluster topology, to recover from a partition isolating a single replica. This limited the rate at which we could create meaningful network faults, and reduced our probability of reaching interesting failure states. Fauna is aiming for recovery times of 500 ms or less—in line with similar consensus systems. Presently, development builds of 2.6.1 can take 30 seconds to recover from some network partitions. Fauna reports that this issue is related to the responsiveness of the Φ-accrual failure detector, and has been fixed in the upcoming 2.6.2 release.

4.2 Possible Surprises

FaunaDB's query documentation says that let bindings and do expressions evaluate their forms sequentially, left-to-right. However, the evaluation semantics for collection literals, like `[1, 2, 3]` or `{type: "cat", sound: "meow"}` were not explicitly specified. As it turns out, those forms are *also* evaluated in array or object literal order, so a query like…

```
{a: Paginate(Match(index ...))
 b: Create(...)
 c: Paginate(Match(index ...))}
```

means that `a` will not reflect the results of the `Create` call, but `c` will. Users should be careful to use order-preserving maps when constructing FaunaDB queries in their language, to avoid the accidental reordering of side effects. Fauna has since documented this.

This also applies to temporal queries: normally, `At(some-timestamp, Paginate(...))` will always return the same results, based on the state of the database at the given timestamp. However, if a temporal query's timestamp is the same as the current transaction timestamp, the value of that temporal query depends on that transaction's prior writes.

4.3 Minor Issues

FaunaDB's data definition language (DDL) is non-transactional: schema changes are asynchronously cached, and may take several seconds to apply. This allows some unusual transient behaviors, including:

- In 2.5.4, you cannot create a class and an index on that class in the same transaction; you have to perform a second transaction to create the index.
- Classes and indices cannot be transactionally upserted. Queries that create a class iff that class does not currently exist may fail to observe an already created class, then, on inserting, throw an "instance not unique" error. This behavior was a known bug in 2.5.4, and a fix is planned for 2.7.
- Creating a class does not necessarily guarantee a subsequent transaction will be able to insert into that class. In FaunaDB 2.5.4, for example, inserts into recently created classes may return errors like "invalid ref: Ref refers to undefined class". A fix is planned for 2.7.
- Newly created indices are queryable, but are built asynchronously. Even if the collection being indexed is empty, transactions using that index could return inconsistent data for several

seconds (and, for clusters undergoing data movement, potentially much longer). Users should be careful to poll newly created indices until they show `active: true`, which indicates that the index is ready for use. We observed this problem in 2.5.4 through 2.6.0-rc10; it is fixed in development builds of 2.6.1.

We also encountered problems changing cluster topology:

- In 2.5.5, we found that one cannot follow the documented procedure for removing a node: asking a node to remove itself from the cluster will always fail. Instead, removes must be initiated on a different node. Fauna fixed this in 2.6.0-rc1.
- We were unable to remove nodes from 2.5.5 clusters, either by removing the node before killing it, or by stopping or killing the node to be removed first, then asking remaining nodes to remove it. Removed nodes would complete the drain process, but never actually leave the cluster. This bug was fixed in 2.5.6-rc4.
- In 2.5.5, configuration and topology changes could result in nodes returning unexpected internal server errors to clients, including "FaunaDB Service is uninitialized" and "Transaction Log is uninitialized." These errors include a warning to contact Fauna's support team, but appear harmless; Fauna fixed these in 2.6.0-rc1.
- In 2.5.5 and 2.5.6-rc4, rebooting nodes to apply topology changes could result in nodes throwing "operator error: No configured replica for key: 0x…" until the topology had stabilized. These errors appear harmless, and Fauna removed them in 2.6.0-rc1.
- In 2.6.0-rc1, Fauna removed the need to manually assign nodes to log shards in the config file, allowing FaunaDB to manage log topology automatically. However, we identified a bug in this system: nodes could leave the cluster before their log partitions had been spun down or migrated to other nodes, which could cause future node leave operations to stall indefinitely. Fauna addressed this in 2.6.0-rc7.
- In 2.6.0-rc7, removing nodes could stall when transaction pipelines blocked, awaiting transactions from a log segment that was already closed (but not yet destroyed). Fixed in 2.6.0-rc10.

And availability issues on startup:

- In 2.5.6-rc4 and -rc9, concurrently rebooting nodes to apply configuration changes, or rebooting nodes when other nodes are inaccessible due to a network partition, could result in nodes connecting to the cluster, but never binding port 8444, which is used for client queries and administrative operations. When a node is unable to join the consensus ring which manages the cluster, it blocks the node from completing the startup process. This issue is addressed in 2.6.0-rc10.

4.4 Clock Skew Unavailability

FaunaDB's replication protocol uses consensus, not wall clocks, to construct its transaction logs. Indeed, in Consistency Without Clocks, Fauna repeatedly claims that "FaunaDB requires no clock synchronization." However, the installation instructions mention that one must first install "NTP, with the clocks synced across nodes".

Specifically, FaunaDB still relies on wall clocks to decide when to seal time windows in the log, which means that clock skew can delay transaction processing. We tested FaunaDB with a range of clock skews from milliseconds to hundreds of seconds, over multi-second windows, and strobing rapidly every few milliseconds, as well as gradually increasing and decreasing offsets. We applied these clock adjustments to randomly selected single nodes, repeatedly to a single node, and to randomly selected subsets.

In none of our clock tests did FaunaDB exhibit new safety violations. However, clock skew *can* cause partial or total unavailability. Specifically, when a single node is skewed by `offset` seconds relative to the rest of the cluster…

- Small positive skews appear to have little impact.
- Skews over 10 seconds can result in 5-second timeouts for all requests on the affected node, until the clock skew is resolved. Once the clock is resynchronized, it takes an additional `offset` seconds for the node to recover.
- Small negative skews can result in elevated latencies (~20 seconds) on single nodes.
- Negative skews more than 10 seconds can result in 60 second timeouts on that node, which persist until the clock skew is resolved. However, they can also cause cluster-wide disruption, where most requests encounter 60 second timeouts, while a few requests proceed normally. Ending the clock skew resolves the unavailability immediately.

Skews on multiple nodes, or skews which change over time, can result in complex combinations of these behaviors, including total unavailability. Predicting exactly how FaunaDB will respond is somewhat difficult; we believe effects depend on whether the primary node for a given log partition was affected, and whether the skew was forward or backwards in time. There may also be a dependence on internal and client timeouts.

Clock synchronization is theoretically not required for FaunaDB availability; Fauna plans to address this issue in 2.7.

4.5 Missing Negative Integer Values

Initial designs of the pagination test inserted positive and negative pairs of integers in the same transaction, e.g `[-5, 5]`. To our surprise, querying the index for all values returned only positive numbers, never negative ones. This occurred because FaunaDB began paginating result sets at 0, not at `MIN_LONG`. Because FaunaDB sorts doubles after longs, and traversal begins at the long value `0`, this problem only affected integers, not floating-point numbers.

We found this issue in 2.5.4, and it was fixed in 2.5.5. As a workaround, users on 2.5.4 and lower can begin any pagination of sets containing negative integer values with `{after: MIN_LONG}`.

4.6 Time is a Flat Circle

As a temporal database, FaunaDB exposes an `events` query which returns the history of events (e.g. creates, updates, and deletes) affecting a given instance, or predicate query. However, when we used event queries to check the history of a single instance, we discovered the returned sequence of events formed an infinite loop.

When paginating through events, FaunaDB returns the most recent page of events first, but in chronological order. That is, if we number events 0, 1, 2, …, then the first page of results might return events `[10, 11, 12, 13]`. This makes some sense; one is typically more interested in recent than ancient history. However, the pointers to request additional pages are broken: there is no `before` page, and the `next` page begins at event 9. Requesting the next page iterates in ascending order from 9, returning the exact same page: `[10, 11, 12, 13]`, rather than `[6, 7, 8, 9]`.

Fauna fixed this issue in 2.6.0-rc10.

4.7 Non-Transactional Pagination

By design, FaunaDB has no way to return unbounded result sets. Instead, one requests a page of results of a certain size—by default, 64 elements. Page objects also include references to the previous and next pages of results, which provides "cursor-like semantics".

The mechanism Fauna used in their Jepsen tests was to fetch the first page of results, and with the `after` cursor from that page, make the same paginated query, but after the given cursor, and to continue traversing the result set until no `after` cursor remained. This is also how the FaunaDB Javascript client iterates through results.

This approach is intuitive, sensible, and wrong. Pagination cursors only encode the *value* that the next page should begin after, not the *time*. Since each page is fetched in a separate query, and since each query executes at a different transaction time, modifications to result sets during traversal may result in inconsistent snapshots. For instance, a transaction could insert the numbers 80 and 81 together, but if the two elements happen to fall on different pages, a paginated query could observe 81 but *not* 80, which violates snapshot isolation.

The documentation says that pagination can be used to "walk the result set in blocks", but doesn't actually claim that result sets are transactional. Whether this behavior violates Fauna's claimed invariants depends on how users interpret the documentation. FaunaDB is a temporal database, so one might reasonably expect pagination cursors to include temporal information. However, each page *is* fetched in a different query, and FaunaDB generally does not enforce transactional isolation across queries.

When paginating, users should be aware of the possibility of read skew, missing elements, duplicated elements, etc., and use the same timestamp for all pages where snapshot isolation is required.

4.8 Definitely Non-Transactional Pagination

To work around this issue, we redesigned how Jepsen iterates through paginated results; we fetch the snapshot time `ts` along with the first page of results, and wrap every subsequent page query in `At(ts, Paginate(...))`, ensuring that every page observes the same logical timestamp. Unfortunately, pagination tests continued to show read skew, both with normal and serialized indices.

```
[{:op
  {:type :ok,
   :f :read,
   :value
   [-9750
    -9282
    ...
    9937
    9991],
   :process 18,
   :time 667358917338,
   :index 82},
  :errors
  #{{:expected #{2392 -1576 -3715 3539},
     :found #{3539}}
    ...
```

In this particular read, several transactions were only partly visible: for instance, 2392, -1576, -3715, and 3539 were inserted in the same transaction, but of those, only 3539 was actually visible in the read. These problems occurred in version 2.5.4, in healthy clusters without faults.

This anomaly manifested regardless of whether indices had one or multiple data partitions. Moreover, careful inspection revealed that not only could FaunaDB exhibit read skew between pages, but that a *single page* could include incomplete writes from other transactions. This suggested a more fundamental problem than pagination—perhaps the index structure itself was improperly isolated.

4.9 Inconsistent Indices

To explore this more fully, we designed two variants of the bank test: one which reads all accounts by making requests for n specific keys, and one which requests all values from an index. We found that although instance reads appeared safe, index reads could observe wildly inconsistent values. For instance, in a system of 8 bank accounts containing $100, with 10 clients making concurrent transfers up to $5, the observed total of all account values could fluctuate between $27 and $126. This clearly violates FaunaDB's claims of snapshot and serializable isolation for indices.

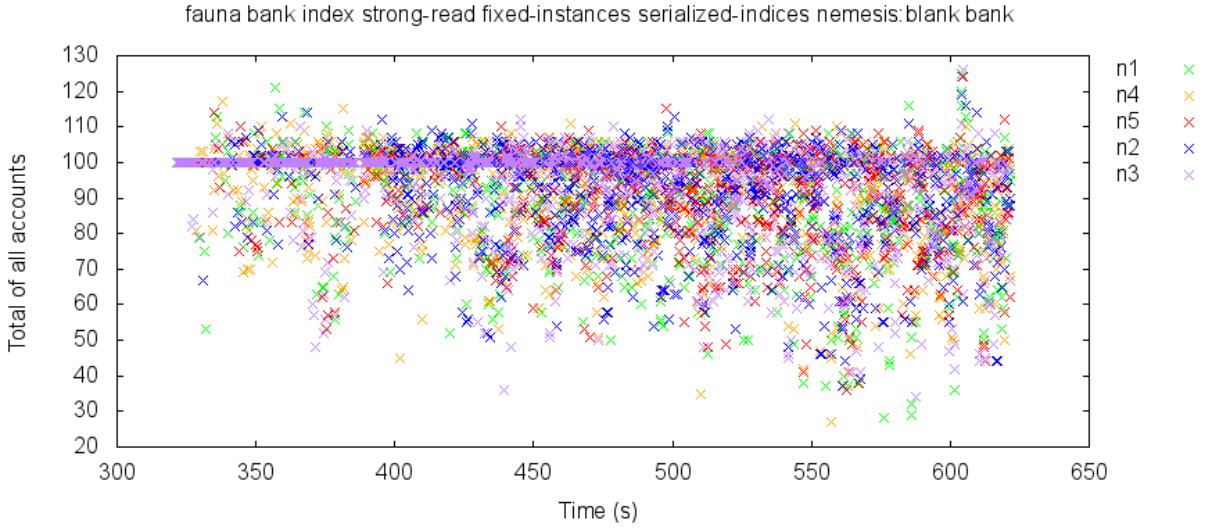

Figure 1: Plot of total balances over time, colored by node. In a snapshot isolated system, every read would return exactly 100.

In healthy clusters, on versions 2.5.4, 2.5.5, and 2.6.0-rc7, we found roughly 60% of reads could observe inconsistent states. This occurred with both serialized and normal indices, and both fixed and dynamic pools of account instances.

These read skew issues stemmed from an incomplete implementation of *bitemporal indices*: while Fauna planned to allow queries to observe a consistent view of an index at any point in time, that system was only partly implemented prior to 2.6.0.

Internally, changes to FaunaDB instances are tracked by assigning each change a distinct *version*, and retaining old versions of instances for a configurable period (by default, 30 days) before garbage collection. To read

the state of an instance r at timestamp t, a transaction would ensure that the server had applied every transaction up to t in the log, then find the version of r with the highest timestamp t_r such that $t_r < t$.

Unlike instances, index entries did not always store a separate version for every change. Instead, if an index entry with the same value already existed, that entry's timestamp t_1 would be *overwritten* with a new timestamp t_2.

Now imagine an index read executed at some time t between t_1 and t_2—either one executed explicitly in the past, or circa some update to the given index entry. That read should observe our index entry, since it was present at t_1. However, because the index entry's timestamp was changed to t_2, the read would *skip* that entry, and instead observe some older state for that index term—perhaps the empty state, or some prior value. Voilà: read skew! Fauna fixed this issue in 2.6.0-rc9.

4.10 Missing Records During Pagination

Unfortunately, bitemporal indices were not the only problem leading to read skew. In FaunaDB 2.6.0-rc7, our pagination test observed that reads of uniformly distributed integers could return fewer records than expected—in some cases, less than a quarter of instances that should have been present. For instance, a read might observe:

```
[-9759 -9748 -9714 ... 4279 4291 5195]
```

In this particular history, elements -9714, -6722, 7406, and 7901 were inserted in the same transaction, but traversal appears to have stopped after 5195: elements 7406 and 7901 are missing. These read skew errors were ubiquitous, even in healthy clusters, with both normal and serialized indices.

This occurred because of a bug in index traversal: the query engine filtered out unapplied transactions from the index, but still counted those unapplied transactions towards the total number of results for a given page—effectively skipping n records at the end of each page, where n was the number of uncommitted transactions visible during the traversal. Fauna fixed this bug in 2.6.0-rc9.

4.11 Non-temporal Temporal Queries

As a temporal database, FaunaDB allows queries to inspect the state of the database at any point in time. Any query can be wrapped with `At(t, ...)` to observe what that query would have seen at timestamp `t`. However, temporal queries for the state of a single instance at time t exhibited inconsistent behavior similar to the bitemporal index problem: queries would be prohibited from returning the state of that instance after t, but could observe any state—not just the most recent state—before t.

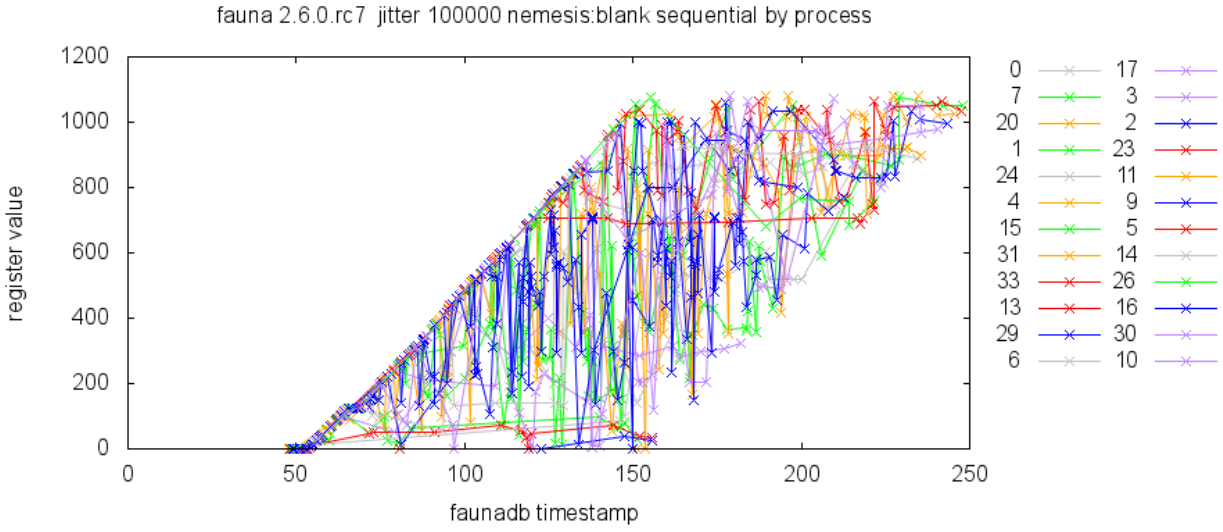

Figure 2: Plot of register values vs FaunaDB timestamps, broken down by process. Queries are performed at the current FaunaDB timestamp, plus or minus 100 seconds. Notice that the values of registers can appear to decrease as timestamps increase, even for a single process talking to a single server.

This resulted in apparent paradoxes where transaction T_1 would read at time t_1, and T_2 read at a later time t_2—but T_2 observed an *earlier* state than T_1. This issue affected instance reads, not just indices.

For example, in the monotonic test, we read an instance with a monotonically increasing value. Because the value never decreases, reads at higher times should show higher values. However, if we perform temporal queries at the current transaction time plus or minus, say, 10 seconds, we observe a distinctly *non*-monotonic relationship between timestamps and values.

In this graph, note that the spread of timestamps which can observe the same value is roughly 100 seconds. This occurs because reads at *future* timestamps (up to 100 seconds from the current time) observe whatever *current* state some node has available, resulting in a spread of possible values for the same timestamp, whereas reads at *past* timestamps observe the fixed value for that timestamp, giving rise to the clean upper bound on observed values.

Moreover, this bug allowed temporal reads of multiple instances to return the state of one instance at t_1, and the state of another at t_2, allowing read skew. For instance, in this bank test, we replace our normal reads with temporal reads, +/- 10 seconds from the current time. Network partitions cause some replicas to lag behind others in applying transactions. Out of 7190 reads, 5 observed an inconsistent state.

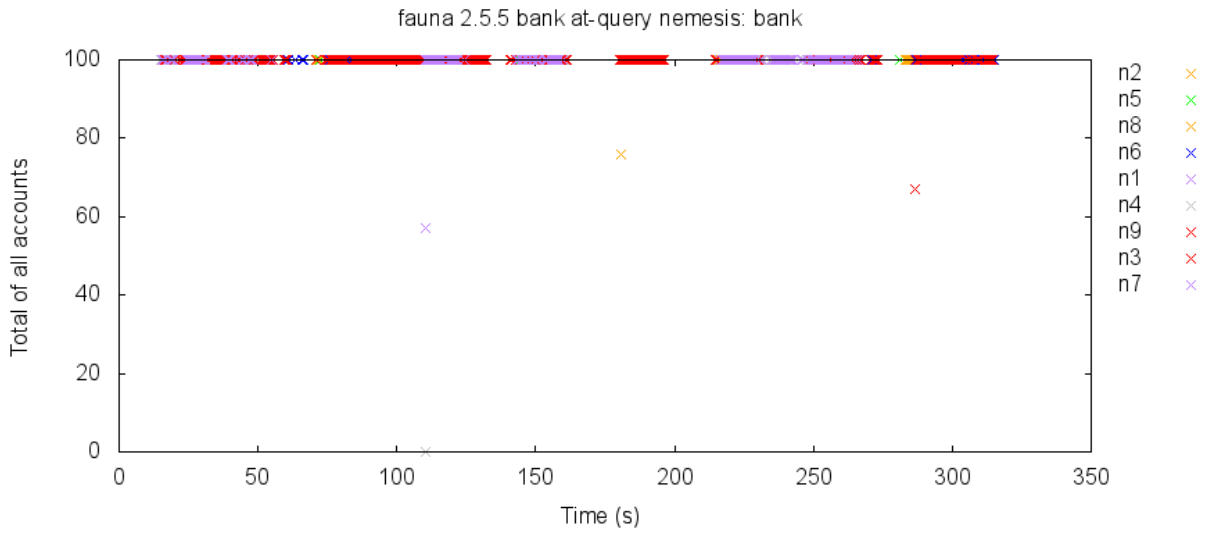

Figure 3: Plot of total balances over time, colored by node. In a snapshot isolated system, every read should have observed $100.

Unusually, some reads observed *no* accounts whatsoever, which would be a legal read of the database state before the test had ever begun. However, this test reads at most 10 seconds into the past—and the read of zero occurred over a hundred seconds into the test. We found that temporal reads in 2.5.5 weren't just able to read recent (but non-monotonic) states—in general, they can read quite old ones, including the empty state.

Fauna was aware of this bug before Jepsen identified it in 2.5.5, and fixed the issue in FaunaDB 2.6.0-rc9. Near-present and future reads no longer result in frequent non-monotonic anomalies.

4.12 Non-monotonicity, Long Fork, Read Skew

While 2.6.0-rc9 resolved the largest cause of read skew in temporal queries, Jepsen continued to see occasional test failures in monotonic, multi-monotonic, and bank tests. We observed these errors sporadically in healthy clusters and more frequently with process crashes and restarts, and they occurred with both normal and temporal queries.

For instance, in monotonic tests (even without temporal reads) a single process reading a single increment-only instance could read 4, 5, 6, then 5 again. This is legal under snapshot isolation, because read-only

transactions are allowed to observe *any* past timestamp. However, this behavior violates FaunaDB's session guarantee: clients should always read a state *at least* as recent as any they have in the past.

As it turns out, Fauna discovered a race condition in the client, which updated the client's highest-seen timestamp *after*, not *before*, returning results to a caller. However, our anomalies were caused by something different: read timestamps *did* increase monotonically, but values occasionally *went backwards*.

We designed the multi-monotonic test to explore this behavior in more detail, and discovered two things. First, this behavior occurs only sporadically, but when it *does* occur, several reads may observe non-monotonic state, which suggests some sort of transition within the FaunaDB cluster may be to blame. Second, the problem is worse than simple non-monotonicity.

To understand why, consider this test run, which exhibited six non-monotonic reads. The fourth (at index 526670) observed a value and write timestamp for key 3676 which were *lower* than observed by a prior read (at index 526569).

Read ts	Value ts	Value
00:38:13.695940	1544747893688800	291
00:38:13.752572	1544747893588600	290

However, these two read transactions observed *more* than key 3676—they read other keys as well. We'll consider just a few of those here, for illustrative purposes:

Key	3297	3676	4189	5432
Read 1	380	291	264	347
Read 2	380	290	265	348

Key 3297 is unchanged, 3676 has decreased, and 4189 & 5432 both increased. Consider the possible orders of writes which could lead to such a snapshot: in order to observe 4189 increasing, read 1 must precede the write increasing 4189 to 265, and read 2 must come thereafter. Therefore, read 1 must precede read 2. However, the exact *opposite* constraint applies on key 3676: read 2 must precede read 1. These snapshots are not compatible with a total order of write transactions.

This particular anomaly is known as "long fork": two write transactions T_1 & T_2 which write disjoint keys can have their writes observed in contradictory orders: one read observes T_1 but not T_2, while another observes T_2 but not T_1. This violates snapshot isolation, although it is legal under parallel snapshot isolation.

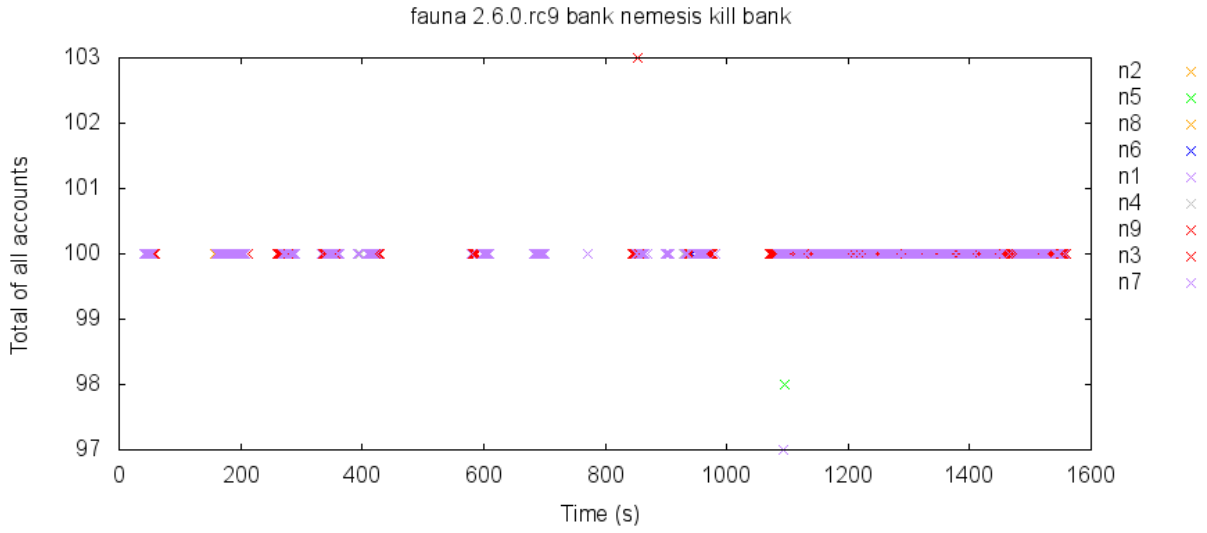

Figure 4: In this run, process crashes and restarts allowed bank queries to occasionally read a total value of accounts slightly lower or higher than expected.

We believe the bug which underlies this issue also allows for sporadic bank test failures. For instance, in this run, version 2.6.0-rc9 returned incorrect total balances for five reads. Since the value returned to normal, rather than shifting permanently, we suspect this read-skew anomaly may only affect read-only transactions.

Careful inspection of the histories suggests that all of these anomalies have a common etiology: reads may fail to observe the most recent state of some (but not necessarily all) instances read in a transaction, instead observing some other, past state. Specifically, when a transaction reads a version of an instance with a timestamp equal to the query snapshot time, that version is interpreted as being in the present, and then rewritten.

This problem was introduced in 2.6.0-rc9 by a patch for the bitemporal index problem, and did not impact production releases. Fauna fixed the issue in 2.6.0-rc10.

4.13 Acute, Persistent Read Skew

Finally, version 2.6.0-rc9 introduced a new class of behavior: under rare conditions, randomized process crashes & restarts could put FaunaDB into a state where almost every read observes an incorrect balance. In some cases, FaunaDB recovers and returns the correct value after a burst of incorrect reads. More often, though, reads remain wildly inconsistent for the remainder of the test, even if we restart every node and allow the cluster to stabilize.

For example, in this test run, we transfer money between a pool of 8 fixed bank accounts, containing $100 total. All reads are performed directly on instances, not indices, and we use normal, non-temporal reads—although this problem manifests with temporal queries too.

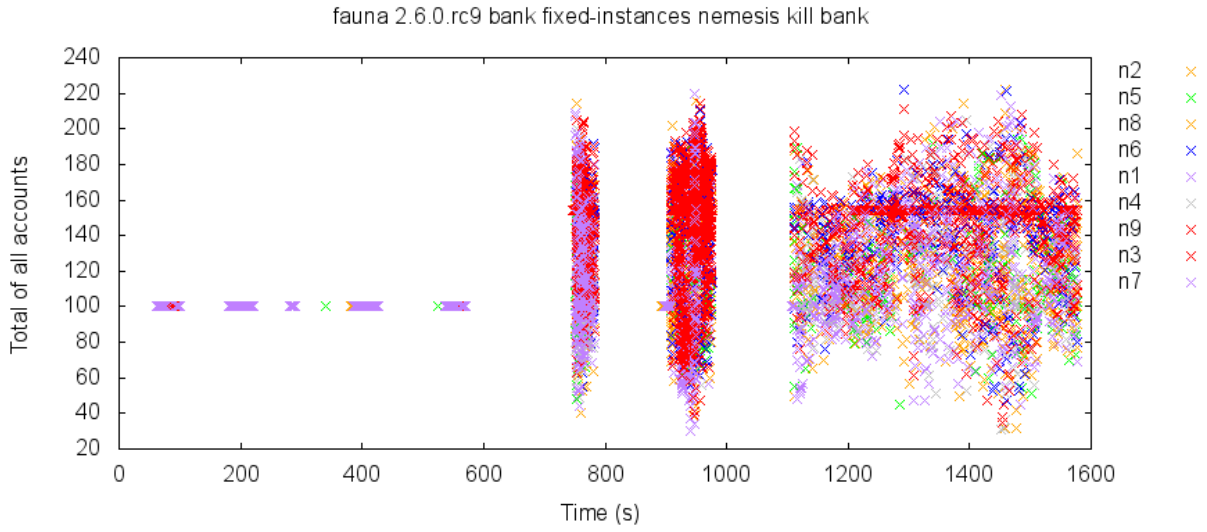

Figure 5: An acute, persistent bank failure, beginning at 1069 seconds. Values fluctuate from 30 to 222, instead of 100.

746 seconds into the test, something terrible happens: observed values fluctuate randomly between 30 and 222. There are some windows of downtime due to additional process crashes and restarts; finally, at 1069 seconds, we restart every node and let the cluster run in a healthy state until 1636 seconds. Despite this long window for recovery, observed values do not stabilize. More worryingly, a dense streak of reads at ~160 suggests that perhaps skewed reads were written back to the database, permanently corrupting state—since the value never stabilizes, it's hard to tell.

During this time, FaunaDB fails some transfer transactions with errors like "Mismatched transaction results Vector", which begin at 750 seconds, and continue throughout the remainder of the test. This error suggests that FaunaDB's internal isolation mechanisms have failed.

Every transaction epoch in FaunaDB is partitioned across segments. In 2.6.0-rc9, Fauna made changes to

the dynamic log management code which introduced an off-by-one error in recovery after a node restarts. This error involves three conditions:

1. The applied transaction state for the current epoch had been partially synced to disk on the node.
2. The node shut down immediately after the partial sync, before additional epochs were applied.
3. The node restarted immediately thereafter.

After restarting, a node has two types of on-disk state to recover from: the transaction log (where every committed entry is fsynced to disk), and the applied transaction state, which is periodically synced—by default, every two minutes, or when buffers are full. The node needs to take the last applied state, and replay transactions from the log against that state to catch up.

However, when the applied transaction state is only partially synced, the node would choose the *next* log segment, rather than the segment which had not been completely applied. Any transactions in the tail of the previous log segment would be skipped on this particular node, which led to nodes disagreeing about the applied transaction state.

Moreover, the timestamps of those missing transactions would be applied to the *next* transaction in the epoch. Future epochs would be unaffected, as would transactions executed on other nodes. This divergence also contributed to read skew.

Operators could recover from this scenario by running a repair task, or by identifying and replacing the affected node, recovering data from their (hopefully correct) peers.

Fauna introduced this issue in 2.6.0-rc9, and fixed it in 2.6.0-rc10. It did not impact production releases.

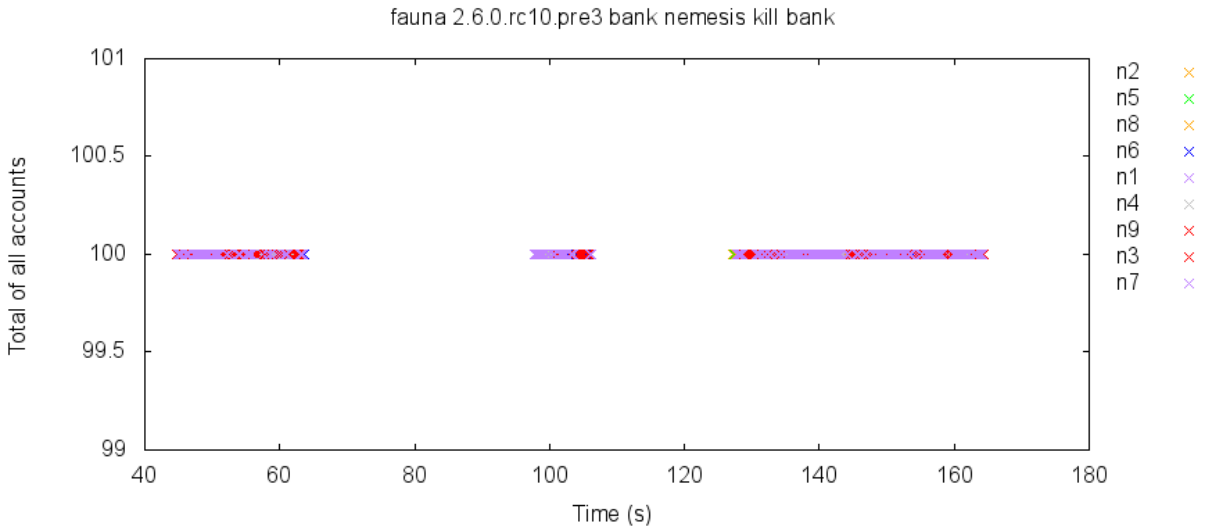

Figure 6: A successful bank test in 2.6.0-rc10, in which no reads observed inconsistent state.

5 Discussion

FaunaDB's core operations on single instances in 2.5.4 appeared solid: in our tests, we were able to reliably create, read, update, and delete records transactionally at snapshot, serializable, and strict serializable isolation. Acknowledged instance updates were never lost to single-instance reads. In 2.6.0-rc10, with serialized indices, FaunaDB even prohibited subtle anomalies like predicate phantoms.

However, we found serious safety issues in index, temporal, and event queries, race conditions in index and class creation, and multiple safety issues in pagination, including read skew and missing records. We also found a number of bugs in cluster topology changes, including nodes getting stuck leaving and rebooting to apply configuration changes. There were serious safety issues in individual-instance operations in release candidates, but these did not impact production releases.

№	Summary	Event Required	Fixed in
1	Can't create a class & index on that class in same txn	None	Unresolved
2	Can't upsert classes or indices	None	Unresolved
3	Can't insert into newly created class	None	Unresolved
4	New indices return inconsistent data	None	2.6.1-dev
5	Can't ask nodes to remove themselves	Node removed	2.6.0-rc1
6	Removed nodes stall, never leave cluster	Node removed	2.5.6-rc4
7	Unexpected, harmless component uninitialized errors	Topology change	2.6.0-rc1
8	Unexpected, harmless operator errors	Topology change	2.6.0-rc1
9	Removed nodes stall due to impossible log topologies	Node removed	2.6.0-rc1
10	Removed nodes stall due to closed transaction pipelines	Node removed	2.6.0-rc10
11	Failure to bind port 8444 when consensus ring unavailable	Restart + isolated	2.6.0-rc10
12	Elevated latencies & unavailability due to clock skew	Clock skew	Unresolved
13	Default pagination never returns negative integers	None	2.5.5
14	Infinite loop paginating instance events	None	2.6.0-rc10
15	Incomplete bitemporal indexes	Index read	2.6.0-rc9
16	Missing records during pagination	None	2.6.0-rc9
17	Temporal queries for future times observe local present	Temporal read	2.6.0-rc9
18	Occasional non-monotonic reads, long fork, read skew	None	2.6.0-rc10
19	Acute, persistent read skew, possible write corruption	Restart after sync	2.6.0-rc10

5.1 Recommendations

Two major bugs impacted transactional safety in released versions of FaunaDB.

First, index queries in 2.5.4 through 2.6.0-rc7 did not necessarily return the current (as of the read timestamp) state of records, due to an incomplete implementation of bitemporal indices. Users can mitigate this problem by fetching specific instances instead of using indices. This problem is fixed in 2.6.0-rc9.

Second, temporal queries in 2.5.5 and prior could return incorrect values for instance reads when a node had not yet applied all updates prior to the requested time. Users can mitigate this problem by avoiding temporal queries for recent (or future) timestamps. This problem is also fixed in 2.6.0-rc9.

While 2.6.0-rc9 fixed several safety issues, it also exhibited new bugs, including occasional non-monotonicity, long fork, and read skew. These issues appear transient and, so far, limited to read-only transactions. In addition, when nodes are restarted, FaunaDB 2.6.0-rc9 can wind up in a state where most queries return dramatically inconsistent data. FaunaDB occasionally recovers on its own, but in the majority of our testing, read skew persists even after nodes are restarted, and the cluster allowed to stabilize. Fauna has since released version 2.6.0, which includes fixes for the problems we identified in 2.6.0-rc9.

Jepsen recommends all users upgrade to 2.6.0 as quickly as practical.

Users should also be aware of issues around pagination: prior to 2.5.5, pagination would skip negative integers by default. In 2.5.4 and prior versions, users can work around this issue by explicitly beginning pagination at `MIN_LONG`. Users should also be aware that pagination may skip some records at the end of each page when records are modified concurrently with pagination; this issue is fixed in 2.6.0-rc9. Finally, users should be careful to explicitly pass a timestamp to paginated queries to avoid duplicated or missing items, or read skew, when traversing multiple pages. Some official client libraries, like the FaunaDB Javascript client, fail to do this correctly.

In 2.5.4 and 2.5.5, schema operations were non-transactional in several ways—for instance, one cannot create a class and index on that class in the same transaction, creating a class does not guarantee that one can immediately[5] insert instances into that class, and classes and indices can not be safely upserted. Since schema operations are generally infrequent in production environments, and rarely concurrent, we do not expect these issues to manifest often in production—although development and testing environments may be more likely to encounter them. These issues are now documented and scheduled to be addressed in 2.7, but users can work around some of them in the mean time: for instance, it's safe to sim-

[5] Class creation may take up to `cache_schema_ttl_seconds` seconds to take effect. By default, this can be up to 60 seconds.

ply ignore duplicate record errors when concurrently creating classes or indices.

Index creation in FaunaDB is asynchronous: newly created indices are queryable, but may return incomplete or transactionally invalid data for several minutes, depending on data volume and cluster state. Users should be careful to poll indices after creation, and avoid querying them until they return `active: true`. Fauna has changed this behavior in 2.6.1, so that querying an incomplete index throws an error.

Our work also uncovered several bugs in topology changes, mostly involving clusters which locked up while trying to remove nodes. Some of these issues were resolved in 2.6.0-rc9, and the remaining problems addressed in 2.6.0. However, our tests for topology changes did not exercise these processes thoroughly, and in general, topology changes are a hard problem. Users should exercise caution when adding and removing nodes.

FaunaDB is based on peer-reviewed research into transactional systems, combining Calvin's cross-shard transactional protocol with Raft's consensus system for individual shards. We believe Fauna's approach is fundamentally sound: the bugs that we've found appear to be implementation problems, and Fauna has shown a commitment to fixing these bugs as quickly as possible.

However, Fauna's documentation for consistency properties was sparse, inconsistent, and overly optimistic: claiming, for example, that FaunaDB offered "100% ACID" transactions and strict serializability, when, in fact, users might experience only snapshot isolation. We recommend that Fauna clarify that strict serializability only applies to read-write transactions using serializable indices, that read-only transactions may observe stale state, and that transactions interacting with default indices may only experience snapshot isolation.

Fauna has since dramatically expanded their documentation for isolation levels, discussing the isolation levels for different types of transactions, how to promote transactions to stronger isolation levels by adding writes and changing index flags, and the impact of those isolation levels on transactional correctness.

5.2 General Comments

FaunaDB's composable query language, temporal queries, and support for transactional consistency models ranging from snapshot isolation to strict serializability are welcome choices, and they work together well. For example, the `At` form establishes lexical temporal scope for any query expression, and allows users to compare two states of the database at different times in a single transaction. Many databases offer a total order of updates through snapshot isolation or even serializability, but making time explicit allows users to obtain consistent views across multiple transactions, and thread causality through multiple actors. We're pleased to see these ideas brought together in FaunaDB.

Many consensus systems rely on fixed node membership, which is cumbersome for operators. FaunaDB is designed to support online addition and removal of nodes with appropriate backpressure. Moreover, Fauna recently removed the need for manual assignment of log shard topologies, making FaunaDB membership fully dynamic. Membership changes are *notoriously* difficult to get right, especially in consensus systems, and we appreciate Fauna's efforts on behalf of their users.

We're also excited to see commercial adaptations of the Calvin paper, as it makes a distinct set of tradeoffs specifically intended for geographically distributed transaction processing. While Jepsen focuses on *safety*, rather than *performance*, we suspect that Calvin-based systems like FaunaDB could play an important future role in the distributed database landscape.

Finally, note that Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. While we believe FaunaDB's replication and transactional algorithm are theoretically sound, and although we make extensive efforts to uncover potential bugs, we cannot prove the correctness of FaunaDB in general.

5.3 Future Work

In keeping with its temporal model, FaunaDB allows any query to be expressed as a stream of change events. Our work did not evaluate this functionality in detail. Nor have we examined conjunctions of node failures with changes to cluster topology; topology changes are sensitive to failure, and our technique for introducing randomized topology changes could have deadlocked or made unsafe changes (e.g. removing the only copy of some data) in the presence of faults.

Indeed, our mechanism for causing automated topology changes remain fragile—our tests deadlock every few hours. While issues with nodes joining, parting, and restarting were addressed in FaunaDB just prior to the conclusion of our research, we have not yet re-

solved deadlock issues in the test suite itself. A more robust test suite could give us more confidence in the correctness of topology changes.

We have also not explored coordinated crashes (e.g. those affecting an entire replica, log partition, or cluster), which might expose weaknesses in error recovery and write-ahead logs, or filesystem-level faults.

FaunaDB has a more complex topology than many systems Jepsen has tested, involving log shards distributed across multiple nodes, each replicated across multiple replicas. The flow of messages through this system is more complex than single-shard systems like Zookeeper or Etcd, which makes the space of distinct meaningful faults larger. Moreover, with FaunaDB, we typically tested clusters of nine (rather than five) nodes, which further increases the space of potential faults. We chose to test the isolation of single nodes, of single replicas, and of partitions *within* replicas, but did not explore asymmetric or generally randomized partitions. Future work could employ more sophisticated failure modes.

This work was funded by Fauna, and conducted in accordance with the Jepsen ethics policy. We wish to thank the Fauna team for their invaluable assistance—especially Evan Weaver, Brandon Mitchell, Matt Freels, Jeff Smick, and Attila Szegedi.