Locking

by Thomas Mahler, Armin Waibel

Table of contents

1. Introduction

Lock management is needed to synchronize concurrent access to objects from multiple transactions (possibly in clustered environments).

An example:
Assume there are two transactions `tx1` and `tx2` running. The first transaction `tx1` modify object `A` and perform an update. At the same time transaction `tx2` modify an object `A'` with the *same* identity `oidA`, so both objects represent the same row in DB table and both operate on the "same" row at the same time, thus the state of object with identity `oidA` is inconsistent.
Assume that `tx1` was committed, now the modified object `A'` in `tx2` based on outdated data (state before `A` changed). If now `tx2` commits object `A'` the changes of `tx1` will be overwritten with the "illegal" object `A'`.

The OJB lock manager is responsible for detecting such a conflict and e.g. doesn't allow `tx2` to read or modify objects with identity `oidA` as long as `tx1` commit or rollback (*pessimistic locking*). In other words, if in a running transaction an object in a with identity `oidA` has a *write lock*, the lock manager doesn't allow other transactions to acquire a *read* or *write lock* on the same identity `oidA` objects (for the sake of completeness: dependent on the used locking isolation level).

OJB supports two kind of locking strategies:

- optimistic locking
- pessimistic locking

OJB provide an pluggable low-level locking-api (located in `org.apache.ojb.broker.locking`) for *pessimistic locking*, which can be used by the top-level api's like ODMG. The PB-api itself does not support *pessimistic locking* out of the box.

The base classes of the locking-api can be found in `org.apache.ojb.broker.locking` and the entry point is class LockManager.

Object locking helps to guarantee data consistency without the need of database locks. During a transaction objects can be locked without the use a database connection, e.g the ODMG implementation lookup a database connection not until the transaction commit was called. If database locks are used, a connection is needed during the whole transaction.

2. Optimistic Locking

To control concurrent access to objects *optimistic locking* uses a version field on each persistent object.

Optimistic locking is supported by all API's (PB-api, ODMG-api, *JDO when it's done*).

Optimistic locking use an additional field/column for each persistent-object/table (*Long*, *Integer* or *Timestamp*) which is incremented each time changes are committed to the object, and is utilizied to determine whether an optimistic transaction should succeed or fail. Optimistic locking is fast, because it checks data integrity only at update time.

1. In your table you need a dedicated column of type `BIGINT`, `INTEGER` or `TIMESTAMP`. Say the column is typed as `INTEGER` and named `VERSION_MAINTAINED_BY_OJB`.
2. You then need a (possibly private) attribute in your java class corresponding to the column. Say the attribute is defined as:
   ```
   private int versionMaintainedByOjb;
   ```
3. in repository.xml you need a field-descriptor for this attribute. This field-descriptor must specify attribute `locking="true"`

4. The resulting field-descriptor will look as follows:

```
<field-descriptor
    name="versionMaintainedByOjb"
    column="VERSION_MAINTAINED_BY_OJB"
    jdbc-type="INTEGER"
    locking="true"
/>
```

Note:

Using of *TIMESTAMP* as optimistic locking field could cause problems, because dependent of the used operating system and database the precision of timestamp values differ (e.g. new value only after 10 ms or 1000 ms). In high concurrency applications this will cause problems.

3. Pessimistic-Locking

To control concurrent access to objects *pessimistic locking* uses shared and exclusive locks on persistent object (more precisely, on the identity object of the persistent object).
Pessimistic locking is currently used by the ODMG-api implementation. The PB-api does not support PL out of the box.

3.1. Supported Isolation Levels

The OJB locking package supports four different isolation level.

- read-uncommitted
- read-committed
- repeatable-read
- serializable
- (none)
- (optimistic)

The object locking isolation levels can be simply characterized as follows:

Uncommitted Reads
Obtaining two concurrent write locks on a given object is not allowed (case 14). Obtaining read locks is allowed even if another transaction is writing to that object (case 13). (Thats why this level is also called *dirty reads*, because you can read lock objects with an existing write lock).

Committed Reads
Obtaining two concurrent write locks on a given object is not allowed. Obtaining read locks is allowed only if there is no write lock on the given object (case 13).

Repeatable Reads
Same as commited reads, but obtaining a write lock on an object that has been locked for reading by another transaction is not allowed (case 7).

Serializable transactions
As Repeatable Reads, but it is even not allowed to have multiple read locks on a given object (case 6).

The isolation level *none* and *optimistic* are self-explanatory:
none - don't lock objects associated with this isolation level
optimistic - don't lock objects associated with this isolation level, because optimistic locking was used instead.
Thus the lock manager will ignore all objects associated with these isolation level.

Note:

It's not needed to declare the *optimistic* isolation level in all persistent objects class-descriptor using this isolation level, because OJB will automatically detect an enabled optimistic locking and will bypass pessimistic locking.
Only the proper settings for optimistic locking are mandatory.

Note:

The locking isolation levels named similar to the database transaction isolation level, but the definitions are different from it, so take care when comparing database transaction isolation level with object locking isolation level.

The proper behaviour of the different locking isolation level is checked by JUnit TestCases that implement test methods for each of the 17 cases specified in the above table. (See code for classes in package `org.apache.ojb.broker.locking` in OJB test suite).

The semantics of the strategies are defined by the following table:

Case	Name of TestCase	Transaction		Transaction-Isolationlevel			
		Tx1	**Tx2**	**ReadUncor**	**ReadComm**	**Repeatable**	**Serializable**
1	SingleRead	R		True	True	True	True
18	ReadThenF	R		True	True	True	True
		R					
2	UpgradeRe	R		True	True	True	True
		U					
3	ReadThenV	R		True	True	True	True
		W					
4	SingleWrite	W		True	True	True	True
5	WriteThenF	W		True	True	True	True
		R					
6	MultipleRea	R	R	True	True	True	False
7	UpgradeWi	R	U	True	True	False	False
8	WriteWithE	R	W	True	True	False	False
9	UpgradeWi	R	R	True	True	False	False
			U				
10	WriteWithM	R	R	True	True	False	False
			W				
11	UpgradeWi	R	R	True	True	False	False
		W					

12	WriteWithM	R	R	True	True	False	False
		W					
13	ReadWithE	W	R	True	False	False	False
14	MultipleWrit	W	W	False	False	False	False
15	ReleaseRe	R		True	True	True	True
		Rel	W				
16	ReleaseUp	U		True	True	True	True
		Rel	W				
17	ReleaseWri	W		True	True	True	True
		Rel	W				
	Acquire ReadLock	R					
	Acquire WriteLock	W					
	Upgrade Lock	U					
	Release Lock	Rel					

The table is to be read as follows. The acquisition of a single read lock on a given object (case 1) is allowed (returns True) for all isolationlevels. To upgrade a single read lock (case 2) is also allowed for all isolationlevels. If there is already a write lock on a given object for tx1, it is not allowed (returns False) to obtain a write lock from tx2 for all isolationlevels (case 14).

Note:

If the low-level locking api was used by hand:
Not all LockManager implementation support the `LockManager#upgrade(...)` method (e.g. upgrade was delegated to write lock) or behavior of this method is a wee bit other than shown above. More detail see javadoc comment of the used *LockManager* implementation.

3.2. How to specify locking isolation level

The locking isolation level can be specified *global* or *per class*.

The global setting is done in the descriptor-repository element:

```
<descriptor-repository version="1.0" isolation-level="read-uncommitted"
       proxy-prefetching-limit="50">
...
</descriptor-repository>
```

The isolation level of a class can be configured with the following attribute to a class-descriptor:

```
<ClassDescriptor isolation-level="read-uncomitted" ...>
    ...
</ClassDescriptor>
```

If no *isolation-level* was specified a default isolation level was used - see interface IsolationLevels. The semantics of isolation levels are described in isolation level section.

3.3. Specify the LockManager Implementation

To specify the used lock manager implementation set the *LockManagerClass* property in OJB.properties file. By default an in memory lock manager is enabled.

```
LockManagerClass=org.apache.ojb.broker.locking.LockManagerInMemoryImpl
 ...
```

3.4. The LockManager Implementations

Below all LockManager implementations shipped with OJB are listed.

The *LockManager* implementation can optionally support

- lock timeout: The locked objects of an *owner* will be released after a specified time
- block timeout: The maximal time to wait for acquire a lock (e.g. when an object was locked by another thread). Implementations which do not support this feature are called *non-blocking*

3.4.1. LockManagerInMemoryImpl

A *non-blocking*, single JVM, in-memory *LockManager* implementation. All `LockManager.upgradeLock(...)` calls are delegated to write locks. It's a simple and fast implementation.

The timeout of locks is supported. The block timeout is ignored, because it's non-blocking.

3.4.2. LockManagerCommonsImpl

This implementation use the locking part of apache's commons-transaction api. The timeout of locks is currently (OJB 1.0.2) not supported, maybe in further versions. This implementation supports *blocking* (with deadlock detection) and *non-blocking* of acquired locks.

3.4.3. LockManagerRemoteImpl

Supports locking in distributed environments based on a servlet. The *LockManagerRemoteImpl* class delegates all locking calls to a remote servlet (`LockManagerServlet`). The URL to contact the servlet have to be set in OJB.properties file using the *LockServletUrl* property, e.g.

```
LockServletUrl=http://127.0.0.1:8080/ojb-lockserver
```

To make deployment of the `LockManagerServlet` on a servlet container easier an Ant target *lockservlet-war* exist, which will build an example *.war* file containing all needed files (maybe some useless files) for deployment.
The generated `web.xml` file look like:

```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

    <display-name>OJB ODMG Lock Server</display-name>
    <description>
```

```
     OJB ODMG Lock Server
    </description>

    <servlet>
        <servlet-name>lockserver</servlet-name>
<servlet-class>org.apache.ojb.broker.locking.LockManagerServlet</servlet-class>
        <init-param>
            <param-name>lockManager</param-name>
<param-value>org.apache.ojb.broker.locking.LockManagerInMemoryImpl</param-value>
        </init-param>
        <init-param>
            <param-name>lockTimeout</param-name>
            <param-value>80000</param-value>
        </init-param>
        <init-param>
            <param-name>blockTimeout</param-name>
            <param-value>1000</param-value>
        </init-param>

        <!--load-on-startup>1</load-on-startup-->
    </servlet>

    <!-- The mapping for the webdav servlet -->
    <servlet-mapping>
        <servlet-name>lockserver</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!-- Establish the default list of welcome files -->
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
    </welcome-file-list>
</web-app>
```

It's possible to use each *LockManager* implementation as backend of the lock manager servlet - only adapt the *lockManager* init-param entry in the `web.xml` file.

4. ODMG-api Locking

The OJB ODMG implementation provides object level transactions as specified by the ODMG. This includes features like registering objects to transactions, persistence by reachability (a toplevel object is registered to a transaction, and also all its associated objects become registered implicitely) and as a very important aspect: object level locking.

The ODMG locking implementation is located in `org.apache.ojb.odmg.locking` and base on the OJB kernel locking code in `org.apache.ojb.broker.locking`. The odmg implementation use it's own internal locking interface `org.apache.ojb.odmg.locking.LockManager` with specific methods to handle transactions as owner of a lock and persistent object Identity objects as resources to lock..

4.1. What it does

The ODMG-Api allows transactions to lock an object `obj` as follows:

```
org.odmg.Transaction.lock(Object obj, int lockMode)
```

where lockMode defines the locking modes:

```
org.odmg.Transaction.READ
org.odmg.Transaction.UPGRADE
org.odmg.Transaction.WRITE
```

A sample session could look as follows:

```
// get odmg facade instance
Implementation odmg = ...

//get open database
Database db = ...

// start a transaction
Transaction tx = odmg.newTransaction();
tx.begin();

MyClass myObject = ... ;

// lock object for write access
tx.lock(myObject, Transaction.WRITE);

// now perform write access on myObject ...

// finally commit transaction to make changes to myObject persistent
tx.commit();
```

The ODMG specification does not say if locks must be acquired explicitely by client applications or may be acquired implicitely. OJB provides implicit locking for the application programmers convenience: On commit of a transaction all read-locked objects are checked for modifications. If a modification is detected, a write lock is acquired for the respective object. If automatic acquisition of read- or write-lock failes, the transaction is aborted.

On locking an object to a transaction, OJB automatically locks all associated objects (as part of the *persistence by reachability feature*) with the same locking level. If application use large object nets which are shared among several transactions acquisition of write-locks may be very difficult. Thus OJB can be configured to aquire only read-locks for associated objects.
You can change this behaviour by modifying the file OJB.properties and changing the entry `LockAssociations=WRITE` to `LockAssociations=READ`.

The ODMG specification does not prescribe transaction isolation levels or locking strategies to be used. Thus there are no API calls for setting isolation levels. OJB provides four different isolation levels that can be configured global or for each persistent class in the configuration files.

5. Locking in distributed environment

In distributed or clustered environments the object level locking (pessimistic locking) have to be consistent over several JVM. The optimistic locking works in clustered/distributed environments without any modifications.

Currently OJB was shipped was simple servlet based LockManager implementation called LockManagerRemoteImpl.

Here is a description how to use it:

1. Change *LockManagerClass* entry in OJB.properties file to the remote implementation: `org.apache.ojb.broker.locking.LockManagerRemoteImpl` and the `LockServletUrl` to the servelt engine where the lock-server servlet will be deployed:

```
LockManagerClass=org.apache.ojb.broker.locking.LockManagerRemoteImpl
...
LockServletUrl=http://127.0.0.1:8080/ojb-lockserver
```

2. Run the `ant lockservlet-war` target to generate the lock-server servlet `.war` application file. The generated file will be found in `[db-ojb]/dist`.

3. Check that all needed libraries be copied in lockservlet-war file.

This implementation has some drawbacks, e.g. it uses one servlet node to deploy the LockMap servlet.

A much better solution will be a JMS- or JavaGroups-based LockManager implementation (hope we can start working on such a implementation some day).

6. Pluggin own locking classes

OJB was shipped with several locking classes implementations.
This may not be viable in some environments. Thus OJB allows to plug in user defined LockManager implementations.
To specify specific implementations change the following entry in the OJB.properties configuration file:

```
LockManagerClass=my.ojb.LockManagerMyImpl
```

Note:
Of course we are interested in your solutions! If you have implemented something interesting, just contact us.