Introduction

Locking plays an important role in concurrent applications. The Java Persistence API provides distinct types of locking mechanisms that may be classified in a couple of major groups: pessimistic and optimistic. Pessimistic locking results in database records being locked by some process until the unit of work is completed so no other process is able to interfere with the locked records until the work is not finished. Optimistic locking does not actually lock anything but relies instead in checks made against the database during data persistence events or transaction commit in order to detect if the data has been changed in the meanwhile by some other process.

This article will cover the following JPA lock modes:

1. OPTIMISTIC
2. OPTIMISTIC\_FORCE\_INCREMENT
3. PESSIMISTIC\_READ
4. PESSIMISTIC\_WRITE
5. PESSIMISTIC\_FORCE\_INCREMENT

Entity versioning in JPA by the means of optimistic locking is detailed in the following article: [**JPA entity versioning (@Version and Optimistic Locking)**](http://www.byteslounge.com/tutorials/jpa-entity-versioning-version-and-optimistic-locking). If you are not familiar with entity versioning you should read that article first since it introduces fundamental concepts that will not be detailed in this article.

The following table represents the illustrative dataset used through the article sections:

**DEPARTMENT:**

|  |  |  |
| --- | --- | --- |
| **ID** | **NAME** | **VERSION** |
| 1 | IT | 3 |
| 2 | Finance | 1 |
| 3 | Human Resources | 7 |

We may map the table against the following entity:

**Department entity**

@Entity

@Table(name = "DEPARTMENT")

public class Department {

@Id

@Column(name = "ID")

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

@Column(name = "NAME", nullable = false)

private String name;

@Column(name = "VERSION")

@Version

private int version;

}

Locking entities

Entity locks may be acquired in the following ways:

**Locking in a find operation**

entityManager.find(Department.class, 1, LockModeType.PESSIMISTIC\_READ);

If we use locking in a find operation the JPA provider will return already locked entities to the application.

**Locking a previously loaded entity**

Department department = entityManager.find(Department.class, 1);

entityManager.lock(department, LockModeType.PESSIMISTIC\_READ);

In this case the entity is only locked when we execute the **lock()** operation.

**Locking query results**

TypedQuery<Department> query = entityManager

.createQuery("select d from Department d", Department.class);

query.setLockMode(LockModeType.PESSIMISTIC\_READ);

List<Department> departments = query.getResultList();

We may also apply a lock to a JPA query. The entities that are returned by the query will be locked.

OPTIMISTIC locking

Optimistic locking for writing operations is used implicitly by the JPA provider. When the provider is persisting an entity that has been optimistically locked, it will compare the current entity version (@Version entity field) with the version of that record that is stored in the database. If the versions don't match at that time it means that some other process has changed the data in the meantime so the provider raises an exception.

The version check is made during UPDATE statement execution using queries with the following semantics:

UPDATE DEPARTMENT  
SET NAME = 'Research', VERSION = **4**  
WHERE ID = 1  
AND VERSION = **3**;

The current version (the one that was read from the database in the first place) is used in the WHERE clause. If the number of updated rows is zero it means that the current record version was changed by another process in the meantime. If the update statement returns a row count of 1, it means that no other process updated the record in the database and we may proceed the execution. Note that we also increment the version in the UPDATE statement.

This mechanism is covered in the following article: [**JPA entity versioning (@Version and Optimistic Locking)**](http://www.byteslounge.com/tutorials/jpa-entity-versioning-version-and-optimistic-locking), so it will not be covered here in detail.

As we have just said, optimistic locking for writing operations is implicit in JPA. The application does not need to take special care of locking because the provider will do it automatically using the entity version field.

Optimistic locking for **read** operations must be explicitly called. It may be used in any of the ways we have seen in the previous section: together with find operations, executed over an already loaded entity or in a JPA query.

entityManager.find(Department.class, 1, LockModeType.OPTIMISTIC);

This type of lock will guarantee that the entity will not be changed by some other process while the current transaction is in progress, and is to be used in entities that we will only read but require that no other process change the entity until our transaction completes. Remember that we are still talking about **optimistic** locking, so other transactions may actually change the locked record, but our transaction will fail as soon as it detects that those changes took place.

In order to illustrate this scenario, consider the following EJB:

**Acquiring an optimistic lock for read operation**

@Stateless

public class TestEJB {

@PersistenceContext

private EntityManager entityManager;

@TransactionAttribute(TransactionAttributeType.REQUIRED)

public void transactionalMethod() {

Department department = entityManager.find(Department.class, 1, LockModeType.OPTIMISTIC);

// Do other work that uses the

// department entity but does

// not change the entity

}

}

We are reading the entity and hypothetically using it in our unit of work. We also require that no other process changes the corresponding database record in the mean time, so we apply an OPTIMISTIC lock in the read operation.

Just **before** the final commit operation, the JPA provider will check if the record version in the database is still the same that it has read in the first place when it loaded the entity. This check is made using a regular SELECT operation. If the version is still the same, the transaction commits. If the version is not the same that it was read, the provider will raise an exception and the transaction will rollback. The SELECT statement used for the check is similar to the following:

SELECT VERSION  
FROM DEPARTMENT  
WHERE ID = 1;

Note that it's still possible for another transaction to change the record between this last SELECT statement check and the final commit, but in terms of business logic it's no longer relevant. What matters is that when we checked the version after doing all the work, the record was still unchanged, so any work that **depends** on that record being unchanged may be considered as correct because we guaranteed that we used an unchanged entity during all the unit of work duration.

Finally, the remaining optimistic locking mode is the **OPTIMISTIC\_FORCE\_INCREMENT**. This locking mode guarantees that the version field will also be incremented even in the case of optimistic locking **read** operations. Write operations do not apply here because the version field is always incremented in write operations.

**Note**: The database connection(s) isolation level must **not** be greater that READ COMMITTED, or else the database itself will guarantee that the non-repeatable read phenomenon does not occur, so the optimistic locking strategy for read operations will not make sense. More details about isolation levels in: [**Spring transaction isolation level**](http://www.byteslounge.com/tutorials/spring-transaction-isolation-tutorial). The article is more towards the Spring framework, but the same concepts apply to isolation levels in general.

As we have seen, optimistic locking strategies do not actually lock anything, so the introduced level of contention is very reduced (or even none). They rely on checks made by the application in order to assert that concurrent changes to the same record didn't happen, and raise exceptions if concurrent changes are detected. It is the application responsibility to deal with these exceptions. If we have a lot of concurrent modifications to the same records it may become impractical to deal with a high number of errors, and this is where the pessimistic locking strategies may assume an important role.

PESSIMISTIC locking

Pessimistic locking is done at the database level. The JPA provider will delegate any pessimistic locking application requests to the database. The database itself will in fact lock the records and will only release the lock(s) after transaction completion (commit or rollback).

This will introduce an higher level of contention but may be useful in scenarios where concurrent updates to the same records are very frequent, and dealing with a high degree of optimistic locking conflicts (errors) becomes impractical.

Pessimistic locking also applies to read and write operations, and we will start with the **write** operation:

**LockModeType.PESSIMISTIC\_WRITE**

entityManager.find(Department.class, 1, LockModeType.PESSIMISTIC\_WRITE);

This instruction will be translated into a **SELECT ... FOR UPDATE** statement:

SELECT ID, NAME, VERSION  
FROM DEPARTMENT  
WHERE ID = 1 **FOR UPDATE**;

The record will become locked for the duration of the current transaction. Any UPDATE operation, SELECT ... FOR UPDATE operation, or pessimistic READ lock operation executed against that same record by other transaction will become locked at the database level until the current transaction - that holds the lock - completes. The semantics are similar to an exclusive write lock.

One may execute the locking statements in a way that they do not wait indefinitely for the lock to be acquired (ex: lock acquire timeouts) but that's outside of this article's scope.

The pessimistic **read** operation follows next:

**LockModeType.PESSIMISTIC\_READ**

entityManager.find(Department.class, 1, LockModeType.PESSIMISTIC\_READ);

The pessimistic lock for read operations is only available in some database systems. When this strategy is not available for the current database, the JPA provider will issue a pessimistic write lock instead. The following SQL statement is the result of executing the pessimistic read lock against a MySQL 5.5.32 instance:

SELECT ID, NAME, VERSION  
FROM DEPARTMENT  
WHERE ID = 1 **LOCK IN SHARE MODE**;

This operation will lock the record in a shared mode, meaning that other transactions may freely read the record and even lock that same record using an identical pessimist **read** strategy and still continue to process. Every write operation or **write** lock against the record will become locked by the database until no other transaction holds a lock to that record. Such write operations that are locked by the database while a transaction holds a read lock include UPDATE operations and SELECT ... FOR UPDATE operations. The semantics are similar to a shared read lock.

Finally, the PESSIMISTIC\_FORCE\_INCREMENT lock mode will result in a pessimistic **write** lock (SELECT ... FOR UPDATE) and will make sure that an eventually existing entity version field is incremented, even if the entity being locked is not changed by the current transaction which acquired the lock.