JEPSEN

RavenDB 6.0.2

Kyle Kingsbury
2024-01-31

RavenDB is a document database which claims to offer ACID transactions, including Snapshot Isolation by default and Serializability with the strongest settings. Following the documentation's claim that a session "represents a single business transaction," we tested RavenDB 6.0.2 and found surprising behavior even in healthy, single-node clusters. Transactions lose updates by default. Both the optimistic concurrency and cluster-wide transaction modes allow fractured read: a serious anomaly forbidden by Snapshot Isolation and several weaker consistency models. Alternatively, RavenDB may not have interactive transactions at all. This work was performed independently without compensation, and conducted in accordance with the Jepsen ethics policy.

1 Background

RavenDB is a distributed document database which repeatedly advertises its support for ACID transactions.[1] It's intended for OLTP workloads, and offers a variety of ETL paths for exporting data to other systems. Its transactional API revolves around a session handle, which "represents a single business transaction on a particular database." Users create a session, perform operations like reads and writes, and finally call `session.saveChanges()` to commit their writes as an atomic unit.

RavenDB can replicate data across a set of nodes with automated failover. Sharding is either a work in progress or ready in 6.0, depending on which part of the documentation you're reading. RavenDB includes secondary indices wrapped with a homegrown query language, multiple revisions of documents, time series datatypes, and CRDT-based counters. In this text, we'll focus on RavenDB's transactional key-value operations.

1.1 Replication

Per RavenDB's High Availability page, the database accepts writes and reads across all nodes in the cluster. It uses the Raft consensus algorithm, which should theoretically allow RavenDB to provide consistency models up to Strong Serializability. However, that page goes on to say operators can "easily setup a topology in which end points operate … independently in case the network is disrupted." In a blog post, CEO Oren Eini repeats this claim:

> If a node is located in a place where the internet connectivity goes down, that node can continue to operate offline, taking in data locally. Once the connection is restored, the node will take the data it processed and replicate it throughout your cluster.

This would make ACID transactions impossible. The "I" in ACID refers to "Isolation": transactions must appear to execute independently, without interference from other transactions. This property is formalized as Serializability: equivalence to some totally ordered, non-concurrent execution of transactions. We know that totally available systems cannot offer Serializability or even Snapshot Isolation. RavenDB might offer Causal or Read Committed, but the stronger consistency models are theoretically off-limits.

A second page on high availability explains that there are two layers within RavenDB, and that Raft is used only for cluster metadata:

- First, the cluster layer is managed by a consensus protocol called Raft. In CAP theorem it is CP (consistent and partition tolerant).
- The second layer, the database layer, is AP (it is always available, even if there is a partition, and it's eventually consistent) and is handled by a gossip protocol between the databases on different nodes, forming multi-master mesh and replicating data between each other.

> RavenDB utilizes the different layers for different purposes. At the cluster layer, the consensus protocol ensures that operators have the peace of mind of knowing that their commands are accepted and followed. At the database layer you know that RavenDB will never lose writes and will always keep your data safe.

[1] As discussed in section 4.1, the interpretation of a "transaction" in RavenDB is complicated. In this report, we identify RavenDB's session API as an interactive transaction.

This is also confusing. AP systems are known for availability, not safety; lost update is a well-understood problem in AP registers. RavenDB claims to offer transactions with ACID guarantees. However, these transactions are apparently routed through an eventually-consistent, totally available replication system. There are databases which couple an (e.g.) Sequential transaction coordinator to an eventually-consistent datastore to provide Serializability, but it's not clear from this documentation how RavenDB links cluster and database layers together to ensure safety.

The Inside RavenDB chapter on cluster design confirms that Raft is used only for cluster metadata. Writes are allowed on every node, and are totally available:

> RavenDB uses multi-master replication inside a database, and it's always able to accept writes.
>
> In other words, even if the majority of the cluster is down, as long as a single node is available, we can still process reads and writes.

On the other hand, RavenDB's ACID Transactions in NoSQL post claims the opposite:

> As in the single node version, RavenDB commits a transaction with just one round of Raft consensus.

If Raft *is* involved in the transactions, RavenDB can offer up to Strong Serializability—but transactions cannot be totally available. Indeed, RavenDB's clustering documentation clarifies there are actually two separate transaction paths. The default mode is called a *single-node* transaction, which allows conflicts "when two clients try to modify the same set of documents on two different database nodes." A *cluster-wide transaction* uses Raft to prevent conflicts, allowing transactions to "favor consistency over availability." To execute a cluster-wide transaction, one must set `TransactionMode = CLUSTER_WIDE`.

What safety properties do these transaction paths guarantee? For this, we need to consider RavenDB's ACID claims in detail.

1.2 ACID

RavenDB's home page prominently advertises "ACID database transactions" "across multiple documents and across your entire cluster." Its ACID Database Transactions page explains that a database without transactions is "not much of a database." It boasts that RavenDB "guarantee[s] ACID without sacrificing performance" and notes that because of its distributed ACID guarantees, "developers are exempt from handling the numerous scenarios of partial data transfers and the intricacies of data storage."

RavenDB's ACID Transactions in NoSQL article explains that RavenDB "was capable of multi-document transactions since version 1.0":

> Because it was optimized with this in mind, there wasn't even a need for a non-ACID option. Any combination of database operations can be combined into an ACID transaction. As a user you never needed to implement ACID guarantees yourself, and you were free to design documents around your own requirements….
>
> RavenDB was designed to make one and only one round trip to the server per transaction. RavenDB's version of the session object tracks a series of commands, collects them as a batch, and sends them all to the server in a single round-trip when the method `session.saveChanges()` is called.

Again, RavenDB claims to have been "the pioneer database to offer ACID in a nonrelational context. In 2010, RavenDB offered ACID consistency across multiple documents." However, these guarantees held only on a single node: concurrent clients on different nodes could violate isolation. RavenDB 4.0, released in fall 2020, introduced the cluster-wide transaction path, which took transactions "from being ACID over multiple documents to being ACID over your entire cluster."

The Transaction FAQ says "all actions performed on documents are fully ACID" but contradicts itself immediately, saying "in a single transaction, all operations operate under snapshot isolation." DBDB takes this to mean that RavenDB offers Snapshot Isolation by default.[2] In a 2020 webinar, CEO Oren Eini confirmed this position: "RavenDB uses Snapshot Isolation by default, and transactions are effectively going to observe Serializable between operations that happen on the same node."

There are hints that RavenDB might provide something much weaker than Snapshot Isolation. Buried in the *Inside RavenDB* book, in the chapter on document modeling, is a section on concurrency control. This section explains that RavenDB (at least in version 4.0) performed no concurrency control, and instead used Last Write Wins conflict resolution by default.

> What happens if two requests are trying to modify the same document at the same time? That depends on what, exactly, you asked RavenDB to do. If you didn't do anything, RavenDB will execute those two modifications one at a time, and the last one will win. There's no way to control which would be last. Note that both operations will execute.

[2] Technically, this isn't clear from the documentation alone: Snapshot Isolation is a property of *histories* of transactions, but the docs discuss behavior only within the scope of a single transaction. RavenDB might have meant some weaker property here—for instance, Prefix Consistency. However, the CEO's claims of Snapshot Isolation by default seem authoritative.

This is the *opposite* of ACID isolation. Isolated transactions appear to execute sequentially, not concurrently. It also contradicts claims of Snapshot Isolation: Last Write Wins registers allow all kinds of anomalies which would be prohibited under Snapshot Isolation, including lost update. However, this book is two major releases out of date; it may not apply to 6.0.2.

What about cluster-wide transactions? The Cluster Transactions page seems definitive. "Concurrent cluster-wide transactions are guaranteed to appear as if they are run one at a time (`serializable` isolation level)."[3]

From this, Jepsen infers that RavenDB's default transaction settings should ensure Snapshot Isolation by default and Serializability in a single-node system. Cluster-wide transactions should ensure Serializability globally.

2 Test Design

In 2020 RavenDB wrote their own Jepsen test and declared in a webinar that per that test, "everything works." Their test checked the linearizability of individual reads and writes against a single document. It did not evaluate multi-operation or multi-document transactions.[4]

We designed a new test harness for RavenDB 6.0.2 running on a single Debian Bookworm node. Our test used RavenDB's JVM client library at version 5.0.4. We did not evaluate multi-node clusters or any kind of faults.

We wrote a single list-append workload using Elle to verify transactional isolation. This workload performs transactions over lists, each list identified by a unique integer ID. Each transaction consists of reads and/or appends of unique integers to those lists. Each worker thread in the test opens a single DocumentStore connected to the same node. Each transaction creates a new session, performs reads and/or appends, then calls `session.saveChanges()` to commit. Reads are encoded as a single call to `session.load(java.util.Map, id)`. Appends call `session.load` to read the current value, add their integer element to the end of the list, then call `session.store(map, key)`.

RavenDB offers a few knobs for tuning transaction safety: `transactionMode` and `optimisticConcurrency`. We ran our tests using the defaults (single-node transactions, no optimistic concurrency), with single-node transactions and optimistic concurrency, and finally with cluster-wide transactions. Cluster-wide transactions cannot be combined with optimistic concurrency.

3 Results

We found surprising safety errors in all three transaction modes.

3.1 Lost Update with Single-Node Transactions (#17927)

By default, RavenDB executes transactions with `transactionMode = SINGLE_NODE` and `optimisticConcurrency = false`. One might assume that `SINGLE_NODE` transactions are safe on single-node clusters. However, we found the default settings caused RavenDB to lose updates constantly, even in single-node clusters without faults.

For instance, in this five-second test run we performed 12,886 transactions over 975 keys. 81 of those keys exhibited a provable lost update. Here are two committed transactions involving key 830:

```
[[:r 830 [1 2]]
 [:append 824 11]
 [:append 807 14]
 [:append 830 3]]

[[:r 830 [1 2]]
 [:r 831 nil]
 [:r 831 nil]
 [:append 830 4]]}]}
```

Both of these transactions read key 830's value as the list `[1, 2]`. Both went on to append a value to key 830: the first transaction appended 3, and the second transaction appended 4. Neither saw the other's effects. In a Snapshot Isolated system, the *first-committer-wins* rule demands that one of these transactions must abort. RavenDB, however, allowed both transactions to commit. This is the definition of a lost update anomaly.

In an isolated transaction system which only ever appends elements to lists, every observed version of a single list must be a prefix of the longest version of that list. However, 454 of the keys in this test violated this prefix property, exhibiting *incompatible orders*. For example, here are all the reads of key 116:

[3] It is tempting to believe that cluster-wide transactions are the ACID transactions RavenDB's marketing boasts, but this cannot be the case. RavenDB says it's offered ACID transactions since 2010, and cluster-wide transactions weren't introduced until roughly a decade later. This may reflect confusion over what "ACID" means.

[4] RavenDB's Jepsen test may not have measured anything at all: at least in the most recent revision, the generator included no client operations of any kind.

Time (s)	Process	Value
3.01	1	[1 2]
3.01	1	[1 2]
3.01	0	[1 2]
3.01	1	[1 2 3]
3.01	0	[1 2 4]
3.01	1	[1 2 4 5]
3.01	0	[1 2 4]
3.02	1	[1 2 4 6]
3.02	1	[1 2 4 6]
3.02	1	[1 2 4 6 9]
3.02	1	[1 2 4 6 9 10]
3.02	1	[1 2 4 6 9 10 11 15]
3.02	0	[1 2 4 6 9 10 11 15]

Just over three seconds into the test, process 1 observed key 116's state as `[1 2 3]`. However, an immediately following read by process 0 saw `[1 2 4]`, and the write of 3 never appeared again. Process 1 then observed `[1 2 4 5]`. This write of 5 was replaced by 6 and never seen again.

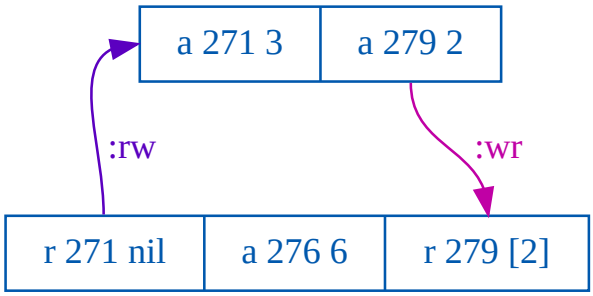
Our lost update checker is conservative: it only infers an anomaly if two transactions read the same version of some key *and* both write to it. However, our append operations are performed by reading the value, then writing back changes. This means these transactions contain reads which are effectively invisible to the checker. It seems likely that these cases of incompatible order also represent lost updates.

In total, 481 out of the 975 keys in this test exhibited lost updates or incompatible orders. These phenomena are prohibited by Serializability, Snapshot Isolation, and Repeatable Read. We've reported this as issue 17927 in RavenDB's issue tracker.

3.2 Fractured Reads with Optimistic Concurrency (#17929)

With the *optimistic concurrency* feature enabled, RavenDB promises to "generate a concurrency exception (and abort all modifications in the current transaction) when the document has been modified on the server side after the client received and modified it." When running on a single node, this setting does appear to prevent lost updates. However, it allows fractured reads—as well as various flavors of G-Single, G-nonadjacent, and G2-item. Again, these anomalies occurred in a healthy single-node system.

For instance, consider this ten-second test run in which every transaction enabled optimistic concurrency. Our checker found hundreds of anomalies like this:

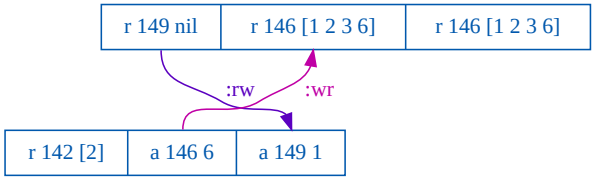


In this diagram the top transaction T_1 appended 3 to key 271, then appended 2 to key 279. The bottom transaction T_2 read key 271 and found nothing, appended 6 to key 276, and finally read key 279's value as `[2]`. Because T_2 failed to observe T_1's append to key 271, we have a read-write anti-dependency, denoted `rw`. Because T_2 observed T_1's append to key 279, we have a write-read dependency, denoted `wr`. In short, T_2 observed some, but not all, of the effects of T_1.

This anomaly is called *fractured read*, and it is prohibited under Read Atomic, Update Atomic, Causal, Prefix, Parallel Snapshot Isolation, Snapshot Isolation, Repeatable Read, and Serializable. RavenDB's Transaction FAQ promises Snapshot Isolation: "even if you access multiple documents, you'll get all of their state as it was in the beginning of the request." In all Snapshot Isolated databases Jepsen is familiar with, snapshots extend across multiple reads. In RavenDB, it appears each read can observe a different state. We've reported this as issue #17929 to RavenDB.

3.3 Fractured Read with Cluster-Wide Transactions (#17928)

Cluster-wide transactions are supposed to be Serializable. However, we found that even healthy, single-node clusters in which every transaction used `CLUSTER_WIDE` mode routinely exhibited fractured reads, as well as G-single, G-nonadjacent, G2-item, and more. Consider this five second test run, which contained hundreds of serializability violations. Here is one of those anomalies:



Here, the bottom transaction T_2 appended 6 to key 146 and 1 to key 149. The top transaction T_1 failed to observe T_2's append to key 149, but *did* observe its append to key 146. This is another instance of fractured read. As before, this behavior appears to be proscribed by RavenDB's documentation, as well as all consistency models above Read Atomic.

We've reported this to RavenDB as issue #17928.

№	Summary	Event Required	Fixed in
17927	Lost update with single-node transactions	None	Unresolved
17929	Fractured read with optimistic concurrency	None	Unresolved
17928	Fractured read with cluster-wide transactions	None	Unresolved

4 Discussion

RavenDB variously claims to offer "fully ACID" transactions, Serializability, or at least Snapshot Isolation. All of these claims appear false. RavenDB 6.0.2's default settings allowed lost updates. Even cluster-wide transactions exhibited fractured reads: a serious anomaly prohibited under Snapshot Isolation, as well as several weaker models. These behaviors occur even in healthy, single-node, single-shard systems, in which all access occurs via primary key.

RavenDB's strongest safety settings violate Read Atomic. It therefore cannot satisfy Update Atomic, Causal, Prefix, Parallel Snapshot Isolation, Snapshot Isolation, Repeatable Read, or Serializable. RavenDB might offer Read Committed or Monotonic Atomic View, but without more rigorous testing, Jepsen is hesitant to make this claim.

RavenDB's weak default behavior is surprising given RavenDB's repeated emphasis on safety. As CEO Oren Eini remarked on MongoDB's transaction safety settings:

> [Default] values matter. They matter quite a lot. Why is that? Because if you choose the bad values, you're absolutely going to get some great numbers in benchmark performance. But then you are going to be hitting those [safety] issues in production. And then there is this classic response: "Oh, you should have read the docs and used the proper configuration."

One wonders: if ACID properties are so important for RavenDB's users, why do the default settings allow lost updates, even on single-key operations? Do users realize their updates can be silently discarded? How many are taking care to use cluster-wide transactions where lost updates would violate safety? Do they know that even cluster-wide transactions allow fractured read?

This report follows a cursory investigation into RavenDB's behavior—it is by no means exhaustive. As always, we caution that Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. There may be other anomalies in RavenDB.

4.1 Does RavenDB Even Have Transactions?

The first sentence of RavenDB's cluster transaction documentation appears quite clear:

> A session represents a single business transaction.

This is echoed by the first sentence of RavenDB's session documentation:

> The Session, which is obtained from the Document Store, is a Unit of Work that represents a single business transaction on a particular database.

… which goes on to say:

> The batched operations that are sent in the `SaveChanges()` will complete transactionally. In other words, either all changes are saved as a Single Atomic Transaction or none of them are. So once SaveChanges returns successfully, it is guaranteed that all changes are persisted to the database.

RavenDB sessions are clearly not intended to work like sessions in typical databases, which are (roughly speaking) one-to-one with client connections.[5] They come with a default limit of 30 network requests; typical database sessions are unbounded. They buffer writes; Jepsen is unaware of any other database whose sessions do this. They cache reads; most sessions do not. They include concurrency control mechanisms like lost update prevention; Jepsen is unaware of any other database which does this at the session level. These are all hallmarks of what most databases would call a transaction.

RavenDB's article ACID Transactions in NoSQL? RavenDB vs MongoDB is emphatic: RavenDB has supported ACID transactions over "any combination of database operations" for over a decade. It certainly appears as if RavenDB sessions are intended for this role!

However, in a response to issue 17927, Eini (a.k.a. Ayende Rahien) explained that sessions are *not* in fact transactions:[6]

> Crucially, RavenDB does not attempt to provide transactional semantics over the entire session, rather it provide[s] transactions over individual requests.

And in response to issue #17928, Eini affirms:

[5] There are scenarios in which a database multiplexes multiple sessions onto a single connection, or migrates a session across connections. It might be more apt to think of a typical database session as "a logical, single-threaded connection."

[6] In the same thread, Eini remarks that "we only consider transactions to be the calls to SaveChanges or other data mutation operations." This is somewhat alarming: it implies that RavenDB transactions don't encompass reads at all. On the other hand, a transaction involving a single HTTP request "applies to reads as well," so it's not quite clear what RavenDB's read safety semantics are.

> A transaction in RavenDB is a request - so TX1 and TX2 above aren't actually single transactions, instead, each of them represent 3 independent transactions.

This is a striking viewpoint: the point of transactions is generally to provide isolation across multiple requests.[7] Moreover, RavenDB's optimistic concurrency and cluster-wide transaction mechanisms are clearly intended to provide transactional isolation which spans from a session's reads to its writes. Furthermore, Eini directly compared RavenDB sessions to MongoDB transactions (which offer typical interactive transaction semantics) and claimed that unlike MongoDB, RavenDB sessions actually satisfied Snapshot Isolation. Yet per Eini's comments, RavenDB *does not have interactive transactions at all*.

Repeatedly advertising "ACID transactions" across "any combination of database operations," telling users that a "session represents a single business transaction," comparing RavenDB sessions to interactive transactions in other databases, offering concurrency control mechanisms whose scope extends across an entire session, and finally expecting users to realize that sessions are not transactions at all—that a transaction is actually limited to a single HTTP request—stretches credulity.

Jepsen strives to evaluate databases in the context of their marketing and documentation. Although RavenDB's CEO now states "we don't support a transaction over more than a single HTTP request," RavenDB's documentation and marketing give every appearance that a session is intended to be a transaction. Jepsen has consulted with several software engineers on their interpretation of these claims, and believes typical database users would come to the same conclusion: RavenDB sessions are transactions. We continue this interpretation throughout this report.

4.2 Recommendations

RavenDB users should be aware RavenDB transactions are not ACID in any meaningful sense.[8] This holds even in single-node, single-shard deployments. The defaults allow lost updates: you should expect some of your writes to be silently discarded. The strongest safety settings allow fractured read: you might observe some, but not all, of another transaction's effects. You could appear to write "into the middle" of another transaction. The two isolation levels RavenDB advertises—Snapshot Isolation and Serializable—appear impossible to obtain.

Users who designed their applications assuming RavenDB provided interactive ACID transactions—or even Snapshot Isolation—should carefully reevaluate their transactions to ensure they are safe in the presence of these anomalies. Consider writing simple tests to verify application invariants are preserved under concurrent execution: the issues in this report are easy to reproduce.

Jepsen recommends RavenDB remove claims of "ACID", "Serializable", and "Snapshot Isolation" from their marketing materials and documentation. RavenDB should instead make specific, accurate, and internally consistent claims about safety properties. For instance, RavenDB might say "transactions offer Read Committed by default, plus internal consistency within the scope of a transaction: once a transaction reads a key, subsequent reads and writes of that key observe the originally read state, plus the effects of that particular transaction's writes. Transactions allow lost update by default. Enabling cluster-wide transactions prevents lost update, but still allows fractured read," and so on.

RavenDB's documentation is remarkably confusing. It repeatedly claims to offer ACID transactions, which implies Serializability. There are specific claims that RavenDB ensures either Serializability or Snapshot Isolation. However, the documentation also says that RavenDB's database layer is an AP system based on Last Write Wins, and the marketing material claims isolated nodes can operate independently. This is impossible: totally available systems cannot provide Serializability or Snapshot Isolation.[9]

There are systems (like Riak & Cassandra) which allow clients to execute either totally available operations with weak consistency, or majority available operations with stronger guarantees, like Linearizability. If RavenDB intends to build a system which supports both modes, they should clearly distinguish those modes throughout marketing and documentation. They have completely different availability, latency, and safety characteristics. Repeated claims that RavenDB provides ACID "without sacrificing performance" are provably impossible, and should be rewritten to clearly explain the tradeoffs involved.

ACID transactions are clearly important to RavenDB. It is therefore alarming that RavenDB's documentation and GitHub comments fundamentally disagree on what a transaction *is*. In one interpretation, RavenDB offers interactive transactions, represented by the session API, which provide relatively weak isolation—certainly not ACID. In another interpretation, RavenDB lacks interactive transactions altogether. Instead, it offers a sort of micro-transaction which (e.g.) writes multiple documents in a single network request. In this world, sessions offer varying, weak consistency constraints that extend *between* micro-transactions.

To resolve this confusion, RavenDB should pick a single definition of "transaction" and stick with it. The

[7] There are databases, like FaunaDB, where transactions are written as small programs and submitted to the database in a single request. RavenDB, like most databases, provides interactive sessions: clients make calls to `load` and `store` interspersed with arbitrary local computation, and call `saveChanges()` to commit their effects. In these kinds of systems, transactions typically encompass multiple read and write requests.

[8] Alternatively, "RavenDB does not have interactive transactions at all." Readers may select their favorite interpretation throughout this report.

[9] At least, not in a network which can partition.

equivalence or difference between a transaction and session should be clearly explained, and these terms used consistently throughout marketing and documentation. RavenDB should provide guidance as to the boundaries of each unit: when are multiple calls to `load` performed in a single transaction? What about `store`? Can a single transaction encompass both a `load` and `store`? The consistency properties of both transactions and sessions should be clearly and formally defined. Are transactions Serializable? Do sessions ensure Monotonic Atomic View? When does a session preclude lost update, and when does it allow it? Above all, do not tell users that sessions "represent a single business transaction" if they are, in point of fact, not transactions at all.

Finally, if RavenDB transactions are truly intended to cover only a single network request, consider using a different term altogether, and avoid comparisons to databases which do have interactive transactions. Some databases call these "mini-" or "micro-transactions," which provides an obvious hint of their limited scope.

4.3 Future Work

This work evaluated only single-node RavenDB clusters without faults. Future research could expand tests across multiple nodes, as well as introducing network, process, and disk faults. We dealt only with key-value operations, and did not evaluate RavenDB's secondary indices. These indices are described as eventually consistent, which raises questions around the integrity of predicate reads. RavenDB also offers server-side transactions using Javascript or a library of built-in patch operations. These might offer different safety characteristics than the interactive transactions we used in this report. Finally, cross-shard transactions are a notoriously challenging problem and deserve careful testing.

Jepsen wishes to thank Irene Kannyo for her invaluable editorial support. Thanks as well to C. Scott Andreas, Taber Bain, Silvia Botros, Coda Hale, Ben Linsay, Kelly Shortridge, Nathan Taylor, Zach Tellman, and Leif Walsh for their comments on early versions of this manuscript. This work was performed independently without compensation, in accordance with the Jepsen ethics policy.