MySQL 8.0.34

Peter Alvaro & Kyle Kingsbury

2023-12-19

MySQL is a popular relational database. We revisit Kleppmann's 2014 Hermitage and confirm that MySQL's Repeatable Read still allows G2-item, G-single, and lost update. Using our transaction consistency checker Elle, we show that MySQL Repeatable Read also violates internal consistency. Furthermore, it violates Monotonic Atomic View: transactions can observe some of another transaction's effects, then later fail to observe other effects of that same transaction. We demonstrate violations of ANSI SQL's requirements for Repeatable Read. We believe MySQL Repeatable Read is somewhat stronger than Read Committed. As a lagniappe, we show that AWS RDS MySQL clusters routinely violate Serializability. This work was performed independently without compensation, and conducted in accordance with the Jepsen ethics policy.

1. Background

MySQL needs little introduction. Over the last 28 years it has become one of the most widely deployed SQL databases. MySQL is primarily used for online transaction processing (OLTP) workloads, but is also deployed as a part of OLAP and queuing systems.

MySQL was designed as a single-server database, but has been extended with various multi-node replication schemes, including several flavors of binlog replication, group replication, NDB cluster, and third-party plugins like Galera Cluster & Percona XtraDB Cluster. Previous Jepsen work discussed Percona XtraDB Cluster and Galera Cluster. In this analysis we focus on single-server MySQL, but we also evaluated clusters with a single writeable primary and read-only secondaries using binlog replication.

MySQL also supports multiple storage engines which have different safety properties. We focus on the default: InnoDB. Throughout this text, we use "MySQL" to mean "MySQL using the InnoDB storage engine."

1.1. ANSI SQL Isolation is Bad, Actually

In order to discuss the nuances of SQL isolation levels, we must first explain some history. In 1977 Gray, Lorie, Putzolu, and Traiger published Granularity of Locks and Degrees of Consistency in a Shared Data Base, which introduced four increasingly safe degrees of transaction consistency. In 1973 IBM developed System R, one of the first relational databases, and shortly thereafter introduced SQL as a query language for it. System R's success spawned a slew of relational databases using SQL, many with distinct flavors of concurrency control. Starting in 1986 ANSI released[1] a series of standards codifying SQL behavior. The third revision of the standard, SQL-92, defined the semantics of concurrent transactions through four transaction isolation levels, again with increasing degrees of safety. As with Gray et al., these isolation levels were related to the behavior of increasingly conservative locking regimes. However, to allow databases which used non-locking concurrency control, ANSI phrased their levels in terms of three possible phenomena which should not occur. As the standard puts it, "the following phenomena are possible:"[2]

P1 ("Dirty Read")
SQL-transaction T1 modifies a row. SQL-transaction T2 then reads that row before T1 performs a COMMIT. If T1 then performs a ROLLBACK, T2 will have read a row that was never committed and that may thus be considered to have never existed.

P2 ("Non-Repeatable Read")
SQL-transaction T1 reads a row. SQL-transaction T2 then modifies or deletes that row and performs a COMMIT. If T1 then attempts to reread the row, it may receive the modified value or discover that the row has been deleted.

P3 ("Phantom")
SQL-transaction T1 reads the set of rows N that satisfy some <search condition>. SQL-transaction T2 then executes SQL-statements that generate one or more rows that satisfy the <search condition> used by SQL-transaction T1. If SQL-transaction T1 then repeats the initial read with the same <search condition>, it obtains a different collection of rows.

ANSI SQL defines four isolation levels in terms of these anomalies. It begins by stating that transactions which execute at the Serializable isolation level must be equivalent to some serial execution, i.e., one in which that set of transactions executed one after the other. Then it says "the isolation levels are different with respect to phenomena P1, P2, and P3." The standard provides the following table which "specifies the phenomena that are possible and not possible for a given isolation level":

Level	P1	P2	P3
Read Uncommitted	Possible	Possible	Possible

Level	P1	P2	P3
Read Committed	Not Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

In 1995 Berenson, Bernstein, Gray,[3] Melton, and the O'Neils published A Critique of ANSI SQL Isolation Levels, which laid out critical flaws in these definitions. "The three ANSI phenomena are ambiguous. Even their broadest interpretations do not exclude anomalous behavior."

For example, P1 says something bad might happen if T_1 were to abort, but doesn't actually say whether it aborts or not. Some people interpreted the standard to require T_1 aborts. This would make it legal under read committed for transactions to read as-yet-uncommitted state from other transactions (so long as they went on to commit). T_1 could write $x = 1$, T_2 could write $y = 2$, and T_1 and T_2 could both see each other's effects. This kind of circular information flow seems bad, but whether the standard allows it is a matter of interpretation. Similar ambiguities exist for P2 and P3.

Even interpreted broadly, preventing P1, P2, and P3 does not ensure Serializability. The standard omits a critical phenomenon P0 ("dirty write"), in which transaction T_1 writes some row, transaction T_2 overwrites T_1's write, and T_1 commits. This is clearly undesirable, but legal under ANSI Read Uncommitted, Read Committed, and Repeatable Read. Furthermore, ANSI SQL P3 only prohibits inserts affecting a predicate, but not updates or deletes.

In 1999, Atul Adya built on Berenson et al.'s critique and developed formal and implementation-independent definitions of various transaction isolation levels, including those in ANSI SQL.[4] As he notes:

> The ANSI definitions are imprecise because they allow at least two interpretations; furthermore, the anomaly interpretation is definitely incorrect. The preventative interpretation [meaning Berenson et al.'s interpretation which added P0, expanded P3, and so on] is correct in the sense that it rules out undesirable (i.e., non-serializable) histories. However, this interpretation is overly restrictive since it also rules out correct behavior that does not lead to inconsistencies and can occur in a real system. Thus, any system that allows such histories is disallowed by this interpretation, e.g., databases based on optimistic mechanisms.

Adya first defines a dependency graph between transactions. There are three main types of dependencies, which we summarize informally:

Write-Write
: Transaction T_1 writes some version x_1 of object x, which transaction T_2 overwrites by installing the next version of x: x_2.

Write-Read
Transaction T_1 writes version x_1, which transaction T_2 reads.

Read-Write
Transaction T_1 reads version x_1, which transaction T_2 overwrites by installing the next version of x: x_2.

Adya then defines portable isolation levels PL-1, PL-2, PL-2.99, and PL-3, which capture what the ANSI SQL standard (arguably) intended. Each level rules out progressively broader kinds of cycles in the transaction dependency graph:

PL-1 ("Read Uncommitted")
Prohibits G0 ("write cycle"): a cycle of write-write dependencies. This is analogous to Berenson's P0 ("dirty write").

PL-2 ("Read Committed")
Prohibits G0 and G1. G1 consists of three anomalies: G1a ("aborted read"), G1b ("intermediate read")[5], and G1c ("cyclic information flow"): a cycle of write-write or write-read dependencies. This captures the essence of the preventative interpretation of P1.

PL-2.99 ("Repeatable Read")
Prohibits G0, G1, and G2-item: a cycle involving write-write, write-read, or read-write edges *without predicates*. This captures the essence of ANSI SQL Repeatable Read, which is distinguished from Serializable only by predicate safety.

PL-3 ("Serializable")
Prohibits G0, G1, and G2: a cycle involving write-write, write-read, or read-write edges (with or without predicates). This guarantees equivalence to a serial execution.

Adya's dependency graph-based isolation levels resolved the ambiguities of the ANSI definitions, and remains the most widely-used formalism for characterizing transaction histories and anomalies. Jepsen generally uses Adya's formalism.

Although the database community has known for decades that ANSI SQL's isolation level definitions are broken, the standard's language remained unchanged. The same ambiguous, incomplete definitions are still present in the 2023 revision of the standard.

1.2. Repeatable Read

ANSI SQL's isolation levels are bad, but some levels have caused more problems than others. The fact that different database vendors provide isolation levels with the same names is useful only if the semantics of a particular level are consistent across vendors. And for three of the isolation levels, this is usually true. Most databases we've evaluated do ensure at least PL-1 for read uncommitted, PL-2 for Read Committed, and PL-3 for Serializable.[6] However, there is less agreement on the semantics of Repeatable Read.

Adya's PL-2.99 definition of Repeatable Read is quite strict, ruling out all dependency cycles except those involving predicate edges. The ANSI definition, while ambiguous, appears similarly strict: it prohibits all listed anomalies except "phantoms," which depend on predicate reads. This is not surprising when we consider the roots of the isolation levels in locking regimes: the original Repeatable Read was the isolation level you got when you followed strict two-phase locking (holding read and write locks until the end of the transaction) but did not enforce predicate locking.

For some reason DB vendors have chosen different definitions of Repeatable Read than Adya and the ANSI standard, and almost no vendors provide the same guarantees at Repeatable Read. In fact, Microsoft SQL Server is the only database that we have tested for which Repeatable Read appears to correspond to PL-2.99 and the ANSI definition. In Postgres, Repeatable Read means Snapshot Isolation, a level that is neither stronger nor weaker than PL-2.99.[7]

With this diversity of implementations in mind, we turn to the question at hand: what does MySQL do?

1.3. MySQL Isolation

The transaction isolation levels documentation for MySQL indicates that MySQL with InnoDB "offers all four transaction isolation levels described by the SQL:1992 standard": Read Uncommitted, Read Committed, Repeatable Read, and Serializable. The documentation goes on to explain how MySQL achieves these isolation levels.

At MySQL Read Uncommitted, transactions should behave "like Read Committed," except for allowing dirty read: an anomaly where a read observes "data that was updated by another transaction but not yet committed."

At MySQL Read Committed, every individual consistent read reads from a fresh snapshot of committed state. A "consistent read" is the default behavior for reads (e.g. `SELECT * FROM problems`) and is the focus of this report. There are also stronger reads (e.g. `SELECT ... FOR UPDATE`) which explicitly request locks, and weaker reads (e.g. `SELECT ... SKIP LOCKED`) which skip some of the default locks.

MySQL Repeatable Read, the default isolation level, ensures safety through a snapshot mechanism:

> Consistent reads within the same transaction read the snapshot established by the first read. This means that if you issue several plain (nonlocking) `SELECT` statements within the same transaction, these `SELECT` statements are consistent also with respect

> to each other.

MySQL's consistent read documentation further emphasizes that reads should operate on a snapshot of the database taken by the first read in a transaction.

> If the transaction isolation level is `REPEATABLE READ` (the default level), all consistent reads within the same transaction read the snapshot established by the first such read in that transaction.…
>
> Suppose that you are running in the default `REPEATABLE READ` isolation level. When you issue a consistent read (that is, an ordinary `SELECT` statement), InnoDB gives your transaction a timepoint according to which your query sees the database. If another transaction deletes a row and commits after your timepoint was assigned, you do not see the row as having been deleted. Inserts and updates are treated similarly.

The documentation for Serializable isolation says Serializable is "like `REPEATABLE READ`, but `InnoDB` implicitly converts all plain `SELECT` statements to `SELECT ... FOR SHARE` if `autocommit` is disabled."

There ends the isolation level documentation. However, if one digs deeper into the consistent read documentation, there is a curious note on the semantics of Repeatable Read:

> The snapshot of the database state applies to `SELECT` statements within a transaction, not necessarily to DML statements. If you insert or modify some rows and then commit that transaction, a `DELETE` or `UPDATE` statement issued from another concurrent `REPEATABLE READ` transaction could affect those just-committed rows, even though the session could not query them. If a transaction does update or delete rows committed by a different transaction, those changes do become visible to the current transaction.

This is confusing: the ANSI SQL standard and MySQL's own reference manual both consider `SELECT` to be a DML statement, but this note seems to think they're different. It appears that writes made by a Repeatable Read transaction can affect rows that the transaction could not read. But what does it mean for a different transaction's updates to become visible to the current transaction? How does that align with MySQL's claim that multiple reads in a Repeatable Read transaction "read the snapshot established by the first read"? What happened to the timepoint assigned by the first read?

This calls for a test.

2. Test Design

We designed a small test suite for MySQL using the Jepsen testing library at version 0.3.4. We used the `mysql-connector-j` JDBC adapter as our client. We tested MySQL

8.0.34, and MariaDB 10.11.3 on Debian Bookworm. Our tests ran against a single MySQL node as well as binlog-replicated clusters with one or two read-only followers, without failover. We also ran our test suite against a hosted MySQL service: AWS's RDS Cluster, using the "Multi-AZ DB Cluster" profile. This is the recommended default for production workloads, and offers a binlog-replicated deployment of MySQL 8.0.34 where secondary nodes support read queries.

Our tests included basic fault injection for process pauses, crashes, and network partitions, as well as the loss of un-fsynced writes to disk. However, almost every finding we discuss in this work occurred in healthy, single-node MySQL instances.

2.1. List Append

Our main workload used Elle's list-append checker for transactional isolation. In a nutshell, Elle infers Adya's write-write, write-read, and read-write dependencies between transactions, then looks for cycles in the resulting dependency graph. Each cycle it finds demonstrates that a particular set of isolation levels do not hold.

At a high level the append workload performs randomly generated transactions comprising reads and appends of unique integer elements to a collection of lists identified by primary key. As in our previous tests of SQL databases, we encoded these lists as a `text` field of comma-separated values, one per row, and used SQL `CONCAT` to append elements.[8] We split these rows across multiple tables with a structure like

```
create table "txn0" (
  id int not null primary key,
  val text
);
```

Over the last few years we've made several improvements to Elle which allow it to detect more anomalies.[9] When some appended elements are never read in a list-append test, Elle now infers ww and rw dependencies, placing them after the last value seen in the longest successful read. We now detect P4 (lost update) anomalies explicitly, even when version orders are uninferable. Elle also searches for cycles involving multiple nonadjacent read-write anti-dependencies which also include real-time and process edges. This lets us detect more subtle violations of both strong and strong session Snapshot Isolation.

2.2. Non-Repeatable Read

When Elle identified internal consistency violations at Repeatable Read, we designed a workload specifically to stress MySQL's Repeatable Read semantics, which works as follows. We create a simple table of people identified by primary key, and populate it with a single row:

```
create table people (
  id     int not null,
  name   text not null,
  gender text not null,
  primary key (id))
);
insert into people (id, name, gender)
  values (0, "moss", "enby");
```

We then perform a series of write transactions which update only the row's name. Concurrently, a second series of transactions each read the row's name, update its gender field, and read the name again. Violations of Repeatable Read manifest as the row's name changing between the two reads. We also perform deletions and re-insertions of row 0, in case they behave differently than plain updates.

2.3. Monotonic Atomic View

We also designed a second targeted workload to illustrate violations of Monotonic Atomic View. This workload creates a single table with two rows:

```
create table mav (
  id      int not null,
  `value` int not null,
  noop    int not null,
  primary key (id)
);
insert into mav (id, `value`, noop)
  values (0, 0, 0);
insert into mav (id, `value`, noop)
  values (1, 0, 0);
```

We perform a mix of write and read transactions. Each write increments the `value` of row 0, then increments row 1. Reads select the value of row 0, set the `noop` field of row 1 to a random value, then read the values of 1 and 0. Under Monotonic Atomic View, these reads should be monotonically increasing. For example, once a reader observes value 2, it should thereafter see every row's value as 2 or higher.

2.4. LazyFS

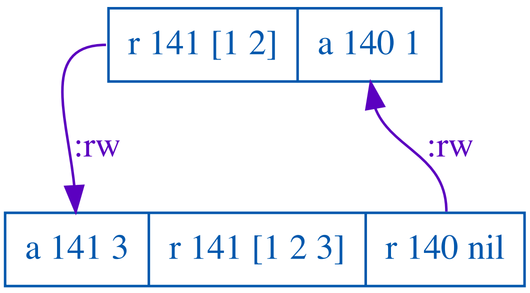
In 2022 Jepsen commissioned the University of Porto's INESC TEC to develop LazyFS: a FUSE filesystem for simulating the loss of un-fsynced writes. LazyFS maintains an in-memory page cache of data which has been written but not fsynced, flushing it to underlying storage only as the cache fills or `fsync` calls are made. A test harness can ask LazyFS to discard its cache at any time, simulating what might happen during a power failure. João Pedro Rodrigues Azevedo's dissertation discusses this work in detail, including several database bugs.

LazyFS has been integrated with Jepsen for a little over a year, but this is the first public Jepsen report including it. We tested MySQL by killing the MySQL process, asking LazyFS to drop uncommitted writes, then restarting the process.

3. Results

3.1. G2-item at Repeatable Read

Adya's Repeatable Read (PL-2.99) prohibits G2-item: a cycle of write-write, write-read, and read-write dependency edges, where those edges do not involve predicates. However, MySQL's Repeatable Read routinely allows G2-item, even on a single healthy node. Kleppmann reported this behavior in 2014 and it still occurs today. Take for example this list-append test, which exhibited 214 cycles in just 40 seconds. Here is one of those cycles comprising two transactions, neither of which saw each other's effects.



In this diagram the top transaction read key 141 and saw the value `[1 2]`, then appended `1` to key 140. The bottom transaction appended `3` to key 141, read key 141 and observed the value `[1 2 3]`, then read key 140 and found it did not exist. The top transaction must have executed before the bottom transaction, since it failed to observe the bottom transaction's append of `3`. But the bottom transaction must have executed before the top transaction, since it read key 140 before any append! This cycle involves purely reads and updates by primary key, and is therefore G2-item.[10]

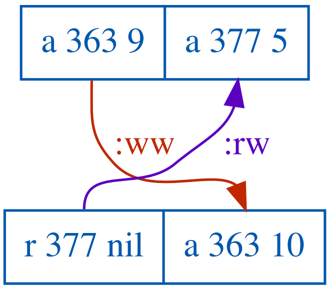
Transactions which fail to see each other's effects could violate important invariants. Consider two independent electricians, each adding a new 20 amp circuit to a breaker panel. Each might visit the site to check[11] that the total load on each circuit (including the one they intend to add) would not exceed the 100 amp capacity of the panel, then return a few days later to add the circuit. Under MySQL's Repeatable Read, both could see a load of 70 amps, add a 20 amp circuit, and create a total load of 110 amps—exceeding the safe load of the panel.[12]

While this behavior is prohibited by PL-2.99 Repeatable Read, it could be interpreted as legal under ANSI SQL Repeatable Read. The standard's definition of P2 (non-Repeatable Read) only discusses a transaction which reads the same row twice and observes some other transaction's effects. Since these transactions never read a row twice, they do not exhibit P2! This is one of many ways in which the standard fails to capture anomalous behavior.

3.2. G-single at Repeatable Read

The example of G2-item we presented above involved a pair of transactions linked by adjacent read-write edges: in short, neither observed the other's effects. However, MySQL Repeatable Read also exhibits G-single (a.k.a. read skew): cycles composed of write-write, write-read, and read-write edges, but where read-write edges are never adjacent to one another. Kleppmann reported this behavior in 2014, and we can confirm it still occurs in MySQL 8.0.34. Like G2-item, G-single cycles involving only item dependencies are prohibited under PL-2.99 Repeatable Read.

Take, for example, this sixty-second append test of a single MySQL node without any faults. At roughly 140 transactions per second it exhibited 244 instances of G-single (plus 305 more instances of G2-item). Since the append test uses no predicate operations, all of these are violations of Repeatable Read. Here is one of those cycles:

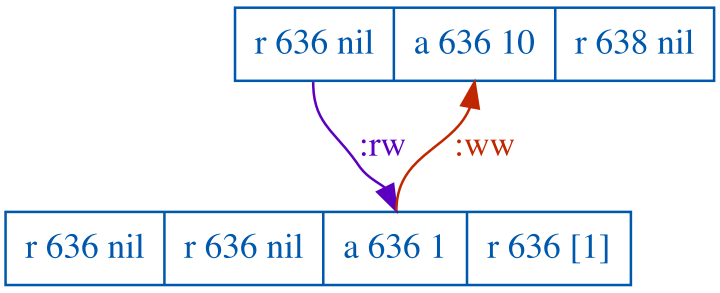


The top transaction here appended `9` to key 363, then `5` to key 377. The bottom transaction failed to observe the append to 377, but also managed to append `10` to key 363 after the top transaction. We know this because a later read observed key 363's value as `[5 6 4 7 8 9 10]`. This violates both Repeatable Read (which rules out any cycle of item edges) and Snapshot Isolation (which rules out G-single in general).

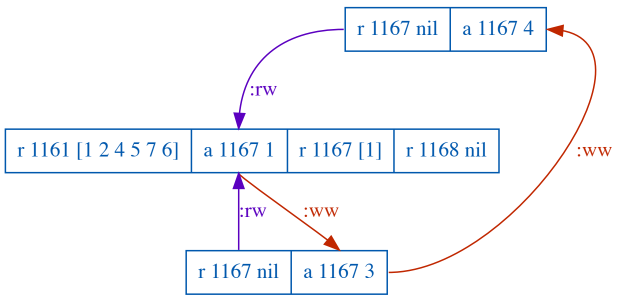
In short: one transaction can both fail to observe but also overwrite another. More complex cycles involving write-read edges also occur. In this case the dependency edges involved different keys, which suggests an interesting question: what would happen if two transactions conflicted on a *single* key?

3.3. Lost Update at Repeatable Read

Phenomenon P4 (lost update) is a special case of G-single in which exactly two transactions are linked by a write-write and read-write cycle on a single key. In other words: two transactions read the same version of some key, and both go on to update it. This is expressly prohibited by Snapshot Isolation and PL-2.99 Repeatable Read. However, Kleppmann showed in 2014 that MySQL Repeatable Read allowed lost update, and we can confirm that it still occurs routinely, even on a single node without faults. Here is a second cycle from the same test run:



Both of these transactions read key 636, found it missing, and went on to write what they thought would be the first element. This is an obvious instance of lost update: at most one of these transactions should have been able to commit.[13] We also have less obvious examples:



This cluster involves two G-single cycles. The smaller, comprising just the bottom two transactions, has no read of key 1167 before the middle transaction's write: it is not a classic instance of lost update. However, its read of key 1167 = [1] implies that the state of that key *prior* to its append of 1 must have been empty, which looks "lost-update-esque." Moreover, the top transaction *also* read the unborn version of key 1167 before appending 4 to it. That, together with the bottom transaction, must be lost update.

A later read `[:r 1167 [1 3 4]]` suggests the following sequence of events. All three transactions must have started before 1167 existed. The middle transaction appended `1` and read `[1]` back. Then the bottom transaction appended `3`, and finally the top transaction appended `4`. All three transactions eventually committed.

These instances of lost update were caught by Elle's cycle detection system, since they were involved in G-single. However, Elle's cycle detection relies on inferring the order of writes to a given key, which we can (mostly) do only if some read observes them. We have recently extended Elle to detect instances of lost update which are invisible to the cycle detector. In these tests, we search for two or more committed transactions which read the same version of some key *k*, then all write *k*. Regardless of whether we see their effects or not, the mere fact that both committed implies lost update. For example:

```
{:key 892,
 :value nil,
 :txns
 [{:process 6,
   :type :ok,
   :f :txn,
   :value [[:r 892 nil]
           [:r 891 nil]
           [:append 892 1]
           [:r 892 [2 5 4 1]]],
   :index 14806,
   :time 49518094450}
  {:process 18,
   :type :ok,
   :f :txn,
   :value [[:r 892 nil]
           [:append 892 8]
           [:r 891 [2 3]]
           [:append 891 9]],
   :index 14842,
   :time 49636093552}]}
```

Both of these committed transactions read the unborn (`nil`) version of key 892 and wrote to it. Out of 9,048 successful transactions in this test, our new checker found 446 distinct transactions involved in 198 instances of lost update. Only 47 of those instances appeared in some cycle.

In short: MySQL Repeatable Read transactions cannot safely read a value and then write it. The standard ORM pattern where a program starts a transaction, loads an object into memory, manipulates it, saves it back to the database, then commits, may find that MySQL silently discards those committed changes. Although PL-2.99 Repeatable Read is supposed to make this pattern safe, MySQL Repeatable Read does not. MySQL users must instead perform their own explicit locking.

An attentive reader may have noticed the above example is more alarming than first meets the eye. The first transaction read the empty state of key 892, appended a single value, then read a version of key 892 including *three additional values*. Where did those come from?

3.4. Non-Repeatable Read at Repeatable Read

MySQL Repeatable Read exhibits *internal consistency anomalies*: consistency violations whose effects are visible within a single transaction. These occur even on a single healthy MySQL node. In that same test run, 126 of 9,048 committed transactions exhibited internal consistency errors. For example:

```
{:op
 {:process 12,
  :type :ok,
  :f :txn,
  :value [[:r 1185 nil]
          [:append 1185 6]
          [:append 1182 8]
          [:r 1185 [3 4 2 6]]],
  :index 19874,
  :time 65980191472},
 :mop [:r 1185 [3 4 2 6]],
 :expected [6]}
```

This transaction read the unborn (`nil`) state of key 1185, and decided to append `6` to it. It then read key 1185 and observed `[3 4 2 6]`. Three elements appeared out of thin air. Or consider:

```
{:op
 {:process 19,
  :type :ok,
  :f :txn,
  :value [[:append 1099 10]
          [:r 1096 [1 2 3]]
          [:append 1096 7]
          [:r 1096 [1 2 3 4 5 6 7]]],
  :index 18404,
  :time 61061580955},
 :mop [:r 1096 [1 2 3 4 5 6 7]],
 :expected [1 2 3 7]}
```

This transaction read key 1096 and obtained the list `[1 2 3]`. It appended `7`, then read the key again, and found three additional values (`4`, `5`, and `6`) inserted in its place. This is forbidden under PL-2.99 Repeatable Read: there must be a read-write dependency from this transaction to some other, and a write-read (or similar) dependency chain leading back. It is forbidden under ANSI Repeatable Read: the transaction performed two reads of the same object and saw different states resulting from a different transaction! The point of Repeatable Read—both for ANSI and Adya—is that once a transaction observes some value, it can count on that value being stable for the remainder of the transaction. MySQL does the opposite: a write is an invitation for another transaction to sneak in and clobber the state you just read.

This behavior allows incredible transactions like the following, recorded during a repeatable-read workload:

```sql
set transaction isolation level Repeatable Read;

start transaction;
select name from people where id = 0;
  --> "pebble"
update people set gender = "femme" where id = 0;
select name from people where id = 0;
```

```
  --> "moss"
commit;
```

This transaction read a person’s name, set their gender, and read their name again. Despite executing at Repeatable Read, their name spontaneously changed from “pebble” to “moss”.

Violations of internal consistency are forbidden under Read Atomic, Causal Consistency, Parallel Snapshot Isolation, Prefix Consistency, Snapshot Isolation, and Serializability. It also seems clear that this transaction satisfies ANSI SQL’s informal definition of a “non-repeatable read.” It violates MySQL’s isolation levels documentation, which claims that “consistent reads within the same transaction read the snapshot established by the first read.” It contradicts MySQL’s consistent read documentation, which specifically states that InnoDB assigns a timepoint on a transaction’s first read, and the effects of concurrent transactions should not appear in subsequent reads.

If we add other transactions which insert or delete the row, we can observe rows popping into existence in the middle of a Repeatable Read transaction:

```
start transaction;
select name from people where id = 0 --> nil
update people set gender = "butch" where id = 0;
select name from people where id = 0; --> "moss"
commit;
```

However, we have not yet observed a row vanishing due to a concurrent delete. Perhaps this is because the update statement updates *no* rows, leaving the snapshot intact. Whatever the reason, the consistent read documentation’s claim that deletes, inserts, and updates “are treated similarly” appears incorrect: deletes seem to work differently from inserts and updates.

3.5. Non-Monotonic View

Kleppmann’s Hermitage lists MySQL Repeatable Read as monotonic atomic view. Per Bailis et al, Monotonic Atomic View ensures that once a transaction T_2 observes an effect of transaction T_1, T_2 observes *all* effects of T_1. Even if MySQL Repeatable Read fetches a fresh snapshot on each write, it might still provide Monotonic Atomic View if the snapshots are monotone. This is how Postgres read committed works.

This is not the case in MySQL. In healthy single-node deployments, MySQL routinely violates Monotonic Atomic View at Repeatable Read. Recall that our Monotonic Atomic View workload has two rows whose `values` are initially `0`. Writer transactions increment the value of row 0, then row 1: both rows’ values should appear to advance in lockstep. However, the first read transaction from this monotonic-atomic-view test observed:

```
start transaction;
select value from mav where id = 0;     --> 0
update mav set noop = 73 where id = 1;
select value from mav where id = 1;     --> 1
select value from mav where id = 0;     --> 0
commit;
```

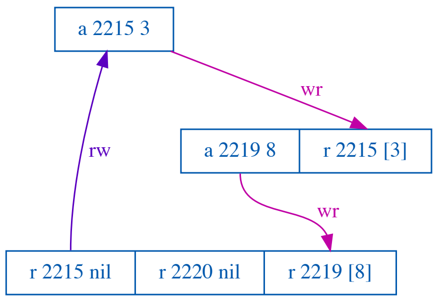
This read transaction saw the state of row 0 prior to the first write transaction. Then it saw the writer's increment of row 1. Under monotonic atomic view, it should have gone on to observe all of the writer's effects—including the increment of row 0. However, when it selected row 0 it saw the old value, not the new one. This is a non-monotonic read!

MySQL's consistent read documentation talks about snapshots extensively, but this behavior doesn't look like a snapshot at all. Snapshot systems usually provide a consistent, point-in-time view of the database state. They are usually atomic: either all of a transaction's effects are included, or none are. Even if MySQL had somehow obtained a non-atomic snapshot from the *middle* of the write transaction, it must have seen the increment of row 0 before the increment of row 1. This is not the case: this read transaction observed the increment of row 1 but *not* row 0. In what sense can this possibly be considered a snapshot?

3.6. Fractured Read-Like Anomalies with RDS Serializable

A common strategy for improving both the availability and throughput of a production MySQL database is to deploy one or more *read replicas*. These replicas continually apply binlogs that are shipped to them by the read/write primary instance, accept connections, and permit read-only transactions to run. Some cloud vendors (e.g. Amazon RDS) configure one or two read replicas as a part of their default production deployment profile.

We found that AWS RDS MySQL routinely violated Serializability at Serializable isolation, even in healthy clusters. Consider this append test which ran on an RDS MySQL cluster with the default recommended production profile. It exhibited several G2-item and G-single anomalies, like the following:



The top transaction appended 3 to key 2215, and that write was visible to the middle transaction. The middle transaction appended `8` to key 2219, which was visible to the bottom transaction. However, the bottom transaction missed the top transaction's write! All G-single anomalies we found involved at least three transactions linked by at least two write-read edges.

Exactly what kind of anomaly is this, and how severe is it? It is clearly an instance of G-single, since it has exactly one read-write edge. It is also G2-item, since it does not involve predicates. This implies RDS MySQL's "Serializable" isolation violates Snapshot Isolation, Repeatable Read, and Serializability.

However, G-single is a broad class of anomalies, and this appears unlike the other instances of G-single we've discussed so far. It is not lost update: no transaction reads then writes the same key. Unlike our previous example of G-single which involved a write-write edge, this anomaly has only write-read and read-write edges. It somewhat resembles fractured read, in which a transaction reads only a subset of another transaction's writes. However, this anomaly involves a reader T_3 which observes a writer T_2's effects, but does not observe an earlier T_1 which was visible to T_2. It is in some sense a "transitive" fractured read.

Regarding severity, we observe first that *any* instance of G-single, when running all transactions at the Serializable isolation level, is significant. The received wisdom in the MySQL community is to avoid using Serializable unless absolutely necessary. The MySQL manual discourages users from using Serializable at all, stating:

> `SERIALIZABLE` enforces even stricter rules than `REPEATABLE READ`, and is used mainly in specialized situations, such as with XA transactions and for troubleshooting issues with concurrency and deadlocks.

Given this guidance, we would expect users to run transactions at this strongest isolation level only when they know they need a high degree of safety, and are willing to pay the performance cost of extra synchronization in order to rule out anomalies. Graver still, fractured read-like anomalies (as instances of G2-item) are forbidden by Repeatable Read. They should occur only at Read Committed and below. That they arise at "Serializable" is troubling.

We suspect this behavior is due in part to RDS's choice of default parameters for production clusters. Among the large variety of configuration parameters that govern the behavior of read replicas is one whose very name should make us uneasy: `replica_preserve_commit_order`.[14]

> For multithreaded replicas (replicas on which `replica_parallel_workers` is set to a value greater than 0), setting `replica_preserve_commit_order=ON` ensures that transactions are executed and committed on the replica in the same order as they appear in the replica's relay log. This prevents gaps in the sequence of transactions that have been executed from the replica's relay log, and preserves the same transaction

> history on the replica as on the source (with the limitations listed below).

Serializable systems are supposed to guarantee transactions execute in (what appears to be) a total order. Failing to preserve that order on a replica seems like it would be a bad thing, even if it permitted more parallelism in applying the log entries. The documentation goes on to say that while this parameter used to be disabled by default, MySQL version 8.0.27 and higher default to `replica_preserve_commit_order=ON`. However, RDS's default parameters still choose `replica_preserve_commit_order=OFF`. If we apply this setting to our local test clusters, we observe similar instances of G-single and G2-item.

№	Summary	Event Required	Fixed in
1	G2-item at Repeatable Read	None	Unresolved
2	G-single at Repeatable Read	None	Unresolved
3	Lost update at Repeatable Read	None	Unresolved
4	Non-Repeatable read at Repeatable Read	None	Unresolved
5	Non-monotonic view at Repeatable Read	None	Unresolved
6	Fractured read-like anomalies at Serializable (in RDS)	None	Unresolved

4. Discussion

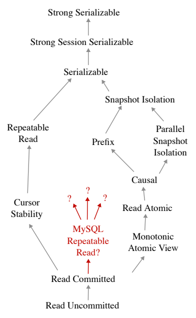
First, the good news. In our testing, MySQL 8.0.34's Read Uncommitted, read committed, and Serializable isolation levels appeared to satisfy PL-1 read uncommitted, PL-2 Read Committed, and PL-3 Serializable, respectively. This held both for single nodes and small clusters with read-only replicas using binlog replication, and through process pauses, crashes, and network partitions.

Our LazyFS fault injection scheme did not discover problems with MySQL's default settings. With `innodb_flush_log_at_trx_commit` at the default setting of `1`, process crashes followed by the loss of un-fsynced data did not result in the loss of committed transactions. When we adjusted that setting to `0`, MySQL fsynced only once every *n* seconds and we observe data loss.

The bad news: MySQL's "Repeatable Read" does not satisfy PL-2.99 Repeatable Read: it exhibits G2-item anomalies including write skew. It does not satisfy Snapshot Isolation: it exhibits G-single, including read skew and lost update. Lost update rules out cursor stability. Reads in MySQL "Repeatable Read" are not repeatable, even under the ambiguous definitions of the ANSI SQL standard. Its transactions violate internal consistency, which rules out Read Atomic, Causal, Consistent View, Prefix, and Parallel snapshot isolation. Kleppmann's 2014 Hermitage suggested MySQL Repeatable Read might be Monotonic Atomic View, but this cannot be true: we found monotonicity violations.

Some authors characterize MySQL Repeatable Read as Snapshot Isolation. For example, Kleppmann's *Designing Data-Intensive Applications* says "PostgreSQL and MySQL call their Snapshot Isolation level Repeatable Read".[15] Formalizations of Snapshot Isolation vary, but most make it appear as if all of a transaction's reads occurred at the transaction start time (plus local changes). This is not true in MySQL: when a write occurs, multiple reads of a key may reveal *newer* versions of that key resulting from other transactions' writes. Moreover, Snapshot Isolated systems generally appear as if all of a transaction's writes occur atomically. MySQL allows a transaction to read some, but not all, of another transaction's writes. This is inconsistent with every version of Snapshot Isolation we are familiar with.

It isn't clear what MySQL Repeatable Read actually *is*. It allows histories which violate Monotonic Atomic View and cursor stability; we know it cannot be equal to or stronger than those models. We have not observed G0 (dirty writes), G1a (aborted reads), G1b (intermediate reads), or G1c (cyclic infomation flow); it appears at least as strong as Read Committed. The repeatability of *some* reads means it is actually stronger than Read Committed.



In this graph of consistency models an arrow from A to B means that B is strictly stronger than A. By this we mean the histories permitted by B are a strict subset of those permitted by A: a system which provides B provides A as well. It seems likely that MySQL Repeatable Read is incomparable to Monotonic Atomic View: it allows violations of Monotonic Atomic View, but also rules out some non-repeatable reads that Monotonic Atomic View allows. Likewise, it is incomparable to Repeatable Read: MySQL Repeatable Read appears to prohibit certain phantoms which are legal under both ANSI and PL-2.99 Repeatable Read. However, we are unsure exactly which other models are strictly stronger than MySQL Repeatable Read. Is every prefix-consistent history legal under MySQL Repeatable Read? Or are they too incomparable? Because MySQL Repeatable Read's behavior is so unusual, and because we lack a formal definition of its properties, we are unsure where to draw additional arrows in the diagram above.

As always, we caution 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 correctness.

4.1. Does the MySQL Community Know?

The behavior of MySQL Repeatable Read appears poorly understood in the MySQL community. Several authors believe Repeatable Read should prevent lost update. However, several others acknowledge it actually does not, and advise (for example) explicit locking tactics. Similarly, many internet sources state (incorrectly) that MySQL repeatable reads are repeatable. This is understandable: MySQL and MariaDB's own documentation makes this claim. Those claims are contradicted by a single sentence in a note buried in the MySQL consistent reads documentation. Other blog posts and articles acknowledge (some indirectly) that MySQL Repeatable Read actually allows non-repeatable reads.

Cabral and Murphy's 2009 *MySQL Administrator's Bible* states that MySQL "supports the four standard isolation levels," and emphasizes at length that Repeatable Read prevents a transaction from observing another transaction's concurrent writes:

> Using the `REPEATABLE READ` isolation level, all reads within a transaction show the same data values, even if a second transaction has committed a data change while the first transaction was still running. If a transaction starts, reads a row, waits 60 seconds, and reads the same row again, both data reads will be the same—even if in those 60 seconds another transaction has changed and committed data. The first transaction has the same data when it repeats the read.…
>
> `REPEATABLE READ` may not seem like a good idea—after all, if the data changes, shouldn't a transaction be aware of that? The problem is that a transaction may take different actions based on the values of the data. Data values changing within a transaction may lead to unexpected consequences.

Cabral and Murphy repeat that Repeatable Read "allows a transaction to see the same data for values it has already read regardless of whether or not the data has been changed." In their section on multi-version concurrency control, they emphasize the independence of transaction snapshots:

> If a second transaction starts, it "checks out" its own copy of the data. If the first transaction makes changes and commits, the second transaction will not see the data. The second transaction can only work with the data it has. There is no way to update the data that the second transaction sees, though the second transaction could issue a ROLLBACK and start the transaction again to see the new data.

This is also wrong: writing a row modifies the transaction's local copy of the data.

Grippa & Kuzmichev's 2021 *Learning MySQL* states that MySQL supports all of the SQL:1992 standard isolation levels. They too claim:

> With the `REPEATABLE READ` isolation level, there are thus no dirty reads and or non-repeatable reads. Each transaction reads the snapshot established by the first read.

However, the section on Serializable isolation actually demonstrates (perhaps inadvertently) that MySQL's Repeatable Read allows both lost update, a change in read snapshot, and a resulting internal consistency violation! It then shows that `Serializable` prevents those anomalies. It doesn't name the anomalies, instead opting to say that "this doesn't make sense", but the behavior is visible to a careful reader. It's not clear if the authors realize the example contradicts their earlier claims about non-repeatable reads and snapshot integrity.

4.2. Recommendations

The core problem is that MySQL claims to implement Repeatable Read but actually provides something much weaker. We see two avenues to resolve this problem.

The first is to keep MySQL's behavior as it is, and to clearly document the consistency model "Repeatable Read" actually provides. There is precedent in other databases: PostgreSQL's Repeatable Read is actually Snapshot Isolation, and exhibits behaviors which violate PL-2.99 Repeatable Read. However, PostgreSQL's documentation eventually mentions that their Repeatable Read implementation is actually Snapshot Isolation. MySQL could similarly document that their "Repeatable Read" means "Read Committed, plus some sort of guarantees that hold until the transaction writes something, at which point mysteries occur." A precise characterization of those mysteries would be most welcome.

The second option is to treat these behaviors as bugs and fix them. Jepsen would be delighted if MySQL and other vendors were to commit to providing PL-2.99 Repeatable Read. However, even satisfying the incomplete, ambiguous ANSI definition of Repeatable Read would be an improvement over current affairs.

In the meantime, MySQL users who require PL-2.99 or ANSI Repeatable Read should be cautious of MySQL Repeatable Read. Reads may not be repeatable, or even reflect a snapshot of committed state. The common ORM pattern in which a transaction reads an object into memory, modifies it, then writes it back within a transaction, may cause committed updates to be silently lost. Users requiring Repeatable Read semantics should use MySQL's Serializable isolation instead. Alternatively, they can selectively strengthen reads performed at `READ COMMITTED` using locking techniques like `SELECT ... FOR UPDATE`.

4.3. RDS

AWS RDS MySQL cluster exhibits read skew and G2-item at its "Serializable" isolation level. Users who rely on Serializability should set `slave_preserve_commit_or-`

`der` to `ON` in their RDS parameter groups. Jepsen suggests that AWS either change the default, or clearly explain the allowed violations of Serializability in the known limitations documentation for RDS MySQL.

4.4. Future Work

MySQL's binlog replication appears fragile. We observed a number of mysterious scenarios in which replication halted in our local Jepsen tests. We also found that a few minutes of testing could completely break AWS RDS's MySQL replication: even a simple `CREATE DATABASE` would succeed on the primary and fail to appear on the secondaries. We waited an hour without observing recovery. MySQL's default settings are known to be unsafe in replicated systems. We made no attempt to promote nodes from secondaries to primaries, or to explore exciting topologies like ring or star replication. Future work might explore these behaviors.

We have begun research into more general-purpose predicate tests, but this work is still early. Once ready, we'd like to evaluate MySQL predicate safety and see if it differs from primary-key operations.

4.5. A Plea to Standards Bodies

Twenty-eight years after Berenson et al. demonstrated that ANSI SQL's isolation levels are ambiguous and incomplete, seven revisions of the ANSI & ISO standards have left its definitions unchanged.[16] P0 is still legal at every level up to Repeatable Read. We still don't know whether circular information flow is legal at Read Committed. P3 still doesn't mention deletes. Internal behavior remains unspecified. The research community has moved on to new formalisms. Many are based on Adya's 1999 thesis, which struggled to capture "what the SQL standard actually meant."

If you happen to sit on the ISO/IEC JTC1/SC 32 Data Management and Interchange committee, please imagine the soft chords of a heart-tugging piano lament have begun to play. A montage of transactional anomalies appears on your screen. Internal anomalies. Lost updates. Dirty writes. Jepsen is looking into the camera, holding a database.

> Hi. This is Jepsen. Will you be an angel for a helpless database? Every day major databases exhibit anomalous behavior which ISO/IEC 9075-2 fails to characterize. For just a few pages of formalism, you can give vendors and users a clear, meaningful, and portable definition of isolation levels.
>
> It's been almost three decades. Act now.

Jepsen wishes to thank INESC TEC and in particular João Azevedo, Ricardo Macedo, João Tiago Paulo, José Pereira, and Maria Ramos, for building LazyFS. Our thanks also to Justin Conklin, who contributed ASM advice and code for a significant perfor-

mance improvement in Jepsen's underlying analysis library. Irene Kannyo provided invaluable editorial support. This work was performed independently by Jepsen without compensation, in accordance with the Jepsen ethics policy.

---

1. Ever sticklers for precision, ANSI reminds readers that while ANSI approves, copyrights, and publishes standards, "there are no ANSI standards, only standards developed by ANSI-approved committees, many operating in accordance with the ANSI Essential Requirements (American National Standards)." In this work, "the ANSI SQL standard" refers to ANSI X3.135, also known as ISO/IEC 9075. ANSI X3.135 (not an ANSI standard) was originally produced by the Accredited Standards Committee (ASC)'s ANSI Database Technical Committee (ANSI X3H2). ISO 9075, a technically identical standard, was published a few months later. Today ISO/IEC 9075 is developed by the ISO/IEC Joint Technical Committee (JTC) 1 for Information Technology, and INCITS (an ANSI-accredited standards developing organization and the successor to ANSI X3), adopts it for use as an American National Standard.↩

2. The relevant portion of the SQL standard costs over two hundred dollars, making it inaccessible to casual readers. Precise phrasing is critical for this work, so we reproduce its definitions verbatim.↩

3. Yes, that Jim Gray, of System R fame!↩

4. Readers looking for a shorter version should try Adya's ICDE paper with Barbara Liskov and Patrick O'Neil.↩

5. We omit a detailed explanation of the non-cyclic anomalies, as well as a discussion of aborted and committed transactions, internal and external reads/writes, version orders, etc., for brevity.↩

6. Vendors rarely come out and *say* they intend to provide the Adya levels, but in practice their implementations either prevent (e.g.) G0 at Read Committed, or, when informed of the behavior, correct it. There are exceptions. For instance, RedPanda considers G0 legal at "Read Committed," and Oracle's "Serializable" is actually Snapshot Isolation. Many databases provide higher isolation than is required under Adya's formalism. For instance, PostgreSQL appears to provide Monotonic Atomic View at read committed.↩

7. Because write skew can occur in Snapshot Isolation but not PL-2.99, and phantoms can occur in PL-2.99 but some are prevented by Snapshot Isolation, neither is strictly stronger than the other. For more details, see Berenson et al., which explains that while A3 (the strict interpretation of phantoms) is prohibited by Snapshot Isolation, it sometimes permits P3 (the broad interpretation).↩

8. There has been some confusion about whether Elle's list-append workload requires a custom datatype not supported by SQL databases. We are happy to share that the SQL standard has included both string and array concatenation since 1999, and that these datatypes and operators are broadly supported by vendors. Elle also includes a workload for plain read-write registers.↩

9. Some authors have claimed some or all of Elle's checkers are unsound. To the best of our knowledge Elle is sound: every anomaly it finds is "real." As our paper discussed, Elle is not complete: it may fail to detect some anomalies. This work makes Elle more complete.↩

10. Write skew (A5B) as presented by Berenson et al requires both transactions read before writing. This particular anomaly is write-skew-esque: the two transactions have overlapping read sets and disjoint write sets causing each to fail to observe the other's effects. However, one transaction happened to read after writing, rather than before.↩

11. Breaker panels have a fixed number of slots, some of which may be left free. Adding a circuit involves installing a circuit breaker in a free slot. In this model of G2-item, the electricians check each slot on the breaker panel by performing a series of primary-key reads. If they used a predicate read, this would be G2.↩

12. This scenario happened to one of your authors. Thankfully the panel limit was not exceeded.↩

13. Some readers might contend that while this meets the formal definition of "lost update," the updates

themselves aren't so much "lost" as simply "stacked on top of each other in potential violation of constraints." Consider, however, that if these update operations were blind writes instead of list appends, the resulting state would be indistinguishable from a history in which one of the writes simply never happened. This is the reason P4 is called "lost update."↩

14. The RDS parameter group for configuring MySQL uses an older name for this setting: `slave_preserve_commit_order`. MySQL is in the process of adopting more inclusive language, and Jepsen uses the newer "replica" term wherever possible.↩
15. Kleppman goes on to note in a later section that MySQL/InnoDB fails to prevent lost update, and that this fails to meet "some authors" definitions of Snapshot Isolation.↩
16. From the late 1980s through 1995 NIST performed conformance testing to evaluate whether databases correctly implemented the SQL standard. One wonders how they would evaluate transaction safety today.↩