https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/pessimistic-lock.html
https://www.baeldung.com/jpa-pessimistic-locking
https://medium.com/@ajones1_999/understanding-mysql-multiversion-concurrency-control-6b52f1bd5b7e

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.

Lock Modes

- *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

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 which allows readers to fetch data that has been already blocked.

—---

Optimistic locking assumes (as being optimistic) that there will be rare chances of multi users read/write conflicts, so it delays the conflict checking till the commit time, whereas pessimistic assumes that there is a high possibility of conflict and acquires a database lock at begging of the transaction.

In following example we will learn how to implement pessimistic locking by using `LockModeType.PESSIMISTIC_READ` and `LockModeType.PESSIMISTIC_WRITE`.

A transaction involving `LockModeType.PESSIMISTIC_READ` is an intent to just read the entity without modifying it, whereas `LockModeType.PESSIMISTIC_WRITE` transaction may modify the entity.

A transaction that has acquired `PESSIMISTIC_READ` lock, prevents any other transaction from acquiring a `PESSIMISTIC_WRITE`. Multiple PESSIMISTIC_READs can still happen simultaneously, that's the reason it is also known as shared lock.

A transaction that has acquired `PESSIMISTIC_WRITE` lock, prevents any other transaction from acquiring either of `PESSIMISTIC_WRITE` or `PESSIMISTIC_READ` lock. It is also known as exclusive lock.

```
@Entity
public class Article {
  @Id
  @GeneratedValue
  private long id;
  private String content;
    ............
}
```

Pessimistic write blocking pessimistic read example

In following example, we are going to use threads to simulate two users. First thread will obtain PESSIMISTIC_WRITE lock and will update the Article entity, during that time another thread will try to obtain PESSIMISTIC_READ lock and will block (or it may fail with exception, depending on underlying locking timeout value) till the first transaction is committed:

```
public class PessimisticLockExample {
  private static EntityManagerFactory entityManagerFactory =
          Persistence.createEntityManagerFactory("example-unit");

  public static void main(String[] args) throws Exception {
      ExecutorService es = Executors.newFixedThreadPool(3);
      try {
          persistArticle();
          es.execute(() -> {
              updateArticle();
          });
          es.execute(() -> {
              //simulating other user by using different thread
              readArticle();
          });
          es.shutdown();
          es.awaitTermination(1, TimeUnit.MINUTES);
      } finally {
          entityManagerFactory.close();
      }
  }

  private static void updateArticle() {
      log("updating Article entity");
      EntityManager em =
entityManagerFactory.createEntityManager();
      em.getTransaction().begin();
      Article article = em.find(Article.class, 1L,
LockModeType.PESSIMISTIC_WRITE);
      try {
          TimeUnit.SECONDS.sleep(2);
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
      article.setContent("updated content .. ");
      log("committing in write thread.");
      em.getTransaction().commit();
      em.close();
      log("Article updated", article);
  }

  private static void readArticle() {
      try {//some delay before reading
          TimeUnit.MILLISECONDS.sleep(100);
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
      log("before acquiring read lock");
      EntityManager em =
entityManagerFactory.createEntityManager();
      em.getTransaction().begin();
      Article article = em.find(Article.class, 1L,
LockModeType.PESSIMISTIC_READ);
```

```
      log("After acquiring read lock", article);
      em.getTransaction().commit();
      em.close();
      log("Article after read commit", article);
  }

  public static void persistArticle() {
      log("persisting article");
      Article article = new Article("test article");
      EntityManager em =
entityManagerFactory.createEntityManager();
      em.getTransaction().begin();
      em.persist(article);
      em.getTransaction().commit();
      em.close();
      log("Article persisted", article);
  }

  private static void log(Object... msgs) {
      System.out.println(LocalTime.now() + " - " +
Thread.currentThread().getName() +
              " - " + Arrays.toString(msgs));
  }
}
```

```
22:39:33.330 - main - [persisting article]

22:39:33.413 - main - [Article persisted, Article{id=1, content='test article'}]

22:39:33.414 - pool-2-thread-1 - [updating Article entity]

22:39:33.514 - pool-2-thread-2 - [before acquiring read lock]

22:39:35.423 - pool-2-thread-1 - [committing in write thread.]

22:39:35.428 - pool-2-thread-1 - [Article updated, Article{id=1, content='updated
content .. '}]

22:39:35.435 - pool-2-thread-2 - [After acquiring read lock, Article{id=1,
content='updated content .. '}]

22:39:35.436 - pool-2-thread-2 - [Article after read commit, Article{id=1,
content='updated content .. '}]
```

As seen in above output, read thread blocked till the commit time of the write transaction.

Pessimistic read blocking pessimistic write example

This example reads Article entity first with PESSIMISTIC_READ lock mode, hence the update thread (PESSIMISTIC_WRITE mode) blocks till the read transaction finishes.

```java
public class PessimisticLockExample2 {
  private static EntityManagerFactory entityManagerFactory =
          Persistence.createEntityManagerFactory("example-unit");

  public static void main(String[] args) throws Exception {
      ExecutorService es = Executors.newFixedThreadPool(3);
      try {
          persistArticle();
          es.execute(() -> {
              //simulating other user read by using different thread
              readArticle();
          });
          es.execute(() -> {
              updateArticle();
          });

          es.shutdown();
          es.awaitTermination(1, TimeUnit.MINUTES);
      } finally {
          entityManagerFactory.close();
      }
  }
  private static void updateArticle() {
      try {//some delay before writing
          TimeUnit.MILLISECONDS.sleep(100);
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
      EntityManager em =
entityManagerFactory.createEntityManager();
      em.getTransaction().begin();
      log("write thread before acquiring lock");
      Article article = em.find(Article.class, 1L,
LockModeType.PESSIMISTIC_WRITE);
      log("write thread after acquiring lock");
      article.setContent("updated content .. ");
      log("committing in write thread.");
      em.getTransaction().commit();
      em.close();
      log("Article updated", article);
  }

  private static void readArticle() {
      log("before acquiring read lock");
      EntityManager em =
entityManagerFactory.createEntityManager();
      em.getTransaction().begin();
      Article article = em.find(Article.class, 1L,
LockModeType.PESSIMISTIC_READ);
      log("After acquiring read lock", article);
      try {
          TimeUnit.SECONDS.sleep(2);
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
      em.getTransaction().commit();
      em.close();
      log("Article after read commit", article);
  }
```

```
    ............
}
```

```
22:40:57.654 - main - [persisting article]

22:40:57.742 - main - [Article persisted, Article{id=1, content='test article'}]

22:40:57.743 - pool-2-thread-1 - [before acquiring read lock]

22:40:57.754 - pool-2-thread-1 - [After acquiring read lock, Article{id=1,
content='test article'}]

22:40:57.844 - pool-2-thread-2 - [write thread before acquiring lock]

22:40:59.755 - pool-2-thread-1 - [Article after read commit, Article{id=1,
content='test article'}]

22:40:59.759 - pool-2-thread-2 - [write thread after acquiring lock]

22:40:59.759 - pool-2-thread-2 - [committing in write thread.]

22:40:59.764 - pool-2-thread-2 - [Article updated, Article{id=1, content='updated
content .. '}]
```

Understanding MySQL Multiversion Concurrency Control

MySQL, under the InnoDB storage engine, allows writes and reads of the same row to not interfere with each other. This is one of those features that we use so often it kind of gets taken for granted, but if you think about how you would build such a thing it's a lot more detailed than it seems. Here, I am going to talk through how that is implemented, as well as some ramifications of the design.

Allowing Concurrent Change

Unsurprisingly, given the title of this post, MySQL's mechanism for allowing you to simultaneously read and write from the same row is called "Multiversion Concurrency Control". They (of course) have documentation on it, but that dives into internal technical details pretty fast.

Instead, let's talk about it at a little higher level. This concept has been around for a long time (the best I can do hunting down an origin is a thesis from 1979). The overall answer for allowing concurrent reads and writes is pretty simple: writes create new versions of rows, reads see the version that was current when they started.

Version tracking

Obviously if we're going to keep track of versions, we need something to differentiate them. This tool needs to

distinguish one version from another, but ideally it would also make it easy to decide which version a read operation should see.

In MySQL, this "version enabling thing" is a transaction id. Every transaction gets one. Even your one-shot update queries in the console get one. These ids are incremented in a way that allows MySQL to determine that one transaction started before another. Every table under InnoDB essentially has a "hidden column" that stores the transaction id of the last write operation to change the row. So, in addition to the columns you may have updated, a write operation *also* marks the row with its transaction id. This allows read operations to know if they can use the row data, or if it has been changed and they need to consult an older version.

Reading older version

For the cases where your read operation hits on rows that have been changed, you'll need an older version of the data. The transaction id comes into play here too, but there's more info needed. Every time MySQL writes data into a row, it *also* writes an entry into the rollback segment. This is a data structure that stores "undo logs" used to restore the row to its previous state. It's called the "rollback segment" because it is the tool used to handle rolling back transactions.

The rollback segment stores undo logs for each row in the database. Every row has *another* hidden column that stores the location of the latest undo log entry, which would restore the row to its state prior to the last write. When these entries are created, they are marked with the *outgoing* transaction id. By walking the undo log for a row and finding the latest transaction *before* a read transaction the database can identify the correct data to present to a transaction.

Handling deletes

Deletion is handled by a marker in the row to indicate a record was deleted. Delete operations *also* set the row's transaction id to their transaction id, so the process above can present a pre-delete version of the row to read operations that started before the delete.

When are versions deleted

MySQL obviously cannot keep a record of every change that happens in the database for all time. It doesn't need to, though. Undo logs can be removed as soon as the last transaction that could possibly want them completes.

Similarly, rows that have been marked as deleted can be outright abandoned once the oldest active transaction started after the deletion. These rows and undo logs are

physically removed to reclaim their disk space by a "purge" operation that happens in its own thread in the background.