JEPSEN

PostgreSQL 12.3

Kyle Kingsbury
2020-06-12

PostgreSQL is a widely-known relational database system. We evaluated PostgreSQL using Jepsen's new transactional isolation checker Elle, and found that transactions executed with serializable isolation on a single PostgreSQL instance were not, in fact, serializable. Under normal operation, transactions could occasionally exhibit G2-item: an anomaly involving a set of transactions which (roughly speaking) mutually fail to observe each other's writes. In addition, we found frequent instances of G2-item under PostgreSQL "repeatable read", which is explicitly proscribed by commonly-cited formalizations of repeatable read. As previously reported by Martin Kleppmann, this is due to the fact that PostgreSQL "repeatable read" is actually snapshot isolation. This behavior is allowable due to long-discussed ambiguities in the ANSI SQL standard, but could be surprising for users familiar with the literature. A patch for the bug we found in serializability is scheduled for the next minor release, on August 13th, and the presence of G2-item under repeatable read could be readily addressed through documentation. This work was performed independently, without compensation, and conducted in accordance with the Jepsen ethics policy.

1 Background

PostgreSQL is a major open-source relational database with a 23-year history and a broad range of features. While Jepsen's work has traditionally focused on distributed systems, our tooling is readily applicable to traditional, single-node databases. In this report, we present the results of applying Jepsen's generative concurrency testing to PostgreSQL 12.3.

Prior to version 9.1, PostgreSQL's documentation claimed to offer up to serializability, "as if transactions had been executed one after another, serially, rather than concurrently…. The Serializable mode provides a rigorous guarantee that each transaction sees a wholly consistent view of the database." However, this was not true: PostgreSQL's "serializable" was in fact snapshot isolation (SI).

Informally, snapshot isolated systems appear to start each transaction with a fixed, instantaneous snapshot of the database, reflecting only committed state. Writes performed in a transaction appear to apply atomically at commit time, and a transaction can only commit if no other transaction has modified the same objects as the transaction has written, since the snapshot was taken. This is (as the paper which formalized snapshot isolation made clear) not serializable: transactions whose write sets do not intersect can commit without observing each other's effects, which could lead to application-level consistency violations.

In version 8.0, PostgreSQL's documentation clarified that "in fact PostgreSQL's Serializable mode does not guarantee serializable execution in this sense," and went on to specify that PostgreSQL lacked a predicate locking system.

In version 9.1, PostgreSQL contributors Grittner and Ports added support for true serializability, based on research by Cahill, Röhm, and Fekete into serializable snapshot isolation (SSI). In short, SSI extends SI by checking, at runtime, for a dependency relationship between transactions called a *dangerous structure*: a pair of adjacent read-write dependencies between three transactions. Preventing these dangerous structures, in addition to snapshot isolation's normal rules, yields only serializable executions. For the last nine years, PostgreSQL's "serializable" mode has justifiably claimed to offer serializability.

PostgreSQL's "repeatable read" remains snapshot isolation, but the concurrency control documentation surprisingly does not mention the term. Instead, it offers:

> The Repeatable Read isolation level only sees data committed before the transac-

> tion began; it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions… This is a stronger guarantee than is required by the SQL standard for this isolation level, and prevents all of the phenomena described in Table 13.1 except for serialization anomalies. As mentioned above, this is specifically allowed by the standard, which only describes the minimum protections each isolation level must provide.

"Serialization anomalies" is a somewhat ambiguous term: the documentation simply describes it as a result which is "inconsistent with all possible orderings of running those transactions one at a time". To better understand what "serialization anomalies" specifically entail, we devised an experiment.

2 Test Design

We designed a test harness for PostgreSQL using the Jepsen testing library. Our test installs PostgreSQL 12.3-1.pgdg100+1 (the current stable version) on a single Debian 10 node, or optionally connects to an existing PostgreSQL installation. We also evaluated versions 9.5.22, 10.13, and 11.8. Our test can kill PostgreSQL processes in random order to help measure crash safety, but our findings here do not require process crashes to reproduce. We used the default configuration provided by PostgreSQL's official Debian packages with only minor changes (e.g. for binding network ports), and, during some tests, shortened autovacuum naptime and enabled more detailed logging.

Our test workload generates randomized transactions of append and read operations across an collection of list objects, chosen with exponential frequency. Each object is identified by a unique integer logical key. We store each object as a row in one of several tables, chosen by the hash of the key. Object keys are stored in two fields: a primary key `id`, and an unindexed secondary key `sk`, which we use to test access by table scans.[1] The value of each list is stored as a comma-separated `TEXT` column.

We append unique integer elements to a list identified by key (either via `id` or `sk`) using `INSERT ... ON CONFLICT DO UPDATE`, or, alternatively, via an update, checking to see if any rows were modified, then backing off to an insert, and if *that* failed, updating again. Reads return the current list of integers for a particular object, e.g. via `SELECT (val) FROM txn0 WHERE id = ?`.

Our test applies these transactions to PostgreSQL using the JDBC PostgreSQL driver (version 42.2.12), and analyzes the resulting history using the Elle transaction isolation checker. Elle infers a transaction dependency graph over experimentally recorded histories, and searches for cycles (and non-cyclic anomalies) in that graph. This allows us to detect a broad range of anomalies from Adya, Liskov & O'Neil's Generalized Isolation Level Definitions, including G0 (dirty write), G1a (aborted read), G1b (intermediate read), G1c (cyclic information flow), G-single (read skew), and G2-item (anti-dependency cycle). We also check for internal consistency, verifying that transactions observe values consistent with their own prior writes, duplicate effects, and garbage values (e.g. elements which were never written).

3 Results

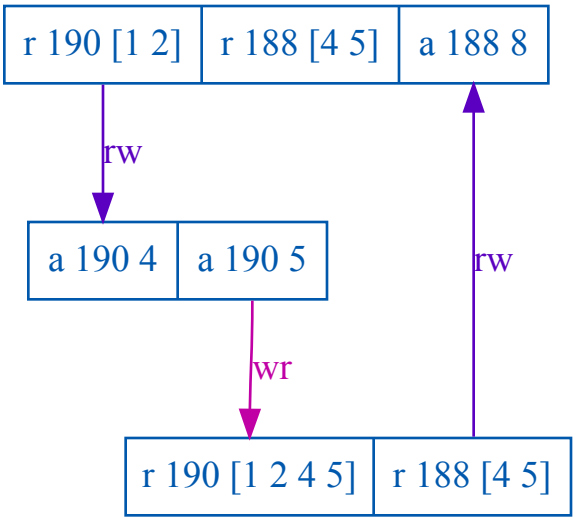
In most respects, PostgreSQL behaved as expected: both read uncommitted and read committed prevent write skew and aborted reads. We observed no internal consistency violations. However, we have two surprising results to report. The first is that PostgreSQL's "repeatable read" is weaker than repeatable read, at least as defined by Berenson, Adya, Bailis, et al. This is not necessarily wrong: the ANSI SQL standard is ambiguous. The second result, which is *definitely* wrong, is that PostgreSQL's "serializable" isolation level isn't serializable: it allows G2-item during normal operation.

3.1 Repeatable Read

PostgreSQL's "repeatable read" isolation level is actually snapshot isolation, and we observed no SI-violating anomalies when using "repeatable read". In fact, the histories we recorded were consistent with strong snapshot isolation, a stronger consistency model which prohibits stale reads and other realtime anomalies.

However, we observed numerous violations of repeatable read, as formally defined by Berenson, Adya, et al. For example, consider this history, which produced roughly 140 anti-dependency cycles per minute. Here's a short cycle from that history consisting of a trio of transactions—each of which appeared to execute before the next.

[1] Neither process crashes, multiple tables, nor secondary-key access is required to reproduce our findings in this report. The technical justification for including them in this workload is "for funsies".



The top transaction begins by reading key 190, and finds the list `[1 2]`. The middle transaction appends 4 to key 190, resulting in the version `[1 2 4]`. Since that write overwrote the state that the top transaction read, we know that the middle transaction must have executed after the top transaction. We call this relationship an *anti-dependency*, and represent it as an edge labeled `rw`.

The middle transaction appended 5 to key 190, which was then visible to the bottom transaction's read of `[1 2 4 5]`. This write-read dependency is represented by an edge labeled `wr`. However, the bottom transaction read key 188, and did *not* observe the top transaction's append of 8. That anti-dependency implies the bottom transaction must have executed before the top transaction: a cycle!

This dependency cycle contains two anti-dependency edges, which makes it a G2 phenomenon in the language of Adya's formalism. Since all of these reads occurred when reading objects by their primary key[2], it is also G2-item: a phenomenon expressly prohibited under Adya's formalization of repeatable read. We believe this is one type of "serialization anomaly" referred to in the PostgreSQL documentation.

However, these anomalies *are* allowable under ANSI SQL's definition of repeatable read, thanks to ambiguously worded plain-English definitions of prohibited phenomena. In fact, this ambiguity is part of what *prompted* Berenson, Bernstein et al. to write A Critique of ANSI SQL Isolation Levels, and to formalize the definition of snapshot isolation in the first place. In that work, Berenson et al. develop two interpretations of the ANSI anomalies: one strict, and one broad. They argue that the strict interpretations fail to capture behaviors which are intuitively incorrect, and that ANSI *meant* to define the broad ones.

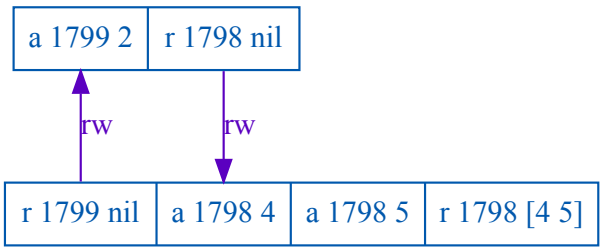
> Strict interpretations A1, A2, and A3 have unintended weaknesses. The correct interpretations are the Broad ones.

Under the broad interpretations preferred by Berenson et al., snapshot isolation is *not comparable* with repeatable read: SI allows histories RR proscribes, and vice versa. Under the strict interpretation, SI is *stronger* than RR (indeed, SI is stronger than anomaly serializable!), and these G2-item anomalies are allowed under repeatable read.

Whether PostgreSQL's repeatable-read behavior is correct therefore depends on one's interpretation of the standard. It is surprising that a database based on snapshot isolation would reject the strict interpretation chosen by the seminal paper on SI, but on reflection, the behavior *is* defensible.

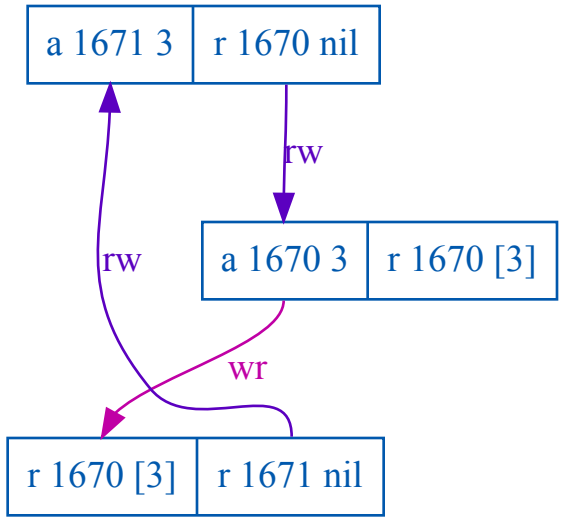
3.2 Serializable

A more serious problem arose when we tested PostgreSQL's serializable isolation level: it *also* exhibited G2-item under normal operation. In this two-minute test run, Jepsen detected six cases of G2-item. For example, consider this pair of transactions, in which each failed to observe the other's insert:

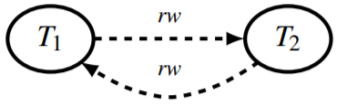


Alternatively, consider the following trio of transactions. The top transaction missed the middle transaction's creation of key 1670, which *was* observed by the bottom, read-only transaction. However, that bottom transaction failed, in turn, to observe the first transaction's creation of key 1671. Notably, the read-write transactions are serializable if taken by themselves. The read-only transaction is necessary for this cycle: it observes the effects of some, but not all, "logically prior" transactions.

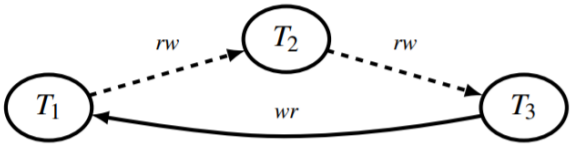
[2] Our tests support updates and reads both by primary key and by secondary key: a predicate read. This type of anomaly occurs in both contexts.



Indeed, these dependency graphs correspond exactly to examples 1 ("Simple Write Skew") and 2 ("Batch Processing") from the PostgreSQL Serializable Snapshot Isolation paper, shown below. Their SQL statements are different, of course—but like Example 1, our first cycle involves a pair of transactions which read one key and write another, each failing to observe the other's effects; and our second involves a read-only transaction which precedes, via two adjacent rw anti-dependencies, a transaction which wrote state which the read-only transaction observed. These cycles are precisely what PostgreSQL's SSI implementation is meant to prevent!



(a) Example 1: Simple Write Skew



(b) Example 2: Batch Processing

Every instance of G2-item we observed under serializable isolation involved at least one read-write conflict for a *freshly inserted row*. Cycles could involve rw anti-dependencies on updates to existing rows, but at least one insert appeared to be necessary.

Following a discussion with PostgreSQL contributors, Peter Geoghegan identified the likely cause of this issue: the conflict detection mechanism could, given three concurrent transactions, incorrectly identify an updating transaction's transaction ID (XID) as responsible for *both* the original and updated versions of a tuple, rather than using the transaction ID which originally created the tuple. By flagging the wrong transaction as a potential conflict, it allowed a transaction to commit while failing to observe a prior transaction's writes. Geoghegan, working with other members of the PostgreSQL community, has written a patch to flag the correct transaction ID, and added a regression test. In their testing, this appears to resolve the issue.

This code has gone essentially untouched since the introduction of serializable snapshot isolation in 2011. Together, we confirmed that this bug was present in PostgreSQL 9.5.22, 10.13, 11.8, 12.3, and 13; we assume it is present in every extant version.

4 Discussion

In our testing of PostgreSQL 12.3, transactions executed at read committed appeared correct: we never observed G0 (dirty write), G1a (aborted read), or G1b (intermediate read). PostgreSQL "repeatable read" appears consistent with strong snapshot isolation, but allows G2-item, which is prohibited in formalizations of repeatable read. However, this behavior could be interpreted as consistent with ANSI SQL repeatable read. Finally, PostgreSQL "serializable" allows G2-item under normal operation, due to a bug in the conflict detection mechanism. A patch has been committed, and this class of serializability violations should be resolved in the next minor release—presently scheduled for August 13th.

PostgreSQL has an extensive suite of hand-picked examples, called `isolationtester`, to verify concurrency safety. Moreover, independent testing, like Martin Kleppmann's Hermitage has also confirmed that PostgreSQL's serializable level prevents (at least some!) G2 anomalies. Why, then, did we immediately find G2-item with Jepsen? How has this bug persisted for so long?

PostgreSQL's isolation tests, Hermitage, and most transactional Jepsen tests (prior to Elle) relied on executing a handful of cleverly constructed transactions with hand-proven invariants. For instance, this isolationtester specification verifies serializability by performing a sequence of transactions proposed by Fekete, O'Neil, & O'Neil in A Read-Only Transaction Anomaly Under Snapshot Isolation. Jepsen's bank test is based on a narrowly-defined class of transactions which preserves a total-balance invariant under snapshot isolation. Hermitage checks for G2-item by performing a pair of symmetric read and update transactions—

which *does* successfully demonstrate G2-item under PostgreSQL "repeatable read", but not under serializable.

Elle, however, is different: it allows us to generate a broad class of transactions, while still inferring strict properties over the resulting histories. This property-based approach allows us to catch unexpected behaviors that no one thought to explicitly test. In this case, it identified the possibility that concurrent updates and inserts could confuse the conflict-detection mechanism into misidentifying which transaction was responsible for a conflict.

That said, the list-append test we devised here verifies only a handful of SQL operations over a simple schema. Mature SQL databases like PostgreSQL are complex organisms with a myriad of interacting components and optimizations. Jepsen assumes that our tests exercise only a fraction of PostgreSQL's possible behaviors.

As always, we note that Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. While we try hard to find problems, we cannot prove the correctness of any distributed system.

4.1 Recommendations

Users should be aware that PostgreSQL's "repeatable read" is in fact snapshot isolation—a fact long-understood in the PostgreSQL community and previously reported by Kleppmann. Since G2-item is prohibited under common formalizations of repeatable read, users may have designed applications assuming this held true for PostgreSQL. In this case, users may wish to run selected transactions under serializable isolation instead, add explicit locking, or redesign those transactions such that they are no longer sensitive to G2-item.

We recommend that the PostgreSQL team update their concurrency control documentation to resolve the ambiguity around "repeatable read". The current documentation does not mention the term "snapshot isolation"—stating that PostgreSQL's "repeatable read" actually means snapshot isolation would immediately clarify matters. The documentation could also provide clearer guidance to users by replacing the ambiguous "serialization anomaly" with G-single, G2-item, and G2; SI prohibits G-single but allows G2-item and G2.

As for whether snapshot isolation is stronger than repeatable read, one possible solution would be to adopt Berenson et al.'s definitions, and state that snapshot isolation is incomparable with repeatable read: SI allows some anomalies which are prohibited under RR (e.g. write skew), but RR allows other anomalies (e.g. phantoms) which are prohibited under SI. Doing so would bring PostgreSQL in line with a twenty-five year thread of scholarship on transactional isolation by Berenson, Adya, Bailis, et al.

However, as Ports & Grittner note in their paper on PostgreSQL's serializable snapshot isolation, the ANSI specification *is* ambiguous, and the G2-item anomalies we observed do not necessarily violate the strict interpretation of the phenomena prohibited by repeatable read. In this case, we suggest that PostgreSQL explicitly state their choice of the strict, rather than broad, interpretation.

It appears that no version of PostgreSQL has ever guaranteed serializability. Users should be aware that concurrent update and insert transactions may exhibit G2-item. High-contention workloads are especially susceptible. The PostgreSQL team has written tests to reproduce the problem and is evaluating a patch; we recommend upgrading once the next minor release becomes available.

One final note: our testing suggests that PostgreSQL provides (or, in the case of serializability, will provide once the G2-item bug is resolved) *more* than snapshot isolation and serializability. Our histories appeared consistent with *strong* snapshot isolation and *strict* serializability, both of which ensure compatibility with a real-time order, in addition to preventing the usual dependency-graph anomalies. We are unsure if this is intentional, or whether it holds in all cases, but if so, PostgreSQL should feel free to claim these stronger consistency models!

4.2 Future Work

PostgreSQL's contributors are evaluating a patch to resolve the serializability violation we discovered, and writing clarifying documentation for snapshot isolation versus repeatable read.

Elle's list-append workload is limited to reads and appends over datatypes which are isomorphic to lists. We have no way to test deletions, replacements, or other list operations: there could be latent issues in those codepaths. We have other workloads available for registers and sets, albeit supporting weaker inferences. Both could be implemented on PostgreSQL, which could help cover additional ground.

It seems unlikely that we can efficiently check, or even model, *all* functionality provided by modern SQL

databases. Aggregations, subqueries, and stored procedures are in common use, and none are verifiable by our current approach. In particular, predicates are a key part of the SQL standard and have been encoded in Adya's formalism—the representation of transactions which underpins Elle. We have thus far punted on how to represent, generate, and verify the correctness of transactions involving predicates. This means that Elle can only identify G2-item, not G2 in generality. We therefore cannot distinguish between repeatable read and serializable. This seems the most promising avenue for future research.

This work was performed independently, without compensation, and conducted in accordance with the Jepsen ethics policy. This work would not have been possible without the kind assistance of PostgreSQL contributors Andres Freund, Peter Geoghegan, Félix Gerzaguet, Thomas Munro, and Daniel Verite. Jepsen also wishes to thank C. Scott Andreas, André Arko, Silvia Botros, Lita Cho, Peter Geoghegan, Félix Gerzaguet, Alex Rasmussen, and James Turnbull for their feedback on early drafts. Kevin Cox and Frank McSherry offered suggestions on the published report.