Pessimistic Locking in JPA

Last modified: March 21, 2020

by baeldung (https://www.baeldung.com/author/baeldung/)

Persistence (https://www.baeldung.com/category/persistence/)

JPA (https://www.baeldung.com/tag/jpa/)

1. Overview

There are plenty of situations when we want to retrieve data from a database. Sometimes we want to lock it for ourselves for further processing so nobody else can interrupt our actions.

We can think of two concurrency control mechanisms which allow us to do that: setting the proper transaction isolation level or setting a lock on data that we need at the moment.

The transaction isolation is defined for database connections. We can configure it to retain the different degree of locking data.

However, **the isolation level is set once the connection is created** and it affects every statement within that connection. Luckily, we can use pessimistic locking which uses database mechanisms for reserving more granular exclusive access to the data.

We can use a pessimistic lock to ensure that no other transactions can modify or delete reserved data.

There are two types of locks we can retain: an exclusive lock and a shared lock. We could read but not write in data when someone else holds a shared lock. In order to modify or delete the reserved data, we need to have an exclusive lock.

We can acquire exclusive locks using '*SELECT … FOR UPDATE*' statements.

2. Lock Modes

JPA specification defines three pessimistic lock modes which we're going to discuss:

- *PESSIMISTIC_READ* – allows us to obtain a shared lock and prevent the data from being updated or deleted
- *PESSIMISTIC_WRITE* – allows us to obtain an exclusive lock and prevent the data from being read, updated or deleted
- *PESSIMISTIC_FORCE_INCREMENT* – works like *PESSIMISTIC_WRITE* and it additionally increments a version attribute of a versioned entity

All of them are static members of the *LockModeType* class and allow transactions to obtain a database lock. They all are retained until the transaction commits or rolls back.

It's worth noticing that we can obtain only one lock at a time. If it's impossible a *PersistenceException* is thrown.

2.1. *PESSIMISTIC_READ*

Whenever we want to just read data and don't encounter dirty reads, we could use *PESSIMISTIC_READ* (shared lock). **We won't be able to make any updates or deletes though.**

It sometimes happens that the database we use doesn't support the *PESSIMISTIC_READ* lock, so it's possible that we obtain the *PESSIMISTIC_WRITE* lock instead.

2.2. *PESSIMISTIC_WRITE*

Any transaction that needs to acquire a lock on data and make changes to it should obtain the *PESSIMISTIC_WRITE* lock. According to the *JPA* specification, holding *PESSIMISTIC_WRITE* lock will prevent other transactions from reading, updating or deleting the data.

Please note that some database systems implement multi-version concurrency control (https://en.wikipedia.org/wiki/Multiversion_concurrency_control) which allows readers to fetch data that has been already blocked.

2.3. *PESSIMISTIC_FORCE_INCREMENT*

This lock works similarly to *PESSIMISTIC_WRITE*, but it was introduced to cooperate with versioned entities – entities which have an attribute annotated with *@Version*.

Any updates of versioned entities could be preceded with obtaining the *PESSIMISTIC_FORCE_INCREMENT* lock. **Acquiring that lock results in updating the version column.**

It's up to a persistence provider to determine whether it supports *PESSIMISTIC_FORCE_INCREMENT* for unversioned entities or not. If it doesn't, it throws the *PersistanceException.*

2.4. Exceptions

It's good to know which exception may occur while working with pessimistic locking. *JPA* specification provides different types of exceptions:

- *PessimisticLockException* – indicates that obtaining a lock or converting a shared to exclusive lock fails and results in a transaction-level rollback
- *LockTimeoutException* – indicates that obtaining a lock or converting a shared lock to exclusive times out and results in a statement-level rollback

- *PersistanceException* – indicates that a persistence problem occurred. *PersistanceException* and its subtypes, except *NoResultException*, *NonUniqueResultException, LockTimeoutException*, and *QueryTimeoutException,* **marks the active transaction to be rolled back.**

3. Using Pessimistic Locks

There are a few possible ways to configure a pessimistic lock on a single record or group of records. Let's see how to do it in JPA.

3.1. Find

It's probably the most straightforward way. It's enough to pass a *LockModeType* object as a parameter to the *find* method:

```
entityManager.find(Student.class, studentId,
LockModeType.PESSIMISTIC_READ);
```

3.2. Query

Additionally, we can use a *Query* object as well and call the *setLockMode* setter with a lock mode as a parameter:

```
Query query = entityManager.createQuery("from Student where studentId =
:studentId");
query.setParameter("studentId", studentId);
query.setLockMode(LockModeType.PESSIMISTIC_WRITE);
query.getResultList()
```

3.3. Explicit Locking

It's also possible to lock manually the results retrieved by the find method:

```
Student resultStudent = entityManager.find(Student.class, studentId);
entityManager.lock(resultStudent, LockModeType.PESSIMISTIC_WRITE);
```

3.4. Refresh

If we want to overwrite the state of the entity by the *refresh* method, we can also set a lock:

```
Student resultStudent = entityManager.find(Student.class, studentId);
entityManager.refresh(resultStudent,
LockModeType.PESSIMISTIC_FORCE_INCREMENT);
```

3.5. NamedQuery

@NamedQuery annotation allows us to set a lock mode as well:

```
@NamedQuery(name="lockStudent",
  query="SELECT s FROM Student s WHERE s.id LIKE :studentId",
  lockMode = PESSIMISTIC_READ)
```

4. Lock Scope

Lock scope parameter defines how to deal with locking relationships of the locked entity. It's possible to obtain a lock just on a single entity defined in a query or additionally block its relationships.

To configure the scope we can use *PessimisticLockScope* enum. It contains two values: *NORMAL* and *EXTENDED*.

We can set the scope by passing a parameter '*javax.persistance.lock.scope*' with *PessimisticLockScope* value as an argument to the proper method of *EntityManager*, *Query*, *TypedQuery* or *NamedQuery*.

```
Map<String, Object> properties = new HashMap<>();
map.put("javax.persistence.lock.scope", PessimisticLockScope.EXTENDED);

entityManager.find(
  Student.class, 1L, LockModeType.PESSIMISTIC_WRITE, properties);
```

4.1. *PessimisticLockScope.NORMAL*

We should know that the *PessimisticLockScope.NORMAL* is the default scope. With this locking scope, we lock the entity itself. When used with joined inheritance it also locks the ancestors.

Let's look at the sample code with two entities:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Person {

    @Id
    private Long id;
    private String name;
    private String lastName;

    // getters and setters
}

@Entity
public class Employee extends Person {

    private BigDecimal salary;

    // getters and setters
}
```

When we want to obtain a lock on the *Employee*, we can observe the *SQL* query which spans over those two entities:

```
SELECT t0.ID, t0.DTYPE, t0.LASTNAME, t0.NAME, t1.ID, t1.SALARY
FROM PERSON t0, EMPLOYEE t1
WHERE ((t0.ID = ?) AND ((t1.ID = t0.ID) AND (t0.DTYPE = ?))) FOR UPDATE
```

4.2. *PessimisticLockScope.EXTENDED*

The *EXTENDED* scope covers the same functionality as *NORMAL*. In addition, **it's able to block related entities in a join table**.

Simply put, it works with entities annotated with *@ElementCollection* or *@OneToOne*, *@OneToMany* etc. with *@JoinTable*.

Let's look at the sample code with the *@ElementCollection* annotation:

```
@Entity
public class Customer {

    @Id
    private Long customerId;
    private String name;
    private String lastName;
    @ElementCollection
    @CollectionTable(name = "customer_address")
    private List<Address> addressList;

    // getters and setters
}

@Embeddable
public class Address {

    private String country;
    private String city;

    // getters and setters
}
```

Let's analyze some queries when searching for the *Customer* entity:

```
SELECT CUSTOMERID, LASTNAME, NAME
FROM CUSTOMER WHERE (CUSTOMERID = ?) FOR UPDATE

SELECT CITY, COUNTRY, Customer_CUSTOMERID
FROM customer_address
WHERE (Customer_CUSTOMERID = ?) FOR UPDATE
```

We can see that there are two '*FOR UPDATE*' queries which lock a row in the customer table as well as a row in the join table.

Another interesting fact we should be aware of is that not all persistence providers support lock scopes.

5. Setting Lock Timeout

Besides setting lock scopes, we can adjust another lock parameter – timeout. **The timeout value is the number of milliseconds that we want to wait for obtaining a lock until the *LockTimeoutException* occurs.**

We can change the value of timeout similarly to lock scopes, by using property '*javax.persistence.lock.timeout*' with the proper number of milliseconds.

It's also possible to specify 'no wait' locking by changing timeout value to zero. However, we should keep in mind that there are database drivers which **don't support setting a timeout value this way.**

```
Map<String, Object> properties = new HashMap<>();
map.put("javax.persistence.lock.timeout", 1000L);

entityManager.find(
  Student.class, 1L, LockModeType.PESSIMISTIC_READ, properties);
```

6. Conclusion

When setting the proper isolation level is not enough to cope with concurrent transactions, JPA gives us pessimistic locking. It enables us to isolate and orchestrate different transactions so they don't access the same resource at the same time.

To achieve that we can choose between discussed types of locks and consequently modify such parameters as their scopes or timeouts.

On the other hand, we should remember that understanding database locks is as important as understanding the mechanisms of underlying database systems. It's also important to have in mind that the behavior of pessimistic locks depends on persistence provider we work with.

Lastly, the source code of this tutorial is available over on GitHub for hibernate (https://github.com/eugenp/tutorials/tree/master/persistence-modules/hibernate-jpa) and for EclipseLink (https://github.com/eugenp/tutorials/tree/master/persistence-modules/spring-data-eclipselink).