JEPSEN

Datomic Pro 1.0.7075

Kyle Kingsbury
2024-05-15

Datomic is a temporal Entity-Attribute-Value OLTP database which supports non-interactive transactions on top of pluggable storage engines. It offers a variety of query mechanisms across thick and thin clients, including Datalog, graph traversal, and an ODM-style API. We evaluated Datomic Pro 1.0.7075 and found its inter-transaction safety properties appear stronger than claimed. Not only was every history Serializable, but sessions bound to a single peer appear Strong Session Serializable, and histories restricted to write transactions and reads using `d/sync` appear Strong Serializable. However, inside of a transaction Datomic behaves as if operations were evaluated concurrently. Depending on how one interprets those operations, this might violate three of the most widely accepted formalizations of Serializability, each of which specify serial intra-transaction semantics. It also creates the potential for invariant violations when composing transaction functions. Datomic has published a companion blog post alongside this report. This work was funded by Nubank (Nu Pagamentos S.A), and conducted in accordance with the Jepsen ethics policy.

1 Background

Datomic is a general-purpose database intended for systems of record. In many ways, Datomic is unusual. At any instant in time, the state of the database is represented by a set of `[entity, attribute, value]` (EAV) triples, known as *datoms*. Each datom declares that some *entity* (like a person) has a particular *attribute* (like a name) with a specific *value* (like "Vidrun"). The types and cardinality of attributes are controlled by a schema.

Datomic is also a temporal database: it models time explicitly. Every transaction is identified by a strictly monotonic logical timestamp `t`, as well as a wall-clock time `txInstant`. Transactions can *assert* a datom, adding it to the database, or they can *retract* a datom, removing it from the database. Every datom also retains a reference to the transaction that asserted or retracted it. A full datom is therefore a five-tuple of `[entity, attribute, value, transaction, asserted-or-retracted?]`. The database is an ever-growing set of these tuples.[1]

Users can request a snapshot state of the database at any logical or wall-clock time—right now or years in the past. They can also obtain a full view of the database's history, allowing users to ask questions like "was there ever a time when Janelle Monáe and Cindi Mayweather were recorded in the same room together?"

Given a state of the database, users may query it via a Datalog-style API, a declarative graph traversal API, or an ODM-style `Entity` datatype which allows lazy access to an entity's associated values, including other entities.

Datomic comes in two flavors. In this report we discuss Datomic Pro, which anyone can run on their own computers. Datomic Cloud runs in AWS and uses a somewhat different architecture.

1.1 Architecture

Datomic Pro comprises several co-operating services. *Transactors* execute write transactions, maintain indices, and write data to storage. *Peers* are thick clients: they embed a JVM library which submits transactions to transactors, executes read queries against storage, and caches results. For applications written in other languages, Datomic also has a traditional client-server model. *Clients* are thin clients which forward transactions and queries to a *peer server*: a peer which runs a small network API.

Internally, Datomic appends each transaction to the *log*: a time-ordered set of transactions. From the log Datomic maintains four indices sorted by different permutations of entity, attribute, value, and time. These indices allow efficient queries like "which entities were modified yesterday," or "who run the world?"[2]

Both log and indices are stored as persistent, immutable trees in a data store like Cassandra or DynamoDB. Because tree nodes are immutable, their backing storage only needs to guarantee eventual consistency. A small pointer to the roots of these trees provides a consistent, immutable snapshot of the database's state. To commit a transaction, a transactor saves new immutable tree nodes to storage,

[1] Datomic also provides an excision mechanism which rewrites history to permanently delete datoms. This is useful for regulatory compliance or removing unwanted PII.

[2] Girls!

then executes a compare-and-set (CaS) operation to advance the root pointer. This CaS operation must execute under Sequential consistency.

Using a Sequential CaS operation ensures a global order of transactions, and limits Datomic's write throughput to the speed of a single transactor. To reduce contention, Datomic tries to have a single active transactor at all times. Operators typically deploy multiple transactors for fault tolerance.

Peers connect directly to storage, and also to transactors. Transactions are forwarded to an active transactor, which executes them. Each peer also maintains a local, monotonically-advancing copy of the root pointer, which allows the peer to read tree nodes from storage. Since tree nodes are immutable, they can be trivially cached. There may be any number of peers, allowing near-linear read scalability.

1.2 Transaction Model

Datomic has an unusual transaction model. Most OLTP databases offer interactive transactions: one begins a transaction, submits an operation, receives results from that operation, submits another, and so on before finally committing. Some databases, like VoltDB, use stored procedures: an operator writes a small program which is installed in the database. Clients invoke that program by name, which mutates database state and returns values to the client. Other databases like FaunaDB allow clients to directly submit miniature programs as text or an abstract syntax tree. Like stored procedures, these programs perform arbitrary reads and writes, mutate state, and return data to the user.

Datomic does something rather different. It enforces a strict separation between read and write paths. There are no interactive transactions. It has stored procedures, but they cannot return values to the caller.

A read obtains an immutable state of the entire database. For instance, the `db` function returns the most recent database state[3] the peer is aware of. To obtain the most recent state across all peers, or a state later than a given time, one calls `d/sync`. To obtain a state from a past time (seconds or years ago), one calls d/as-of. These states are cheap, highly cacheable, and never block other writers or readers.

Given a database state, one can run any number of queries using (e.g.) `q` or `pull`. Queries lazily fetch datoms from cache or storage. Since database states are immutable, any number of queries run against the same state occur at the exact same logical time. In this sense, all queries run on the same state take place in a single atomic transaction—even two queries executed on different machines, months apart.

Write transactions[4] are represented as an ordered list of operations.[5]

> A transaction is simply a list of lists and/or maps, each of which is a statement in the transaction.

For example, here is a transaction of three operations, all involving entity `123`:

```
[[:db/add 123 :person/name "N. K. Jemisin"]
 [:db/cas 123 :author/hugo-count 2 3]]
 [:author/add-book 123 "The Stone Sky"]]
```

Those operations may be simple assertions (`:db/add`) or retractions (`:db/retract`) of datoms, or they may be calls to transaction functions: either built-in or user-defined. In this example, the built-in `db/cas` function performs a CaS operation, asserting the number of Hugo awards for this author is 3 if and only if that number is currently 2. One can also store a function (represented as a Clojure AST or Java string) in the database just like any other value. Alternatively, one may write a function in any JVM language, and provide it in a `jar` file on the transactor's classpath. Once a function has been installed, any transaction may invoke it by providing the function's name and arguments. Here, the `author/add-book` function receives the state of the database as of the start of the transaction, as well as any arguments from the transaction. It can perform arbitrary (pure) computation, including running queries against the database state. It then returns a new set of operations for the transaction—for instance, assertions, retractions, or calls to other functions. Function calls are recursively expanded until only assertions and retractions remain.

While transaction functions can make decisions based on the results of reads they perform internally, there is no channel to return those reads (or other information) to the caller of `transact`. Transactions *only* return effects. This means there is no direct analogue for an arbitrary read-write transaction in Datomic! For example, you can write a function which performs a conditional write,[6] but you can't inform the caller whether the write took place or not. This constraint nudges Datomic users towards pulling reads out of the write transaction path—a key factor in obtaining good performance from a system which can logically execute only one write transaction at a time.

[3] Datomic refers to an immutable version of the database as a "value". To avoid confusion with other kinds of values in this report, we call this a "database state".

[4] Most systems use "transaction" to refer to a group of operations, including reads or writes, executed as a unit. Datomic uses "transaction" to refer specifically to a write transaction—i.e. a call to `d/transact`. However, Datomic's reads are trivially transactional as well. We refer to both reads and writes as transactions in this work—it significantly simplifies our discussion of consistency models.

[5] At the start of our collaboration, Datomic used "statement", "operation", and "data" to refer to elements of a transaction. We use "operation" in this report for consistency with the database literature, and to avoid confusion with other kinds of data. Datomic intends to refer to transaction elements solely as "data" going forward.

[6] Datomic wishes to note that transaction functions (for instance, `db/cas`) do not actually perform writes. They produce data structures which represent *requests* for writes. Those writes are performed during the final stages of transaction execution. Delayed evaluation of transaction effects is a common database technique; we use the term "write" loosely with this understanding.

Instead of offering arbitrary return values from transactions, every call to `transact` returns the database state just before the transaction, the database state the transaction produced, and the set of datoms the transaction expanded to. This allows callers to execute read-write transactions by splitting them in twain: they submit a transaction which performs some writes, then use the pre-state of the database to determine what data that transaction would have read. Peers can also examine the post-state and set of datoms produced by the transaction to (e.g.) determine whether a conditional write took place.

From the perspective of traditional database systems, this sounds absurd. Mixed read-write transactions are a staple of OLTP workloads—how could you get anything done without them? Datomic offers a view of an alternate universe: one where database snapshots are cheap, efficient, and can be passed from node to node with just a timestamp. From this point of view, other databases feel impoverished. What do you mean, Postgres can't give you the state of the entire database a transaction observed? The lack of a return channel for transaction functions may be annoying, but Datomic's other strengths generally allow it to solve the same kinds of problems as a traditional, interactive transaction system. For example, NuBank (Datomic's current developers) offers financial services to nearly 94 million users, processing an average of 2.3 billion user transactions per day. Almost all of their products use Datomic as a system of record.

1.3 Consistency

Datomic advertises ACID transactions and means it: their ACID documentation makes detailed, specific promises with respect to consistency models and durability guarantees. Transactions are "written to storage in a single atomic write," which precludes intermediate read anomalies. Every peer "sees completed transactions as of a particular point in time," and observes *all* transactions, totally ordered, up to that time. Transactions are always flushed to durable storage before client acknowledgement.

When our analysis began in early January 2024, Datomic's documentation informally claimed write transactions were Serializable:

> The Isolation property ensures that concurrent transactions result in the same system state that would result if the transactions were executed serially.

Since write transactions are Serializable and execute atomically, and since read-only queries execute against committed snapshots, it seems plausible that histories of both read and write transactions should also be Serializable.[7]

Serializability does not impose real-time or session ordering constraints: in a Serializable system, it is legal for a client to execute a transaction which inserts object x, then execute a second transaction which fails to observe x. While Datomic's documentation does not make this claim, it seems plausible that Datomic's transactor design might provide Strong Serializability over write transactions, preventing real-time anomalies.

Since `d/db` returns an asynchronously updated copy of the database, we expect peers to observe stale reads. Indeed, Datomic is explicit that peer reads may not observe some recently committed transactions. However, it would be straightforward for peers to ensure that their time basis advances monotonically; if we say that every session is bound to a single peer node, we would expect to observe Strong Session Serializable histories.

In addition to these possible realtime and session constraints, Datomic has multiple synchronization mechanisms. Clients can block until they observe a value of the database at or later than some time `t`. This enables clients to ensure consistency when threading state through side channels. Calling `d/sync` forces the client to synchronize with the transactor, preventing stale reads. We expect histories which always use `d/sync` to be Strict Serializable as well.

Datomic's documentation also described it as a "single-writer" system:

> A single thread in a single process is responsible for writing transactions. The Isolation property follows automatically from this, because there are no concurrent transactions. Transactions are always executed serially.

This is wrong in two senses. First, Datomic is fault-tolerant: one can and should run several transactor nodes on different computers. Typically one transactor is active and the others are in standby. When a transactor's failure detector believes there is no active transactor, it will attempt to promote itself to active. However, perfect failure detectors are impossible in asynchronous networks. There may be times when a standby transactor believes it should take over, but another active transactor is still running. This means transactions may actually execute concurrently. During this window Datomic is not a single-writer system, but a multi-writer one!

Second, even if there were a perfect failure detector which ensured a single Datomic transactor at a time, its messages to storage could be arbitrarily delayed by the network and arrive interleaved with messages from other transactors. Thankfully this doesn't matter: Datomic's safety property follows directly from the Sequential consistency of the storage system's CaS operation. Any number of concurrent transactors ought to be safe.

[7] As Fekete, O'Neil, and O'Neil point out, adding read-only transactions to a history which is otherwise Serializable can actually yield non-Serializable histories! However, this paper applies specifically to Snapshot Isolation, where two transactions may read the same state and write new values concurrently. Datomic's design ensures transactions are atomic, in the sense that no two transactions overlap in the window between their read and write timestamps.

2 Test Design

We designed a test suite for Datomic using the Jepsen testing library. Our test installed Datomic Pro 1.0.7075 on a cluster of Debian Bookworm nodes. For storage, it provisioned a DynamoDB table in AWS. Two of the test nodes ran transactors, and the remaining nodes ran peers.

Our peers were small Clojure programs which used the Datomic peer library. They connected to storage and transactors and exposed a small HTTP API for performing test suite operations. For each operation the test used an HTTP client to submit that operation to some peer. The peer executed that operation using the peer library, and returned a result to the client. We ran our workloads both using `d/db`, which may yield stale reads, and also with `d/sync`, which blocks but guarantees recency.

Our test harness injected faults into both transactors and peers, including process pauses, crashes, and clock errors. We created network partitions between nodes (including both transactors and peers) and between nodes and the storage system. We also requested Datomic perform garbage collection.

Datomic transactors kill themselves when they cannot maintain a stable connection to storage. When we ran transactors with the default 5-second timeout settings on nodes outside AWS, transactors routinely killed themselves every few minutes due to normal network fluctuations. With a 1-second timeout, even transactors running in our EC2 test environment would kill themselves roughly every 10–20 minutes. To work around this, Datomic advises that operators run their own supervisor daemons to restart transactors. We used a systemd service with `Restart=on-failure`.

Our test suite included four workloads.

2.1 List Append

We designed an append workload for use with the Elle transaction checker. Logically, this workload operates over lists of integer elements, with each list identified by an integer primary key. Clients perform transactions comprising random operations. Each operation may read the current value of a list, or append a unique element to the end of a list. Elle then performs a broad array of checks on the history of transactions. It looks for aborted and intermediate reads, violations of internal consistency, and inconsistent orders of elements across different reads of a list. From the element orders, it infers write-write, write-read, and read-write dependencies between transactions. From the order of transactions on each logical process, and the global order of transactions, it infers per-process and real-time orders, respectively. Elle then searches for cycles in the resulting dependency graphs. Various cycles correspond to violations of different consistency models, like Strict Serializability.

We encoded our lists in Datomic as follows. Each list was represented by a single entity with two attributes. One, `append/key`, served as the primary key. The other, `append/elements`, was a many-valued attribute which stored all the integer elements in a given list.

Performing the writes in a transaction was straightforward: given a write, we emitted a single operation for the transaction stating that the given key now had that element: `{:append/key k, :append/elements element}`. To perform a read of `k`, we read a local cache of `k`'s elements. We populated that cache with an initial series of read queries, then used a small state machine to simulate internal reads.

Note that multi-valued attributes represent an unordered *set*, not an ordered list. Elle's inference uses the order of list elements to infer the serialization order of transactions. To obtain this order, we took advantage of the fact that Datomic is a temporal database: every datom includes a reference to the transaction which wrote it. When we read the elements for a given key, we also included their corresponding transactions. We then sorted the elements by transaction times, which provides exactly the order Elle needs.

Elle's list-append workload is designed for databases which offer mixed read-write transactions, but Datomic doesn't have this concept. As previously mentioned, any read-write transaction can be expressed in Datomic by running the writes in a transaction function, then using the returned database state to determine what the transaction's reads would have been. We used this technique in our workload: a single function executes the transaction, simulates internal reads, and produces side effects (for the write transaction) and completed reads (to be returned to the client). We execute this function twice: once using a stored procedure via `transact`, then a second time on the peer to fill in reads, using the pre-state of the database `transact` returned.

2.2 List Append with CaS

Many Datomic users use the built-in compare-and-set function `db/cas` to control concurrent updates to an attribute of an entity outside a transaction. For example, they might read the current DB state using `d/db`, read the value of a counter as 4, then increment the counter's value using `[:db/cas 123 :counter/value 4 5]`. The CaS function asserts the new value 5 if and only if the current value is 4.

Datomic guarantees transactions are always Serializable, but a user might want to express a logical "user transaction" consisting of a read followed by a separate write transaction. Since Datomic database states are always complete snapshots, and transactions are Serializable, using `db/cas` for every write[8] allows users to build an ad hoc Snapshot Isolation over these user transactions.

[8]We say "every" write for safety and clarity. In practice, users often arrange for all transactions requiring concurrency control to conflict on a single attribute of an entity. A single CaS operation on, say, a customer's `version` attribute could ensure that any number of updates to that customer occur sequentially.

Our append-cas workload provides the same logical API as the list-append workload, but uses this CaS pattern to ensure Snapshot Isolation. Instead of multi-valued elements, we encoded each list as a single-valued, comma-separated string. We performed a read at the start of each transaction, applied reads and writes locally, maintaining a buffer of written values, then constructed a transaction of CaS operations which ensured that any values we wrote had not been modified since they were read.

2.3 Internal

Our two list-append workloads measured safety between transactions, but because they simulated the results of internal reads, they did not measure Datomics intra-transaction semantics. We designed an internal workload which measures internal consistency with a suite of hand-crafted transactions. For instance, we assert that the value for some attribute of an entity is 1, then 2. We assert and retract a fact in the same transaction. We assert a value, then try to CaS it to something else. We perform multiple CaS operations—trying to change 1 to 2, then 2 to 3. We create an entity, then modify it using a lookup ref. Using a transaction function, we attempt to increment a value twice, and so on.

2.4 Grant

To ensure that transaction functions preserved function invariants, we designed a grant workload which simulates a simple state machine using transaction functions. Grants are first created, then can either be approved or denied. We encode a grant as a single entity with three attributes: `created-at`, `approved-at`, and `denied-at`.

No grant should be *both* approved and denied. We ensure this invariant by writing a pair of transaction functions `approve` and `deny`. Each first checks that the grant under consideration has not been approved or denied already, aborting the transaction if necessary. If the grant hasn't been approved or denied yet, `approve` adds the grant's `approved-at` date. Our `deny` function works the same way.

Our grant workload creates a new grant in one transaction. In subsequent transactions it tries to approve and/or deny the grant. We repeat this process, exploring different combinations of functions and transaction boundaries. We check to make sure that no grant is both approved and denied.

3 Results

We found no behavior which violated Datomic's core safety claims. Transactions appeared to execute as if they had been applied in a total order, and that order was consistent with the local order of operations on each peer. Histories restricted to just those transactions performing writes, and histories in which reads used `(d/sync conn)` to obtain a current copy of the database, were consistent with real-time order.

However, we did observe unusual behavior within transactions. This intra-transaction behavior is generally consistent with Datomic's documentation, but it represents a significant departure both from typical database behavior and the major formalisms used to model transactional isolation. We discuss those divergences here.

3.1 Internal Consistency

Virtually all databases and formalisms Jepsen is familiar with provide serial execution semantics within a transaction. For example, a transaction like `set x = 1; read x;` would print `1`, rather than the value of `x` when the transaction started.

Although Datomic transactions are ordered lists of operations, Datomic does not preserve this order in execution. Instead, all operations within a transaction (adds, retracts, and transaction functions) are executed as if they were concurrent with one another. Transaction functions always observe the state of the database at the beginning of the transaction. They do not observe prior assertions, retractions, or transaction functions. For example, consider these results from our internal workload. Imagine entity `123` currently has an `:internal/value` of `0`, and we execute the following transaction:

```
[[:db/cas 123 :internal/value 0 1]
 [:db/cas 123 :internal/value 0 1]]
```

In a serial execution model, this transaction would fail: the first CaS would alter the value of key `123` from `0` to `1`, and the second CaS would fail, since the current value was `1` and not `0`. In Datomic, both CaS operations observe the initial state `0`, and both succeed. They produce a pair of redundant assertions `[:db/add 123 :interval/value 1]`, and the value of entity `123` becomes `1`.

This means that state transitions may not compose as one expects. For instance, here is a transaction function that increments the value of the entity with key `k`:

```
(defn increment
  [db k]
  (let [{:keys [id value]} (read db k)]
    [[:db/add id :internal/value (inc value)]]))
```

What value does the following transaction produce, given an entity with key `"x"` and value `0`?

```
[['internal/increment "x"]
 ['internal/increment "x"]]
```

In a serial model, the result of two increments would be 2. In Datomic, it's `1`: both `increment` functions receive the database state from the start of the transaction. Similarly, transaction functions do not observe lexically prior assertions or retractions.

```
[[:db/add id-of-x :internal/value 1]
 ['internal/increment "x"]]
```

This produces a final value of 1, not 2.

Likewise, lookup refs use the state of the database as of the start of the transaction. This means a transaction which adds an entity cannot use a lookup ref to refer to it later in that same transaction. The following transaction aborts with an `Unable to resolve entity` message:

```
[; Create an entity with key "x"
 [:db/add "x" :internal/key "x"]
 ; And set the value of the entity with key x
 ; to 0:
 [:db/add [:internal/key "x"] :internal/value 0]]
```

Many of the above transactions included multiple assertion requests with the same entity, attribute, and value. What happens if the values conflict? Imagine this transaction executes on a state where x's value is 0.

```
[[:db/add id-of-x :internal/value 2]
 ['internal/increment "x"]]
```

In a database with serial intra-transaction semantics, this would produce the value 3. In Datomic, the increment observes the start-of-transaction value 0. It completes successfully, and the transaction expands to the following:

```
[[:db/add id-of-x :internal/value 2]
 [:db/add id-of-x :internal/value 1]]
```

If this were executed by a serial database, it would produce the value 1. But Datomic's order-free semantics have another rule we have not yet discussed. If two assertions in the same transaction have different values for the same single-cardinality attribute of the same entity, the transaction aborts with `:db.error/datoms-conflict`. This transaction aborts!

This in-transaction conflict detection mechanism likely rules out many cases where the use of transaction functions would produce surprising results. A pair of increments will silently produce a single increment, but this is only possible because they all expand to compatible `[entity, attribute, value]` triples. Since there are an infinite number of incompatible values, and a single compatible choice for any `[entity, attribute]` pair, it seems likely that users who accidentally compose transaction functions incorrectly will find their transactions fail due to conflicts, and recognize their mistake.

This behavior may be surprising, but it is generally consistent with Datomic's documentation. Nubank does not intend to alter this behavior, and we do not consider it a bug.

3.2 Pseudo Write Skew

The fact that transactions appear to execute in serial, but the operations *within* a transaction appear to execute concurrently, creates an apparent paradox. A set of transaction functions might be correct when executed in separate transactions, but *incorrect* when executed in the same transaction! While Datomic's in-transaction conflict checker prevents conflicts on a (single-cardinality) `[entity, attribute]` pair, it does nothing to control concurrency of functions which produce disjoint `[entity, attribute]` pairs.

We designed the grant workload to illustrate this scenario. Following the documentation's advice that transaction functions "can atomically analyze and transform database values," and can be used to "ensure atomic read-modify-update processing, and integrity constraints," we wrote a pair of transaction functions `approve` and `deny`. These functions encode the two legal state transitions for a single grant.

```
(defn approved?
  "Has a grant been approved?"
  [db id]
  (-> '{:find  [?t]
        :in    [$ ?id]
        :where [[?id :grant/approved-at ?t]]}
      (d/q db id)
      count
      pos?))

(defn ensure-fresh
  "Throws if the given grant ID
  is approved or denied."
  [db id]
  (when (approved? db id)
    (throw+ {:type :already-approved}))
  (when (denied? db id)
    (throw+ {:type :already-denied})))

(defn approve
  "Approves a grant by ID. Ensures the
  grant has not been approved or denied."
  [db id]
  (ensure-fresh db id)
  [[:db/add id :grant/approved-at (Date.)]])
```

The `denied?` and `deny` functions are identical to `approved?` and `approve`, except they use the `denied-at` attribute; we omit them for brevity.

By ensuring that the given grant ID is fresh (i.e. neither approved nor denied), these functions ensure an important invariant: no sequence of `approve` and/or `deny` calls can produce a grant which is both approved and denied. And indeed, Datomic's Serializable transactions guarantee this invariant holds—so long as calls to `approve` and `deny` only ever take place in *different* transactions.

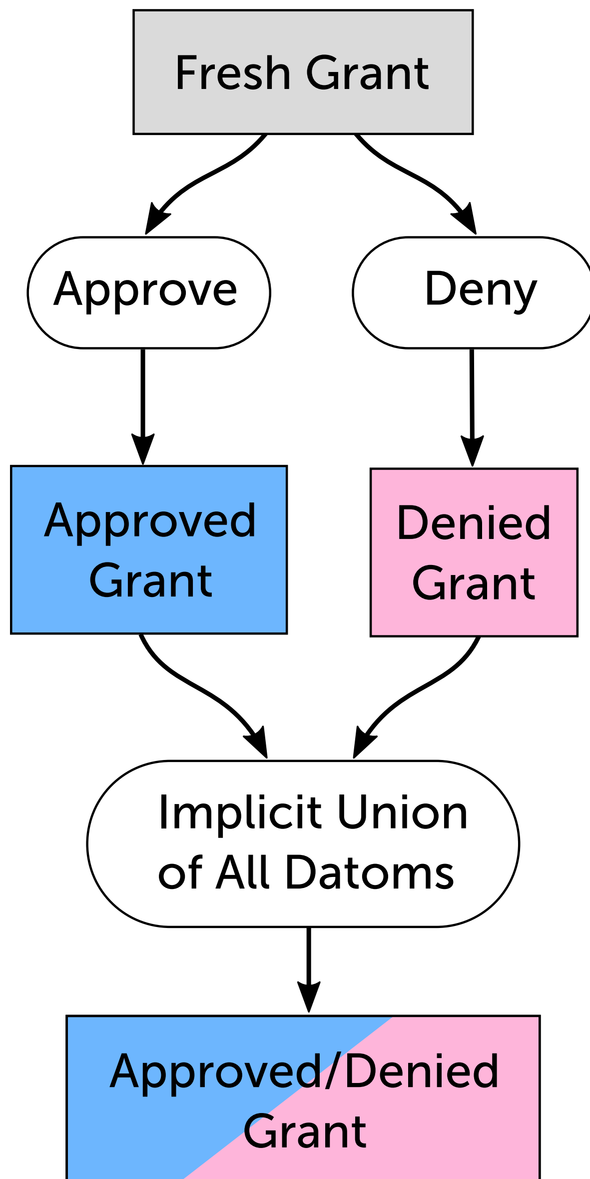
However, if a single transaction happens to call both `approve` and `deny`, something very interesting occurs:

```
[['grant/approve id]
 ['grant/deny id]]
```

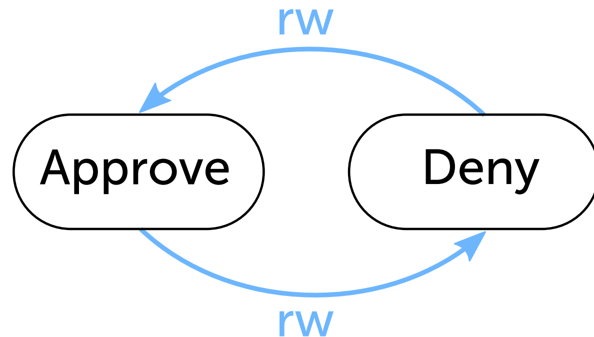
This transaction produces a grant with the following state:

```
{:db/id             17592186045426,
 :grant/created-at  #inst "2024-02-01...",
 :grant/denied-at   #inst "2024-02-01...",
 :grant/approved-at #inst "2024-02-01..."}
```

This grant is both approved and denied at the same time. Our invariant has been violated! Datomic's in-transaction conflict checker did not prevent this behavior because the `approve` and `deny` functions returned assertion requests for disjoint `[entity, attribute]` pairs.



If we were to draw a data dependency graph between these two functions using the language of Adya's formalism, we'd see something like the following:



The `approve` function wrote a new version of the grant's `approved-at` attribute, but when `deny` read that attribute, it observed the previous (unborn) version from the start-of-transaction database state. This is analogous to a read-write (`rw`) anti-dependency edge in Adya's Direct Serialization Graph. Symmetrically, `deny` wrote a new version of the grant's `denied-at` attribute, but `approve` saw the previous unborn version of `denied-at`. This gives rise to a dependency cycle: each transaction function failed to observe the other's effects.

If these `approve` and `deny` boxes were transactions, we'd call this cycle G2-item: an isolation anomaly proscribed by Repeatable Read and Serializability. Indeed, this phenomenon is analogous to a concurrency anomaly Berenson et al called called Write Skew:

> Suppose `T1` reads x and y, which are consistent with C(), and then a `T2` reads x and y, writes x, and commits. Then `T1` writes y. If there were a constraint between x and y, it might be violated.

There are some similarities between the inter-transaction concurrency control of Berenson et al's Snapshot Isolation and the intra-transaction concurrency control of Datomic's end-of-transaction conflict checker. When the write sets (assertion requests) of two transactions (transaction functions) intersect on some object (an entity and cardinality-one attribute), the first-committer-wins principle (conflict checker) prevents concurrent execution by forcing an abort. When their write sets are disjoint, invariants preserved by two transaction functions individually may be violated by the transaction as a whole.

Like the internal consistency findings above, this behavior may be surprising, but it is broadly consistent with Datomic's documentation. Nubank intends to preserve Datomic's concurrent intra-transaction semantics. We consider this expected behavior for Datomic, rather than a bug.

3.3 Entity Predicates

From Datomic's point of view, the grant workload's invariant violation is a matter of user error. Transaction functions do not execute atomically in sequence. Checking that a precondition holds in a transaction

function is unsafe when some other operation in the transaction could invalidate that precondition!

However, Datomic offers a suite of constraints for enforcing database invariants, including type, uniqueness, and arbitrary predicates on specific attributes. One of the most general constraints is an entity predicate.

Entity predicates are functions which receive a candidate state of the database with all transaction effects applied, the ID of an entity, and return `true` if the transaction should be allowed to commit that state. "Entity" is something of a misnomer: these predicates have access to the *entire* state of the database, and can therefore enforce arbitrary global constraints, not just those scoped to a particular entity.

We can use entity predicates to ensure grants are never approved and denied. To start, we write an entity predicate function `valid-grant?`.

```
(defn valid-grant?
  [db eid]
  (let [{:grant/keys [approved-at denied-at]}
        (d/pull db '[:grant/approved-at
                     :grant/denied-at]
                   eid)]
   (not (and approved-at denied-at))))
```

Then we add an entity spec to the schema which references that function.

```
(def schema
 [...
  {:db/ident          :grant/valid?
   :db.entity/preds ['grant/valid-grant?]
   :db/doc           "Ensures the given grant
                      is not approved *and*
                      denied"}])
```

Unlike other schema constraints, which are enforced for every transaction, entity specs (and their associated entity predicates) are only enforced when transactions explicitly ask for them. Datomic believes that whether or not to enforce an entity spec is a domain decision, and that this approach is more flexible than making entity specs mandatory. Therefore our transition functions assert the grant's `approved-at` or `denied-at` attribute, then request the entity spec be enforced by adding a special request for a *virtual datom*, binding the attribute `:db/ensure` to our entity spec.

```
(defn approve
  [db id]
  [[:db/add id :grant/approved-at (Date.)]
   [:db/add id :db/ensure :grant/valid?]])

(defn deny
  [db id]
  [[:db/add id :grant/denied-at (Date.)]
   [:db/add id :db/ensure :grant/valid?]])
```

Using this entity spec, attempts to approve and deny a grant within the same transaction throw an error, preserving our intended invariant.

```
{:cognitect.anomalies/category
 :cognitect.anomalies/incorrect,
 :cognitect.anomalies/message
 "Entity 17592186045427 failed pred
  #'jepsen.datomic.peer.grant/valid-grant?
  of spec :grant/valid?",
  :db.error/pred-return false,
  :db/error :db.error/entity-pred}
```

4 Discussion

In our testing, Datomic's inter-transaction semantics were consistent with Strong Session Serializability. Intra-transaction semantics appeared strictly concurrent: the operations within a transaction seemed to be executed simultaneously, and the resulting effects merged via set union. This combination satisfies a common high-level definition of Serializability: "equivalence to a serial execution of transactions." However, it does seem to violate the definitions of Serializability in the most broadly-adopted academic formalisms for transactional isolation. Datomic argues—and Jepsen is willing to entertain—that these formalisms should not be applied to Datomic; they are fundamentally different kinds of databases.

While some details of the documentation were inaccurate or misleading, Datomic's inter- and intra-transaction behavior appeared consistent with its core safety claims. Indeed, we believe Datomic's inter-transaction safety properties are stronger than promised.

As always, we caution that Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. We also note that correctness errors in the storage system underlying Datomic could cause violations of Datomic's guarantees; Datomic atop DynamoDB is only as safe as DynamoDB's compare-and-set operation.

4.1 Inter-Transaction Semantics

If one considers a session as being bound to a single peer, Datomic appears to guarantee Strong Session Serializability. Histories of transactions appear indistinguishable from one in which those transactions had executed in some total order, and that order is consistent with the order observed on each peer.

Histories restricted to write transactions (i.e. calls to `d/transact`) appear Strict Serializable. So too do histories where readers use `d/sync` to obtain an up-to-date state of the database rather than `d/db`, which could be stale.

4.2 Intra-Transaction Semantics

Most transactional systems provide serial semantics within a single transaction.[9] Each operation—writes, reads, procedure calls, etc.—within a transaction appears to take place after the previous operation in that same transaction. This property is explicitly encoded in the major formalisms for transactional isolation. Adya, Liskov, and O'Neil begin their database model by defining transactions as ordered, and explicitly specify later operations observe earlier ones:

> Each transaction reads and writes objects and indicates a total order in which these operations occur….
>
> If an event $w_i(x_{i.m})$ is followed by $r_i(x_j)$ without an intervening event $w_i(x_{i.n})$ in E, x_j must be $x_{i.m}$. This condition ensures that if a transaction modifies object x and later reads x, it will observe its last update to x.

Similarly, the abstract execution formalism of Cerone, Bernardi, and Gotsman defines an *internal consistency axiom* preserved by all consistency models from Read Atomic through Serializable:

> The internal consistency axiom INT ensures that, within a transaction, the database provides sequential semantics: a read from an object returns the same value as the last write to or read from this object in the transaction. In particular, INT guarantees that, if a transaction writes to an object and then reads the object, then it will observe its last write.

Crooks, Alvisi, Pu, and Clement's client-centric formalism similarly specifies transactions include a total order, and uses that order to ensure reads observe the most recent write to that object within the current transaction:

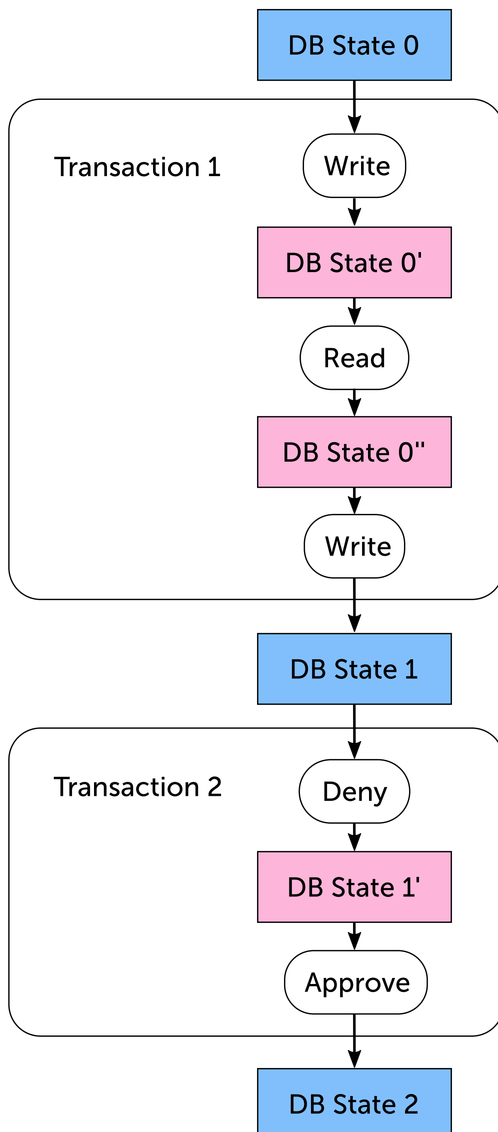
> Further, once an operation in T writes v to k, we require all subsequent operations in T that read k to return v.

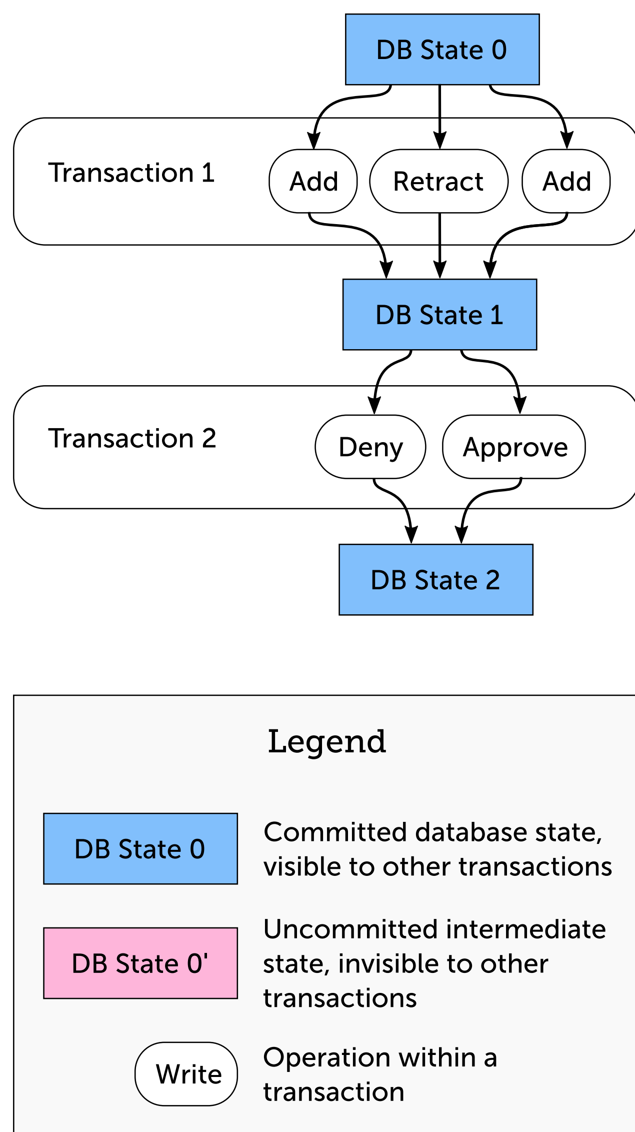
Even as far back as 1979, Kung and Papadimitriou defined transactions as a finite sequence of *transaction steps*. "Thus, our transactions are straight-line programs," they explain.

In all of these models, a Serializable system behaves equivalently to one which begins with an initial database state db_0, picks some transaction T, applies the first operation in T producing an intermediate database state db_0', applies the second operation in T to db_0' producing db_0'', and so on until the transaction has completed, producing a committed database state db_1. Then it moves to a second transaction, and the process continues.

Datomic's semantics are quite different. As previously discussed, the operations within a transaction (assertions, retractions, transaction functions, etc.) are evaluated logically concurrent with one another. Every transaction function in a transaction T observes the state of the database when T began, and produces a new set of operations. They do not observe the other assertions, retractions, or functions in T. These operations are recursively evaluated until only assertions and retractions remain. Those assertions and retractions are merged with set union, checked for conflicts (e.g. contradictory assertions about the value of a single-cardinality attribute on some entity), and then applied to the database state to produce a new, committed version of the database.

[9] Of course, typical Serializable databases may not *actually* execute operations in serial order. However, they (ought to) behave indistinguishably from a system which had. Similarly, Datomic may not execute transaction functions in parallel—but it guarantees concurrent semantics. For concision, we say "serial semantics" instead of "behavior which is indistinguishable from a serial execution," and so on.





This behavior may be surprising to users familiar with other databases, but it is (to some extent) documented. The lookup ref documentation explains that refs use the before-transaction database state. The database functions documentation says the transaction processor calls transactions "in turn", which hints at ordered execution, but explicitly notes that functions are passed "the value of the db (currently, as of the beginning of the transaction)." On the other hand, that same documentation goes on to say that "[t]ransaction functions are serialized by design," which is true *between* transactions, but not *within* them.

Datomic's concurrent semantics yield advantages and drawbacks. For one, a common axiom of database systems is that committed database state is always *consistent*, in the business-rules sense. Read Committed and above proscribe phenomenon G1b (*intermediate read*) in which one transaction sees intermediate state from another transaction. Datomic goes one step further: it is impossible to observe your *own* transaction's intermediate state. One can never[10] produce or observe an inconsistent view of the system—full stop! In some sense, the concept of intermediate state is inherently confusing; Datomic does away with it altogether. This choice also simplifies Datomic's model of time: everything in a transaction happens "at once", and every datom is *always* associated with a single, totally-ordered time.

On the other hand, Datomic's model reintroduces one of the problems Serializability has long been used to prevent. As Papadimitriou's 1979 paper The Serializability of Concurrent Database Updates[11] concisely

[10]Unless one produces new, transient database states using `d/with`.

[11]Intriguingly, Papadimitriou's paper begins with transactions which perform a set of reads, then a set of writes; this formalism might be more readily applicable to Datomic transactions. Later in the paper he addresses "multistep transactions," which are analogous to the serial formalisms discussed in this section.

argues:

> Another way of viewing serializability is as a tool for ensuring system correctness. If each user transaction is correct—i.e., when run by itself, it is guaranteed to map consistent states of the database to consistent states—and transactions are guaranteed to be intermingled in a serializable way, then the overall system is also correct.

It seems plausible that users would want to write transaction functions that transform data while preserving some kind of correctness invariant. Datomic's transaction functions documentation originally suggested as much:

> Transaction functions run on the transactor inside of transactions, and thus can atomically analyze and transform database values. You can use them to ensure atomic read-modify-update processing, and integrity constraints… A transaction function can issue queries on the db value it is passed, and can perform arbitrary logic in the programming language.

If one writes a set of transaction functions which independently preserve some invariant—say, that a grant must never be both approved and also denied, or that the circuits in a home never exceed the capacity of the main panel—one would like to say (analogous to Serializable transactions) that any composition of these functions also preserves that invariant. But as we've demonstrated, within a single Datomic transaction this is not true! A transaction which calls multiple transaction functions might produce an outcome incompatible with the atomic application of those functions. It might violate integrity constraints. Paradoxically, combining two transactions into one can actually make the system *less* safe.

It seems likely that Datomic's behavior violates the major modern transaction formalisms: Cerone et al's internal consistency axiom, Adya's program order, Crooks et al's in-transaction order, etc. It may be possible to contort Datomic's model into alignment with these formalisms: say, by defining Datomic as containing only one object (the entire database), or through a non-local translation of Datomic operations to the formalism's sequence of reads and writes, in which reads are reordered to the beginning of the transaction, and writes to the end. However, these approaches strain intuition. Datomic databases obviously contain independently addressable entities and attributes. Datomic transactions are clearly made up of individual parts, those parts are written in order, and this looks very much like how other databases would express a transaction with serial semantics. Convincing users to ignore that intuition seems a challenging lift.

An easier path might be to abandon these formalisms altogether: they are clearly not designed to apply to Datomic's concurrent intra-transaction semantics. Instead, we could follow the classic informal definition of Serializability. The internal structure of transactions is completely opaque; all that matters is that the history of transactions is equivalent to one which executed in a serial order. Under this interpretation, Datomic does ensure Serializability, Strong Session Serializability, and so on—just with different intra-transaction rules. To avoid confusion, we carefully distinguish between inter- and intra-transaction consistency throughout this report.

4.3 Recommendations

We found no evidence of safety bugs in Datomic, or serious divergence between documentation and system behavior. Datomic's concurrency architecture is refreshingly straightforward, and its transactional correctness easy to argue. Jepsen believes users can rely on Datomic's inter-transaction Serializability.

However, Datomic users should be aware of the concurrent execution semantics within transactions. These are specified in the documentation, but remain an unusual choice which creates the potential for subtle invariant violations. Users should be careful when calling multiple transaction functions in the same transaction. In particular, watch out for intersecting read sets and disjoint write sets. Also be aware of the possibility that multiple updates (e.g. increments) to a single value might quietly collapse to a single update.

In practice, we believe several factors protect Datomic users against encountering anomalies. First, users often try to create a schema and use it in the same transaction, or try to use a lookup ref to refer to an entity created in the same transaction. Both of these scenarios fail, which guides users towards re-reading the documentation and internalizing Datomic's model. Second, the in-transaction conflict checker likely prevents many of the anomalies that could arise from logically-concurrent transaction functions: if two transaction functions produce different values for a single-cardinality attribute of an entity, the transaction aborts.

In addition, users can use attribute predicates to constrain individual values, and entity specs (which must be requested on each transaction) to constrain all attributes of a single entity, or even an entire database. However, users must take care to explicitly request the appropriate entity specs within every transaction that might require them.

Another potential surprise: Datomic goes to great pains to ensure every database state is business-rules consistent: there are no intermediate states, every state is the product of a committed transaction, and so on. However, not all schema constraints apply to extant data. In particular, attribute predicates are only enforced on newly-added datoms, not on existing datoms.

A small operational note: Datomic transactors kill themselves after a few minutes of not being able to talk to storage. We recommended Datomic add a retry loop to make transactors robust to network fluctuations.

4.4 Documentation Changes

Following our collaboration, Datomic has made extensive revisions to their documentation.

First, we worked together to rewrite Datomic's transaction safety documentation. It now reflects the stronger safety properties we believe Datomic actually offers: Serializability globally, monotonicity on each peer, and Strict Serializability when restricted to writes, or reads which use `sync`. Datomic also removed the "single-writer" argument from their safety documentation.

Datomic's docs now include a comprehensive explanation of transaction syntax and semantics. It covers the structure of transaction requests, the rules for expanding map forms and transaction functions, and the process of applying a transaction. Expanded documentation for transaction functions explains Datomic's various mechanisms for ensuring consistency, how to create and invoke functions, and the behavior of built-in functions. The transaction function documentation no longer says they can be used to "atomically analyze and transform database values", nor does it claim transaction functions can "ensure atomic read-modify-write processing".

Datomic used to refer to the data structure passed to `d/transact` as a "transaction", and to its elements as "statements" or "operations". Going forward, Datomic intends to refer to this structure as a "transaction request", and to its elements as "data". The `[:db/add ...]` and `[:db/retract ...]` forms are "assertion requests" and "retraction requests," respectively. This helps distinguish between assertion *datoms*, which are `[entity, attribute, value, transaction, added-or-removed?]` tuples, and the incomplete `[entity, attribute, value]` assertion *request* in a transaction request.

Datomic has also added documentation arguing for a difference between Datomic transactions and SQL-style "updating transactions." There is also a new tech note which discusses the differences between transaction functions and entity predicates when composing transactions.

4.5 Future Work

Our tests did not evaluate excision or historical queries. Nor did we investigate the Datomic client library—though we believe its behavior is likely similar to the peers we designed in this test. We also limited ourselves to a single storage engine: DynamoDB. Datomic runs atop a variety of storage systems; testing others might be of interest. Finally, we have not evaluated Datomic Cloud, which uses a slightly different architecture.

Jepsen is aware of few systems or formalisms which provide inter-transaction Serializability but intra-transaction concurrent semantics. Datomic's behavior suggests fascinating research questions.

First, what *are* Datomic transactions? Is there a sense in which they are a *dual* to typical database transactions? Rather than happening entirely in series, everything happens all at once. What are the advantages and drawbacks of such a "co-transaction" model? Can the drawbacks be mitigated through static analysis, runtime checks, or API extensions? And does this actually matter in practice, or are users unlikely to write transactions which could violate invariants?

Second, are there other databases with concurrent intra-transaction semantics? Conversely, what about other temporal databases with serial intra-transaction semantics? How does Datomic's model fit into this landscape?

Alvaro's Dedalus, a research project exploring temporal Datalog, comes to mind. Like Datomic, its transactions happen "all at once." As in Datomic, this creates the apparent paradox that breaking up operations into multiple transactions can actually make them safer. Consider also Fauna, a temporal database supporting up to Strong Serializability. Like Datomic, Fauna transactions are small programs that the database evaluates, rather than an interactive session driven by a client. Unlike Datomic, Fauna's transactions provide (what appears to be) serial execution with incremental side effects within each transaction. Are Fauna's in-transaction temporal semantics sound? How do their models compare?

The similarity between Datomic's end-of-transaction conflict checker and Snapshot Isolation's first-committer-wins rule suggests new research opportunities. How close is the relationship between Snapshot Isolation and Datomic's in-transaction semantics, and what parts of the existing literature on Snapshot Isolation could we apply to Datomic? Can we show that within a Datomic transaction, cycles between transaction functions must always involve a pair of adjacent read-write anti-dependency edges. Clearly Datomic does not prevent the intra-transaction analogue of lost update, since it collapses multiple increments. What about Fractured Read? Does it allow something like the read-only transaction anomaly described by Fekete, O'Neil, and O'Neil? Or Long Fork? Are there analogues to other G2-item and G2 cycles, perhaps involving predicates?

Finally, one wonders whether there might be a connection to Hellerstein & Alvaro's CALM theorem. Could we show, for instance, that transaction functions which are logically monotonic are safe to combine in a single Datomic transaction? Datalog programs without negation are logically monotonic. Can we show that those programs are also safe under this execution model? Jepsen encourages future research.

Jepsen wishes to thank the entire Datomic team at Nubank, and in particular Dan De Aguiar, Guilherme Baptista, Adrian Cockcroft, Stuart Halloway, Keith Harper, and Chris Redinger. Peter Alvaro offered key insights into concurrent semantics. Irene Kannyo provided invaluable editorial support. This work was funded by Nubank (Nu Pagamentos S.A), and conducted in accordance with the Jepsen ethics policy.