Overview

Data integrity is most vital for any transaction. The way in which the concurrent access to data is managed is governed by concurrency control. It addresses conflicting requirements of applications that read data against those of applications that update it.

Pramati Server offers a choice between pessimistic and optimistic concurrency control for CMP entity beans. Application server resource management modules typically provide mechanisms that enable applications to influence how a resource manager manages concurrency control.

Typical resource managers implement a form of locking to provide concurrency control. An application deployer may then use one of the two common concurrency control modes: pessimistic or optimistic. Pramati Server provides a locking system for the pessimistic concurrency control method. There are no locks available for the Optimistic Concurrency control method. Using locks synchronizes pessimistic Concurrent transactions. These locks are maintained by the application server or pushed to the underlying persistent store. Such a mechanism is useful when there are relatively more concurrent transactions on the same entity identity.

In case of optimistic concurrency, clients are written to handle possible rollbacks of some of the parallel transactions. Clients of the entity with pessimistic concurrency have no such limitations.

Pessimistic Concurrency Control

An application server that will block an entity bean from being read by some transaction if some other transaction is currently working with that entity bean is explicitly managing concurrent access to the entity bean, and can thus be said to use pessimistic concurrency. These types of application servers usually enforce pessimistic concurrency by having only one entity bean in-memory, per primary key. Serializing access to this entity bean then becomes easier.

Optimistic Concurrency Control

The serialization of transaction severely limits the scalability of application. When the number of concurrent users increases, the number of parallel transactions also increases. But serialization of a transaction will make other transactions wait for the currently executing transaction to complete. Therefore to achieve high scalability, there should be a way to
execute these transactions concurrently without compromising on the ACID properties of the transaction.

When transactions are executed in parallel, they are called concurrent transactions. This mechanism of execution is known as optimistic concurrency control. The main advantage of optimistic concurrency control method is faster response time, but on the downside you cannot be sure that the action will be performed until the lock has been accepted or rejected. This depends on the probability of concurrent access. For example, if there are many users the probability of concurrent access is greater and so pessimistic concurrency control might be used, whereas if many users don't exist, concurrent access will be rarer so optimistic concurrency control might be better used. By default the pessimistic mode is selected.

**Lock Management**

Pramati Server has an elaborate lock manager framework that manages the locks obtained by the container on various entities. The lock manager framework is used to determine which lock mechanism is used.

**Local Lock Server (LLS)**

This is a lock server running within a standalone Pramati server (or in each node of a cluster). The lock-server is used to lock resources within the single server/node. The local lock server is used to lock Entity Beans in standalone servers.

**Super Lock Server (SLS)**

SLS is specific to Bean Managed Persistence. Super-lock-servers are used in the Pramati cluster. The super-lock servers work with the local lock servers. The lock manager manages the access and the cluster semantics. When a client requests access to a particular object on a node, the lock manager checks for compatibility with the existing locks. LLS acquire a lock on a resource from the SLS. All further lock accesses to that resource are managed from within the LLS until another node in the cluster, for example node 1, requests for the same resource. Node 1 then requests the SLS for the lock. SLS then requests the LLS of node 1 to release the lock and passes the lock to node 2.

**DB Lock Module**

The DB Lock module enables in locking resources by setting UPDATE locks on the database. When a CMP bean requires to be locked, SELECT FOR UPDATE SQL statement can be used. This acquires a lock on the database. The lock is released only when the transaction completes.

**Locking in Clusters**

In case of clusters, whenever a client requests access to a particular object in a specified node, the lock manager checks for the compatibility with the existing locks. A lock on a resource is acquired from the SLS to the LLS. Once acquired, all further lock accesses to that resource is managed from within the LLS. Until one of the other nodes (for example: Node 1 in the cluster needs the same
resource.) This node then requests the SLS for the lock, and the SLS will request the Node 1 LLS to release the lock and then SLS passes the lock to Node 2.

**Commit Options**

The J2EE specification does not require the application to necessarily have exclusive database access, but it provides various Commit Options (A,B,C) specified in 9.1.10 section of the EJB 1.1 specification and section 10.5.9 of EJB 2.0 specification.

Commit Options:
- Non-Exclusive
- Exclusive

The default option is non-exclusive, which can be set to exclusive if the container has exclusive access to DB.

**Non-exclusive**

This is applicable for both standalone and clustered server nodes. Since the application does not have exclusive access to the database, the bean is always loaded fresh from the database at the beginning of the transaction.

**Performance Optimization**

When bean activation and de-activation are not expensive operations, commit option can be set to non-exclusive. The bean gets freshly loaded every time in the beginning of transaction even if it was in ready state. So the bean pool size can be kept smaller, reducing memory consumption. Clustered deployment always works in non-exclusive mode.

**Exclusive**

In the Exclusive mode no external entity application will access or modify the database. This is the default mode in stand-alone servers.

**Performance advantage**
- `ejbLoad()` callback avoided at the beginning of the transaction.
- Bean state need not be synched with the DB.

**Performance Optimization**

If the commit option is set to exclusive, the expensive EJB Loads can be avoided if the bean is in ready State. Hence, for higher performance, the bean pool size for a particular bean should be set equal to the DB size if the DB is of reasonable size like <5000, even if the concurrency is not so high on such beans to justify a pool size so high. If the DB is large for the given bean and the access pattern is quite random, then a higher pool size with exclusive DB access will not greatly enhance the performance. A pool size as high as the database size for big data is not advisable, since exceptionally higher pool size for a bean implies larger size data structures maintained by the
container. This would imply higher search times and higher consumption of memory too. Hence, it would be wiser to load when the bean is not found in the ready state. With Pramati server you can choose between Commit Option A and B.

**Concurrent Access to EJB**

Enterprise applications often try to balance the application scalability with concurrency issues. To improve the concurrent access for entity enterprise beans, Pramati Server allows different concurrency modes. It enables pessimistic concurrency at the application server level or defers locking services to the corresponding database. When writing the entity bean business methods, the bean provider does not have to worry about concurrent access from multiple transactions. The provider may assume that the container will ensure appropriate synchronization for entity objects that are accessed concurrently from multiple transactions. Usage of locks synchronizes concurrent transactions. These locks are maintained by the application server or pushed to the underlying persistent store. This mechanism is useful when there are relatively more concurrent transactions on the same entity identity.

**Repeatable Read**

This is the default setting for entity beans deployed in Pramati EJB Container. A single bean instance is created irrespective of number of concurrent requests; the application server takes care of serializing requests to this particular instance. A load happens before the beginning of a transaction, if the application doesn't have exclusive access to the database while in the standalone mode. The state of the bean is always updated based on the outcome of an invalid check.

**Case 1: Exclusive DB**

In exclusive lock on the instance's state in the DB is postponed till a `ejbStore()` callback happens as part of commit on a transaction. A delayed lock implies connections keep locked for limited time. Since it is a limited resource usage, it results in higher performance.

**Case 2: Non-Exclusive DB**

At the beginning of a transaction, a bean is loaded fresh and an exclusive lock is acquired on the particular row in db (`SELECT FOR UPDATE`). Connections are locked up for the complete transaction starting from beginning to end.

**Optimistic Repeatable Read**

For multiple instances of beans created one per each concurrent transaction, the methods in the beans execute concurrently. Bean's state may get loaded fresh from DB depending on if the bean has exclusive access to DB. But on committing a transaction, a verified update is done always irrespective of if the bean state got modified as part of the transaction. The container validates the state of the cache against the database as follows:
• If the bean was modified in a transaction, a verified update is performed. That is, the instance is updated with the new values after ensuring that the data that was read is still as is (and unmodified by any other transaction) in the database.

• If the bean was not modified, the container will perform a verified update to ensure that no other transaction has modified the instance since the load of the data by this transaction.

• In both cases, if the verified update fails, then the transaction is rolled back.

• When two or more transactions happen at the same time, the first transaction will be allowed to change the bean and the other transactions will be rolled back. In such cases, the owner has to retry them.

Read Committed
For multiple instances of bean created one per each concurrent transaction, the methods in the beans execute concurrently. Bean’s state always gets loaded fresh from DB in the beginning of transaction. At the time of committing, the transaction the bean’s CMF are compared and an update is done only if the state is illegal.
## Concurrency Control Strategies

Table 1: Concurrency control strategies

<table>
<thead>
<tr>
<th>Concurrency Control</th>
<th>Read Committed</th>
<th>Repeatable Read</th>
<th>Optimistic Repeatable Read</th>
<th>Read Only</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exclv</td>
<td>Non-exclv</td>
<td>Exclv</td>
<td>Non-exclv</td>
<td>Exclv</td>
</tr>
<tr>
<td>Load</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>App server locks</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DB locks</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes (if DB locks supported)</td>
<td>No</td>
</tr>
<tr>
<td>Deadlocks</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Tx rollbacks</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes (if DB locks supported)</td>
<td>Yes</td>
</tr>
<tr>
<td>Multiple bean access</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Multiple threads wait</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Range of concurrent Tx</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Nature of concurrent Tx</td>
<td>High read, low update</td>
<td>High read, low update</td>
<td>Any read, high update</td>
<td>Any read, high update</td>
<td>Any (prefer high read, low update)</td>
</tr>
</tbody>
</table>

**Terms used in the table**
- **Tx**: transaction. **Exclv**: Exclusive. **Non-exclv**: Non-exclusive. **DB**: Database. **App**: Application. **RC**: Read Committed. **ORR**: Optimistic Repeatable Read.
Summary

In this chapter we saw the various concurrency control mechanisms provided by Pramati Server. We looked at what pessimistic and optimistic concurrency control are. Pramati Server has an elaborate lock manager to manage locks that are obtained by the container. We discussed the various commit options that are provided by the container, and those specified in the EJB Specifications. Finally, we discussed the various concurrent access provided to EJBs such as repeatable read, optimistic repeatable read and read committed.