VoltDB 6.3

2016-07-12

In the last Jepsen analysis, we found that RethinkDB could lose data when a network partition occurred during cluster reconfiguration. In this analysis, we'll show that although VoltDB 6.3 claims strict serializability, internal optimizations and bugs lead to stale reads, dirty reads, and even lost updates. Fixes are now available in version 6.4. This work was funded by VoltDB, and conducted in accordance with the Jepsen ethics policy.

1 Background

VoltDB is a distributed SQL database intended for high-throughput transactional workloads on datasets which fit entirely in memory. All data is stored in RAM, but backed by periodic disk snapshots and an on-disk recovery log for crash durability. Data is replicated to at least `k+1` nodes to tolerate `k` failures. Tables may be replicated to every node for fast local reads, or sharded for linear storage scalability.

As an SQL database, VoltDB supports the usual ad-hoc SQL statements, with some caveats (e.g. no auto-increment, no foreign key constraints, etc.) However, its approach to multi-statement transactions is distinct: instead of `BEGIN ... COMMIT`, VoltDB transactions are expressed as *stored procedures*, either in SQL or Java. Stored procedures must be deterministic across nodes (a constraint checked by hashing and comparing their resulting SQL statements), which allows VoltDB to pipeline transaction execution given a consensus on transaction order.

That consensus is obtained through a custom consensus algorithm. Update operations on a single partition are ordered by that partition's Single-Partition Initiator, or SPI: a stable leader which ensures transactions don't interleave. All replicas of a partition follow the SPI's updates. In contrast to updates, pure-read transactions are *not* ordered by the SPI, and read local state directly off any replica. Operations across multiple partitions are ordered by a single Multi-Partition Initiator (MPI) for the entire cluster, which issues operations to relevant SPIs for execution on their partitions.

This design allows VoltDB to provide high throughput for single-partition transactions, while still supporting occasional multi-partition queries—all at strict serializability. Stored procedures which operate on data within a single partition can be efficiently executed by the SPIs, and transaction throughput scales as node counts (or the number of SPIs per node) rise. Multi-partition procedures must pass sequentially through the MPI, and their throughput slowly drops with node count due to coordination costs. In the regime where single-partition work dominates, throughput scales semilinearly with nodes, but VoltDB's internal benchmarks suggest multi-partition transactions are practically limited to a few hundred updates/sec, or tens of thousands of reads/sec (with no updates). This is why VoltDB's performance numbers use entirely single-partition workloads.

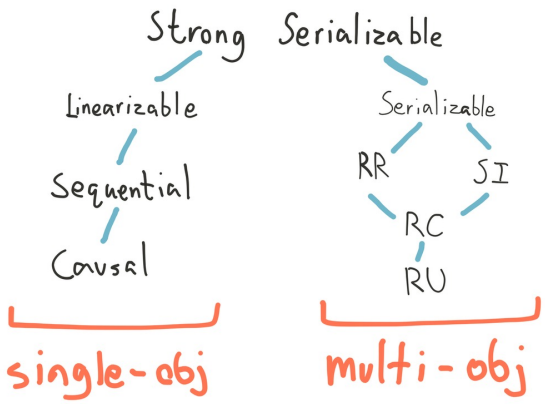
As it turns out, coordination in real-world transactional workloads is less common than one might expect. The industry-standard benchmark TPC-C, for instance, is largely shardable, since the bulk of its transactions occur within the scope of a single district. In fact, TPC-C can even be implemented without any coordination between nodes. Many OLTP systems involve high volumes of single-key operations coupled with periodic analytic rollups. Others, like SaaS offerings, have strong boundaries isolating one customer from another, and only administrative transactions cross customer boundaries. VoltDB targets these applications—and offers them the strongest claims of any database we've tested with Jepsen: strict serializable isolation. In this work, we aim to verify those safety claims.

2 Consistency

Unlike most SQL databases, which default to weaker isolation levels for performance reasons, VoltDB

chooses to provide strict serializable isolation by default: the combination of serializability's multi-object atomicity, and linearizability's real-time constraints.

Serializability is the strongest of the four ANSI SQL isolation levels: transactions must appear to execute in some order, one at a time. It prohibits a number of consistency anomalies, including lost updates, dirty reads, fuzzy reads, and phantoms.

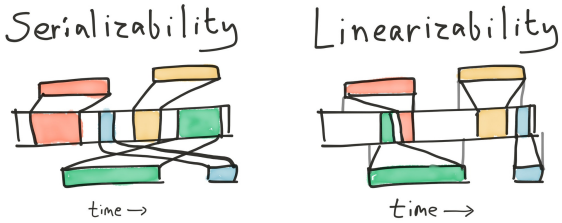


Serializability requires transactions appear to execute in *some* order, but doesn't specify *what* that order should be. This allows for some unintuitive behaviors. For instance, read-only transactions may execute at *any* logical time, regardless of when the query is performed. Under serializability, `SELECT COUNT(*) FROM USERS` may always return 0, regardless of the number of users currently in the table, because when the table was first created, it had no contents. It could also return the count from five minutes ago. We call these reads-in-the-past *stale reads*.

Serializable systems are also free to discard write-only transactions by reordering them arbitrarily far into the future. This also applies to read-modify-update. For instance, we can `UPDATE videos SET view_count = view_count + 1 WHERE id = 123` at any time, because you can increment *any* number, and if there's *no* row 123, the `where` clause wouldn't match. It would be legal to apply this transaction just prior to the heat death of the universe—and if that happened, nobody would see the increment. Therefore, a serializable system is not required to apply this update *at all*. Similar arguments allow serializable systems to discard transactions whose consequences would be overwritten by some already-executed transaction, and so on.

If reading from the past and throwing away blind writes is considered legal, perhaps serializability is not the only constraint we care about. Can we do better?

Ideally, we'd like a transaction to take place sometime *after* we send it to the database, and some time *before* the database confirms it has committed. That way, we could guarantee that once a transaction is complete, any future transaction will see its effects. This real-time constraint is called linearizability, and when applied to multi-object transactions, we obtain a consistency model called strict serializability.



VoltDB's documentation explicitly claims serializability, but implicitly claims strict serializability as well. For instance, their transaction whitepaper asserts:

> Because VoltDB always performs synchronous replication of read-write transactions within a partition, end-users are guaranteed to read the results of prior writes even when reads bypass the SPI sequencer

The guarantee that prior writes are visible to clients suggests that VoltDB's transactions obey linearizability's real-time constraint. VoltDB's engineers confirm this interpretation: it should provide strict serializability. Because strict serializable systems are also linearizable, we can use Jepsen's existing linearizability checker to verify VoltDB's correctness—both on single objects in the database, and on systems of multiple rows.

3 Stale reads

VoltDB can shard tables into logical *partitions* (not to be confused with network partitions), and each of those partitions is replicated to `k+1` *sites* for redundancy. Transactions which only interact with data in a single partition can be executed by that partition's SPI, without coordinating with other partitions. Our goal is to test whether each SPI ensures linearizability within a single partition, by performing reads, writes, and compare-and-sets on a single database row.

To begin, we'll create a simple table of registers, each identified by a primary key `id`, and partition the table by those `ids`. Each node in the cluster will own some fraction of the keyspace.

```
(voltdb/sql-cmd!
  "CREATE TABLE registers (
    id          INTEGER UNIQUE NOT NULL,
    value       INTEGER NOT NULL,
    PRIMARY KEY (id)
  );
  PARTITION TABLE registers ON COLUMN id;")
```

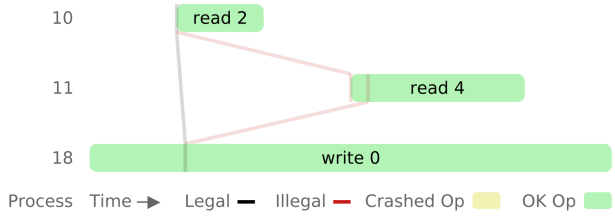
Then, on a given register, we'll perform three types of operations: a read, a write, and a compare-and-set. Reads and writes are easy: VoltDB predefines stored procedures called `REGISTERS.select` and `REGISTERS.upsert` which take the primary key. For compare-and-set, we'll define an SQL stored procedure. Then we'll call those procedures to perform operations on the database.

We define generators for each operation type, and have 5 clients (one for each node) perform a mix of writes and CaS ops, while another 5 clients perform reads—roughly once a second for each client. We dedicate specific clients to reads for two reasons: first, VoltDB has an optimized path for read-only transactions, and second, if a client blocks writing a value to a specific node, say, during a failure, we'd like another client to have a chance to *see* if the transaction succeeded or failed before the failure resolves. Sometimes consistency errors manifest during that window.

We'll use Jepsen's independent/concurrent-generator to run *several* of these single-register tests concurrently—improving our chances of catching an error in any given time period. Each single-register test lasts for 30 seconds.

After 25 seconds, Jepsen partitions the network into two randomly selected components, and waits another 25 seconds before healing the fault.

This test detects a linearizability violation almost immediately.



This diagram shows three processes (10, 11, and 18) concurrently executing a read of 2, a read of 4, and a write of 0. Time flows left to right. We know the value must be 2 during process 10's read, and that read could be preceded or followed by a write of 0—but neither 0 nor 2 allows process 11 to read the value 4. This read of 4 is *inconsistent* with a linearizable register.

Looking at the full history suggests an explanation. Process 13 begins and completes a read of 4, the current value. At some point shortly thereafter, a network partition takes effect, isolating two nodes from the other three.

```
13  :invoke :read nil
13  :ok :read 4
18  :invoke :write  0    ; succeeds
15  :invoke :cas  [0 1]  ; fails
17  :invoke :cas  [1 2]  ; succeeds
19  :invoke :write  1    ; succeeds
16  :invoke :cas  [4 3]  ; fails
10  :invoke :read nil
10  :ok :read 2
11  :invoke :read nil
11  :ok :read 4
```

Writes block (pending timeout) as nodes wait for acknowledgement from their disconnected peers, but reads do *not* require coordination in VoltDB and complete successfully. We see a consistent pattern until the partition resolves: process 11 reads 4, but the other nodes see 2. One possible interpretation is that one component kept 4, while the other wrote 0, wrote 1, compare-and-set 1 to 2, then read 2.

```
12  :invoke :read nil
12  :ok     :read 2
14  :invoke :read nil
14  :ok     :read 2
13  :invoke :read nil
13  :ok     :read 2
10  :invoke :read nil
10  :ok     :read 2
11  :invoke :read nil
11  :ok     :read 4
```

This is a type of split-brain: different nodes have internally consistent but differing states. The fact that writes block, however, prevents the system from *diverging*. One node is trapped in the past, but it has not accepted conflicting writes. This history is not *strict* serializable, but it is *serializable*—because we could *reorder* the transactions such that the alternating reads made sense.

This anomaly arises from an optimization we mentioned earlier: read-only transactions don't pass through VoltDB's transaction-ordering system:

> As an optimization, read-only transactions skip the SPI sequencing process and are routed directly to a single copy of a partition. There is no useful reason to replicate

> reads. Effectively, this optimization load-balances reads across replicas. Because VoltDB always performs synchronous replication of read-write transactions within a partition, end-users are guaranteed to read the results of prior writes even when reads bypass the SPI sequencer.

The argument here is mostly sound: because reads don't change the state of a replica, they can be freely executed at any replica without coordination. The result is always serializable—but not *strict* serializable, because we might read from a stale replica. If a node is isolated by a network partition, it might deliver old results for read requests until it detects a fault and steps down. We saw this problem in etcd, consul, and mongodb: all assumed local state was sufficient to ensure linearizable reads.

4 Dirty reads

When an SPI receives a write, it first orders the write in its internal transaction queue. It then broadcasts that write to all other replicas for that partition, journals that write to disk (when synchronous command logging is enabled), and applies the write locally. It then blocks, awaiting a response from all replicas. Once all replicas have acknowledged the write, VoltDB returns the transaction's results to the client.

Read-only transactions don't go through this synchronous replication process, but rather, execute on any replica's local state and return immediately. This allows for stale reads when a successful transaction is incompletely replicated, but *also* suggests the possibility of dirty reads when an aborted transaction's results are made visible.

For instance, we might insert a unique number `n` into a table, which is received and applied locally by some SPI. Before the SPI receives acknowledgement from its peers, a concurrent read on that SPI's local replica could see `n`. If a new SPI is subsequently elected *without* having received `n`, then the insert would appear never to have happened. This implies (at best) a dirty read.

```
{:dirty-reads
 {:valid? false,
  :read-count 28800,
  :strong-read-count 28733,
  :unseen-count 26,
  :dirty-count 93,
  :dirty
```

Since VoltDB allows stale reads, we might not be able to tell which transactions committed or not. We need a way to perform a final *strong read*—one which is guaranteed to see all prior transactions. To do this, we can write a VoltDB stored procedure which includes an *unused* insert statement. VoltDB statically analyzes stored procedures to identify whether they are read-only, and the possibility of a write forces this transaction to go through the SPI—hopefully preventing stale reads.

Our dirty-read client will create a table (`dirty_reads`) with a single integer column (`id`). Then we'll handle `read` operations by trying to read a specific ID, `write` ops by inserting the given value, and `strong-read` ops by calling our strong-read stored procedure to select *all* IDs in the table.

Based on our hunches about the way a dirty read might happen, we'll keep track of the most recently attempted insert on each node in the cluster, and have reads against that node try to read that value. We'll reserve a single process for writing to each node, and the remaining processes will perform reads. Then we'll have each process perform roughly a hundred ops per second, while Jepsen's *nemesis* wreaks havoc with the cluster. At the end of the test, we'll heal the cluster, and have each client perform a final strong read.

We suspect that dirty reads require a node performing a write to become isolated from some of its peers, but still service concurrent reads from clients. Isolated nodes will kill themselves after discovering they can no longer see a majority, but just for good measure, we'll kill the node ourselves, after it's been isolated for a few seconds, then restart it and rejoin it to the cluster. To make sure the cluster continues running, we'll make sure to keep a majority intact at all times.

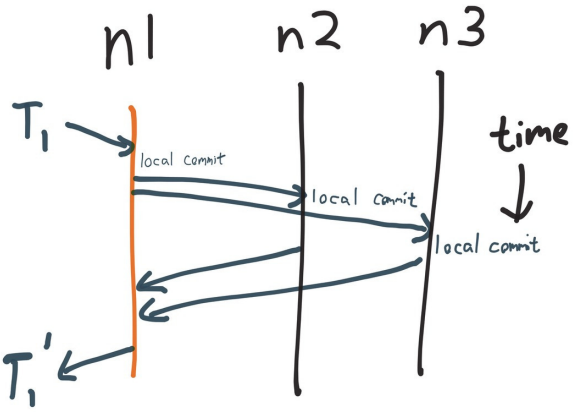
To verify correctness, we'll examine the history of successful operations, looking at the set of successful writes, reads, and strong reads. When we don't see an inserted value with a normal read, we'll call that `unseen`—a measure of our tests's resolving power. Conversely, if we see a value in a normal read, but it's not present in a final strong read, we know it saw uncommitted state, and call it *dirty*.

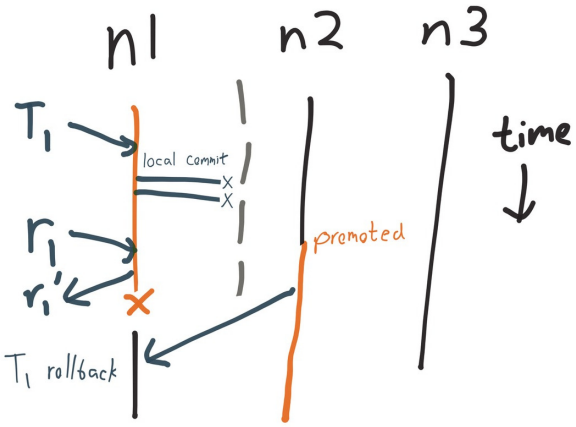
```
#{21713 21714 21715 21716 21717 21718 21719 21720 21721 21722 21723
  21724 21725 21726 21727 21728 21729 21730 21731 21732 21733 21734
  21735 21736 21737 21738 21739 21740 21741 21742 21743 21744 21745
  21746 21747 21748 21749 21750 21751 21752 21753 21754 21755 21756
  21757 21758 21759 21760 21761 21762 21763 21764 21765 21766 21767
  21768 21769 21770 21771 21772 21773 21774 21775 21776 21777 21778
  21779 21780 21781 21782 21783 21784 21785 21786 21787 21788 21789
  21790 21791 21792 21793 21794 21795 21796 21797 21798 21799 21800
  21801 21802 21803 21804 21805},
```

As suspected, this test suggests the existence of dirty reads. A node which crashes while waiting for its writes to replicate could expose uncommitted transaction state to concurrent reads.

Normally, transactions are committed locally on the SPI, and may only return when all replicas have acknowledged the transaction. However, if a partition interrupts replication, that transaction is visible for reads on some replicas *before* it has fully committed. If a new SPI is elected without that transaction, then those reads saw data from an *uncommitted* transaction: a dirty read has occurred.

Both stale reads and dirty reads are addressed by ENG-10389, which forces reads to wait for writes to complete before they can return. This is the new default behavior for VoltDB in 6.4, and is configurable with a global option. VoltDB may introduce per-transaction options for users who wish to perform selective unsafe reads in exchange for lower latencies.





4.1 Lost updates

We've assumed, in our dirty-read test, that inserted values *not* present in the final read set failed—but this assumption isn't necessarily justified. It could be that the insert *did* commit, but its data was later *lost*: a lost update. To verify this assumption, we'll check for successfully inserted values that aren't present in a final read.

```
{:dirty-reads
 {:valid? false,
  :read-count 27612,
  :strong-read-count 26799,
  :unseen-count 53,
  :dirty-count 866,
  :dirty #{12227 12228 12235 ...
           13631 13635 13636},
  :lost-count 866,
  :lost #{12227 12228 12235 ...
          13631 13635 13636}}
```

Not only is uncommitted transaction state visible to concurrent reads, but confirmed transactions can be *lost entirely* when nodes are isolated from a majority of the cluster. In this particular test, every dirty read was in fact a lost update—in fact, it's difficult to *prove* that dirty reads exist at all, if committed transactions can be arbitrarily discarded.

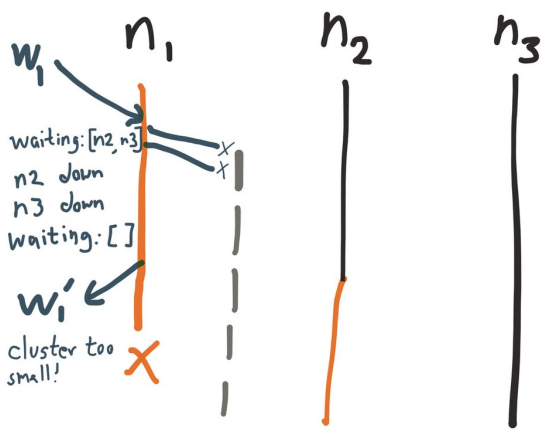
Lost-update anomalies also appear (much less frequently) in single-register linearizability tests: we can

use strong reads or no reads at all to rule out read-only transaction anomalies.

Why is this possible? How can a transaction be acknowledged to the client if it's not present on every node? We said earlier that transactions can only return if they're acknowledged by *every* replica.

Wait a minute—if the coordinator needs *every* replica, how can VoltDB tolerate the loss of a node?

Like Kafka 0.8's replication algorithm, VoltDB can give up on nodes which are unresponsive. These nodes are ejected from the cluster. With unanimous consensus from the remaining reachable nodes, VoltDB is free to declare a *new* cluster (a subset of the old one), and continue running. Since the cluster no longer includes the unreachable nodes, the SPI is free to return writes which weren't replicated to them.

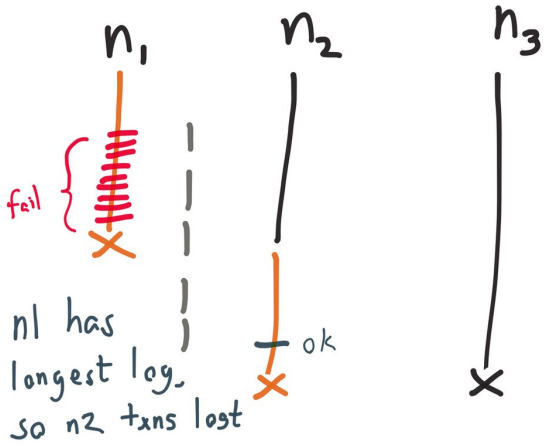


This is not the only source of lost updates: VoltDB's crash recovery system can *also* cause write loss.

During recovery, the recovery planner chooses the longest disk log from all nodes as the authoritative copy. Since operations are journaled to the log immediately, before the node receives acknowledgement, nodes on the minority side of a partition may have longer logs than those on the majority. This means the recovery planner may discard acknowledged writes on the majority, if some minority node accepted more requests for that partition before crash. This is ENG-10486, and has been fixed in 6.4 by reconstructing the final cluster topology from the logs.

This works well for node *crashes*, but in the event of a network partition, nodes on both sides would declare the others dead, splitting into two independent clusters. To prevent this split-brain scenario, VoltDB has a *partition detector* which watches their internal cluster consensus system (termed ZooKeeper, but actually a homegrown consensus algorithm which provides the ZK API) for updates to the cluster state. When a cluster shrinks, and the new cluster is not a majority of the previous cluster, the partition detector shuts down the node to prevent divergence.

Since ZK watches are asynchronous, it's possible for waiting transactions to be released to the client *before* the partition detection code can run and shut down the node. Therefore, writes on the minority side of a partition—which *should* fail—can be successfully acknowledged to the client. This is ENG-10453, and is addressed in VoltDB 6.4 by performing partition detection before releasing pending client responses.



5 Multi-key transactions

Our single-register linearizability test only evaluated transactions on a single partition. We'd also like to evaluate the MPI (Multi-Partition Initiator), to see if transactions that operate on multiple keys satisfy strict serializability.

For simplicity, we'll restrict ourselves to read and write operations on a set of registers, identified by key. We'll represent operations on those registers with a tuple of function, key, and value, e.g. `[:read :x 2]`, or `[:write :y 3]`. A transaction is just a sequence of those operations, which should be applied in order and atomically. We'll generate transactions which operate on any subset of the keyspace, to allow for concurrency.

To keep the state space small, we'll perform a read before every write. This helps our analyzer prune invalid linearizations, because while blind writes can *always* succeed, a specific read restricts the value of its register.

In Knossos (Jepsen's linearizability checker), we'll define a new datatype for these transactional k/v systems. This model defines the semantics of a *singlethreaded* multi-register system: transactions are applied by taking each operation in turn, updating values for writes, and returning inconsistent states when an operation tries to read the wrong value for a given key.

```
(defrecord MultiRegister []
  Model
  (step [this op]
    (assert (= (:f op) :txn))
    (reduce (fn [state [f k v]]
              ; Apply this particular op
              (case f
                :read  (if (or (nil? v)
                               (= v (get state k)))
                         state
                         (reduced
                           (inconsistent
                             (str (pr-str (get state k)) " " (pr-str v)))))
                :write (assoc state k v)))
            this
            (:value op))))
```

Note that `nil` reads are always legal in this model. When a read is attempted and crashes, we don't know what value it would have read, and use `nil` to express that it could have been anything.

We'll create a table, much like the single-register test, composed of *systems*, each of which has many *keys* mapping to *values*. We'll test multiple systems simultaneously to improve our chances of finding a consistency violation.

```
CREATE TABLE multi (
  system      INTEGER NOT NULL,
  key         VARCHAR NOT NULL,
  value       INTEGER NOT NULL,
  PRIMARY KEY (system, key)
);
PARTITION TABLE multi ON COLUMN key;
```

Next, we need a stored procedure to execute our generated transactions. We define general read and write SQL statements, and take arrays for the functions, keys, and values for each operation. Then we zip through those arrays, building up a queue of SQL statements to apply. Calling `voltExecuteSQL()` applies each statement, and returns an array of results to the client.

```
public class MultiTxn extends VoltProcedure {
  public final SQLStmt write =
    new SQLStmt("UPDATE multi SET value = ? WHERE system = ? AND key = ?");
  public final SQLStmt read =
    new SQLStmt("SELECT * FROM multi WHERE system = ? AND key = ?");

  // Arrays of the function, key, and value for each op in the transaction.
  // We assume string keys and integer values.
  public VoltTable[] run(int system, String[] fs, String[] ks, int[] vs) {
    assert fs.length == ks.length && ks.length == vs.length;
```

```
    for (int i = 0; i < fs.length; i++) {
      if (fs[i].equals("read")) {
        voltQueueSQL(read, system, ks[i]);
      } else if (fs[i].equals("write")) {
        voltQueueSQL(write, vs[i], system, ks[i]);
      } else {
        throw new IllegalArgumentException(
                    "Don't know how to interpret op " + fs[i]);
      }
    }
    return voltExecuteSQL();
  }
}
```

Our client applies transactions to the system by calling that stored procedure, and copying any values the transaction read back into the completion operation, so we can verify their correctness.

In several days of test runs, through partitions, node crashes, rejoins, and disk recoveries, this multi-transaction test has yet to find a nonlinearizable case. This is somewhat surprising, because we *know* VoltDB loses updates. It could be that the multi-partition code-path introduces additional serialization points which prohibit the anomalies we saw in single-partition tests—or perhaps there are simply performance differences that mask anomalies. The state space for multi-register tests is significantly larger than for single registers, which limits our resolving power.

One possibility is that the global order imposed by the MPI means that minority replicas can't receive enough pending requests to cause divergence on recovery. ENG-10486 may not be possible when *all* transactions pass through the MPI, but concurrent single-partition transactions could still cause divergence and data loss in multi-partition transactions, by forcing minority write logs to be longer. I've experimented with concurrent single-partition workloads, but haven't found a nonlinearizable case yet.

6 Discussion

To summarize, **VoltDB 6.3 allows stale reads, dirty reads, and lost updates due to network partitions and fault recovery**. It cannot satisfy its claims of strict serializability; nor can it satisfy any of the weaker SQL isolation levels: repeatable read, snapshot isolation, read committed, and even read uncommitted are out of the question. However, the VoltDB team is determined to fix consistency bugs and choose safe defaults, even when doing so would reduce performance. Users of 6.4 should have a much safer experience.

Until upgrading, users can mitigate VoltDB's stale and dirty reads by using a stored procedure including an unused update statement for read-only transactions. The VoltDB analyzer will run these queries through the normal update path, instead of the optimized read path. This does not ensure correctness: VoltDB will still allows nonlinearizable histories due to lost updates—but it does significantly reduce the probability of stale and dirty reads.

Given `n` nodes and `k+1` replicas, VoltDB believes that lost updates should be impossible in clusters where `n < 2k`: an isolated component of a VoltDB cluster will kill itself immediately if it does not have at least a single copy of *every* data partition. As node counts rise, network partitions are more likely to kill the entire cluster. Rack-aware replica placement can mitigate the risk of total shutdown by ensuring rack-isolating partitions will preserve a copy of every replica on some subset of racks, but this reintroduces the possibility of lost updates where `k >= rack-count - 1`.

In testing, Jepsen also uncovered a few minor issues which don't appear to affect safety: rejoining more than one node to the cluster at once can cause some to crash with mysterious errors, and you can rejoin to nodes which are about to kill themselves, causing both nodes to crash. The Java client's auto-reconnect thread never stops trying, even after client close. Finally, identical schema changes, like creating the same table twice, are subject to a mostly harmless race condition.

Most consensus systems we've tested with Jepsen have a well-defined cluster membership and use majority quorums: network partitions can cause some nodes to go unavailable, but service continues so long as a ma-

jority of the cluster remains alive and connected. Minority nodes typically pause and reconnect when the network heals. VoltDB behaves differently: network partitions cause minority nodes to shut down permanently; operator intervention is required to restore full service.

VoltDB also does not require a majority of the *original* cluster—rather, it needs a majority of the *current* cluster to continue. This means a cluster can shrink from five nodes to three, then two, then possibly a single (blessed) node—so long as the remaining cluster has at least one copy of every logical partition. This allows VoltDB to tolerate more failures than most strongly consistent systems, but also reduces durability guarantees: acknowledged transactions may not be present on as many nodes as you'd think.

Version 6.4 includes fixes for all the issues discussed here: stale reads, dirty reads, lost updates (due to both partition detection races and invalid recovery plans), and read-only transaction reordering are all fixed, plus several incidental bugs the VoltDB team identified. After 6.4, VoltDB plans to introduce per-session and per-request isolation levels for users who prefer weaker isolation guarantees in exchange for improved latency. VoltDB's pre-6.4 development builds have now passed all the original Jepsen tests, as well as more aggressive elaborations on their themes. Version 6.4 appears to provide strict serializability: the strongest safety invariant of any system we've tested thus far. This is not a guarantee of correctness: Jepsen can only demonstrate faults, not their absence. However, I am confident that the scenarios we identified in these tests have been resolved. VoltDB has also expanded their internal test suite to replicate Jepsen's findings, which should help prevent regressions.

These tests explored simple majority/minority network partitions, process crashes, rejoin, and recovery, but there are several avenues for future research. VoltDB requires that transactions be deterministic, and shuts down upon detecting nondeterministic execution to avoid data corruption. How well does this mechanism preserve safety? Databases vary in their ability to detect and compensate for single-bit and truncation errors at the network and disk level; we could investigate VoltDB's error correction behavior. Partial network partitions are well-handled by Paxos, ZAB, etc., but can confuse algorithms which use fault detectors to enforce correctness: how well does VoltDB tolerate partial failure? Finally, VoltDB has several implementations of k-ordered flake IDs for assorted internal purposes: we might explore the impact of clock skew.

VoltDB has also published an in-depth discussion of these issues, and a Consistency FAQ that may be of interest.

This work was funded by VoltDB, and conducted in accordance with the Jepsen ethics policy. My sincerest thanks to the VoltDB team, especially John Hugg, Ruth Morgenstein, and Ning Shi, for their help in testing, and hard work in fixing bugs. I am also indebted to Camille Fournier, Marc Hedlund, Peter Alvaro, Peter Bailis and Caitie McCaffrey for their valuable peer review.