Oracle Blog

Enterprise Tech Tips

Get tips on using enterprise Java technologies and APIs, such as those in Java Platform, Enterprise Edition (Java EE).

« A Sampling of EJB... | Main | Using CDI and Depend... »

Locking and Concurrency in Java Persistence 2.0

By edort on Sep 11, 2009

by Carol McDonald

The Java Persistence API (informally referred to as JPA) provides a plain old Java object (POJO)-based persistence model for Java EE and Java SE applications. It handles the details of how relational data is mapped to Java objects, and it standardizes Object/Relational (O/R) mapping. The latest update to JPA, Java Persistence 2.0, adds a number of new features such as additional O/R mapping functionality and new query language capabilities. Another area that has been enhanced in JPA 2.0 is locking and concurrency.

This Tech Tip highlights the new locking and concurrency features in JPA 2.0 and provides an application that demonstrates these capabilities.

Locking and Concurrency

Locking is a technique for handling database transaction concurrency. When two or more database transactions concurrently access the same data, locking is used to ensure that only one transaction at a time can change the data.

There are generally two locking approaches: optimistic and pessimistic. *Optimistic locking* assumes that there will be infrequent conflicts between concurrent transactions, that is, they won't often try to read and change the same data at the same time. In optimistic locking, the objective is to give concurrent transactions a lot of freedom to process simultaneously, but to detect and prevent collisions. Two transactions can access the same data simultaneously. However, to prevent collisions, a check is made to detect any changes made to the data since the data was last read.

Pessimistic locking assumes that transactions will frequently collide. In pessimistic locking, a transaction that reads the data locks it. Another transaction cannot change the data until the first transaction commits the read.

Optimistic locking works best for applications where concurrent transactions do not conflict. Pessimistic locking works best where concurrent transactions do conflict.

With JPA it is possible to lock an entity. This allows you to control when, where, and which kind of locking to use for an entity. Recall that in JPA, an entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, with each entity instance corresponding to a row in that table.

Locking Support in JPA 1.0

JPA 1.0 only supports optimistic read or optimistic write locking. In this support, any transaction can read and update an entity. However, when a transaction commits, JPA checks the `version` attribute of the entity to determine if it was updated since the entity was last read. If the `version` attribute was updated since the entity was last read, JPA throws an exception. The advantage of this approach is that no database locks are held. This can result in better scalability than for pessimistic locking. The disadvantage of this approach is that the user or application must refresh and retry failed updates.

A versioned entity is marked with the `@Version` annotation, as illustrated in the following code snippet:

```
public class User {
    @ID int id;
    @Version int version;
```

and its corresponding database schema has a version column, such as that created by the following SQL statement:

```
CREATE TABLE USER
(ID NUMBER NOT NULL, VERSION NUMBER),
   PRIMARY KEY (ID));
```

The `version` attribute can be an int, short, long, or timestamp. It is incremented when a transaction successfully commits. This results in an SQL operation such as the following:

```
UPDATE User SET ..., version = version + 1
  WHERE id = ? AND version = readVersion
```

Figure 1 illustrates optimistic locking.

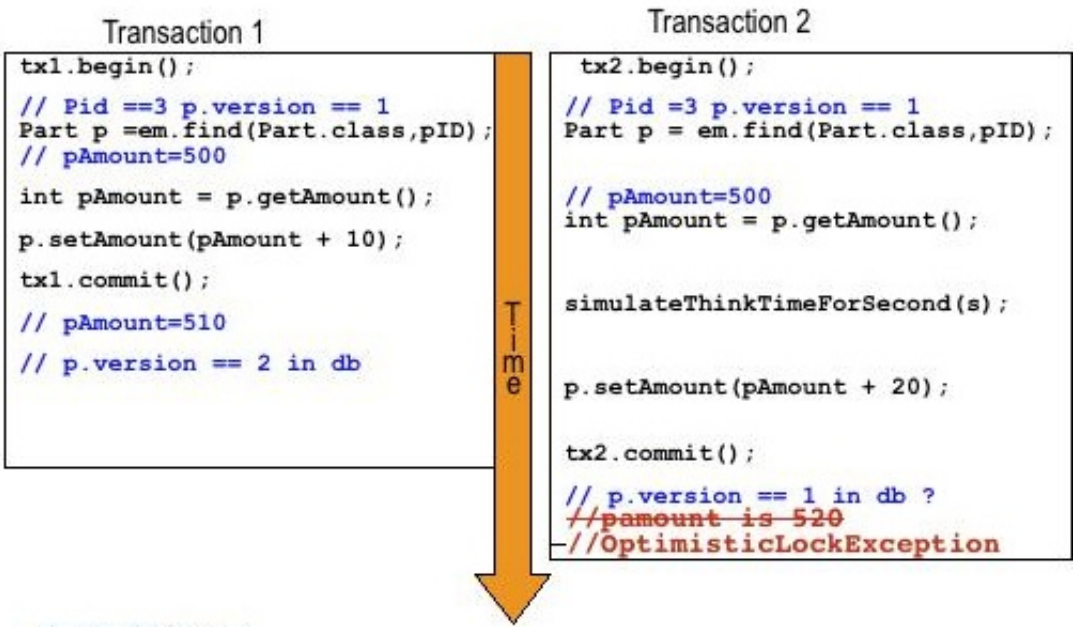

Figure 1. *Optimistic Locking*

Here, two concurrent transactions attempt to update Part `p`. Transaction 1 commits first. In response, JPA increments the `version` attribute for the `p` entity. When Transaction 2 commits, JPA throws an `OptimisticLockException` because the `version` attribute for the `p` entity is higher than it was when Transaction 2 last read the `p` entity. As a result, Transaction 2 is rolled back.

You can further control the way JPA manages locking on a versioned entity by specifying a lock mode. You do this through the `lock()` method of the `EntityManager` class. Here is the method signature:

```
public void lock(Object entity, LockModeType lockMode);
```

The first method parameter is the entity instance that needs to be locked in the transaction. The second method parameter is the lock mode.

In JPA 1.0, the lock mode value could only be one of the following:

- `READ`. In this case, the JPA entity manager performs the optimistic locking operations as previously described. It locks the entity and before a transaction commits, checks the entity's version attribute to determine if it has been updated since the entity was last read. If the version attribute has been updated, the entity manager throws an `OptimisticLockException` and rolls back the transaction.
- `WRITE`. In this case, the entity manager performs the same optimistic locking operations as for the READ lock mode. However, it also updates the entity's version column.

Additional Locking Support in JPA 2.0

JPA 2.0 adds five new lock modes. Two of these are used for optimistic locking. JPA 2.0 also adds support for pessimistic locking and provides three lock modes for pessimistic locking. The two new optimistic lock modes are:

- `OPTIMISTIC`. This is the same as the `READ` lock mode. The `READ` lock mode is still supported in JPA 2.0, but specifying `OPTIMISTIC` is recommended for new applications.
- `OPTIMISTIC_FORCE_INCREMENT`. This is the same as the `WRITE` lock mode. The `WRITE` lock mode is still supported in JPA 2.0, but specifying `OPTIMISTIC_FORCE_INCREMENT` is recommended for new applications.

The three new pessimistic lock modes are:

- `PESSIMISTIC_READ`. The entity manager locks the entity as soon as a transaction reads it. The lock is held until the transaction completes. This lock mode is used when you want to query data using repeatable-read semantics. In other words, you want to ensure that the data is not updated between successive reads. This lock mode does not block other transactions from reading the data.
- `PESSIMISTIC_WRITE`. The entity manager locks the entity as soon as a transaction updates it. This lock mode forces serialization among transactions attempting to update the entity data. This lock mode is often used when there is a high likelihood of update failure among concurrent updating transactions.
- `PESSIMISTIC_FORCE_INCREMENT`. The entity manager locks the entity when a transaction reads it. It also increments the entity's `version` attribute when the transaction ends, even if the entity is not modified.

JPA 2.0 also provides multiple ways to specify the lock mode for an entity. You can specify the lock mode in the `lock()` and `find()` methods of the `EntityManager`. In addition, if you call the `EntityManager.refresh()` method, it refreshes the state of the entity instance from the database and locks it based on the entity's lock mode.

You can also set the lock mode for a query through the `setLockMode()` method of the `Query` interface. And you can specify a lock mode for the results returned by a named query through the `setLockMode` element of the `@NamedQuery` annotation.

Let's look at some examples of the new locking support in JPA 2.0.

`OPTIMISTIC` Lock Mode

The typical use case for `OPTIMISTIC` lock mode is where an entity has an intrinsic dependency on one or more entities to ensure consistency, for example, when there is a relationship between two entities. In the example shown in Figure 2, Transaction 1 on the left updates the price for part `p1`. This increments `p1`'s `version` attribute. Transaction 2 on the right submits a bid for a user, `u1`. If the part price is lower than the user's current bid, Transaction 2 increases the bid.

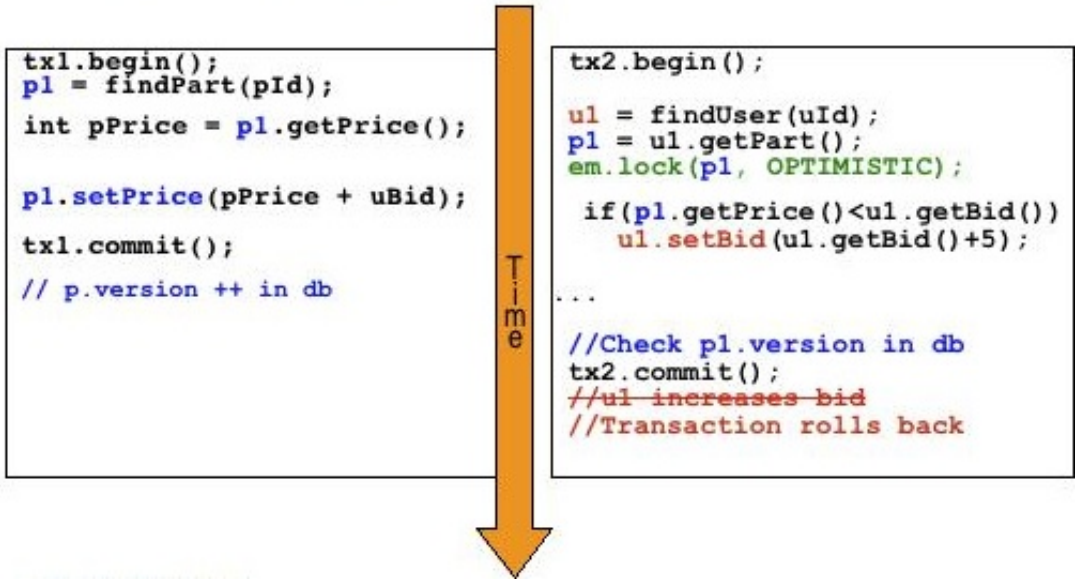

Figure 2. *Using `OPTIMISTIC` Lock Mode*

In this scenario, you don't want Transaction 2 to commit if Transaction T1 changes the price for the part after Transaction T2 reads the price. So `OPTIMISTIC` lock mode is a good choice:

```
em.lock(p1, OPTIMISTIC);
```

Before committing Transaction 2, the entity manager checks the `version` attribute for the `p1` entity. The `p1` `version` attribute is higher than when p1 was last read, so the entity manager throws an `OptimisticLockException` and rolls back Transaction2. Note that checking `u1`'s `version` attribute for an update would not throw an exception. That's because Transaction 1 updates `p1`'s `version` attribute — it does not increment `u1`'s `version` attribute.

`OPTIMISTIC_FORCE_INCREMENT` Lock Mode

`OPTIMISTIC_FORCE_INCREMENT` lock mode causes an optimistic lock failure if another transaction tries to modify the locked entity. The common use for this lock is to guarantee consistency among entities in a relationship.

Figure 3 shows an example of `OPTIMISTIC_FORCE_INCREMENT` lock mode.

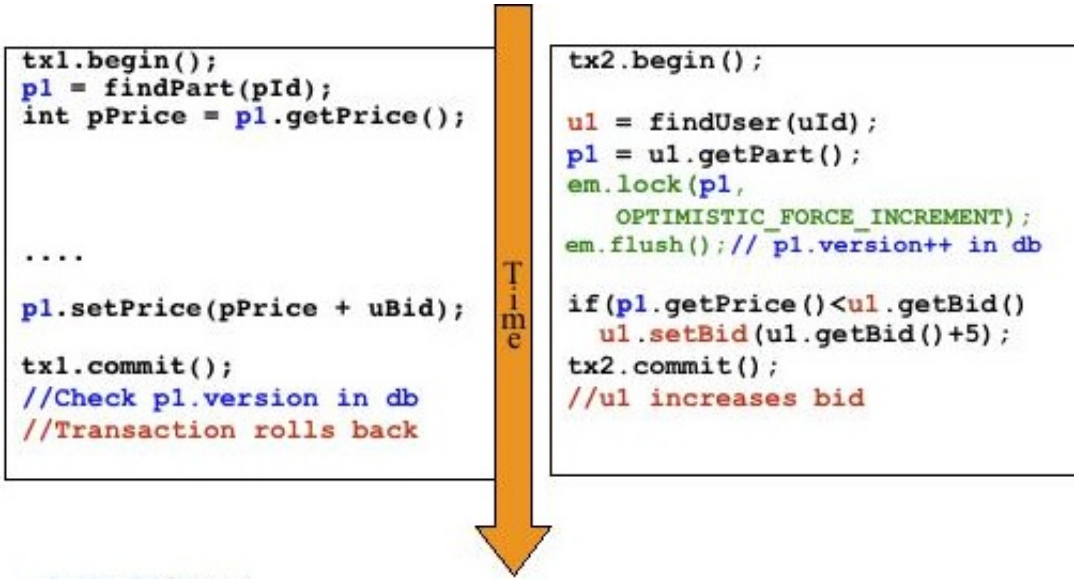

Figure 3. *Using `OPTIMISTIC_FORCE_INCREMENT` Lock Mode*

Transaction 2 on the right wants to ensure that the price for a part `p1` does not change during the transaction, so it locks the `p1` entity as follows:

```
em.lock(p1, OPTIMISTIC_FORCE_INCREMENT);
```

Transaction 2 then calls `em.flush()`— this increments `p1`'s `version` attribute in the database. Any parallel attempt to update `p1` will throw an `OptimisticLockException` and roll back. As you can see, Transaction 1 attempts to update `p1`'s price after Transaction 2 calls `em.flush()`. When Transaction T1 attempts to commit, the entity manager checks the `p1` `version` attribute. Because the attribute has been updated since the last read, the entity manager throws an `OptimisticLockException` and rolls back Transaction T1.

`PESSIMISTIC` Lock Modes

The pessimistic lock modes lock a database row when data is read. This is the equivalent to the action taken in response to the SQL statement `SELECT . . . FOR UPDATE [NOWAIT]`. Pessimistic locking ensures that transactions do not update the same entity at the same time. This can simplify application code, but it limits concurrent access to the data, something that can cause poor scalability and may cause deadlocks. Pessimistic locking is better for applications with a higher risk of contention among

concurrent transactions.

The following figures show various examples of `PESSIMISTIC` lock modes:

- Figure 4 shows an example of reading an entity and in a later step setting it in `PESSIMISTIC_READ` lock mode.
- Figure 5 shows an example of reading an entity and at the same time setting it in `PESSIMISTIC_WRITE` lock mode.
- Figure 6 shows an example of reading an entity and in a later step setting it in `PESSIMISTIC_WRITE` lock mode.

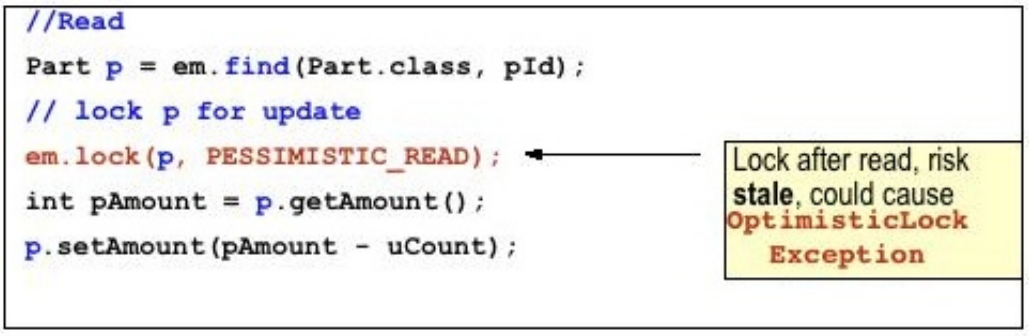

Figure 4. *Setting `PESSIMISTIC_READ` Lock Mode After Reading an Entity*

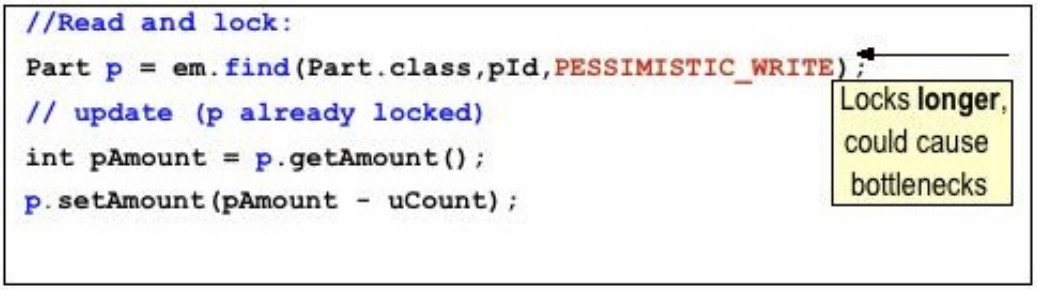

Figure 5. *Setting `PESSIMISTIC_WRITE` Lock Mode While Reading an Entity*

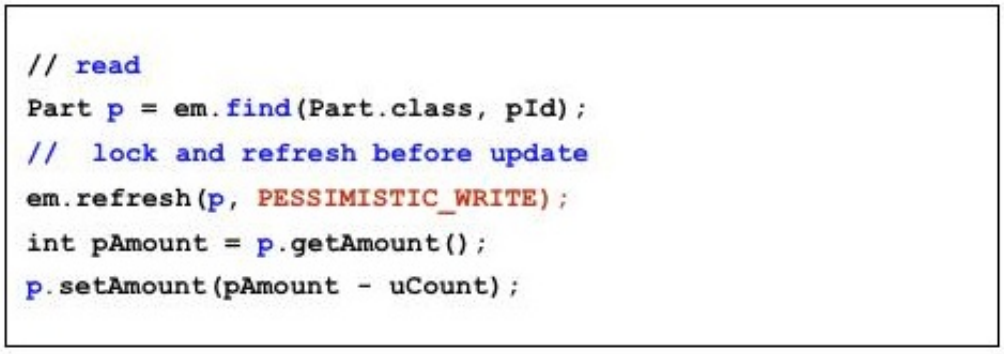

Figure 6. *Setting `PESSIMISTIC_WRITE` Lock Mode After Reading an Entity*

The right locking approach to use depends on your application. Some questions you might want to ask to help make the decision are:

- What is the risk of contention among concurrent transactions?
- What are the requirements for scalability?
- What are the requirements for user retrying after a failure?

Sample Application

Accompanying this tip is a sample application that demonstrates some of the locking support in JPA

2.0. The application is also available in the Java EE 6 SDK Preview release — look for "The Java Persistence API Locking Sample Application" in the `samples` directory of the Java EE 6 SDK Preview release download package.

The application consists of a client, a servlet, entity classes for part and user data, and stateless session beans that provide the logic for accessing and updating the data. The client calls the servlet to initialize the data. The client then makes multiple requests to the servlet that simulate parallel read and update operations. These operations are performed by the beans. Some of the operations are performed using optimistic locking, some using pessimistic locking. For example, the following method, `updateWithOptimisticReadLock()` demonstrates parallel operations performed using optimistic locking.

```
public boolean updateWithOptimisticReadLock(int uID, int s) {
     boolean updateSuccessfull = true;
     // find part update price
     partEJB.updatePrice(uID, s);
     simulateThinkTimeForSecond(s);
     // find user and part, lock part OPTIMISTIC, update user
     userEJB.updateBid(uID, s);

     try {
         em.flush();
     } catch (OptimisticLockException e) {
         System.out.println("updateWithOptimisticReadLock OptimisticLockException
         + "The transaction will be rolled back");
         updateSuccessfull = false;
     } catch (PersistenceException e) {
         System.out.println("Got Exception while updating with optimstic lock" +
         updateSuccessfull = false;
     }
     System.out.println("updateWithOptimisticReadLock " + " updateSuccessful? " +
     return updateSuccessfull;
 }
```

The `updateWithOptimisticReadLock()` method calls the `updatePrice()` method in the `partEJB` bean to find a user and then update the price of a part. The `updateWithOptimisticReadLock()` method then waits to allow parallel method calls to find other users before calling the `updateBid()` method in the `userEJB` bean. The `updateBid()` method sets an optimistic lock for the part and then submits a user bid that is based on the part price, as shown below:

```
public void updateBid(int uID, int s) {

    User u = em.find(User.class, uID);
    int pID = u.getPart().getId();
    Part p = em.find(Part.class, pID);
    em.lock(p, LockModeType.OPTIMISTIC);

    System.out.println("UserManagmentBean updateBid " + " for userId " + uID);
    if (p.getPrice() <= u.getBid() && ! (p.isSold())) {
        u.setBid(u.getBid() +10);
    }

 }
```

The `updateWithOptimisticReadLock()` method then calls `em.flush()`. At that point, the entity manager performs a version check on the part entity. If any transaction submitted by any of the other users

updates the part while it is locked, the entity manager increments the part's version attribute. If the version attribute for the part is higher than it was when the part was last read by the set bid transaction, the `updateWithOptimisticReadLock()` method throws an `OptimisticLockException` and rolls back that transaction.

You can find the source code for the application in the `/samples/javaee6/jpa/locking` directory.

To run the sample application, do the following:

1. If you haven't already done so, download the Java EE 6 SDK Preview release. Also be sure to have an installed version of the Java Platform Standard Edition (Java SE) 6 SDK.
2. Download the sample application.
3. Set up your build environment and configure the application server by following the common build instructions.
4. Change to the *samples_install_dir*`/javaee6/jpa/locking` directory, where *samples_install_dir* is where you installed the sample application.
5. Build, deploy, and run the sample application by entering the following command on the command line:

```
ant all
```

You can replace the `ant all` command with the following set of commands:

`ant default` — compiles and packages the application.

`ant dir` — deploys the application to the application server.

`ant run` — runs the test Java client.

In response, you should see output similar to the following:

```
    [java] LockingJavaClient: Test is starting
    [java] Calling URL:http://localhost:8080/locking/test/?tc=initData&nc=6
    &ns=3&np=3
    [java]
    [java] Starting parallel updates with 9 users for operation: updateWOL
    [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWOL&uid=1
    [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWOL&uid=2
    [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWOL&uid=7
    [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWOL&uid=9
    [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWOL&uid=5
    [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWOL&uid=3
    [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWOL&uid=8
    [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWOL&uid=6
    [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWOL&uid=4
    [java] Result for operation updateWOL for userId 1 is Success
    [java] Result for operation updateWOL for userId 2 is Success
    [java] Result for operation updateWOL for userId 5 is Failure
    [java] Result for operation updateWOL for userId 6 is Failure
    [java] Result for operation updateWOL for userId 7 is Failure
    [java] Result for operation updateWOL for userId 8 is Success
    [java] Result for operation updateWOL for userId 9 is Success
    [java] Result for operation updateWOL for userId 3 is Success
    [java] Result for operation updateWOL for userId 4 is Failure
    [java] Parallel updates executed with 9 users for operation: updateWOL Time
 taken:6146 miliseconds
    [java] ...
    [java]
    [java] Starting parallel updates with 9 users for operation: updateWPL
```

```
     [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWPL&uid=2
     [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWPL&uid=5
     [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWPL&uid=3
     [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWPL&uid=9
     [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWPL&uid=1
     [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWPL&uid=4
     [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWPL&uid=7
     [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWPL&uid=6
     [java] Calling URL:http://localhost:8080/locking/test/?tc=updateWPL&uid=8
     [java] Result for operation updateWPL for userId 5 is Success
     [java] Result for operation updateWPL for userId 3 is Success
     [java] Result for operation updateWPL for userId 9 is Success
     [java] Result for operation updateWPL for userId 2 is Success
     [java] Result for operation updateWPL for userId 1 is Success
     [java] Result for operation updateWPL for userId 8 is Success
     [java] Result for operation updateWPL for userId 4 is Success
     [java] Result for operation updateWPL for userId 6 is Success
     [java] Result for operation updateWPL for userId 7 is Success
     [java] Parallel updates executed with 9 users for operation: updateWPL Time
taken:15054 miliseconds
     [java] LockingJavaClient: Test is ended
```

The operations, which are identified in the `tc` parameter values in the URL calls, are as follows:

- `updateWOL`. Finds a part. Simulates think time to allow parallel threads to find users in parallel. Updates the part using optimistic locking.
- `updateWOR`. Finds a part and a user. Simulates think time to allow parallel threads to find users in parallel. Locks the part using optimistic locking. Updates the user.
- `updateWOW`. Finds a part and a user. Simulates think time to allow parallel threads to find users in parallel. Locks the part using an OPTIMISTIC_FORCE_INCREMENT lock. Updates the user.
- `updateWRP`. Finds a part. Simulates think time to allow parallel threads to find users in parallel. Locks the part using a PESSIMISTIC_READ lock. Updates the part.
- `updateWRR`. Finds a part. Simulates think time to allow parallel threads to find users in parallel. Refreshes using a PESSIMISTIC_WRITE lock. Updates the part.
- `updateWPL`. Finds a part using a PESSIMISTIC_WRITE lock. Simulates think time to allow parallel threads to find users in parallel. Updates the part.

Notice that some update operations that use optimistic locking, such as `updateWOL`, fail, while all update operations that use pessimistic locking, such as `updateWPL`, are successful. However, the time it takes to update using pessimistic locking is much higher than that taken using optimistic locking.

Use the command `ant clean` to undeploy the sample application and to remove temporary directories.

Further Reading

For more information, see the following resources:

- Preventing Non-Repeatable Reads in JPA Using EclipseLink
- Java Persistence API 2.0: What's New ?
- What's New and Exciting in JPA 2.0
- Beginning Java EE 6 Platform with GlassFish 3: From Novice to Professional
- Java Persistence API: Best Practices and Tips

About the Author

Carol McDonald is a Java Technology Evangelist at Sun Microsystems. As a software developer since 1986, Carol's experience has been in the technology areas of distributed network applications and protocols, including Java EE technology, XML, Internet/Intranet applications, LDAP, Distributed Network Management (CMIP,SNMP) and Email (X.400,X.500). Besides Java, Carol is also fluent in French and German. Read Carol McDonald's blog.

Category: Enterprise Java technology

Tags: carol concurrency java jpa locking mcdonald optimistic persistence pessimistic

Permanent link to this entry

« A Sampling of EJB... | Main | Using CDI and Depend... »
Comments:

Nice to see that JPA 2 gets additional features for concurrency control. But what I'm completely missing in your story is that concurrency control is very database implementation specific and that not understanding the concurrency schema used, leads to a false sense of security.

Imho abstracting too much away from the database leads to all kinds of problems (performance / correctness) but also more subtle liveness problems like livelocking and starvation.

So having a comprehensive API still doesn't fix problems. You really need to know the database to see what is going on under the hood (and you also need to check the generated sql to see if it is what you would expect).

Posted by **Peter Veentjer** on September 14, 2009 at 07:40 PM PDT #

O/R Frameworks significantly reduce the code and development time for programmers. However this does not mean you can ignore what's going on with the database. For example with MySQL locking maybe by row with MVCC or by table depending on which storage engine you use. I wrote 2 blog entries on this: JPA Performance, Don't Ignore the Database http://blogs.sun.com/carolmcdonald/entry/jpa_performance_don_t_ignore and MySQL for Developers http://blogs.sun.com/carolmcdonald/entry/mysql_for_developers

Posted by **carol mcdonald** on September 15, 2009 at 01:50 AM PDT #

Hi,
I had difficulties for using the locking features before.Now I think after reading this article now I can use the locking features more efficiently.

Posted by **externe festplatte** on November 09, 2009 at 07:40 PM PST #

Can these types of locking be effectively used in a container managed transaction context, if the Session bean is running with CMT and it is using JPA and database ?

Posted by **IQTF** on September 10, 2010 at 01:04 AM PDT #

Post a Comment:
Comments are closed for this entry.

About

edort

Search

Enter search term:

Search only this blog

Recent Posts

- Security Token Service and Identity Delegation with Metro
- DataSource Resource Definition in Java EE 6
- Asynchronous Support in Servlet 3.0
- POST-REDIRECT-GET and JSF 2.0
- Using CDI and Dependency Injection for Java in a JSF 2.0 Application
- Locking and Concurrency in Java Persistence 2.0
- A Sampling of EJB 3.1
- A Common Ant Build File for Metro-Based Services and Clients
- Jersey and Spring
- Tech Tips Quiz

Top Tags

- .net
- 109
- 196
- 2.0
- 299
- 3.0
- 314
- 330
- annotation
- api
- application
- beans
- components
- composite
- developer
- ee
- ejb
- enterprise
- faces
- facses
- glassfish
- java
- javabeans
- javaserver
- jax-rs
- jaxb
- jersey
- jmaki
- jpa
- jsf
- jsr
- metro

- mysql
- performance
- persistence
- phobos
- quiz
- rest
- security
- services
- servlet
- sip
- spring
- tech
- technology
- tips
- ui
- view
- web
- wsit

Categories

- Enterprise Java technology
- Enterprise JavaBeans Technology
- JAX-RS
- Java Persistence
- JavaServer Faces (JSF) technology
- Metro
- OpenSolaris
- Personal
- Phobos
- Servlet API
- Session Initiation Protocol (SIP)
- Students
- Sun
- datasource
- jMaki
- performance
- quiz
- security
- web services

Archives

« July 2015

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Today

Bookmarks

- BigAdmin blog
- Core Java Technologies Tech Tips
- Enterprise Java Technologies Tech Tips Archive
- Java EE
- Java Technology Fundamentals
- SDN Program News
- blogs.sun.com
- java.com
- java.net
- opensolaris.org

Menu

- Blogs Home
- Weblog
- Login

Feeds

RSS

- All
- /Enterprise Java technology
- /Enterprise JavaBeans Technology
- /JAX-RS
- /Java Persistence
- /JavaServer Faces (JSF) technology
- /Metro
- /OpenSolaris
- /Personal
- /Phobos
- /Servlet API
- /Session Initiation Protocol (SIP)
- /Students
- /Sun
- /datasource
- /jMaki
- /performance
- /quiz
- /security
- /web services
- Comments

Atom

- All
- /Enterprise Java technology
- /Enterprise JavaBeans Technology
- /JAX-RS
- /Java Persistence
- /JavaServer Faces (JSF) technology

- /Metro
- /OpenSolaris
- /Personal
- /Phobos
- /Servlet API
- /Session Initiation Protocol (SIP)
- /Students
- /Sun
- /datasource
- /jMaki
- /performance
- /quiz
- /security
- /web services
- Comments