Vlad Mihalcea's Blog

Teaching is my way of learning

Hibernate Locking Patterns – How do PESSIMISTIC_READ and PESSIMISTIC_WRITE work

FEBRUARY 24, 2015APRIL 1, 2015 / VLADMIHALCEA

Introduction

Java Persistence API (http://en.wikipedia.org/wiki/Java_Persistence_API) comes with a thorough concurrency control mechanism, supporting both implicit and explicit locking. The implicit locking mechanism is straightforward and it relies on:

- Optimistic locking: Entity state changes (2014/08/07/a-beginners-guide-to-jpahibernate-flush-strategies/) can trigger a version incrementation
- Row-level locking: Based on the current running transaction isolation level (2014/12/23/a-beginners-guide-to-transaction-isolation-levels-in-enterprise-java/), the INSERT/UPDATE/DELETE statements may acquire exclusive row locks

While implicit locking (2015/01/12/a-beginners-guide-to-java-persistence-locking/) is suitable for many scenarios, an explicit locking mechanism can leverage a finer-grained concurrency control.

In my previous posts, I covered the explicit optimistic lock modes:

- OPTIMISTIC (2015/01/26/hibernate-locking-patterns-how-does-optimistic-lock-mode-work/)
- OPTIMISTIC_FORCE_INCREMENT (2015/02/09/hibernate-locking-patterns-how-does-optimistic_force_increment-lock-mode-work/)
- PESSIMISTIC_FORCE_INCREMENT (2015/02/16/hibernate-locking-patterns-how-does-pessimistic_force_increment-lock-mode-work/)

In this post, I am going to unravel the explicit pessimistic lock modes:

- PESSIMISTIC_READ (http://docs.oracle.com/javaee/7/api/javax/persistence/LockModeType.html#PESSIMISTIC_READ)
- PESSIMISTIC_WRITE (http://docs.oracle.com/javaee/7/api/javax/persistence/LockModeType.html#PESSIMISTIC_WRITE)

Readers–writer lock

A database system is a highly concurrent environment, therefore many concurrency theory idioms (2014/05/28/the-data-knowledge-stack/) apply to database access as well. Concurrent changes must be serialized to preserve data integrity, so most database systems use a two-phase locking (http://en.wikipedia.org/wiki/Two-phase_locking)

strategy, even if it's usually supplemented by a Multiversion concurrency control (http://en.wikipedia.org/wiki/Multiversion_concurrency_control) mechanism.

Because a mutual exclusion locking (http://en.wikipedia.org/wiki/Mutual_exclusion) would hinder scalability (treating reads and writes equally), most database systems use a readers-writer locking (http://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock) synchronization scheme, so that:

- A shared (read) lock blocks writers, allowing multiple readers to proceed
- An exclusive (write) lock blocks both readers and writers, making all write operations be applied sequentially

Because the locking syntax is not part of the SQL Standard, each RDBMS (http://en.wikipedia.org/wiki/Relational_database_management_system) has opted for a different syntax:

Database name	Shared lock statement	Exclusive lock statement
Oracle	FOR UPDATE	FOR UPDATE
MySQL	LOCK IN SHARE MODE	FOR UPDATE
Microsoft SQL Server	WITH (HOLDLOCK, ROWLOCK)	WITH (UPDLOCK, ROWLOCK)
PostgreSQL	FOR SHARE	FOR UPDATE
DB2	FOR READ ONLY WITH RS	FOR UPDATE WITH RS

Java Persistence abstraction layer hides the database specific locking semantics, offering a common API that only requires two Lock Modes. The shared/read lock is acquired using the PESSIMISTIC_READ (http://docs.oracle.com/javaee/7/api/javax/persistence/LockModeType.html#PESSIMISTIC_READ) Lock Mode Type, and the exclusive/write lock is requested using PESSIMISTIC_WRITE (http://docs.oracle.com/javaee/7/api/javax/persistence/LockModeType.html#PESSIMISTIC_WRITE) instead.

PostgreSQL row-level lock modes

For the next test cases, we are going to use PostgreSQL (http://www.postgresql.org/) for it supports both exclusive and share explicit locking (http://www.postgresql.org/docs/9.4/static/explicit-locking.html).

All the following tests will use the same concurrency utility, emulating two users: Alice and Bob. Each test scenario will verify a specific read/write locking combination.

```
1  private void testPessimisticLocking(ProductLockRequestCallable primaryLockRequestCallable
2      doInTransaction(session -> {
3          try {
4              Product product = (Product) session.get(Product.class, 1L);
5              primaryLockRequestCallable.lock(session, product);
6              executeAsync(
7                  () -> {
8                      doInTransaction(_session -> {
9                          Product _product = (Product) _session.get(Product.class, 1L);
10                         secondaryLockRequestCallable.lock(_session, _product);
11                     });
12                 },
13                 endLatch::countDown
14             );
15             sleep(WAIT_MILLIS);
16         } catch (StaleObjectStateException e) {
17             LOGGER.info("Optimistic locking failure: ", e);
18         }
19     });
20     awaitOnLatch(endLatch);
21 }
```

Case 1: PESSIMISTIC_READ doesn't block PESSIMISTIC_READ lock requests

The first test will check how two concurrent PESSIMISTIC_READ lock requests interact:

```
@Test
public void testPessimisticReadDoesNotBlockPessimisticRead() throws InterruptedException
    LOGGER.info("Test PESSIMISTIC_READ doesn't block PESSIMISTIC_READ");
    testPessimisticLocking(
        (session, product) -> {
            session.buildLockRequest(new LockOptions(LockMode.PESSIMISTIC_READ)).lock(prc
            LOGGER.info("PESSIMISTIC_READ acquired");
        },
        (session, product) -> {
            session.buildLockRequest(new LockOptions(LockMode.PESSIMISTIC_READ)).lock(prc
            LOGGER.info("PESSIMISTIC_READ acquired");
        }
    );
}
```

Running this test, we get the following output:

```
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Test PESSIMISTIC_READ

#Alice selects the Product entity
[Alice]: Time:1 Query:{[
SELECT lockmodepe0_.id          AS id1_0_0_,
       lockmodepe0_.description AS descript2_0_0_,
       lockmodepe0_.price       AS price3_0_0_,
       lockmodepe0_.version     AS version4_0_0_
FROM   product lockmodepe0_
WHERE  lockmodepe0_.id = ?
][1]}

#Alice acquires a SHARED lock on the Product entity
[Alice]: Time:1 Query:{[
SELECT id
FROM   product
WHERE  id =?
AND    version =? FOR share
][1,0]}
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - PESSIMISTIC_READ acqui

#Alice waits for 500ms
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Wait 500 ms!

#Bob selects the Product entity
[Bob]: Time:1 Query:{[
 SELECT lockmodepe0_.id          AS id1_0_0_,
       lockmodepe0_.description AS descript2_0_0_,
       lockmodepe0_.price       AS price3_0_0_,
       lockmodepe0_.version     AS version4_0_0_
FROM   product lockmodepe0_
WHERE  lockmodepe0_.id = ?
][1]}

#Bob acquires a SHARED lock on the Product entity
[Bob]: Time:1 Query:{[
SELECT id
FROM   product
WHERE  id =?
AND    version =? FOR share
][1,0]}
[Bob]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - PESSIMISTIC_READ acquire

#Bob's transactions is committed
```

```
[Bob]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection

#Alice's transactions is committed
[Alice]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection
```

In this scenario, there is no contention whatsoever. Both Alice and Bob can acquire a shared lock without running into any conflict.

Case 2: PESSIMISTIC_READ blocks UPDATE implicit lock requests

The second scenario will demonstrate how the shared lock prevents a concurrent modification. Alice will acquire a shared lock and Bob will attempt to modify the locked entity:

```
@Test
public void testPessimisticReadBlocksUpdate() throws InterruptedException {
    LOGGER.info("Test PESSIMISTIC_READ blocks UPDATE");
    testPessimisticLocking(
        (session, product) -> {
            session.buildLockRequest(new LockOptions(LockMode.PESSIMISTIC_READ)).lock(prc
            LOGGER.info("PESSIMISTIC_READ acquired");
        },
        (session, product) -> {
            product.setDescription("USB Flash Memory Stick");
            session.flush();
            LOGGER.info("Implicit lock acquired");
        }
    );
}
```

The test generates this output:

```
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Test PESSIMISTIC_READ

#Alice selects the Product entity
[Alice]: Time:0 Query:{[
SELECT lockmodepe0_.id          AS id1_0_0_,
       lockmodepe0_.description AS descript2_0_0_,
       lockmodepe0_.price       AS price3_0_0_,
       lockmodepe0_.version     AS version4_0_0_
FROM   product lockmodepe0_
WHERE  lockmodepe0_.id = ?
][1]}

#Alice acquires a SHARED lock on the Product entity
[Alice]: Time:0 Query:{[
SELECT id
FROM   product
WHERE  id =?
AND    version =? FOR share
][1,0]}
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - PESSIMISTIC_READ acqui

#Alice waits for 500ms
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Wait 500 ms!

#Bob selects the Product entity
[Bob]: Time:1 Query:{[
SELECT lockmodepe0_.id          AS id1_0_0_,
       lockmodepe0_.description AS descript2_0_0_,
       lockmodepe0_.price       AS price3_0_0_,
       lockmodepe0_.version     AS version4_0_0_
FROM   product lockmodepe0_
WHERE  lockmodepe0_.id = ?
][1]}

```

```
#Alice's transactions is committed
[Alice]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection

#Bob can acquire the Product entity lock, only after Alice's transaction is committed
[Bob]: Time:427 Query:{[
UPDATE product
SET    description = ?,
       price = ?,
       version = ?
WHERE  id = ?
       AND version = ?
][USB Flash Memory Stick,12.99,1,1,0]}
[Bob]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Implicit lock acquired

#Bob's transactions is committed
[Bob]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection
```

While Bob could select the Product entity, the UPDATE is delayed up until Alice transaction is committed (that's why the UPDATE took **427ms** to run).

Case 3: PESSIMISTIC_READ blocks PESSIMISTIC_WRITE lock requests

The same behaviour is exhibited by a secondary PESSIMISTIC_WRITE lock request:

```
@Test
public void testPessimisticReadBlocksPessimisticWrite() throws InterruptedException {
    LOGGER.info("Test PESSIMISTIC_READ blocks PESSIMISTIC_WRITE");
    testPessimisticLocking(
        (session, product) -> {
            session.buildLockRequest(new LockOptions(LockMode.PESSIMISTIC_READ)).lock(prc
            LOGGER.info("PESSIMISTIC_READ acquired");
        },
        (session, product) -> {
            session.buildLockRequest(new LockOptions(LockMode.PESSIMISTIC_WRITE)).lock(pr
            LOGGER.info("PESSIMISTIC_WRITE acquired");
        }
    );
}
```

Giving the following output:

```
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Test PESSIMISTIC_READ

#Alice selects the Product entity
[Alice]: Time:0 Query:{[
SELECT lockmodepe0_.id          AS id1_0_0_,
       lockmodepe0_.description AS descript2_0_0_,
       lockmodepe0_.price       AS price3_0_0_,
       lockmodepe0_.version     AS version4_0_0_
FROM   product lockmodepe0_
WHERE  lockmodepe0_.id = ?
][1]}

#Alice acquires a SHARED lock on the Product entity
[Alice]: Time:1 Query:{[
SELECT id
FROM   product
WHERE  id =?
AND    version =? FOR share
][1,0]}
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - PESSIMISTIC_READ acqui

#Alice waits for 500ms
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Wait 500 ms!

```

```
#Bob selects the Product entity
[Bob]: Time:1 Query:{[
SELECT lockmodepe0_.id          AS id1_0_0_,
       lockmodepe0_.description AS descript2_0_0_,
       lockmodepe0_.price       AS price3_0_0_,
       lockmodepe0_.version     AS version4_0_0_
FROM   product lockmodepe0_
WHERE  lockmodepe0_.id = ?
][1]}

#Alice's transactions is committed
[Alice]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection

#Bob can acquire the Product entity lock, only after Alice's transaction is committed
[Bob]: Time:428 Query:{[
SELECT id
FROM   product
WHERE  id = ?
       AND version = ?
FOR UPDATE
][1,0]}
[Bob]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - PESSIMISTIC_WRITE acquir

#Bob's transactions is committed
[Bob]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection
```

Bob's exclusive lock request waits for Alice's shared lock to be released.

Case 4: PESSIMISTIC_READ blocks PESSIMISTIC_WRITE lock requests, NO WAIT fails fast

Hibernate provides a PESSIMISTIC_NO_WAIT (https://docs.jboss.org/hibernate/orm/4.3/javadocs/org/hibernate/Session.LockRequest.html#PESSIMISTIC_NO_WAIT) timeout directive, which translates to a database specific NO_WAIT lock acquire policy.

The PostgreSQL NO WAIT (http://www.postgresql.org/docs/9.4/static/sql-select.html#SQL-FOR-UPDATE-SHARE) directive is described as follows:

> To prevent the operation from waiting for other transactions to commit, use the NOWAIT option. With NOWAIT, the statement reports an error, rather than waiting, if a selected row cannot be locked immediately. Note that NOWAIT applies only to the row-level lock(s) — the required ROW SHARE table-level lock is still taken in the ordinary way (see Chapter 13). You can use LOCK with the NOWAIT option first, if you need to acquire the table-level lock without waiting.

```
@Test
public void testPessimisticReadWithPessimisticWriteNoWait() throws InterruptedException {
    LOGGER.info("Test PESSIMISTIC_READ blocks PESSIMISTIC_WRITE, NO WAIT fails fast");
    testPessimisticLocking(
        (session, product) -> {
            session.buildLockRequest(new LockOptions(LockMode.PESSIMISTIC_READ)).lock(prc
            LOGGER.info("PESSIMISTIC_READ acquired");
        },
        (session, product) -> {
            session.buildLockRequest(new LockOptions(LockMode.PESSIMISTIC_WRITE)).setTime
            LOGGER.info("PESSIMISTIC_WRITE acquired");
        }
    );
}
```

This test generates the following output:

```
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Test PESSIMISTIC_READ

```

```
#Alice selects the Product entity
[Alice]: Time:1 Query:{[
SELECT lockmodepe0_.id          AS id1_0_0_,
       lockmodepe0_.description AS descript2_0_0_,
       lockmodepe0_.price       AS price3_0_0_,
       lockmodepe0_.version     AS version4_0_0_
FROM   product lockmodepe0_
WHERE  lockmodepe0_.id = ?
][1]}

#Alice acquires a SHARED lock on the Product entity
[Alice]: Time:1 Query:{[
SELECT id
FROM   product
WHERE  id =?
AND    version =? FOR share
][1,0]}
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - PESSIMISTIC_READ acqui

#Alice waits for 500ms
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Wait 500 ms!

#Bob selects the Product entity
[Bob]: Time:1 Query:{[
SELECT lockmodepe0_.id          AS id1_0_0_,
       lockmodepe0_.description AS descript2_0_0_,
       lockmodepe0_.price       AS price3_0_0_,
       lockmodepe0_.version     AS version4_0_0_
FROM   product lockmodepe0_
WHERE  lockmodepe0_.id = ?
][1]}

#Bob tries to acquire an EXCLUSIVE lock on the Product entity and fails because of the NO
[Bob]: Time:0 Query:{[
SELECT id
FROM   product
WHERE  id = ?
       AND version = ?
FOR UPDATE nowait
][1,0]}
[Bob]: o.h.e.j.s.SqlExceptionHelper - SQL Error: 0, SQLState: 55P03
[Bob]: o.h.e.j.s.SqlExceptionHelper - ERROR: could not obtain lock on row in relation "pr

#Bob's transactions is rolled back
[Bob]: o.h.e.t.i.j.JdbcTransaction - rolled JDBC Connection

#Alice's transactions is committed
[Alice]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection
```

Since Alice already holds a shared lock on the Product entity associated database row, Bob's exclusive lock request fails immediately.

Case 5: PESSIMISTIC_WRITE blocks PESSIMISTIC_READ lock requests

The next test proves that an exclusive lock will always blocks a shared lock acquire attempt:

```
@Test
public void testPessimisticWriteBlocksPessimisticRead() throws InterruptedException {
    LOGGER.info("Test PESSIMISTIC_WRITE blocks PESSIMISTIC_READ");
    testPessimisticLocking(
        (session, product) -> {
            session.buildLockRequest(new LockOptions(LockMode.PESSIMISTIC_WRITE)).lock(pr
            LOGGER.info("PESSIMISTIC_WRITE acquired");
        },
        (session, product) -> {
            session.buildLockRequest(new LockOptions(LockMode.PESSIMISTIC_READ)).lock(prc
```

```
            LOGGER.info("PESSIMISTIC_WRITE acquired");
        }
    );
}
```

Generating the following output:

```
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Test PESSIMISTIC_WRITE

#Alice selects the Product entity
[Alice]: Time:1 Query:{[
SELECT lockmodepe0_.id          AS id1_0_0_,
       lockmodepe0_.description AS descript2_0_0_,
       lockmodepe0_.price       AS price3_0_0_,
       lockmodepe0_.version     AS version4_0_0_
FROM   product lockmodepe0_
WHERE  lockmodepe0_.id = ?
][1]}

#Alice acquires an EXCLUSIVE lock on the Product entity
[Alice]: Time:0 Query:{[
SELECT id
FROM   product
WHERE  id = ?
       AND version = ?
FOR UPDATE
][1,0]}
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - PESSIMISTIC_WRITE acqu

#Alice waits for 500ms
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Wait 500 ms!

#Bob selects the Product entity
[Bob]: Time:1 Query:{[
SELECT lockmodepe0_.id          AS id1_0_0_,
       lockmodepe0_.description AS descript2_0_0_,
       lockmodepe0_.price       AS price3_0_0_,
       lockmodepe0_.version     AS version4_0_0_
FROM   product lockmodepe0_
WHERE  lockmodepe0_.id = ?
][1]}

#Alice's transactions is committed
[Alice]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection

#Bob can acquire the Product entity SHARED lock, only after Alice's transaction is commit
[Bob]: Time:428 Query:{[
SELECT id
FROM   product
WHERE  id =?
AND    version =? FOR share
][1,0]}
[Bob]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - PESSIMISTIC_WRITE acquir

#Bob's transactions is committed
[Bob]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection
```

Bob's shared lock request waits for Alice's transaction to end, so that all acquired locks are released.

Case 6: PESSIMISTIC_WRITE blocks PESSIMISTIC_WRITE lock requests

An exclusive lock blocks an exclusive lock as well:

```
@Test
public void testPessimisticWriteBlocksPessimisticWrite() throws InterruptedException {
```

```
    LOGGER.info("Test PESSIMISTIC_WRITE blocks PESSIMISTIC_WRITE");
    testPessimisticLocking(
            (session, product) -> {
                session.buildLockRequest(new LockOptions(LockMode.PESSIMISTIC_WRITE)).loc
                LOGGER.info("PESSIMISTIC_WRITE acquired");
            },
            (session, product) -> {
                session.buildLockRequest(new LockOptions(LockMode.PESSIMISTIC_WRITE)).loc
                LOGGER.info("PESSIMISTIC_WRITE acquired");
            }
    );
}
```

The test generates this output:

```
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Test PESSIMISTIC_WRITE

#Alice selects the Product entity
[Alice]: Time:1 Query:{[
SELECT lockmodepe0_.id          AS id1_0_0_,
       lockmodepe0_.description AS descript2_0_0_,
       lockmodepe0_.price       AS price3_0_0_,
       lockmodepe0_.version     AS version4_0_0_
FROM   product lockmodepe0_
WHERE  lockmodepe0_.id = ?
][1]}

#Alice acquires an EXCLUSIVE lock on the Product entity
[Alice]: Time:0 Query:{[
SELECT id
FROM   product
WHERE  id = ?
       AND version = ?
FOR UPDATE
][1,0]}
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - PESSIMISTIC_WRITE acqu

#Alice waits for 500ms
[Alice]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - Wait 500 ms!

#Bob selects the Product entity
[Bob]: Time:1 Query:{[
SELECT lockmodepe0_.id          AS id1_0_0_,
       lockmodepe0_.description AS descript2_0_0_,
       lockmodepe0_.price       AS price3_0_0_,
       lockmodepe0_.version     AS version4_0_0_
FROM   product lockmodepe0_
WHERE  lockmodepe0_.id = ?
][1]}

#Alice's transactions is committed
[Alice]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection

#Bob can acquire the Product entity SHARED lock, only after Alice's transaction is commit
[Bob]: Time:428 Query:{[
SELECT id
FROM   product
WHERE  id =?
AND    version =? FOR update
][1,0]}
[Bob]: c.v.h.m.l.c.LockModePessimisticReadWriteIntegrationTest - PESSIMISTIC_WRITE acquir

#Bob's transactions is committed
[Bob]: o.h.e.t.i.j.JdbcTransaction - committed JDBC Connection
```

Bob's exclusive lock request has to wait for Alice to release its lock.

Conclusion

Relational database systems use locks for preserving ACID guarantees (2014/01/05/a-beginners-guide-to-acid-and-database-transactions/), so it's important to understand how shared and exclusive row-level locks inter-operate. An explicit pessimistic lock is a very powerful database concurrency control mechanism and you might even use it for fixing an optimistic locking race condition (2015/02/03/how-to-fix-optimistic-locking-race-conditions-with-pessimistic-locking/).

Code available on GitHub (https://github.com/vladmihalcea/hibernate-master-class).

If you have enjoyed reading my article and you're looking forward to getting instant email notifications of my latest posts, consider following my blog (follow-me/).

Categories: Hibernate, Java, Transactions Tags: concurrency control, explicit locking, hibernate, Hibernate training, Hibernate tutorial, pessimistic locking, PESSIMISTIC_READ, PESSIMISTIC_WRITE, PostgreSQL