JEPSEN

TiDB 2.1.7

Kyle Kingsbury
2019-06-12

TiDB is a distributed, auto-sharded SQL database based on Google's Percolator model. Despite promising snapshot isolation, TiDB 2.1.7 through 3.0.0-beta.1-40 allowed read skew and lost updates by default, thanks to two auto-retry mechanisms which blindly re-applied updates when a transaction conflicted. TiDB also supports a `select ... for update` *statement which mostly, but not entirely, prevents write skew. We found a minor race condition in table creation, reduced durability & fault tolerance in fresh clusters, and several crashes on startup. With both auto-retry mechanisms disabled, TiDB 2.1.8 through 3.0.0-beta.1-40 passed our tests for snapshot isolation and single-key linearizability. 3.0.0-rc.2, which disables auto-retry by default, also passes. Finally, TiDB has a theoretical dependence on* `CLOCK_MONOTONIC_RAW` *for safety, but we have not yet observed anomalies due to clock skew, possibly due to tooling limitations. PingCAP has published a companion blog post to this report. This work was funded by PingCAP, the makers of TiDB, and conducted in accordance with the Jepsen ethics policy.*

1 Background

In 2010, Google engineers Daniel Peng & Frank Dabek published Large-scale Incremental Processing Using Distributed Transactions and Notifications, which described Percolator: a Google-internal database for random access transactional workloads where low latencies are not required. Percolator uses a global coordination service called a *timestamp oracle* (TSO, for short) to allocate a monotonically increasing sequence of transaction timestamps. Data is stored in Bigtable, a horizontally scalable key-value store, which offers atomic read-modify-write on individual records. Transactions are executed by stateless clients, which use the timestamp oracle and Bigtable to provide a snapshot isolated multi-dimensional key-value store.

Rows in Percolator are stored as a sequence of distinct Bigtable records: one for each version. A single metadata record tracks the state of the row, including whether that row is locked by a transaction, or a pointer to that row's current version. A transaction begins by acquiring a *start timestamp* from the TSO. It then acquires a lock (via Bigtable compare-and-set) on each row to be written, checking to make sure that no other transaction has locked that record. Once a lock is acquired, the transaction writes a new version of that row at the transaction's start timestamp. If every cell was locked and written successfully, the transaction may commit; it obtains a *commit timestamp* from the TSO and replaces each lock record with a write record pointing to that transaction's written value.

Crashed transactions are cleaned up lazily, when new transactions encounter conflicting locks. Every transaction designates one of its locks as the *primary lock*; this record is used to determine whether the entire transaction commits. If the primary lock is committed, the crashed transaction is rolled forward—otherwise, it is rolled back. Compare-and-set on the primary lock prevents transactions from racing to commit or abort.

PingCAP's database, TiDB, adapts Percolator's model for use as a general-purpose SQL database. Like Percolator, it's comprised of three components:

- *Placement Driver* (PD): allocates timestamps and coordinates shards
- *TiKV*: a sharded, horizontally-scalable key-value store
- *TiDB*: a transactional SQL layer which stores data in TiKV

Placement Driver is replicated via the Raft consensus algorithm for fault tolerance. Likewise, each TiKV shard is replicated with Raft, and stores data on local disks using RocksDB, a log-structured merge-tree. Shards automatically split and merge as data volumes change, or as hotspots arise.

TiDB nodes, by contrast, are stateless clients of TiKV:

any number can run independently. They expose a mostly MySQL-compatible interface for clients.

There are some wall-clock dependencies in TiDB. Placement Driver, for instance, allocates timestamps based on local wall-clocks, but ensures monotonicity through Raft plus a leader lease. Clock skew between PD nodes can cause allocated timestamps to diverge from wall time, but shouldn't result in prolonged unavailability: PD will allocate timestamps in the future if its own clock falls behind what's committed to Raft.

To improve throughput, PD nodes reserve a batch of timestamps in a single Raft operation. No other node can reserve the same batch, which means that the reserving node is free to allocate those timestamps independently. However, this creates the possibility that two leaders (say, due to a network partition) might allocate non-monotonic timestamps concurrently. PD uses leader leases based on `CLOCK_MONOTONIC_RAW` to mitigate this.

1.1 Consistency

TiDB's transaction isolation documentation explains that TiDB provides snapshot isolation, but, for compatibility with MySQL, calls it repeatable read. The documentation is therefore somewhat confusing: some of its descriptions of repeatable read actually refer to repeatable read, and other parts refer to snapshot isolation. To make matters worse, MySQL's "repeatable read" isn't actually repeatable read either—it's monotonic atomic view, a weaker isolation level.

Moreover, the documentation seems to imply that TiDB "repeatable read" might be different from both MySQL's "repeatable read" *and* snapshot isolation:

> The consistency of MySQL Repeatable Read isolation level is weaker than both the snapshot isolation level and TiDB Repeatable Read isolation level.

Monotonic atomic view is strictly weaker than snapshot isolation: it allows anomalies (predicate-many-preceders, lost updates, and read skew), which snapshot isolation prohibits. But how does TiDB's "repeatable read" differ from snapshot isolation? The documentation goes on to say it doesn't:

> According to the standard described in the A Critique of ANSI SQL Isolation Levels paper, TiDB implements the snapshot isolation level, and it does not allow phantom reads but allows write skews.

However, that paper, by Berenson et al states that snapshot isolation only disallows the *anomaly definition* of the ANSI SQL standard's phantoms: phenomenon A3. The *preventative definition* of phantoms, P3, is "sometimes possible" under snapshot isolation. In discussion with Jepsen, PingCAP's engineers clarified that TiDB does, in fact, allow some phantoms.

There are a few other interesting divergences from standard SQL—most notably, although TiDB supports foreign key constraints, the database does not actually enforce them.

2 Test Design

Jepsen is a distributed systems testing toolkit, and has found safety and liveness issues in dozens of databases. With Jepsen, one runs a real distributed system, rather than relying on static analysis or model-checking. Jepsen connects to that system via a client library, performs some work (optionally while injecting faults into the system), and finally verifies that the history of all client operations satisfies some invariants.

We tested TiDB 2.1.7, 2.1.8, 3.0.0-beta.1, 3.0.0-beta.1-40, and 3.0.0-rc.2. Our tests ran on a five-node Debian Stretch cluster, with PD, TiKV, and TiDB on each node. We used a replication factor of 3 for each TiKV region. To stress cross-region transactions, we used TiDB's `split-table` option to ensure each table had a distinct region, and executed queries across multiple tables.

For faults, we paused (using `SIGSTOP`), forcibly killed (using `SIGKILL`), and isolated (using `iptables`) individual nodes, as well as introducing network partitions dividing the cluster in two, or into overlapping rings of 3/5 nodes each, such that every node observed a majority, but no two nodes agreed on what that majority was. We used special TiDB schedules to shuffle leaders, shuffle regions, and merge regions together. We tested using exponentially distributed clock skews up to 232 seconds, as well as strobing the clock up and down every few milliseconds. We also used `libfaketime` to simulate some node clocks, both `CLOCK_REALTIME` and `CLOCK_MONOTONIC`, running up to 5x faster than others.

PingCAP had written their own Jepsen tests, which we reviewed and expanded upon during our collaboration. The resulting test suite encompasses several workloads from previous Jepsen analyses, as well as some new ones.

2.1 Set

In the set test, we insert unique integers as distinct rows into a table, and concurrently read the entire table to see what integers are present. We verify that every successfully inserted element (either those whose inserts were acknowledged, or which appeared in reads) are present in subsequent reads. We consider an inserted element *lost* if there exists some time after which every read fails to observe that element. A stricter variant of this test ensures that every element is immediately visible, as opposed to allowing stale reads.

2.2 Bank

In the bank test, we create a pool of simulated bank accounts, and transfer money between them using transactions which read two randomly selected accounts, subtract and increment their balances accordingly, and write the new account values back. Under snapshot isolation, the total of all accounts should be constant over time. We read the state of all accounts concurrently, and check for changes in the total, which suggests read skew or other snapshot isolation anomalies.

2.3 Long Fork

Long fork is an anomaly prohibited by snapshot isolation, but allowed by the slightly weaker model *parallel snapshot isolation*. In a long fork, updates to independent keys become visible to reads in a way that isn't consistent with a total order of those updates. For instance:

T_1: `w(x, 1)`
T_2: `w(y, 1)`
T_3: `r(x, 1), r(y, nil)`
T_4: `r(x, nil), r(y, 1)`

Under snapshot isolation, T_1 and T_2 may execute concurrently, because their write sets don't intersect. However, every transaction should observe a snapshot consistent with applying those writes in *some* order. Here, T_3 implies T_1 happened before T_2, but T_4 implies the opposite. We run an n-key generalization of these transactions continuously in our long fork test, and look for cases where some keys are updated out of order.

2.4 Register

TiDB promises snapshot isolation, but stores individual keys in Raft, which suggests that operations on a single key may be linearizable. We perform reads, writes, and compare-and-set operations on a set of registers, and verify each register independently using the Knossos linearizability checker.

2.5 Sequential

Sequential consistency requires that the orders of operations observed by each individual client be consistent with one another. We evaluate sequential consistency by having one process perform two transactions, each inserting a different key, and, concurrently, reading those keys in the reverse order using a second process:

T_1: `w(x, 1)`
T_2: `w(y, 1)`
T_3: `r(y)`
T_4: `r(x)`

A serializable system could allow x and y to be inserted in either order, and likewise, could evaluate the reads at any point in time: reads could see neither, only x, only y, or both. A sequentially consistent system, however, can never observe y alone, since the same process inserted x prior to y.

2.6 Monotonic & Append

Some of Jepsen's checkers (e.g. linearizability) allow the verification of arbitrary histories. However, our usual tests for transactional systems rely on a carefully selected "artisanal" set of transactions with hand-proven invariants, as in the bank, long-fork, and sequential tests.

For TiDB, we developed a new, more general approach for verifying transactional isolation, based on cycle detection.[1] The monotonic test finds dependency cycles over increment-only registers, the txn-cycle test finds write-read dependency cycles over read-write registers, and the append test uses appends of unique elements to lists. We include realtime dependencies (e.g. T_1 completes before T_2 begins) in our analysis. TiDB doesn't formally promise realtime guarantees, but our results hold both with and without those constraints.

[1] The details of our cycle detection strategy are somewhat involved, and will be published in an upcoming paper.

3 Results

3.1 Crashes on Startup

When their connection to Placement Driver is lost, TiKV nodes try indefinitely to reconnect, and come back online once PD is available again. However, a recently started TiKV node is special: it will only try to connect a *finite* number of times, and if those attempts fail, the process crashes.[2] In 2.1.7 through 3.0.0-beta.1, restarting every process could (depending on the state of KV & PD processes, and the network) result in a completely unusable cluster. We found problems like this in various components—for instance, another crash in TiKV, one in TiDB when TiKV's regions aren't yet available, and another in TiDB when binlog pump instances aren't registered.

This is not a severe problem—many databases share this behavior. However, it makes the startup process more fragile, and could potentially cause issues in production. For instance, if an init system automatically restarts TiKV, it may fail several times in a row, causing the init system to declare the service broken, and give up altogether. Operators attempting to recover from an outage could also restart processes in the wrong order, only to discover the daemon they had just restarted was not, in fact, running. In general, TiDB restarts must be carefully sequenced.

TiDB has patches for #4500 and #10495 in 3.0.0-rc.2. Comments in #10240 suggest it may be fixed later. PingCAP says the crash observed in #10470 is working as designed, and will not be fixed. We encouraged PingCAP to develop a standardized policy for crashes vs retries on transient failures.

3.2 Created Tables May Not Exist

As with many of the databases Jepsen has evaluated, changes to schemata in TiDB might not take effect immediately. For instance, TiDB 3.0.0-beta.1 could successfully execute a `create table cats ...` query, then (occasionally) throw `table cats doesn't exist` if one tries to insert a record into `cats`.

This is bug 10410, which PingCAP thinks is linked to a race condition shortly after cluster bootstrap. PingCAP reports this issue is fixed in 3.0.0-rc.2.

3.3 Under-Replicated Regions

TiDB 3.0.0-rc.2, by design, starts up with a region with only a single replica, regardless of the configured target number of replicas. PD then gradually adds additional replicas until the target replica count is reached. In addition, any regions which are split from this initial region *also* start with the same number of replicas as the target region, until PD can expand them to new nodes.

This is not a problem in itself, but it does lead to an awkward possibility: in the early stages of a TiDB cluster, data may be acknowledged, but stored only on a single node, when the user expected that data to be replicated to multiple nodes. A single-node failure during that period could destroy acknowledged writes, or render the cluster partly, or totally, unusable. In our tests, default-sized clusters reached a fully-replicated state in 1-2 minutes; regions capped at 500 elements might fail to converge after 80+ minutes. When a network partition occurs under those conditions, it could result in a partial or total outage, because some regions, lacking a full set of 3 replicas, cannot tolerate the loss of any single node.

Conferring with the PingCAP team, we believe this issue only affects new clusters; once TiDB has stabilized, split regions should inherit their parents' replica counts. During topology changes (e.g. in response to a dead node), TiDB adds new nodes before removing old ones, which should prevent scenarios where the number of replicas falls below the configured target. In addition, waiting for full replication *before* starting TiDB (or any other service which uses TiKV), can dramatically speed up convergence. TiDB has a patch adding an initialized flag to the upcoming 3.0.0-rc.3 release; deployment systems could use this flag to ensure full region replication as a part of cluster setup.

3.4 Read Skew & Lost Updates

Under normal operation, without faults, TiDB 2.1.7, 2.1.8, and 3.0.0-beta.1 exhibited frequent cases of read skew (G-single, A5A) and other anti-dependency cycles (G2, G2-item), due to a retry mechanism which ignored transactional boundaries. Writes could also appear to take place in mutually incompatible orders between transactions. Updates could be lost altogether.

To begin, consider this trio of transactions from an append test. We write `r(34, [2 1])` to denote the value

[2] Technically, the process logs a fatal error, prints a stacktrace, and exits. PingCAP considers this a normal exit. We use "crash" to refer to any early termination.

of key 34 was read as the vector [2 1], and `append(36, 5)` to denote appending 5 to the current value of key 36.

T_1: `r(34, [2 1]), append(36, 5), append(34, 4)`
T_2: `append(34, 5)`
T_3: `r(34, [2 1 5 4])`

Because T_1 did not see T_2's append of 5 to 34, we know T_1 must have executed before T_2. We call this an *anti-dependency*, because T_1 read state which T_2 overwrote. However, we can infer from T_3's read of key 34 that the append of 4 must have executed *after* the append of 5, which means that T_2 must have executed before T_1. We call this a *write dependency*, because T_2 wrote data which T_1 later modified.[3]

Adya, Liskov, and O'Neil call this anomaly (G2) an *anti-dependency cycle*, because it includes at least one anti-dependency edge. Because there's exactly one anti-dependency, it is also Adya's *G-single*, which Berenson calls A5A, or *read skew*. Because these reads and writes are of individual records, as opposed to predicates, these dependencies are *item anti-dependencies*, and this history is also *G2-item*—an anomaly prohibited by repeatable read. Intuitively, T_1 both observes, but also fails to observe, T_2. In fact, it does both on a single record!

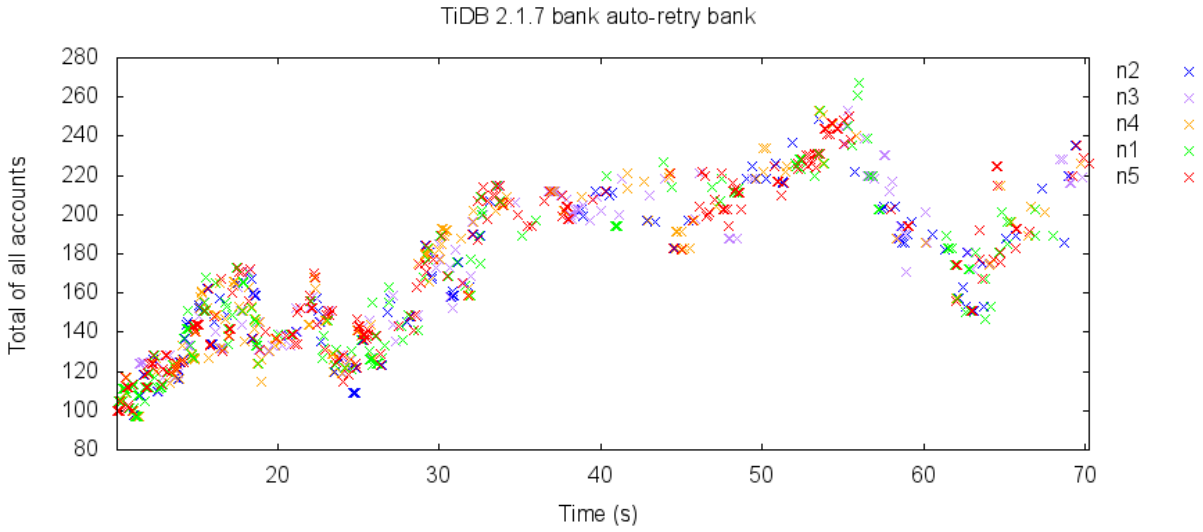


Figure 1:

To make this concrete, consider this run of Jepsen's bank workload on a healthy cluster: transactions reading multiple accounts could observe part, but not all, of a concurrent transfer transaction. If transfer T_1 moves $5 from account A to account B, a read could observe the decrement on A but *not* the increment on B, or vice versa. Consequently, the total of all accounts fluctuates over time. Moreover, transfer transactions could commit values based on skewed reads back into the database, permanently corrupting logical state.

In this plot, we show the total of all accounts over time, as observed by read-only transactions. Colors indicate which node (e.g. "n1") each query was executed on. Under snapshot isolation, this total should remain constant. However, the total drifts rapidly, doubling in under thirty seconds.

In addition, we observed numerous *incompatible orders*. Consider these three transactions involving key 7, from the same test run as our first anti-dependency example. We have elided some operations on other keys for clarity:

T_1: `... r(7, [1 2 3 4])`
T_2: `append(7, 7), r(7, [1 2 3 7])`
T_3: `r(7, [1 2 3 4 7])`

Even if transactions were *not* required to be atomic, this is not consistent with a single order of operations on key 7! We appended both 4 and 7 to key 7, but one

[3]The same test also contains numerous anti-dependency cycles involving *read dependencies* (T_2 reads state which T_1 wrote), instead of write dependencies.

[4]This is, in fact, the technical term.

transaction observes the list `[1 2 3 4]`, and one observes `[1 2 3 7]`. Then, a mystery occurs[4], and the append of 7 slots in *after* 4.[5]

T_1 and T_2 appear to observe *separate copies* of key 7. If there were separate copies, did TiDB merge them together? Some databases, like Cassandra, might pick one copy (say, `[1 2 3 7]`) and discard the other, on the basis of e.g. a last-modified timestamp—but this cannot have happened in TiDB, because we observe *both* 4 and 7 in a subsequent read. Did TiDB merge the *list* structures? Doubtful, because we (anticipating this possibility) chose to store these list values as *strings*, which are opaque to TiDB. It seems more likely that instead of merging *states*, TiDB is merging the append *operations*.

Indeed, this is exactly what happens. When one transaction conflicts with another, TiDB tries to *hide* the conflict from the user by automatically re-trying the transaction again. Here, T_2's append of 7 likely conflicted with the concurrent append of 4, and was forced to abort and retry. Let that retry be T_{2r}:

T_1: `... r(7, [1 2 3 4])`
T_2: `append(7, 7), r(7, [1 2 3 7]), abort`
T_{2r}: `append(7, 7), r(7, [1 2 3 4 7])`
T_3: `r(7, [1 2 3 4 7])`

However, we did *not* observe `[1 2 3 4 7]` from T_2's read! This is a second problem in TiDB's retry mechanism: it returns the *reads* from the aborted transaction T_2, then retries T_2's *writes*, without bothering to return the new values that would have been observed in T_{2r}.

T_1: `... r(7, [1 2 3 4])`
T_2: `append(7, 7), r(7, [1 2 3 7]), abort`
T_{2r}: `append(7, 7)`
T_3: `r(7, [1 2 3 4 7])`

Even if retried transactions *did* return their second reads, TiDB would still exhibit *lost updates*. Consider the following history, from a set CaS test. In this variant of the set test, operations are transactions on a single key, which stores a set of numbers in a text field. We write `2 :invoke :read nil` to signify that process 2 began a transaction to read the current state of that key. `2 :ok :read (0 1 2 3 4 5)` means that process 2 completed its read transaction successfully, and found the value to be the list `(0 1 2 3 4 5)`.

```
2 :ok     :read (0 1 2 3 4)
2 :invoke :read nil
1 :ok     :add  5
4 :invoke :read nil
4 :ok     :read (0 1 2 3 4 5)
1 :invoke :add  7
2 :ok     :read (0 1 2 3 4 5)
1 :ok     :add  7
3 :invoke :read nil
3 :ok     :read (0 1 2 3 4 5 7)
0 :ok     :add  6
0 :invoke :add  8
0 :ok     :add  8
4 :invoke :read nil
4 :ok     :read (0 1 2 3 4 6 8)
3 :invoke :read nil
3 :ok     :read (0 1 2 3 4 6 8)
```

Process 1 adds element 5 to the set by reading its current value (presumably, `(1 2 3 4)`), then writing back `(1 2 3 4 5)`. A concurrent add of 6, however, failed to observe process 1's write, and consequently destroyed it. It's possible for multiple "chains" of values to exist concurrently, as transactions squabble to be the last writer. Reads can observe updates popping into and out of existence repeatedly, over the course of multiple seconds. In this 30-second test with ~12 updates per second, TiDB lost 64 out of 378 insertions.

TiDB's automatic transaction retry mechanism *was* documented, but poorly: TiDB claimed repeatedly to support snapshot isolation, while providing essentially none of snapshot isolation's guarantees. The documentation for auto-retries was titled "Description of optimistic transactions", and it simply said that the automatic-retry mechanism "cannot guarantee the final result is as expected"—but did not describe *how*. We suspect users may be unaware of how badly broken this behavior is.

In 3.0.0-rc.2, automatic retries are disabled by default, and this anomaly does not occur unless users re-enable them.

3.5 Auto-Retry Redux

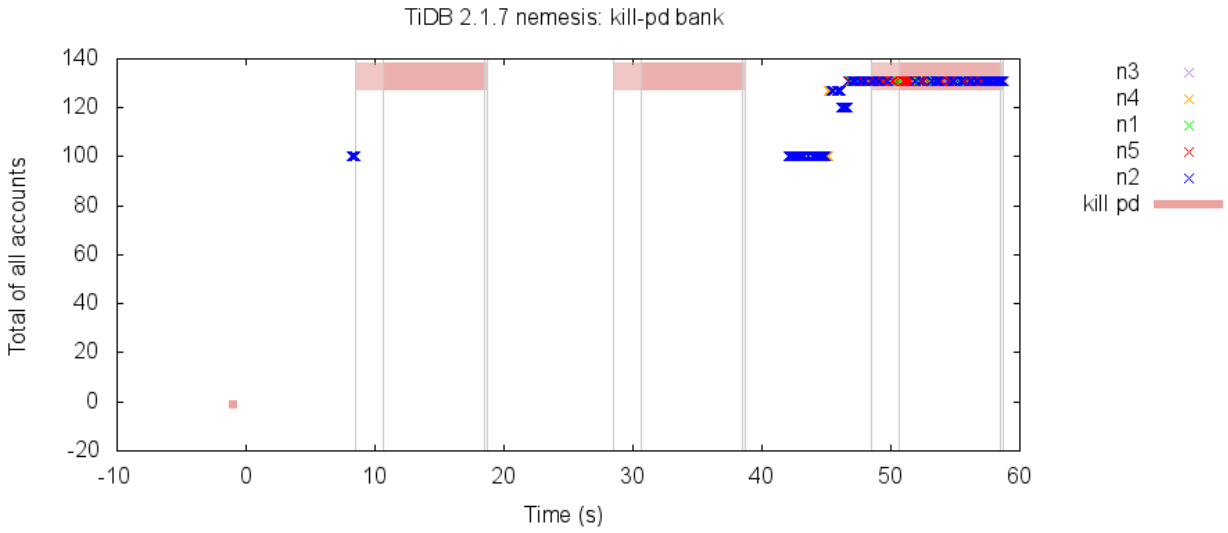
The documentation stated:

> To disable the automatic retry of explicit transactions, configure the `tidb_disable_txn_auto_retry` global variable...

[5] One could argue that this anomaly is a *write cycle*—Adya's G0. Let T_1 = `append(x, 1), r(x, [1])`, and T_2 = `append(x, 2), r(x, [2])`. Since T_1 was the first to write to `x`, it must have written before T_2. Likewise, since T_2 was also the first to write to `x`, it must have written before T_1. The astute reader may have noticed a small problem here: we cannot infer the version order for `x`, which we use to reconstruct the serialization graph for a history, because *no version order exists*. This so deeply violates the assumptions in Adya's formalism that we aren't sure *what* to call it.

However, `tidb_disable_txn_auto_retry` only disables *one* of TiDB's auto-retry mechanisms. A second mechanism, configured with `tidb_retry_limit`, comes into play when TiDB loses its connection to Placement Driver, e.g. due to a network partition, process pause, crash, or simply because the query was a bit slow.

For instance, in this bank test, the value of all bank accounts abruptly fluctuated, then rose from $100 to $136, shortly after recovering from a crash of Placement Driver.



All the same anomalies are present: read skew, incompatible orders, lost updates—they're simply much less frequent. Even *without* process crashes, we've observed occasional instances of read skew in healthy clusters.

With `tidb_retry_limit = 0`, TiDB 2.1.7 through 3.0.0-beta.1-40 passes tests for snapshot isolation, under partitions, process pauses, crashes, and clock skew. In 3.0.0-rc.2, `tidb_retry_limit` still defaults to 10, but `tidb_disable_auto_retry` defaults to on, and applies to both retry mechanisms. 3.0.0-rc.2 therefore passes with default settings.

3.6 Select ... For Update

Curiously enough, Vadim Tkachenko pointed out that TiDB allowed read skew in February 2017, and when Jepsen discussed their initial read skew histories with the PingCAP team, PingCAP's response was the same: TiDB only offers snapshot isolation, and users should use `select ... for update` to prevent instances of write skew. Indeed, PingCAP's initial Jepsen tests used select for update as well. Should users do the same?

The answer, in this case, is no: TiDB, as a snapshot isolated system, is supposed to prevent read skew, and users should not need to resort to `select ... for update` to prevent this anomaly. Instead, use `select ... for update` to prevent *write skew*: where transactions read intersecting data, but their write sets are disjoint. Using `select ... for update` on a read is somewhat like promoting that read to a write, allowing the conflict detector to check for write skew on that query. Of course, write skew may still occur on rows which don't use `select ... for update`—it is up to the application developer to decide when and how to use it.

Even with automatic retries enabled, using `select ... for update` with every read dramatically reduced the probability of transactional anomalies. For starters, it forces two transactions to conflict if any of the keys they read, or write, intersect. But in addition, it disables automatic retry of that specific transaction. It also, in our tests, dramatically lowered transaction throughput, which further reduces the probability of observing errors in any given test.

However, both with and without automatic retries, we still observed anti-dependency cycles using `select ... for update` on all reads, with versions 2.1.8, 3.0.0-beta.1, and 3.0.0-beta.1-40. Consider this history from an append test:

T_1: `r(3, nil) r(4, nil) append(4, 2)`

T_2: `r(3, nil) r(4, nil) append(3, 1)`

Since T_1 observes the initial state of key 3, it must precede T_2. Since T_2 observes the initial state of key 4, it must precede T_1: a contradiction! Since T_1 and T_2's read sets intersect, and their writes are disjoint, this history is also an example of write skew.

Every G2 anomaly we've observed with `select ... for update` involves the *first* write to some key. The cause? TiDB's locking mechanism can't lock keys which haven't been created yet, which allows write skew to manifest!

Whether this is actually illegal is somewhat up for debate. PingCAP's engineers have said repeatedly that using `select ... for update` prevents write skew, which definitely contradicts this behavior. However, the PostgreSQL and MySQL documentation describe `select ... for update` in terms of locking behavior on specific rows, rather than preventing anomalies. PingCAP's official documentation did not describe what `select ... for update` *should* have done. That documentation now states that locks are not acquired on rows that "do not exist in the result sets" for a given query.

№	Summary	Event Required	Fixed in
10410	Created tables may not exist	Create table	3.0.0-rc.2
4500	TiKV crash on startup	PD unavailable	3.0.0-rc.2
10495	TiKV crash on startup	No PD leader	3.0.0-rc.2
10470	TiDB crash on startup	KV region not ready	Won't fix
10657	New clusters may not fully replicate records	New cluster	Unresolved
10444	G2 & write skew despite select … for update	None	Documented
10075	Auto-retry read skew, lost updates	None	3.0.0-rc.2
10076	Less frequent auto-retry read skew, lost updates	None	3.0.0-rc.2

4 Discussion

Based on our cycle detector results, we believe TiDB's default isolation level (in versions 2.1.7, 2.1.8, 3.0.0-beta.1, and 3.0.0-beta.1-40) to be something *like* read committed, in that it prohibits dirty reads (G1a), but weaker than read uncommitted, in that it can apply operations from a transaction multiple times, which undermines the assumption of a total version order for each key. We have opted, conservatively, not to infer dependencies for contradictory orders. Under those assumptions, we have not observed write cycles (G0), dirty reads (G1a), intermediate reads (G1b), or cyclic information flow (G1c), but we emphasize that this decision is debatable. Even with these conservative assumptions, TiDB's default settings allow P4 (lost updates), and G-single (read skew), both of which are illegal under snapshot isolation.

With `tidb_retry_limit = 0`, and by default in 3.0.0-rc.2, TiDB appears to provide snapshot isolation, through network partitions, process crashes, and pauses. Note that `select ... for update` does not prevent read skew, or other kinds of G2 anomalies, during the first write to a particular key.

PingCAP updated the TiDB transaction documentation to note that TiDB could lose updates by default, and to identify the correct setting (`tidb_retry_limit`) used to disable both auto-retry features. As of June 11, 2019, the docs stated that retries were disabled by default, and that `tidb_disable_txn_auto_retry` controlled both retry mechanisms. These statements will likely apply to the upcoming release of 3.0.0, but do not reflect the behavior of current GA releases.

The transaction documentation now notes that SI allows some classes of phantoms (P3), and prohibits others (A3). In addition, the docs for DML now explain that `select ... for update` does not lock keys which do not exist as of the transaction's start time.

Jepsen is not a good measure of database performance; we evaluate pathological workloads with fixed concurrency, rather than realistic workloads with fixed request rates. We also note that Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. While we make extensive efforts to find problems, we cannot prove the correctness of any distributed system.

4.1 Recommendations

Users should be aware that TiDB 2.1.7, 2.1.8, 3.0.0-beta.1, and 3.0.0-beta.1-40, in their default configuration, do not provide snapshot isolation. TiDB exhibits stale reads, read skew, lost updates, and incompatible orders of updates due to two separate, improperly

designed, transactional retry mechanisms. We recommend that users upgrade to 3.0.0 once it stabilizes. In the meantime, users can prevent these anomalies by disabling retry mechanisms.

PingCAP has recommended using `select ... for update` to prevent write skew, but users may observe write skew even with `select ... for update`, while a record is first being created. Users can work around this issue by ensuring a record exists *before* transactions where write skew would create problems.

TiDB processes are fragile during initialization; they may start, run for a short while, and crash after a few minutes if their dependencies are unavailable, slow, or otherwise degraded. Users may consider wrapping PD, TiKV, and TiDB with a process supervisor which automatically restarts them. Be aware that service managers, like systemd, may disable a service when it restarts too often.

Be advised that new TiDB clusters will accept writes without fully replicating them, which could lead to the loss of committed data if a single node fails. Users should poll the PD API to ensure that all regions have the desired replica count before writing data. Avoid starting TiDB, or any other process that writes data, until TiKV's regions are fully replicated.

TiDB uses leader leases based on `CLOCK_MONOTONIC` to ensure that PD timestamps are monotonically increasing. PingCAP and Jepsen believe this could lead to consistency anomalies when monotonic clocks are not well behaved. However, we don't have experimental evidence to confirm this hypothesis. We encourage users to instrument their clock rates and alert on divergence, just in case.

4.2 Future Work

We recommended that PingCAP disable auto-retry mechanisms by default, and they did so in 3.0.0-rc.2. There *are* safe ways to do transactional auto-retry, which PingCAP may explore in the future, but this is an easy fix which offers users safety right away. We anticipate that 3.0.0, like 3.0.0-rc.2, will offer snapshot isolation by default. Some issues we found with crashes on TiKV startup should be fixed in 3.0.0 as well; others are still under consideration.

The absence of observed transactional anomalies during clock skew, including running `CLOCK_MONOTONIC` faster or slower than realtime on various nodes, is disconcerting: we believe this *should* lead to anomalies. One possible cause of this behavior is that our `libfaketime` shims don't intercept all timing calls that PD makes—for instance, log messages are written with correct, rather than accelerated, timestamps. Our tests may also be insufficiently rigorous to detect transient consistency violations, or our randomized schedules may not (frequently) create conditions where those anomalies could occur. We'd like to explore this more carefully in the future.

We have not evaluated filesystem or disk faults with TiDB, and cannot speak to crash recovery. Nor have we tested dynamic membership changes. Both might be fruitful avenues of investigation for future research.

This work was funded by PingCAP, and conducted in accordance with the Jepsen ethics policy. Our thanks to the PingCAP team for their assistance with this analysis—especially Shen Taining, Tang Liu, Morgan Tocker, Shuaipeng Yu, Li Shen, Menglong Huang, Huang Dongxu, Yin Chengwen, and Kevin Xu. We're grateful to Tim Kordas and Keyur Govande for their technical assistance with libfaketime. Finally, we wish to thank Kit Patella for her supporting work on the Jepsen library, and to Peter Alvaro for theoretical guidance.