**Locking strategies**

Optimistic

Optimistic locking ssumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back[[1](https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html" \l "ftn.d0e2213)].

Pessimistic

Pessimistic locking assumes that concurrent transactions will conflict with each other, and requires resources to be locked after they are read and only unlocked after the application has finished using the data.

Hibernate provides mechanisms for implementing both types of locking in your applications.

**5.1. Optimistic**

When your application uses long transactions or conversations that span several database transactions, you can store versioning data, so that if the same entity is updated by two conversations, the last to commit changes is informed of the conflict, and does not override the other conversation's work. This approach guarantees some isolation, but scales well and works particularly well in *Read-Often Write-Sometimes* situations.

Hibernate provides two different mechanisms for storing versioning information, a dedicated version number or a timestamp.

Version number

Timestamp

**Note**

A version or timestamp property can never be null for a detached instance. Hibernate detects any instance with a null version or timestamp as transient, regardless of other unsaved-value strategies that you specify. Declaring a nullable version or timestamp property is an easy way to avoid problems with transitive reattachment in Hibernate, especially useful if you use assigned identifiers or composite keys.

**5.1.1. Dedicated version number**

The version number mechanism for optimistic locking is provided through a @Version annotation.

**Example 5.1. The @Version annotation**

@Entity

public class Flight implements Serializable {

...

    @Version

    @Column(name="OPTLOCK")

    public Integer getVersion() { ... }

}

Here, the version property is mapped to the OPTLOCK column, and the entity manager uses it to detect conflicting updates, and prevent the loss of updates that would be overwritten by a *last-commit-wins* strategy.

The version column can be any kind of type, as long as you define and implement the appropriate UserVersionType.

Your application is forbidden from altering the version number set by Hibernate. To artificially increase the version number, see the documentation for properties LockModeType.OPTIMISTIC\_FORCE\_INCREMENT or LockModeType.PESSIMISTIC\_FORCE\_INCREMENTcheck in the Hibernate Entity Manager reference documentation.

**Database-generated version numbers**

If the version number is generated by the database, such as a trigger, use the annotation @org.hibernate.annotations.Generated(GenerationTime.ALWAYS).

**Example 5.2. Declaring a version property in hbm.xml**

<version

        column="version\_column"

        name="propertyName"

        type="typename"

        access="field|property|ClassName"

        unsaved-value="null|negative|undefined"

        generated="never|always"

        insert="true|false"

        node="element-name|@attribute-name|element/@attribute|."

/>

|  |  |
| --- | --- |
| column | The name of the column holding the version number. Optional, defaults to the property name. |
| name | The name of a property of the persistent class. |
| type | The type of the version number. Optional, defaults to integer. |
| access | Hibernate's strategy for accessing the property value. Optional, defaults to property. |
| unsaved-value | Indicates that an instance is newly instantiated and thus unsaved. This distinguishes it from detached instances that were saved or loaded in a previous session. The default value, undefined, indicates that the identifier property value should be used. Optional. |
| generated | Indicates that the version property value is generated by the database. Optional, defaults to never. |
| insert | Whether or not to include the version column in SQL insert statements. Defaults to true, but you can set it to false if the database column is defined with a default value of 0. |

**5.1.2. Timestamp**

Timestamps are a less reliable way of optimistic locking than version numbers, but can be used by applications for other purposes as well. Timestamping is automatically used if you the @Version annotation on a Date or Calendar.

**Example 5.3. Using timestamps for optimistic locking**

@Entity

public class Flight implements Serializable {

...

    @Version

    public Date getLastUpdate() { ... }

}

Hibernate can retrieve the timestamp value from the database or the JVM, by reading the value you specify for the @org.hibernate.annotations.Source annotation. The value can be either org.hibernate.annotations.SourceType.DB or org.hibernate.annotations.SourceType.VM. The default behavior is to use the database, and is also used if you don't specify the annotation at all.

The timestamp can also be generated by the database instead of Hibernate, if you use the @org.hibernate.annotations.Generated(GenerationTime.ALWAYS) annotation.

**Example 5.4. The timestamp element in hbm.xml**

<timestamp

        column="timestamp\_column"

        name="propertyName"

        access="field|property|ClassName"

        unsaved-value="null|undefined"

        source="vm|db"

        generated="never|always"

        node="element-name|@attribute-name|element/@attribute|."

/>

|  |  |
| --- | --- |
| column | The name of the column which holds the timestamp. Optional, defaults to the property namel |
| name | The name of a JavaBeans style property of Java type Date or Timestamp of the persistent class. |
| access | The strategy Hibernate uses to access the property value. Optional, defaults to property. |
| unsaved-value | A version property which indicates than instance is newly instantiated, and unsaved. This distinguishes it from detached instances that were saved or loaded in a previous session. The default value of undefined indicates that Hibernate uses the identifier property value. |
| source | Whether Hibernate retrieves the timestamp from the database or the current JVM. Database-based timestamps incur an overhead because Hibernate needs to query the database each time to determine the incremental next value. However, database-derived timestamps are safer to use in a clustered environment. Not all database dialects are known to support the retrieval of the database's current timestamp. Others may also be unsafe for locking, because of lack of precision. |
| generated | Whether the timestamp property value is generated by the database. Optional, defaults to never. |

**5.2. Pessimistic**

Typically, you only need to specify an isolation level for the JDBC connections and let the database handle locking issues. If you do need to obtain exclusive pessimistic locks or re-obtain locks at the start of a new transaction, Hibernate gives you the tools you need.

**Note**

Hibernate always uses the locking mechanism of the database, and never lock objects in memory.

**5.2.1. The LockMode class**

The LockMode class defines the different lock levels that Hibernate can acquire.

|  |  |
| --- | --- |
| LockMode.WRITE | acquired automatically when Hibernate updates or inserts a row. |
| LockMode.UPGRADE | acquired upon explicit user request using SELECT ... FOR UPDATE on databases which support that syntax. |
| LockMode.UPGRADE\_NOWAIT | acquired upon explicit user request using a SELECT ... FOR UPDATE NOWAIT in Oracle. |
| LockMode.READ | acquired automatically when Hibernate reads data under Repeatable Read or Serializable isolation level. It can be re-acquired by explicit user request. |
| LockMode.NONE | The absence of a lock. All objects switch to this lock mode at the end of a Transaction. Objects associated with the session via a call to update() or saveOrUpdate() also start out in this lock mode. |

The explicit user request mentioned above occurs as a consequence of any of the following actions:

* A call to Session.load(), specifying a LockMode.
* A call to Session.lock().
* A call to Query.setLockMode().

If you call Session.load() with option UPGRADE or UPGRADE\_NOWAIT, and the requested object is not already loaded by the session, the object is loaded using SELECT ... FOR UPDATE. If you call load() for an object that is already loaded with a less restrictive lock than the one you request, Hibernate calls lock() for that object.

Session.lock() performs a version number check if the specified lock mode is READ, UPGRADE, or UPGRADE\_NOWAIT. In the case of UPGRADE or UPGRADE\_NOWAIT, SELECT ... FOR UPDATE syntax is used.

If the requested lock mode is not supported by the database, Hibernate uses an appropriate alternate mode instead of throwing an exception. This ensures that applications are portable.

Automatic versioning is a simple technique for assuring data integrity.  
  
Consider the following: clients A and B load the same record R. After a time A commits back R with data changed, and after a while B commits too .. changing data of R - but not from the state of R after commit A.  
  
Automatic versioning does the following: a field is used to version the row data. In the above scenario A and B obtain the record R with version t. After the commit A the version is changed, so when B tries to commit the check upon the version fails.  
  
This is somehow similar to a simple one using timestamps - which is not recommended.

1. Hibernate issues an update statement like   
     
   UPDATE FOOS SET BAR='bar', VERSION=2  
   WHERE ID=1234 AND VERSION=1  
     
   (2) Hibernate checks the [JDBC](http://www.coderanch.com/forums/f-3/JDBC) row count, and throws a StaleObjectStateException if no rows were updated

### Optimistic

When using optimistic locking, you map a special attribute (a number, a timestamp) as a version (so you actually have a column for it). This version is read when you retrieve an entity and included in the where clause during an update and incremented by Hibernate.

To illustrate how this works, let's imagine you load a Person entity by id=1 and with a current version=1. After a save, Hibernate will perform something like this:

update PERSON set ID=1, NAME='NAME 1', VERSION=2 where ID=1 and VERSION=1;

So, now, imagine you have two concurrent transactions running, each of them loading the **same** entity (same version number) and changing the name.

Let's say transaction #1 is committed first, the following query is performed:

update PERSON set ID=1, NAME='NAME 1', VERSION=2 where ID=1 and VERSION=1;

It succeeds and the version gets incremented.

Then transaction #2 is committed, the following query is performed:

update PERSON set ID=1, NAME='NAME 2', VERSION=2 where ID=1 and VERSION=1;

This one won't update anything because the where clause won't match any record. This is where you'll get an optimistic concurrency exception.

This strategy is appropriate when you don't maintain the connection, when concurrent accesses are not frequent, and scales really well. And everything is of course handled transparently by Hibernate for you, as long as you map a version attribute.

### Pessimistic

When using pessimistic locking, Hibernate locks a record for your exclusive use until you have finished with it (typically using a SELECT ... FOR UPDATE). Any other concurrent transaction trying to access the same record will get suspended until the lock is removed. This strategy gives better predictability, at the price of performance and doesn't scale indefinitely.

**Note**:

* Remember friends, first we must run the logic to **save** the object then hibernate will inset 0 (Zero) by default in the version column of the database, its very important point in the interview point of view also
* First save logic to let the hibernate to insert zero in the version column, then any number of update logic’s (programs) we run, hibernate will increments +1 to the previous value
* But if we run the update logic for the first time, hibernate will not insert zero..! it will try to increment the previous value which is NULL in the database so we will get the exception.

# [Differences between Pessimistic and Optimistic Locking](https://mjabr.wordpress.com/2011/06/10/differences-between-pessimistic-and-optimistic-locking/)

Locking is an RDBMS feature that prevents users from different transactions from causing data conflicts. When locking is acquired on a row, it prevents other transactions from changing that row until the transaction ends. In this post I will explain the differences between pessimistic and optimistic locking in the context of ADF framework.

**pessimistic locking**:

Most Oracle developers are already familiar with pessimistic locking, which was the default locking in BC4J (now optimistic is the  default in 11.1.2.x)). This means that the row is locked in advance once one of  its attribute is changed through a call to setAttribute() method. If anyone else attempts to acquire a lock of the same row during this process, he will be forced to wait until the first transaction has completed. This is achieved by using the familiar SELECT…FOR UPDATE syntax. This is the safest locking mode because two transactions will never make inconsistent change to the same row. However, this locking mode has disadvantages such that:

1. If a user selects a record for update, and then leaves for lunch without  
   finishing or aborting the transaction. All other users that need to update that record are forced to wait until the user returns and completes the transaction, or until the DBA kills the offending transaction and releases the lock.
2. The Deadlock – Users A and B are both updating the database at the same time. User A locks a record and then attempt to acquire a lock held by user B – who is waiting to obtain a lock held by user A.

Pessimistic locking, which is the default, should not be used for web applications as it creates pending transactional state in the database in the form of row-level locks. If pessimistic locking is set, state management will work, but the locking mode will not perform as expected. Behind the scenes, every time an application module is recycled, a rollback is issued in the JDBC connection. This releases all the locks that pessimistic locking had created.

An example about pessimistic locking based on well known hr schema, suppose user1 and user2 are two different users (two distinct transactions) using pessimistic locking, both of them try to change the same row of data as follows:

1. User1 calls EmployeesImpl.setSalary(1000) on a particular row, so user1 immediately acquire a lock on that row.
2. Now user2 calls EmployeesImpl.setSalary(2000) on the same row, user2 tries to acquire a lock on the row and receives [oracle.jbo.AlreadyLockedException](http://download.oracle.com/docs/cd/E15051_01/apirefs.1111/e10653/oracle/jbo/AlreadyLockedException.html).

**Optimistic Locking:**

Optimistic locking assumes that multiple transactions can complete without affecting each other. Oracle recommends using optimistic locking for web applications. instead of locking a row as soon as it is changed, under optimistic locking, BC4J waits until changed row is posted before attempting to acquire a lock. An exception is not thrown until the conflicting transactions attempt to post their changes to the database.

An example about optimistic locking, suppose user1 and user2 are two different users (two distinct transactions) using optimistic locking, both of them try to change the same row of data as follows:

1. User1 calls EmployeesImpl.setSalary(1000) on a particular row, user1 does not immediately acquire a lock on that row.
2. User2 calls EmployeesImpl.setSalary(2000) on  the same row. User1 and User2 now have different entity cache for the same row.
3. User2 calls commit() action, as part of the commit cycle the changed row is posted to the database. before the update can be executed, user2 acquires a lock on that row. The lock expires immediately, when the commit command is sent to the database.
4. User1 now calls commit() action, BC4J tries to post the changed row to the database, right before posting it, it attempts to acquire a lock on that row. BC4J recognizes that the row has been changed by another user and that updating the row would overwrite another transaction’s changes, so it throws an [oracle.jbo.RowInconsistentException.](http://download.oracle.com/docs/cd/E12839_01/apirefs.1111/e10653/oracle/jbo/RowInconsistentException.html)

Finally, Whatever you use, you can lock a row at any time by calling EntityImpl.lock() on the corresponding entity object instance, even if the locking mode is optimistic.

# Locking in JPA

JPA 2 supports both **optimistic locking** and **pessimistic locking**. Locking is essential to avoid update collisions resulting from simultaneous updates to the same data by two concurrent users. Locking in ObjectDB (and in JPA) is always at the database object level, i.e. each database object is locked separately.

**Optimistic locking** is applied on transaction commit. Any database object that has to be updated or deleted is checked. An exception is thrown if it is found out that an update is being performed on an old version of a database object, for which another update has already been committed by another transaction.

When using ObjectDB, optimistic locking is [enabled by default](http://www.objectdb.com/java/jpa/setting/database#The_locking_element_) and fully automatic. Optimistic locking should be the first choice for most applications, since compared to pessimistic locking it is easier to use and more efficient.

In the rare cases in which update collision must be revealed earlier (before transaction commit) **pessimistic locking** can be used. When using pessimistic locking, database objects are locked during the transaction and lock conflicts, if they happen, are detected earlier.

This page covers the following topics:

* [Optimistic Locking](http://www.objectdb.com/java/jpa/persistence/lock#Optimistic_Locking_)
* [Pessimistic Locking](http://www.objectdb.com/java/jpa/persistence/lock#Pessimistic_Locking_)
* [Other Explicit Lock Modes](http://www.objectdb.com/java/jpa/persistence/lock#Other_Explicit_Lock_Modes_)
* [Locking during Retrieval](http://www.objectdb.com/java/jpa/persistence/lock#Locking_during_Retrieval_)

## Optimistic Locking

ObjectDB maintains a version number for every entity object. The initial version of a new entity object (when it is stored in the database for the first time) is 1. In every transaction in which an entity object is modified its version number is automatically increased by one. Version numbers are managed internally but can be exposed by defining a [version field](http://www.objectdb.com/java/jpa/entity/fields#Version_Field).

During [commit](http://www.objectdb.com/api/java/jpa/EntityTransaction/commit) (and [flush](http://www.objectdb.com/api/java/jpa/EntityManager/flush)), ObjectDB checks every database object that has to be updated or deleted, and compares the version number of that object in the database to the version number of the in-memory object being updated. The transaction fails and an [OptimisticLockException](http://www.objectdb.com/api/java/jpa/OptimisticLockException) is thrown if the version numbers do not match, indicating that the object has been modified by another user (using another [EntityManager](http://www.objectdb.com/api/java/jpa/EntityManager)) since it was retrieved by the current updater.

Optimistic locking is completely automatic and [enabled by default](http://www.objectdb.com/java/jpa/setting/database#The_locking_element_) in ObjectDB, regardless if a [version field](http://www.objectdb.com/java/jpa/entity/fields#Version_Field) (which is required by some ORM JPA providers) is defined in the entity class or not.

## Pessimistic Locking

The main supported pessimistic lock modes are:

* [PESSIMISTIC\_READ](http://www.objectdb.com/api/java/jpa/LockModeType/PESSIMISTIC_READ) - which represents a shared lock.
* [PESSIMISTIC\_WRITE](http://www.objectdb.com/api/java/jpa/LockModeType/PESSIMISTIC_WRITE) - which represents an exclusive lock.

### Setting a Pessimistic Lock

An entity object can be locked explicitly by the [lock](http://www.objectdb.com/api/java/jpa/EntityManager/lock_Object_LockModeType) method:

em.[lock](http://www.objectdb.com/api/java/jpa/EntityManager/lock_Object_LockModeType)(employee, [LockModeType](http://www.objectdb.com/api/java/jpa/LockModeType).[PESSIMISTIC\_WRITE](http://www.objectdb.com/api/java/jpa/LockModeType/PESSIMISTIC_WRITE));

The first argument is an entity object. The second argument is the requested lock mode.

A [TransactionRequiredException](http://www.objectdb.com/api/java/jpa/TransactionRequiredException) is thrown if there is no active transaction when [lock](http://www.objectdb.com/api/java/jpa/EntityManager/lock_Object_LockModeType) is called because explicit locking requires an active transaction.

A [LockTimeoutException](http://www.objectdb.com/api/java/jpa/LockTimeoutException) is thrown if the requested pessimistic lock cannot be granted:

* A PESSIMISTIC\_READ lock request fails if another user (which is represented by another EntityManager instance) currently holds a PESSIMISTIC\_WRITE lock on that database object.
* A PESSIMISTIC\_WRITE lock request fails if another user currently holds either a PESSIMISTIC\_WRITE lock or a PESSIMISTIC\_READ lock on that database object.

For example, consider the following code fragment:

em1.[lock](http://www.objectdb.com/api/java/jpa/EntityManager/lock_Object_LockModeType)(e1, lockMode1);

em2.[lock](http://www.objectdb.com/api/java/jpa/EntityManager/lock_Object_LockModeType)(e2, lockMode2);

em1 and em2 are two EntityManager instances that manage the same Employee database object, which is referenced as e1 by em1 and as e2 by em2 (notice that e1 and e2 are two in-memory entity objects that represent one database object).

If both lockMode1 and lockMode2 are PESSIMISTIC\_READ - these lock requests should succeed. Any other combination of pessimistic lock modes, which also includes PESSIMISTIC\_WRITE, will cause a [LockTimeoutException](http://www.objectdb.com/api/java/jpa/LockTimeoutException) (on the second lock request).

### Pessimistic Lock Timeout

By default, when a pessimistic lock conflict occurs a [LockTimeoutException](http://www.objectdb.com/api/java/jpa/LockTimeoutException) is thrown immediately. The "javax.persistence.lock.timeout" hint can be set to allow waiting for a pessimistic lock for a specified number of milliseconds. The hint can be set in several scopes:

For the entire [persistence unit](http://www.objectdb.com/java/jpa/entity/persistence-unit) - using a [persistence.xml](http://www.objectdb.com/java/jpa/entity/persistence-unit#persistence.xml) property:

<properties>

<property name="javax.persistence.lock.timeout" value="1000"/>

</properties>

For an [EntityManagerFactory](http://www.objectdb.com/api/java/jpa/EntityManagerFactory) - using the [createEntityManagerFacotory](http://www.objectdb.com/api/java/jpa/Persistence/createEntityManagerFactory_String_Map) method:

Map<String,Object> properties = new HashMap();

properties.put("javax.persistence.lock.timeout", 2000);

EntityManagerFactory emf =

Persistence.[createEntityManagerFactory](http://www.objectdb.com/api/java/jpa/Persistence/createEntityManagerFactory_String_Map)("pu", properties);

For an [EntityManager](http://www.objectdb.com/api/java/jpa/EntityManager) - using the [createEntityManager](http://www.objectdb.com/api/java/jpa/EntityManagerFactory/createEntityManager_Map) method:

Map<String,Object> properties = new HashMap();

properties.put("javax.persistence.lock.timeout", 3000);

EntityManager em = emf.[createEntityManager](http://www.objectdb.com/api/java/jpa/EntityManagerFactory/createEntityManager_Map)(properties);

or using the [setProperty](http://www.objectdb.com/api/java/jpa/EntityManager/setProperty_String_Object) method:

em.[setProperty](http://www.objectdb.com/api/java/jpa/EntityManager/setProperty_String_Object)("javax.persistence.lock.timeout", 4000);

In addition, the hint can be set for a specific [retrieval operation](http://www.objectdb.com/java/jpa/persistence/lock#Locking_during_Retrieval_) or [query](http://www.objectdb.com/java/jpa/query/setting#Query_Hints_Execution_Timeout_and_Lock_Timeout_).

### Releasing a Pessimistic Lock

Pessimistic locks are automatically released at transaction end (using either [commit](http://www.objectdb.com/api/java/jpa/EntityTransaction/commit) or [rollback](http://www.objectdb.com/api/java/jpa/EntityTransaction/rollback)).

ObjectDB supports also releasing a lock explicitly while the transaction is active, as so:

em.[lock](http://www.objectdb.com/api/java/jpa/EntityManager/lock_Object_LockModeType)(employee, [LockModeType](http://www.objectdb.com/api/java/jpa/LockModeType).[NONE](http://www.objectdb.com/api/java/jpa/LockModeType/NONE));

## Other Explicit Lock Modes

In addition to the two main pessimistic modes (PESSIMISTIC\_WRITE and PESSIMISTIC\_READ, which are discussed above), JPA defines additional lock modes that can also be specified as arguments for the [lock](http://www.objectdb.com/api/java/jpa/EntityManager/lock_Object_LockModeType) method to obtain special effects:

* [OPTIMISTIC](http://www.objectdb.com/api/java/jpa/LockModeType/OPTIMISTIC) (formerly [READ](http://www.objectdb.com/api/java/jpa/LockModeType/READ))
* [OPTIMISTIC\_FORCE\_INCREMENT](http://www.objectdb.com/api/java/jpa/LockModeType/OPTIMISTIC_FORCE_INCREMENT) (formerly [WRITE](http://www.objectdb.com/api/java/jpa/LockModeType/WRITE))
* [PESSIMISTIC\_FORCE\_INCREMENT](http://www.objectdb.com/api/java/jpa/LockModeType/PESSIMISTIC_FORCE_INCREMENT)

Since optimistic locking is applied automatically by ObjectDB to every entity object, the OPTIMISTIC lock mode has no effect and, if specified, is silently ignored by ObjectDB.

The OPTIMISTIC\_FORCE\_INCREMENT mode affects only clean (non dirty) entity objects. Explicit lock at that mode marks the clean entity object as modified (dirty) and increases its version number by 1.

The PESSIMISTIC\_FORCE\_INCREMENT mode is equivalent to the PESSIMISTIC\_WRITE mode with the addition that it marks a clean entity object as dirty and increases its version number by one (i.e. it combines PESSIMISTIC\_WRITE with OPTIMISTIC\_FORCE\_INCREMENT).

# [How DB locking system works in Hibernate (In Concurrency operations)](http://javacompleteexamples.blogspot.com/2009/07/how-db-locking-system-works-in.html)

Database *locking* can apply to any database applications.  
There are two common strategies when dealing with updates to database  
records, **pessimistic locking** and **optimistic locking**.  
"**Optimistic locking**" is more scalable than pessimistic locking when dealing with a  
highly concurrent environment. However pessimistic locking is a better solution for situations  
where the possibility of simultaneous updates to the same data by multiple sources  
(for example, different applications use same database to update) is common, hence making the possibility of "**data clobbering**" , a  
likely scenario. Lets look at a brief explanation of each of these two locking strategies.  
Pessimistic locking is when you want to reserve a record for exclusive update by  
locking the database record(entire table). Hibernate supports pessimistic locking  
(using the underlying database, not in-memory) via one of the following methods:  
  
***1. Session.get***  
***2. Session.load***  
***3. Session.lock***  
***4. Session.refresh***  
***5. Query.setLockMode***  
  
Although each of the methods accepts different params, the one common parameter  
across all is the LockMode class, which provides various locking modes such as **NONE,**  
**READ, UPGRADE, UPGRADE\_NOWAIT**, and **WRITE**.  
For example, to obtain a *Timesheet* record for updating,  
we could use the following code: (assume database supports locking system):  
  
***public Timesheet getTimesheetWithLock(int timesheetId)***  
***{***  
***Session session = HibernateUtil.getSessionFactory().getCurrentSession();***  
***session.beginTransaction();***  
***Timesheet timesheet = (Timesheet)session.get(Timesheet.class,***  
***new Integer(timesheetId), LockMode.UPGRADE);***  
***session.getTransaction().commit();***  
***session.close();***  
***return timesheet;***  
***}***   
Optimistic locking means that you will not lock a given database record or table and  
instead check a *property* of some sort (for example, a timestamp column) to  
ensure the data has not changed since *you read* it. Hibernate supports this using a  
version property, which can either be checked manually by the application or automatically  
by Hibernate for a given session. For example, the following code excerpt is taken  
verbatim out of the Hibernate reference documentation and shows how an application  
can manually compare the oldVersion with the current version using a getter method  
(for example):  
  
**// foo is an instance loaded by a previous Session**  
***session = factory.openSession();***  
***int oldVersion = foo.getVersion();***  
***session.load( foo, foo.getKey() );***  
 ***if ( oldVersion!=foo.getVersion )  
throw new StaleObjectStateException();   
  
foo.setProperty(“bar”);***  
***session.flush();***  
***session.connection().commit();***  
***session.close();***

===========================JDBC==================================

Here are the results of the experiments I have done so far:

**Experiment 1 - Rely on read uncommitted**

1. Read the db record
2. Lock the record by id in another table, as part of the global JTA transaction
3. Process the record A second transaction which tries to lock the same record will fail, will drop the record. But for this to work the RDBMS should allow dirty reads. Unfortunately Oracle does not support read uncommitted isolation level.

**Experiment 2 - Lock record in local transaction**

1. Read the db record
2. Lock the record by id in another table, as a separate local transaction
3. Process the record and delete the record when the transaction commits successfully A second transaction which tries to lock the same record will fail, will drop the record. This approach is based on committed data, should work fine. Here is the problem - Since the lock transaction and the global parent are different, if the processing fails rolling back the main transaction, I should compensate by rolling back the lock transaction, which I do not know how to do - **Need help here**

If Iam not able to rollback the record locking transaction, would have to write some dirty logic around the record locking code. I dont prefer this.

##### ransaction levels in JDBC - posted by Nishant

### What are different transaction Isolation levels in JDBC?

**Answer**Transaction Isolation levels is used to implement locking. They decide how one process isolated from other is. We have four Transaction Isolation Levels

Read Committed   
No two transactions can change the data at the same time. The shared lock is held for the record for the duration of the transaction and no other transaction can access the record.

Read Uncommitted   
This allows least restrictions to the record, meaning no shared locks and no exclusive locks. This kind offers best data concurrency but threat to data integrity.

Read Table Read  
This level allows new rows that can be inserted in the table and can even be read by the transactions.

Serializable  
This applies most restrictive setting holding shared locks on the range of data.

##### Transaction levels in JDBC - posted by Vidya Sagar

### Explain the different transaction Isolation levels in JDBC.

Transaction Isolation levels in JDBC:

The transaction levels specifies the data that is visible to the statements within a transaction.

The following are the transaction isolation levels :

**JDBC\_TRANSACTION\_NONE:**

This constant allows JDBC driver not to support transactions.

**JDBC\_TRANSACTION\_READ\_UNCOMMITTED**

Locking prevents **concurrent access** to some object.

# [Why is optimistic locking faster than pessimistic locking?](http://dba.stackexchange.com/questions/35812/why-is-optimistic-locking-faster-than-pessimistic-locking)

|  |  |  |  |
| --- | --- | --- | --- |
| up vote 3 down vote [favorite](http://dba.stackexchange.com/questions/35812/why-is-optimistic-locking-faster-than-pessimistic-locking)  **1** | Both forms of locking cause a process to wait for a correct copy of the record if its currently in use by another process. With pessimistic locking, the lock mechanism comes from the DB itself (a native lock object), whereas with optimistic locking, the lock mechanism is some form of row versioning like a timestamp to check whether a record is "stale" or not.  But both cause a 2nd process to hang. So I ask: why is optimistic locking generally considered faster/superior than pessimistic locking? And, are there are use cases where pessimistic is preferred over optimistic? Thanks in advance!  [performance](http://dba.stackexchange.com/questions/tagged/performance) [locking](http://dba.stackexchange.com/questions/tagged/locking) [rdbms](http://dba.stackexchange.com/questions/tagged/rdbms)   |  |  | | --- | --- | | [share](http://dba.stackexchange.com/q/35812)[improve this question](http://dba.stackexchange.com/posts/35812/edit) | asked Mar 2 '13 at 19:21  [[https://www.gravatar.com/avatar/80e3de8507f690c11e71eb96604265f5?s=32&d=identicon&r=PG](http://dba.stackexchange.com/users/20781/mara)](http://dba.stackexchange.com/users/20781/mara)  [Mara](http://dba.stackexchange.com/users/20781/mara) 12714 | |
|  | |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | | 3 |  | | A very short explanation exists in the naming. Optimistic locking works well when the chance of a conflicting lock is low. We are optimistic about the interaction of multiple processes. Pessimistic locking works well when the chance of a conflicting lock is high. We are pessimistic about the interaction of multiple processes. Both will perform sub-optimally where their opposite would be more appropriate. –  [Mark Storey-Smith](http://dba.stackexchange.com/users/2374/mark-storey-smith) [Mar 2 '13 at 21:37](http://dba.stackexchange.com/questions/35812/why-is-optimistic-locking-faster-than-pessimistic-locking#comment63399_35812) | | |  |  | | --- | --- | |  |  | | optimistic locking may or may not be faster than pessimistic locking, depending on your workload. –  [A-K](http://dba.stackexchange.com/users/4839/a-k) [Mar 4 '13 at 0:48](http://dba.stackexchange.com/questions/35812/why-is-optimistic-locking-faster-than-pessimistic-locking#comment63444_35812) |   add a comment |

## 4 Answers

[active](http://dba.stackexchange.com/questions/35812/why-is-optimistic-locking-faster-than-pessimistic-locking?answertab=active#tab-top) [oldest](http://dba.stackexchange.com/questions/35812/why-is-optimistic-locking-faster-than-pessimistic-locking?answertab=oldest#tab-top) [votes](http://dba.stackexchange.com/questions/35812/why-is-optimistic-locking-faster-than-pessimistic-locking?answertab=votes#tab-top)

|  |  |
| --- | --- |
| up vote 4 down vote accepted | Duplicate question from:  <http://stackoverflow.com/questions/129329/optimistic-vs-pessimistic-locking>  **Copy/Pasting answer from the above link:**  Optimistic Locking is a strategy where you read a record, take note of a version number and check that the version hasn't changed before you write the record back. When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit.  If the record is dirty (i.e. different version to yours) you abort the transaction and the user can re-start it.  This strategy is most applicable to high-volume systems and three-tier architectures where you do not necessarily maintain a connection to the database for your session. In this situation the client cannot actually maintain database locks as the connections are taken from a pool and you may not be using the same connection from one access to the next.  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. To use pessimistic locking you need either a direct connection to the database (as would typically be the case in a two tier client server application) or an externally available transaction ID that can be used independently of the connection.  In the latter case you open the transaction with the TxID and then reconnect using that ID. The DBMS maintains the locks and allows you to pick the session back up through the TxID. This is how distributed transactions using two-phase commit protocols (such as XA or COM+ Transactions) work.  **Edit (Adding more info to address the performance question):**  Performance wise it depends on your environment. Take in the following factors to decide:  you're going to find optimistic will be better due to concurrency in most situations. Depending on the RDBMS and environment this might be less or more performant however. Typically with Optimistic locking you will find that the value needs to be row versioned somewhere. |

**Pessimistic locking (pessimistic lock):**

Usually by the database mechanisms, the data locked in the whole process (query), as long as the transaction does not release (commit or rollback), any user can view and modify. Lock key is to resolve concurrency problems.

Achieved through jdbc sql statement as long as the entire statement after the increase for update can be. For example: select ... for update

Hibernate load a persistent class need to use this method of load method is overloaded ordinary load.

Such as: Student inv = (Student) session.load (Student class, "1001", LockMode.UPGRADE may be acquired upon explicit);

Then need to explain a problem, so that the locks can be of type the LOCKMODE value decision. Lock, Hibernate lazy strategy fails, then you made ??a sql statement, because of the need to load this object come in lock, so the SQL statement. When a pessimistic locking mechanism to load an object, the object is a series of operations, during the operation, that is, as long as the transaction is not submitted, the lock has been in existence. Another way to load the object (two object is an object, that is the only identifier values ??are the same) it will only issue a query, stops, because the previous method uses pessimistic locking mechanism to load this object, and not the end of the transaction (commit or rollback), then exclusive. When the first method to commit the transaction, the second method can be successfully loaded and executed in accordance with their own wishes all its operations.

Use pessimistic locking: pessimistic locking to solve the problem of lost updates (lost update), but it also brings the concurrency issues  concurrent bad. However, a demand, need a lot of very frequent concurrent operations, in particular, an operation or transaction will take up a lot of time, other operations can only be static and then the. Depends on the specific application.

**Hibernate lock mode:**

Ø LockMode.NONE: No lock mechanism.

Ø LockMode.WRITE: Hibernate Insert and Update record automatically

Get.   
Ø LockMode.READ: Hibernate read the record will automatically get.

Three lock mechanisms are generally used by the internal Hibernate, such as Hibernate in order to ensure that the Update

Process object will not be modified by the outside world, will automatically save method to achieve target object plus WRITE lock.

Ø LockMode.UPGRADE: database locked for update clause.

Ø LockMode. UPGRADE\_NOWAIT: Oracle-specific implementation of Oracle for

update nowait clause to achieve lock.

**Optimistic locking (optimistic):**

Optimistic locking is actually not a lock, so it is not locked, but added a field to the database table (the version number (version), you can also make a timestamp (timestamp)), or all fields / dirty field (in a manner suitable for the legacy of the past system, without changing the original table structure when using this strategy) to determine whether the data has been modified, the general application of the data version (version) implementation, read data when the version read out, save data to determine the version value is less than the value of the database version, less than updates are not allowed, otherwise it can be updated.

Version optimistic locking (recommended): a transaction commits will change the value of the version in the database, the database will automatically add 1. Implementation steps  1 version attribute in the persistent class to generate getter and setter methods. 2 the <class> tab in the configuration file to configure a property optimistic-lock = "version" (the property's default value is the version can not be configured, it is recommended configuration), the version field mapping, the use of version tag (the mapping of this field must be behind the id tag first)  <version   name = "version" />

A detailed analysis  loading a persistent class, a series of operations on this object in one transaction, but not to commit the transaction, At the same time, other things can also load the persistent class and completed a series of operation and submit the transaction, then the first transaction This is also the time to commit the transaction up, this will throw an exception  org.hibernate.StableObjectStateException: the Rows, WAS updated OR deleted by another transaction ... The reason is this: will issue a transaction commit such an SQL statement  update table set table properties =? where id (unique identifier value) =? and version =? version = this statement? is the most critical. In the example above, the first transaction to take the old version values ??for more updates submitted after the second transaction has changed the version into a new version, so that the first issued in a transaction when the transaction is committed update table set table properties =? WHERE id (unique identifier value) =? and version =? statement will fail! Throws the above exception information.

Optimistic locking: Suitable for high concurrency.