

[Live Webinar] Gain a Competitive Edge with Graph Analytics

[Sign Up Now](#)

DZone > Java Zone > Transaction Management with Spring and MySQL

Transaction Management with Spring and MySQL

by Ali Saker Ali · Apr. 09, 20 · Java Zone · Tutorial

MariaDB Platform: Access to all historical data NOW – no waiting for BI and data scier
Interactive, real-time, on-demand analytics. [Try It Today](#)

Presented by MariaDB

A transaction is a unit of program execution that accesses and possibly updates various data items. It contains multiple steps that must appear to a user as a single, indivisible unit which either executes in its entirety or not at all. We are going to discuss different aspects of transaction management in the MySQL InnoDB environment and how Spring implements those aspects using proxies.

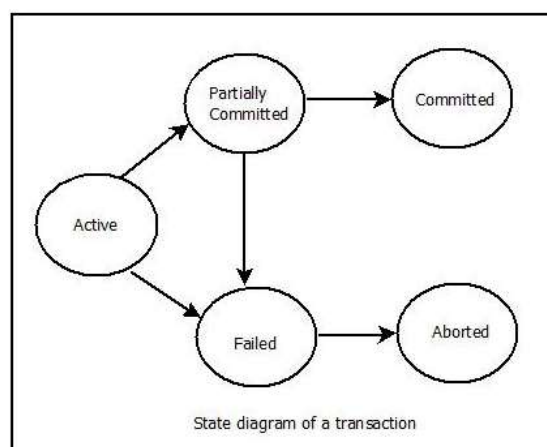
Table of Contents

- ACID
- Serializability
- Isolation levels
- How MySQL implements each level of isolation
- Examples with Spring
- References

ACID

Database system maintains the following properties of the transactions:

1. **Atomicity:** Either all the operations of the transaction are reflected properly in the database or none are. MySQL ensures atomicity using undo logs that store information about the updates that are made by each transaction.
2. **Consistency:** If the database is consistent before the execution of the transaction, the database remains consistent after the execution of the transaction. There are two types of consistency. The first is ensured by the database system; for example, the data integrity constraints while the other is the responsibility of the programmer who codes the transaction to ensure the application-dependent consistency constraints.
3. **Durability:** To understand durability we first must understand the state diagram of a transaction.



In the above diagram, we notice the “Partially Committed” state. The transaction enters this state after the execution of the last statement. In fact, all those updates are made on memory so any system failure (e.g. power outage) can result in a loss of

changes corresponding to this transaction. Durability means that after a transaction commits, the updates should be saved despite any possible system failure, and for that reason, information about the updates should be saved on stable storage (MySQL uses redo logs). After that, the transaction moves from “Partially Committed” state to the “Committed” state so that database system can reconstruct updates whenever they are needed.

4. Isolation: The basic idea behind isolation that each transaction should be unaware of other transactions running at the same time. Let T1, T2 be two transactions running at the same time

T1	T2
READ(A)	
A = A - 50	
WRITE(A)	
	READ(A)
	A = A * 10
FAIL AND ROLLBACK	WRITE(A)
	COMMIT

This table is called *S schedule* which represents the chronological order in which instructions are executed in the system. Clearly a schedule for a set of transactions must consist of all instructions of those transactions. The database system can generate different schedules by forcing the transactions to wait at any single point of processing.

You will notice that T2 read a value that is never committed and this will leave the database inconsistent.

The best solution for this problem is to run transactions serially (one after the other). We will talk about this solution in the next section, “Serializability” but this solution has negative performance effects where transactions should wait for each other so that, many solutions have been developed to compromise between performance and consistency, and we will talk about those solutions in the “**Isolation levels**” section.

Serializability

As we see in table above, transactions running at the same time may leave the database inconsistent, but there is a type of schedule, called a Serial Schedule that doesn’t affect the database consistency:

A Serial Schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.

T1	T2
	READ(A)
	A = A - 10
	WRITE(A)
	COMMIT
READ(A)	
COMMIT	

Many database systems can’t generate such a schedule to execute concurrently transactions. Imagine that T1 and T2 started at the same time and the database system chose T2 then T1. In this case, T1 will wait for T2 to be committed which may remain active for hours!

active for hours:

A more efficient type of schedule that also preserves the consistency and isolation properties is the Serializable Schedule.

To understand serializable schedules we should first discuss the situations where two concurrent transactions conflict:

Please note that the examples used below are part of a schedule so it isn't a complete one and each example may contains multiple additional instructions and each transaction may have rolls back or commits

1- Read/Read situation:

T1	T2
READ(A)	
	READ(A)

T2 reads record A after T1 but in fact, order here does not matter since the same value of A is read by T1 and T2, regardless of the order, so there is no conflict here and we can swap between the two operations and obtains a new schedule that is equivalent to the original one.

2. Read/Write situation:

T1	T2
READ(A)	READ(A)
	WRITE(A)
WRITE(B)	

If T1 reads the value of A before it is written by T2 (the situation above) we will get a different value from the situation where T1 reads the value of A after it is written by T2. There is a conflict here and we can't swap between the two operations.

3. Write/Read situation:

Same (Read/Write) situation and there is a conflict here and we can't swap between the two operations.

4. Write/Write situation (A.K.A Dirty Write):

T1	T2
READ(A)	READ(A)
	WRITE(A)
WRITE(A)	

The value of A will be different if it is written finally by T1 or T2 and if the schedule above contains an additional READ(A)

operation it will be affected by the order so there is a conflict here and we can't swap between the two WRITE/WRITE operations.

We say that two transactions T1 and T2 **conflict** if they contain two operations (one for each transaction) on the **same data item**, and at least one of these instructions is a **write** operation.

Thus, only in the case of (read/read) does the relative order of their execution not matter and we can swap between the two instructions. By doing such a swap the database system generates (conflict equivalence) schedules to the original one.

Now we can define the Serializable Schedule: as a schedule that is conflict equivalent to a Serial Schedule, and this is the type of schedule that is generated when you set the isolation level to Serializable.

Isolation Levels

As we discussed before, serializability has its own performance issues where any case of conflict will cause one of the transactions to wait for the other to be committed. so many solutions have been developed to compromise between consistency and performance and they are called isolation levels.

Note that dirty write in all the following solutions will cause the conflicting transaction to wait until the other to be committed.

1. Serializable: where the database system generates a serializable schedule to execute the concurrent transactions.
2. Repeatable Read: a transaction eliminates the (read/write) and (write/read) conflicts by ignoring any database update made by another concurrent transaction. So the only conflict that forces a transaction to wait is the dirty write.
3. Read Committed: a transaction eliminates the (read/write) and (write/read) conflicts by reading only the updates that are made by a committed concurrent transaction and ignoring updates that are made by uncommitted transactions. So the only conflict that forces a transaction to wait is the dirty write.
4. Read Uncommitted: a transaction eliminates the (read/write) and (write/read) conflicts by reading the updates that are made by a committed/uncommitted concurrent transaction. So the only conflict that forces a transaction to wait is the dirty write.

You can notice how the last three solutions affect the consistency, especially the Read Uncommitted solution, where the transaction can read uncommitted data and depends on it to do other operations. It is the responsibility of the application developers to ensure consistency when they choose one of these three solutions.

How MySQL Implements Each Level of Isolation

To understand how MySQL implements each level of isolation we should discuss some definitions:

- Shared Lock: Permits the transaction that holds the lock to read a row. Multiple active transactions may have a shared lock on the same row.
 - Exclusive Lock: Permits the transaction that holds the lock to update or delete a row. A transaction can exclusively lock a row only if it isn't (shared or exclusive) locked by another transaction.
 - Snapshot: A representation of data at a particular time, which remains the same even as changes are committed by other transactions. Used by certain isolation levels to allow consistent reads.
 - Consistent Nonlocking Reads: A consistent read means that MySQL InnoDB uses multi-versioning to present to a query a snapshot of the database at a point in time (it depends on the database and the information stored in the undo log to build that snapshot). The query sees the changes made by transactions that committed before that point of time, and no changes made by later or uncommitted transactions. The exception to this rule is that the query sees the changes made by earlier statements within the same transaction.
 - Locking Reads: Select records with a shared lock or an exclusive lock.
 - Dirty Reads: An operation that retrieves unreliable data, data that was updated by another transaction but not yet *committed*.
1. Serializable: MySQL uses Locking Reads with this type of isolation to ensure that the generated schedule is a serializable one. So if transaction T1 reads a record A another one T2 can't update A until T1 commits.

2. Repeatable Read: MySQL uses Consistent Nonlocking Reads with this type of isolation, where the transaction obtains only one snapshot with the first consistent read operation that didn't change until the transaction commits. So it ignores any database update made by other transactions.
3. Read Committed: MySQL uses Consistent Nonlocking Reads with this type of isolation, but the difference from Repeatable Read level is that each consistent read within a transaction sets and reads its own fresh snapshot. So it reads the updates made by committed concurrent transactions.
4. Read Uncommitted: MySQL uses Dirty Reads with this type of isolation depending on the database and changes on the memory log buffer.

Examples with Spring

We will depend on the following entity:

Java

```
1 @Entity
2 public class TestTransaction implements Serializable {    @Id
3     @GeneratedValue(strategy = GenerationType.IDENTITY)
4     private Long id;    @Column(length = 50)
5     private String name;    public Long getId() {
6         return id;
7     }    public String getName() {
8         return name;
9     }
10
11     public void setName(String name) {
12         this.name = name;
13     }
14 }
```

Serializable Isolation Level:

Java

```
1 @Lazy
2 @Autowired
3 private TestService testService;
4
5 public TestService(TestTransactionRepository repository) {
6     this.repository = repository;
7 }
8
9 @Transactional(isolation = Isolation.SERIALIZABLE)
10 public void T1() {
11     TestTransaction t = repository.findById(1L).get();
12     String name = String.format("Name - %s", new Date().getTime());
13     t.setName(name);
14     // we used saveAndFlush to flush the mysql instructions
15     // to the database system so the row get locked
16     repository.saveAndFlush(t);
17
18     testService.T2();
19 }
20
21 // We use propagation = REQUIRES_NEW to start a new transaction T2
22 // different from T1
23 @Transactional(propagation = Propagation.REQUIRES_NEW,
24 isolation = Isolation.SERIALIZABLE)
25 public void T2() {
26     // Transaction T2 will stuck here
27     // because it needs a shared lock on the same row that is    already exclusively locked by T1
28     TestTransaction t = repository.findById(1L).get();
29 }
```

Repeatable Read:

Java

```
1 @PersistenceContext
2 private EntityManager entityManager;
```

```

3
4 @Lazy
5 @Autowired
6 private TestService testService;
7
8 public TestService(TestTransactionRepository repository) {
9     this.repository = repository;
10 }
11
12 @Transactional(isolation = Isolation.REPEATABLE_READ)
13 public void T1() {
14     TestTransaction t = repository.findById(1L).get();
15     System.out.print(t.getName()); // output: X
16     testService.T2(); // updates the record then commits
17
18     // we used the entity manager proxy provided by spring
19     // So that it will manage the persistence context related
20     // to the current active transaction (T1)
21     entityManager.detach(t);
22
23     // fetch the record another time
24     t = repository.findById(1L).get();
25
26     System.out.print(t.getName()); // output: X/* Output: X
27 * Why? T1 obtains only one snapshot
28 * with the first consistent read operation
29 * that didn't change until the transaction commits.
30 * So it ignores any database update made by other transactions.
31 */}
32
33 @Transactional(propagation = Propagation.REQUIRES_NEW,
34 isolation = Isolation.READ_COMMITTED)
35 public void T2() {
36     TestTransaction t = repository.findById(1L).get();
37     t.setName("Y");
38 }

```

Read Committed:

Java

```

1 @PersistenceContext
2 private EntityManager entityManager;
3
4 @Lazy
5 @Autowired
6 private TestService testService;
7
8 public TestService(TestTransactionRepository repository) {
9     this.repository = repository;
10 }
11
12 @Transactional(isolation = Isolation.READ_COMMITTED)
13 public void T1() {
14     TestTransaction t = repository.findById(1L).get();
15     System.out.print(t.getName()); // output: X
16     testService.T2(); // updates the record then commits
17
18     // we used the entity manager proxy provided by spring
19     // So that it will manage the persistence context related
20     // to the current active transaction (T1)
21     entityManager.detach(t);
22
23     // fetch the record another time
24     t = repository.findById(1L).get();
25
26     System.out.print(t.getName()); // output: Y/* Output: Y
27 * Why? Each consistent read within a transaction T1
28 * sets and reads its own fresh snapshot. So it reads
29 * the updates made by committed concurrent transactions.
30 */}
31
32 @Transactional(propagation = Propagation.REQUIRES_NEW,
33 isolation = Isolation.READ_COMMITTED)
34 public void T2() {
35     TestTransaction t = repository.findById(1L).get();
36     t.setName("Y");
37 }

```

Boost Your Remote Work Environment



LIVE WEBINAR! Are you managing remote developers? Register and let the community can be a vibrant source of collaboration for your developer team.

Like This Article? Read More From DZone



Transaction configuration with JPA and Spring 3.1



Spring Transaction Management Over Multiple Threads




Optimizing Relationships Between Entities in Hibernate



**Free DZone Refcard
Getting Started With Headless CMS**

Topics: DATABASE , JPA 2.0 , SPRING , TRANSACTION , JAVA , MYSQL , ISOLATION

Published at DZone with permission of Ali Saker Ali . [See the original article here.](#) 
Opinions expressed by DZone contributors are their own.
