**Optimistic Locking VS Pessimistic Locking - Precise – 2022**

**Optimistic Locking**

**Optimistic Locking is a strategy where you read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version hasn't changed before you write the record back**. Optimistic locking doesn't necessarily use a version number. Other strategies include using (a) a timestamp or (b) the entire state of the row itself. **Timestamps are a less reliable way of optimistic locking than version numbers**.

In order to use optimistic locking, we need to have an entity including a property with **@Version annotation**. Before the transaction wants to make an update, it checks the version property again. If the value has changed in the meantime an **OptimisticLockException** is thrown. **Optimistic locking is based on detecting changes on entities by checking their version attribute**. If any concurrent update takes place, OptmisticLockException occurs.

**When to use**

**Optimistic Locking is suitable for applications which do much more reads than updates or deletes.**

**On the contrary, pessimistic locking mechanism involves locking entities on the database level.**

@Entity

public class Student {

@Id

private Long id;

**@Version**

**private Integer version;**

// getters and setters

}

We should know that we can retrieve a value of the version attribute via entity, but **we mustn't update or increment** it.

**Lock Modes**

JPA provides us with two different optimistic lock modes (and two aliases):

**LockModeType.OPTIMISTIC** – obtains an optimistic read lock, **LockModeType.READ – it's a synonym for OPTIMISTIC :** Whenever we request the OPTIMISTIC lock mode, a persistence provider will prevent our data from **dirty reads as well as non-repeatable reads**.

**LockModeType.OPTIMISTIC\_FORCE\_INCREMENT** – it obtains an optimistic lock the same as OPTIMISTIC and additionally increments the version attribute value - **LockModeType.WRITE** – it's a synonym for OPTIMISTIC\_FORCE\_INCREMENT.

To specify a lock on a custom query method of a Spring Data JPA repository, we can annotate the method with @Lock and specify the required lock mode type:

**@Lock(LockModeType.OPTIMISTIC\_FORCE\_INCREMENT)**

**@Query("SELECT c FROM Customer c WHERE c.orgId = ?1")**

**public List<Customer> fetchCustomersByOrgId(Long orgId);**

**Versionless optimistic locking**

Although the default @Version property optimistic locking mechanism is sufficient in many situations, sometimes, you need rely on the actual database row column values to prevent lost updates. Hibernate supports a form of optimistic locking that does not require a dedicated "version attribute". This is achieved through the use of the **@OptimisticLocking annotation** which defines a single attribute of type org.hibernate.annotations.**OptimisticLockType**. 🡺 Hibernate specific.

Here are **4 available OptimisticLockTypes**:

**OptimisticLockType.NONE**: optimistic locking is disabled even if there is a @Version annotation present

**OptimisticLockType.VERSION** (the default): performs optimistic locking based on a @Version as described above

**OptimisticLockType.ALL**: performs optimistic locking based on all fields as part of an expanded WHERE clause restriction for the UPDATE/DELETE SQL statements

**OptimisticLockType.DIRTY**: performs optimistic locking based on dirty fields as part of an expanded WHERE clause restriction for the UPDATE/DELETE SQL statements

@Entity(name = "Person")

**@OptimisticLocking(type = OptimisticLockType.ALL)**

**@DynamicUpdate**

public static class Person {

@Id

private Long id;

@Column(name = "`name`")

private String name;

//Getters and setters are omitted for brevity

}

**When using OptimisticLockType.ALL, you should also use @DynamicUpdate because the UPDATE statement must take into consideration all the entity property values.**

**Versionless optimistic locking using OptimisticLockType.DIRTY**

**The OptimisticLockType.DIRTY differs from OptimisticLockType.ALL in that it only takes into consideration the entity properties that have changed since the entity was loaded in the currently running Persistence Context.**

@Entity(name = "Person")

**@OptimisticLocking(type = OptimisticLockType.DIRTY)**

**@DynamicUpdate**

**@SelectBeforeUpdate**

public static class Person {

@Id

private Long id;

@Column(name = "`name`")

private String name;

//Getters and setters are omitted for brevity

}

**The main advantage of OptimisticLockType.DIRTY over OptimisticLockType.ALL and the default OptimisticLockType.VERSION used implicitly along with the @Version mapping, is that it allows you to minimize the risk of OptimisticLockException across non-overlapping entity property changes.**

**When using OptimisticLockType.DIRTY,** you should also use **@DynamicUpdate** because the UPDATE statement must take into consideration all the dirty entity property values, and also **@SelectBeforeUpdate annotation** so that detached entities are properly handled by the Session#update(entity) operation**.**

Another way to implement this is by using annotations

**@Table(name = "Avenger")**

**@OptimisticLocking(type=OptimisticLockingType.VERSION\_COLUMN**)

public class Avenger implements Serializable {

private String heroName ;

**@Version**

**private long version;**

...

}

**Multiple threads update same row in database at a time how to maintain consistency? There are two possible ways to go.**

Either you choose a pessimistic approach and lock rows, tables or even ranges of rows Or you work with versioned Entities (Optimistic Locking).

**Pessimistic Locking**

**Pessimistic Locking is when you lock the record for your exclusive use until you have finished with it**. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid Deadlocks. **The Pessimistic Locking strategy is ugly but avoids the need for a dedicated version column**. Pessimistic locking is used when a collision is anticipated.

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.

**When to use**

**Pessimistic locking is useful if there are a lot of updates** and relatively high chances of users trying to update data at the same time.

**Pessimistic locking is also more appropriate in applications that contain small tables that are frequently updated.**

**Lock Modes**

**LockModeType.PESSIMISTIC\_READ** – **a shared lock** and prevent the data from being updated or deleted.

**LockModeType.PESSIMISTIC\_WRITE** – **an exclusive lock** and prevent the data from read, updates or deletes

**LockModeType.PESSIMISTIC\_FORCE\_INCREMENT** – works like PESSIMISTIC\_WRITE and it additionally increments a version attribute of a versioned entity

**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.

**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**.

To enforce the lock on predefined repository methods such as findAll or findById(id), we have to declare the method within the repository and annotate the method with the Lock annotation:

**@Lock(LockModeType.PESSIMISTIC\_READ)**

**public Optional<Customer> findById(Long customerId);**

**Setting Transaction Lock Timeouts**

**@Lock(LockModeType.PESSIMISTIC\_READ)**

**@QueryHints({@QueryHint(name = "javax.persistence.lock.timeout", value = "3000")})**

public Optional<Customer> findById(Long customerId);

**Exceptions**

**PessimisticLockException** – indicates that obtaining a lock or converting a shared to exclusive lock fails and results in a transactionlevel rollback

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

**entityManager.find(Student.class, studentId, LockModeType.PESSIMISTIC\_READ);**

**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**);

**Applying Pessimistic Locking with @Lock Annotation**

public interface ArticleRepository extends CrudRepository<Article, Long> {

**@Lock(LockModeType.PESSIMISTIC\_WRITE)**

**@Query("select a from Article a where a.id = :id")**

Article findArticleForWrite(@Param("id") Long id);

**@Lock(LockModeType.PESSIMISTIC\_READ)**

**@Query("select a from Article a where a.id = :id")**

Article findArticleForRead(@Param("id") Long id);

}

**@Transactional and @Lock**

You need to start a transaction before executing a locking query. You can only lock something in the context of a database transaction.

You use @Lock(LockModeType.PESSIMISTIC\_WRITE) on non transactional context.

You need add @Transactional.

@Lock(LockModeType.PESSIMISTIC\_WRITE)

@Transactional

@Query(value = "SELECT so FROM SampleObject so WHERE so.parameter1 = ?1 AND so.parameter2 = ?2")

SampleObject findSampleObjectByParameter1AndParameter2(String parameter1, String parameter2);

**If you want transactional behavior then add @transactional and if your usecase requires locking and as per use case use appropriate locking.**

**The two main tools we use to cope with concurrency are database transactions and distributed locks. These two are not interchangeable. You can't use a transaction when you need a lock. You can't use a lock when you need a transaction.** [source](https://makandracards.com/makandra/31937-differences-between-transactions-and-locking).

If you look carefully at the javadoc of @Lock it states that: Annotation used to specify the LockModeType to be used when executing the query. So the lock is active only during the query execution.. after the query is done.. the lock is released. It is not active for the duration of the entire transaction as you expect it to be.