Best Practices to improve performance in EJB

This topic illustrates best practices to improve performance in EJB with the following sections:

Overview of EJB

Enterprise Java Bean is a component that simplifies the job of developer by providing the following built-in services

1. RMI/RMI-IIOP (Distributed computing)
2. Transactions
3. Security
4. Resource pooling
5. Thread safe
6. Client state
7. Persistence
8. Messaging
9. Life cycle etc.

As a developer, you need not write infrastructure code for these services, rather vendors provide EJB container/server that support all these services. You simply use these services without writing any code for them. But all these services do not come for free , they are expensive and come with a price. So before writing an application you need to carefully look into all the aspects for optimum use of your resources and calculate the tradeoffs involved.

This section focuses on how to improve performance when using the following types of EJBs.

1. Stateless session bean
2. Stateful session bean
3. Entity bean
4. Message driven bean

This section focuses on EJB 1.1 and EJB 2.0 specification. Initially we will discuss common tuning practices among some/all types of beans and then discuss each of them separately.

Note: This Section assumes that reader has some basic knowledge of EJBs.

Common tuning practices in EJBs

Here we discuss common tuning practices among some/all types of enterprise java beans.

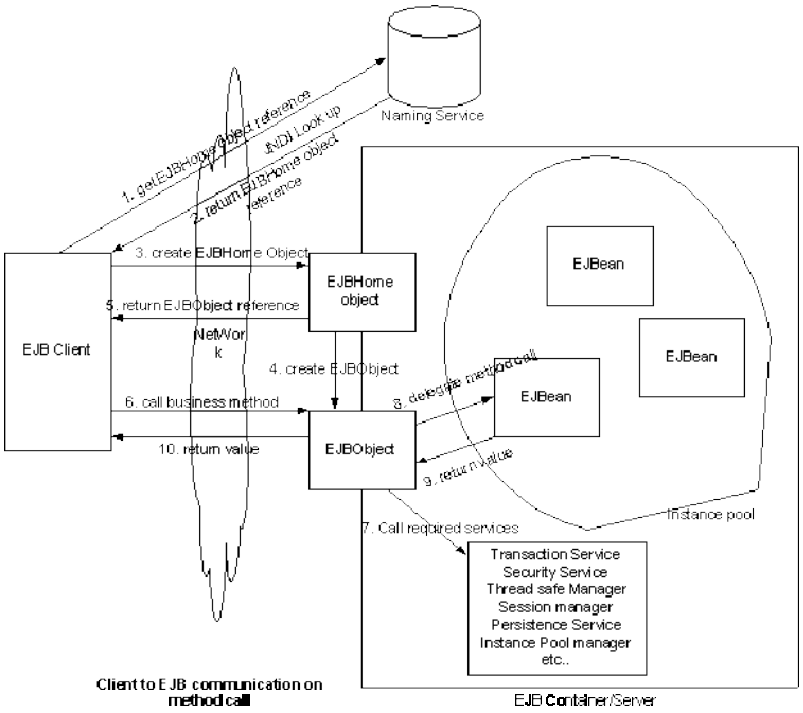
Choosing between EJB and non-EJB

The first thing that you can question yourself is why use EJB and why not a normal java object since EJB with services such as transactions, RMI/RMI-IIOP, security, client state management, persistence etc are associated with a cost. If you want these services for your application then EJB is the best choice otherwise a normal java object is a better option. And also the decision between them depends on how many services you want for your application, if you want only transaction management then you can use JTA (Java Transaction API) rather than using EJB or incase you want your application to have only distributed computing then RMI/RMI-IIOP will suffice without EJB.

EJB is a reusable component unlike normal java object that is backed by component architecture, you can deploy an EJB in any EJB compliant server, that can be reused.

The following figure illustrates the underlying technology about how a client communicates with an EJB :

Figure1: Client to EJB communication



This figure gives a clear picture about how the communication between client and the EJB is achieved, the overall process is expensive.

So use java object if you do not want some/all the services that come with an EJB, otherwise choose EJB.

Choosing between local EJB and remote EJB

Once you decide to use an EJB application the first thing that you have to decide is choosing between local/remote EJB. EJB 2.0 introduced the concept of local interfaces which work as local EJB without distributed computing capability (RMI/RMI-IIOP) while EJB 1.1 supports only remote EJBs.

Remote EJB means that it extends remote interface so that client communicates with EJB through RMI/RMI-IIOP which involves serialization, marshalling/unmarshalling for every method call and this has costs associated which can be avoided if the client is on the same machine/JVM.

So if you want to deploy your EJB client and EJB in the same server/JVM then you don't need the remote EJB but can use the local EJB .The local EJB does not use RMI/RMI-IIOP for communication, it bypasses this overhead and uses pass-by-reference as normal java mechanism and hence is faster and efficient when compared to remote EJBs.

The following figures illustrate the difference between local and remote EJBs.

Figure2: Remote EJB

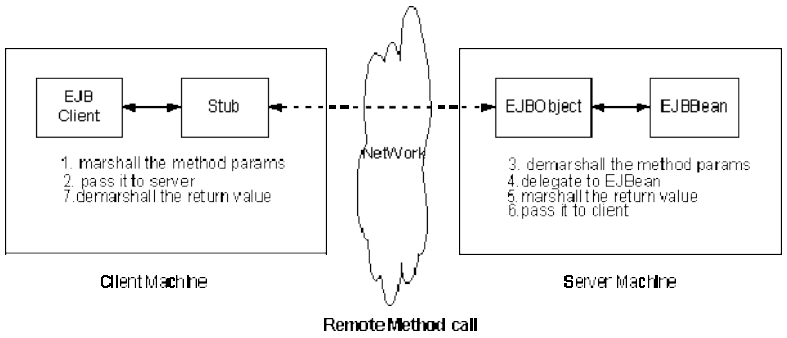
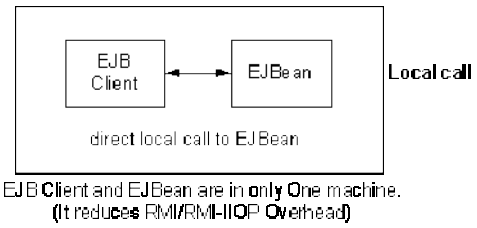


Figure3: Local EJB



In EJB2.0, use Local interfaces to reduce RMI/RMI-IIOP overhead.

Optimize with pass-by-reference

Unlike EJB2.0, EJB 1.1 does not support local interfaces so you have only one option that is to treat all objects as remote objects. If you want to deploy client and EJBean on the same machine then also you cannot avoid the overhead of the objects being treated as remote in spite of being local. However some EJB servers solve this problem if you configure their deployment descriptor files. For example in Weblogic server's weblogic-ejb-jar.xml file has an element to define wether the element is to be passed by reference or by value. That element is

<enable-call-by-reference > true</enable-call-by-reference >

By default this element is set to true.

In JBoss servers the jboss.xml file has an element

```
<container-invoker-conf>
    <optimized>true</optimized>
</container-invoker-conf>
```

Some vendors may not support pass-by-reference as default so look at your server documentation for detailed information and configure.

In EJB1.1, configure the pass-by-reference property in vendors specific deployment descriptor file to make EJBs as local EJBs whenever you deploy both client and EJBs in the same machine/server to improve performance.

Wrap entity beans with session bean

If you call an entity bean remotely from java swing, JSP or a servlet, you will end up with a number of remote network calls that increase network overhead. Wrapping entity beans with session beans solves this problem. The best method is to make entity beans as local EJBs and wrap with session bean. Generally to accomplish a request we need to call a lot of entity beans, for example if client wants to display an order with account info then we need to call PersonEntityBean, OrderEntityBean, and AccountEntityBean. Here three remote calls are involved, we can minimise the number of remote calls by wrapping these three entity beans with an OrderManager session bean thus requiring only one remote call. Session bean gets the request and processes three calls locally and returns values in a single call. To make entity beans as local you can follow the above mentioned approach depending on the EJB version.

The following figure illustrates this process :

Figure4: calling entity bean directly through network

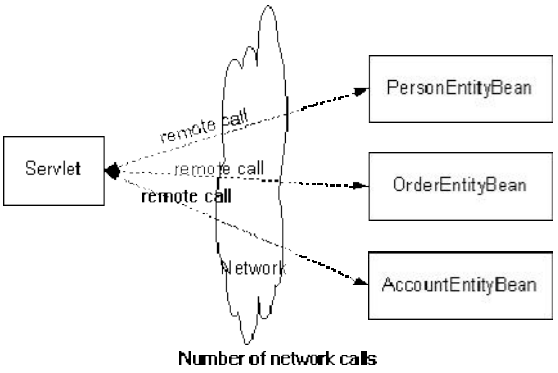
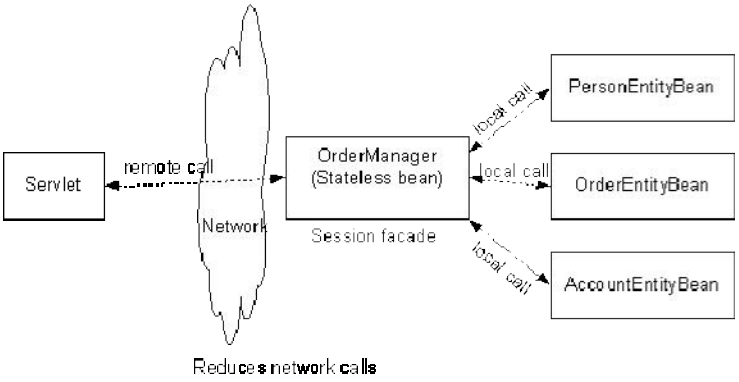


Figure5: Wrap entity bean with session bean to reduce network calls



So to reduce network calls and increase performance, wrap entity beans with session beans. This technique not only improves performance but also gives good declarative transaction support and design.

There is a pattern available for this process, see Session Facade.

Reduce method calls with coarse granularity

When you call a methods of remote object for example like this :

```
remoteObject.getName();
remoteObject.getCity();
remoteObject.getState();
remoteObject.getZipCode();
```

Here, there are four network calls from client to the remote object. You can minimize these network calls by reducing the number of method calls like this :

```
// create an object and fill that object locally
PersonInfo person = new PersonInfo();
person.setName("name");
person.setCity("Austin");
person.setState("TX");
person.setZipCode("78749");
remoteObject.getPersonInfo(person); // send object through network
```

Here, there is only one network call instead of three network calls.

In the first case we set attribute values one by one and send them over the network .This approach is called fine grained approach,this approach needs more network calls.

In the second approach we wrap all the four attributes in an object at the client side and send it as an object rather than each attribute separately. This approach is known as coarse grained approach. This approach when compared to fine grained approach needs less network calls.

So use coarse grained approach to minimize the number of network calls there by improving performance .

Control serialization in remote EJBs

When you decide to write your code for distributed /remote object you need to carefully choose what method parameters you want to send over the network,for example when you pass an object like this :

```
remoteObject.setPersonInfo(person); // call remote object by passing object
```

here, not only the PersonInfo object will be serialized and sent over network but all the total PersonInfo

object graph (variables and it's super class variables except transient variables) will also be sent through network because of default behavior. You might want to send only PersonInfo object only but not total object graph.

To avoid this problem, use 'transient' key word for the attributes that need not be sent over the network. See Serialization section for more detailed information.

Cache EJBHome object references

When you want to call an EJBean from the client, first you need to get EJBHome object reference through JNDI , This is illustrated in the figure above, the code for this process is shown below :

```
// get InitialContext
public Context getInitialContext() throws NamingException{
      Properties props= new Properties();
      props.put (Context.PROVIDER_URL,"urlname");
      props.put(Context.INITIAL_CONTEXT_FACTORY,"nameservice");
      return new InitialContext(props);
}
// get home reference from JNDI
public AccountHome getHome () throws NamingException{
      Context ctx = setInitialContext();
      Object home=ctx.lookUp("JNDIlookupName);
      return (AccountHome) PortableRemoteObject.narrow(home,AccountHome.class);
}
```

This code needs to be executed each time the client wants to call an EJBean. This is redundant and expensive. So to solve this problem, you can cache EJBHome object references first time and reuse it again and again from cache without repeating the JNDI look up process.

The following code snippet shows how you can cache EJBHome object references.

```
import javax.ejb.*;
import javax.rmi.*;
import java.util.*;
import javax.naming.*;
public class EJBHomeCache {
        // cache home references in Hashtable
        private static Hashtable homes = new Hashtable();
        Context ctx;
  public EJBHomeCache() throws NamingExeption {
  }
  public static synchronized EJBHome getHome(Class homeClass) throws NamingExeption {
        EJBHome home = (EJBHome) this.homes.get(homeClass);
        if(home == null) {
          ctx = getInitialContext();
          home = (EJBHome) PortableRemoteObject.narrow(ctx.lookup(homeClass.getName()),homeClass);
          this.homes.put(homeClass,home);
        }
        return home;
  }
  private Context getInitialContext() throws NamingException {
     Properties props= new Properties();
     props.put (Context.PROVIDER_URL,"urlname");
     props.put(Context.INITIAL_CONTEXT_FACTORY,"nameservice");
     return new InitialContext(props);
  }
} //end class
// client code for getting AccountHome object reference
AccountHome accountHome = (AccountHome)EJBHomeCache.getHome(AccountHome.Class);
```

Here we are getting Home object reference the first time and putting it in the cache and reusing it without doing expensive JNDI look up second time onwards.There is a pattern available for this technique, see EJBHomeFactory Pattern (Service Locator Pattern) for detailed implementation.

Control transaction

In EJB, You can use transactions either declaratively or programmatically.

The declarative transactions in EJB are at method level that means transaction starts (begins) when method starts and transaction ends (commits) when method ends. And also transaction propagates into the sub methods if the parent method uses these sub methods. For example, if you write a session bean method that calls four of the entity bean methods, transaction starts when the session method begins and transaction ends when that method ends, in between transaction propagates into four of the entity bean methods and gets back to session bean method. It works like chain of transaction propagations.

Declarative transactions have six transaction attributes they are Required, RequiredNew, Mandatory, Supports, NotSupported and Never. You need to declare any of these attributes in ejb-jar.xml deployment descriptor file for each method like this.

```
<ejb-jar>
<session><transaction-type>Container<<transaction-type><session>
<assembly-descriptor>
   <container-transaction>
      <method>
         <ejb-name>EJBName</ejb-name>
         <method-name>methodName / *</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
   </container-transaction>
</assembly-descriptor>
</ejb-jar>
```

You can give each method name separately or you can give asterisk(*) for all the methods.

The problem with declarative transactions is they look innocent, because you declare transaction attributes in deployment descriptor file instead of writing code for a transaction. For example if you use asterisk (*), all methods are involved in the transaction. Why do we need a transaction for non transactional methods? it is unnecessary and takes more time to execute every method. So you need to control transaction to avoid unnecessary transaction propagation on every method. What you can do is that you can divide a bean's methods into transactional methods and non transactional methods and assign transaction attributes to only transactional methods, assign 'NotSupported' or 'Never' to non transactional methods so that you avoid transaction propagation. But note that 'NotSupported' or 'Never' attributes can't be used for entity beans because entity beans need to involve in transaction to commit data, so use these attributes for session bean's non transactional methods. In this process you are controlling transaction propagation if any method uses other session beans but you have to be careful whether sub beans need a transaction or not.

You can write your own transactional code (Bean Managed transaction) using javax.transaction.UserTransaction interface that has methods like begin(), commit(), rollback() etc to write transactional code.

And one more thing is that the transaction mechanism should span for minimum time possible because transaction locks the database data till it completes and it does not let other clients access this data.

So control transaction by avoiding transactions for non-transactional methods.

Set optimal transaction age

If you declare a transactional attribute, how long does a transaction run?. If you want to make sure that your transaction should take place with in a certain time limit, set the transaction time out (age) value in vendor specific deployment descriptor file. For example in weblogic server's weblogic-ejb-jar.xml file has an element

```
<transaction-descriptor>
      <trans-timeout-seconds>120</trans-timeout-seconds>
</transaction-descriptor>
```

Look at vendor's documentation for other severs. So set maximum transaction time by this process to control maximum time out of a transaction, make sure that the time-out you set for transactionis appropriate.

Use clustering for scalability

What happens if you deploy an EJB application and later the number of clients increase, your hardware resources might not be enough to handle the increased number of clients and the response time may go up thus degrading the performance. To avoid such situations we need to go for clustering.

Clustering is a group of servers that service the clients, but the client is unaware of presence of multiple servers he feels that he is interacting with a single server. Once you deploy an application in clustered environment, you can increase number of servers depending upon the increasing number of clients. Clustering gives good scalability when the number of clients increase and also gives high availability. Some of the major EJB vendors support clustering. For example if you deploy a cluster aware EJB application in three servers and if the first server is overloaded, clients are routed automatically to the next server depending upon the load balancing algorithms. And also clients are routed to the other server if the

first server dies, in this situation client state is passed to the other server to handle from that point (this is called as fail over). When you deploy your application on a single server and don't have future anticipation of increase in number of clients then do not use clustering because cluster aware EJBeans have overheads involved.

You need to configure vendor specific deployment descriptor file(or other vendor specific manner) in order to get cluster aware EJB application. When you configure for clustering, EJB Container/Server generates cluster aware EJBeans. Look at vendor specific documentation for details on clustering. So use clustering to get high scalability and fail over.

Tune thread count

Your EJB server may have a facility to tune the number of simultaneous operations/threads (thread count) it can run at a time. If the default value of thread count provided by your server is less than the capability of the server, the clients requesting for an application may be put in a queue. Depending on your resources and the capability of the server you can increase the thread count to improve performance. If you are not clear about the thread count then it is better to leave the default value as it is because increasing thread count without sufficient resources may degrade performance. For more information on this see your vendor documentation.

Choosing between HttpSession and Stateful session bean

Both Servlet's HttpSession object and EJB's Stateful session bean are meant to maintain client state, so which one is a better option? let's look into the advantages and disadvantages of both the mechanisms :

1 . Stateful session bean

Advantages :

- It supports transaction service ,security service, life cycle management, RMI, instance cache, thread safe etc. You need not write code for these services.
- It can be used by both web based and non web based clients (like swing etc.) .
- It can be used for multiple operations for a single http request.

Disadvantages :

- Since it supports a number of services mentioned above it takes more time to process a request.

2. HttpSession object

Advantages:

- It is a simple java object (perhaps a Hashtable) and takes very less time and resources to maintain a client state

Disadvantages:

- It does not support the features discussed above
- It cannot process multiple operations for a single http request.

So depending on your application's requirement you can choose the one best suited for you, if you want the bean only for client state management then HttpSession object gives better performance than Stateful session bean.

Choosing an EJB server

We have more than thirty vendors who provide EJB containers/servers. So when you have to choose a server you need to carefully look into the following issues and consider the trade-offs between them. The following features need to be looked into before you decide on EJB server

1. Clustering
2. Load balancing
3. Instance pool and instance caching
4. Lazy loading
5. Pass by reference for EJB 1.1
6. Different locking strategies
7. Connection pooling
8. Controlling callback methods such as ejbLoad(),ejbStore() in CMP etc

If an EJB server provides all these features then how can you make sure about the performance of your server? The answer is to use the benchmarking specifications available such as TPC-W, @Bench etc and test the server. But the problem with these bench marking policies is that they test only for a specific feature of your server, so in order to test the overall performance of your server you can use ECperf

specification released by SUN to test your server.

Ecperf benchmark specification is developed under java community process that is meant mainly for EJB server performance testing. It provides EJB code and driver to test an EJB server vigorously. For ECperf information, have a look at the following links

http://java.sun.com/j2ee/ecperf/

You will also find benchmark results of various servers at the following site.

http://ecperf.theserverside.com/ecperf/

http://www.cmis.csiro.au/adsat/

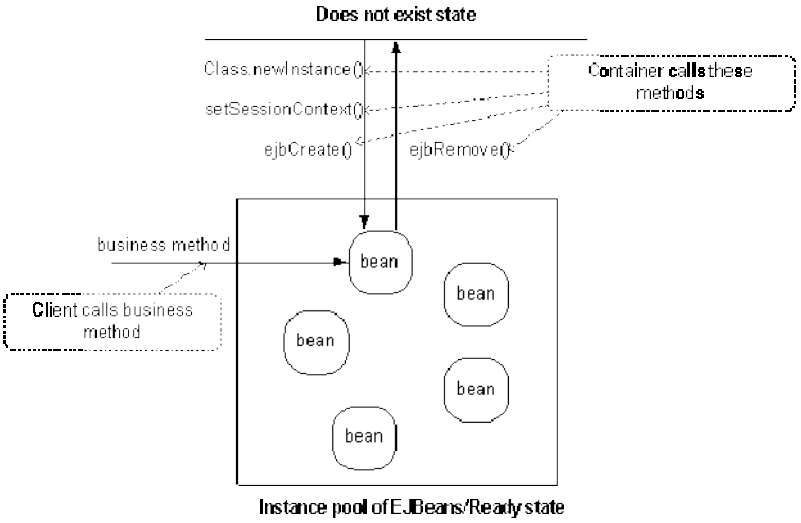
In order to choose best EJB server, analyze features of different EJB servers, test using ECperf tool kit and see available benchmarks (see above links for already available benchmarks) and finally decide suitable server for your application.

Tuning Stateless session beans

We already discussed common practices in the above sections. Those practices are applicable for Stateless session beans also. Here we discuss specific practices for Stateless session beans. In order to get clear picture of the practices, we will initially discuss Stateless session bean life cycle, since it drives some of the practices to improve performance. Note that the life cycle of all the beans are dissimilar.

Stateless session bean life cycle

Life cycle means when an EJBean is created and removed by EJB Container, and when the Container calls methods of EJBean. The following figure illustrates the life cycle of Stateless session bean.



Stateless session bean life cycle

You can control life cycle by mentioning instance pool size in vendor specific deployment descriptor. For example weblogic server's weblogic-ejb-jar.xml has element for instance pool size

```
<pool>
    <max-beans-in-free-pool>100</max-beans-in-free-pool>
    <initial-beans-in-free-pool>50</initial-beans-in-free-pool>
</pool>
```

and JBoss server's jboss.xml has an element <instance-pool> to mention pool size. See vendors documentation for information for other servers. Here you can specify initial beans and maximum beans in pool. If you mention for example 50 beans in initial beans and 100 for maximum beans in the pool, the life cycle starts when the server starts up.

When the server starts up, the EJB Container/Server creates 50 beans using Class.newInstance() method and puts it in the pool and it calls the following call back methods of the bean.

setSessionContext(ctx) and

ejbCreate() methods

these 50 beans are ready to be accessed by 50 clients concurrently. If the client accesses exceed 50 then the Container creates beans and calls the above methods till the number of beans is equal to 100 (maximum beans in the pool). So at any time the Container can only have a maximum of 100 beans to serve clients. What will happen if the concurrent clients are more than 100? Then the container puts clients in the queue. We will discuss about this in the next section on how you can tune the pool size.

The Container removes the beans from the pool if the number of clients accessing beans are less. When the Container removes the beans depending on its specific algorithms (perhaps LRU, Least Recently Used). At this time Container calls ejbRemove() method of the bean.

If the client calls the home.create() method, the Container creates EJBObject and assigns existing bean from the pool to the the client, at this time client neither creates a bean nor calls ejbCreate() method because it is already called when it created and is put in the pool. In the same manner, if the client calls home.remove() / remote.remove() method, the Container removes EJBObject and deassigns the bean from the client and puts it back to the pool but does not remove from the pool. In Stateless session bean life cycle, Client does not have control over bean's life cycle and bean's call back methods, it does not know when the creation, destruction and call back methods occur. So In Stateless session beans the creation, destruction and call back methods depend upon the pool size, clients concurrent access and Container algorithms.

As the name implies, a Stateless session bean does not maintain any state (instance variables values) across methods, that is the reason why ejbActivate() and ejbPassivate() methods do not have significance in Stateless session bean. So the Container can assign different beans from the pool to the client for successive methods.

With this discussion, we understand the importance of pool size and when the call back methods are executed in Stateless session beans. Now let us discuss how we can tune Stateless session beans.

Tune Stateless session beans instance pool size

The creation and destruction of beans is expensive. To reduce this cost, The EJB Container/Server creates pool of beans that depending upon vendor specific configuration, you need to give a proper value for this pool size to increase performance. As we discussed above, configure pool size, for example weblogic's weblogic-ejb-jar.xml has an element <pool> and JBoss server's jboss.xml has an element <instance-pool>. see your vendor documentation for configuring your EJB server pool size.

The number of maximum beans in pool impacts performance. If this is less, then the Container has to put the clients in the queue when the number of clients accessing is more than the maximum pool size. This degrades the performance and clients take more time to execute. For best performance, give maximum beans as equal to number of maximum clients accesses.

Use setSessionContext() or ejbCreate() method as cache

In Stateless session bean life cycle, The container invokes the setSessionContext(SessionContext sc) and ejbCreate() methods when it creates the bean instance first time and puts it in the pool and later it will be used for the other clients till it is removed by the Container,You can use these methods to acquire resources like home object references of other session or entity beans or DataSource references and put it in instance variables. Once you acquire these resources in these methods you need not acquire resources for each client because those resources are already acquired and available. Obviously these resources will be specific to a bean but not available globally. For global reuse, it is better to use the technique that we discussed in Cache EJBHome object references. You can use this technique to acquire other resources also. Remember that you should not acquire physical resources like database connections in these methods if the concurrent clients are more and pool size is more, it is better to acquire that type of resources in each method and remove them in that method. Use setSessionContext (SessionContext sc) and ejbCreate() methods to cache bean specific resources that are needed for other clients.

Release resources in ejbRemove() method

The Container calls ejbRemove() method just before removing a bean from the pool. So whatever resources you acquired in your bean like those discussed above must be released in this method.

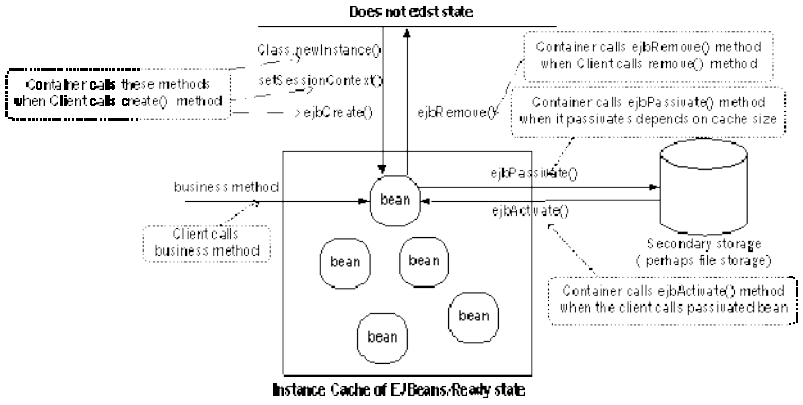
Tuning Stateful session beans

The common practices that we discussed above are applicable for Stateful session beans also. Here we discuss about specific practices for Stateful session beans. In order to get a clear picture, we will initially discuss Stateful session bean's life cycle.

Stateful session bean life cycle

The life cycle of Stateful and Stateless bean is differs. The reason is, Stateful session bean has to maintain state (instance variables values) across the methods for a client. That means once a client assigns values for instance variables using one method, those values are available for other methods also. The following figure illustrates the life cycle of Stateful session bean.

Figure7: Stateful session bean life cycle



Stateful session bean life cycle

Here you see the instance cache instead of instance pool. Cache maintains beans that have state (Stateful) whereas pool maintains beans that don't have state (Stateless). You can control life cycle by describing instance cache size in vendor specific deployment descriptor file. For example weblogic's weblogic-ejb-jar.xml has element for instance cache size

```
<stateful-session-cache>
    <max-beans-in-cache>100</max-beans-in-cache>
</stateful-session-cache>
```

and JBoss server's jboss.xml has an element

```
<instance-cache>
<container-cache-conf>
    <cache-policy>
        <cache-policy-conf>
            <min-capacity>5</min-capacity>
            <max-capacity>10</max-capacity>

        </cache-policy-conf>
    </cache-policy>
</container-cache-conf>
</instance-cache>
```

For detailed information, look at their documentation and for other servers look at vendors documentation for instance cache configuration.

Here you can specify minimum beans and maximum beans in cache. Some vendors such as weblogic do not support configuring minimum beans but support configuring maximum beans. So look at your vendor documentation for detailed information on what your EJB Container/server supports.

Here life cycle of stateful session bean starts when the client calls create() method. When the the client calls create() method, the Container creates the bean using Class.newInstance() and calls

setSessionContext(ctx) and

ejbCreate() methods

and puts it in the instance cache. Then onwards, the client is assigned to that bean till client calls remove() method, thus container calls ejbRemove() and destroys the bean. In Stateful session bean, the client controls life cycle (creation and removal but not activation and passivation). So when does container call ejbActivate() and ejbPassivate() methods? For example, if you set maximum beans in cache as 100 beans, when clients are accessing a bean concurrently, container creates beans till 100 (maximum bean limit) and assigns those beans to clients. After this limit, if the 101st client accesses this bean, Container passivates some of the idle beans that depending on Container algorithms. Just before passivating beans, it calls ejbPassivate() method. Here passivation means, it stores the bean's state (instance variables values) in the secondary storage (file storage or database). The passivation happens through Serialization. Later if the the idle client accesses the bean again, then Container activates the bean and reassigns the passivated values to its instance variables and calls ejbActivate() method.

Here what we understand is that client controls creation and destruction of beans but not activation and passivation. Activation and passivation are controlled by Container and depend on instance cache size.

With this discussion, we understand the importance of instance cache size and when the call back methods are executed in Stateful session beans. Now let us discuss how we can tune Stateful session beans.

Tune Stateful session beans instance cache size

As discussed above, You control activation and passivation indirectly by describing instance cache size in vendor specific deployment descriptor file . Activation and passivation is expensive because of serialization process. If the instance cache size is less and concurrent active clients are more than instance cache size then activation and passivation occur often, thus degrading performance. So in order to increase performance, give optimal cache size.

Set optimal bean age for Stateful session beans

The removal of EJBean instance by the Container from the instance cache depends not only on when the client calls remove() method but also on EJBean time out value (bean age) that you can configure in the vendor specific descriptor file. If the beans time out value is less, the Container needs to remove and create often,which is expensive. So set optimal bean age to minimise removal and creation process.

Control Serialization in Stateful session beans

We already discussed about Control Serialization in remote EJBs that describes how you need to control serialization when you pass method parameters from client to EJBean through network. Here we discuss about the serialization that is specific to Stateful session bean.

When the Container wants to passivate a Stateful session bean it needs to serialize the instance variable values and store it in the secondary storage and needs to de-serialize those values from storage when it activates. The Serialization occurs for all instance variables except 'transient' variables. If the instance variable has a huge object graph, you may not want to serialize total graph or all the instance variables. So to avoid serialization for unwanted variables, use 'transient' keyword for those variables so that serialization process will be reduced. For more detailed information about serialization, See the Serialization section.

Remove Stateful session beans explicitly

The Container keeps the bean in the instance cache till the bean's time out occurs or when the client calls remove() method explicitly otherwise the container passivates the bean if the bean is idle for more time. So if the client finishes work with the bean and does not remove explicitly, the Container keeps the the bean in the instance cache or passivates the bean. This process consumes unnecessary memory. So remove the bean explicitly from the client using the remove() method.

Tuning Entity beans

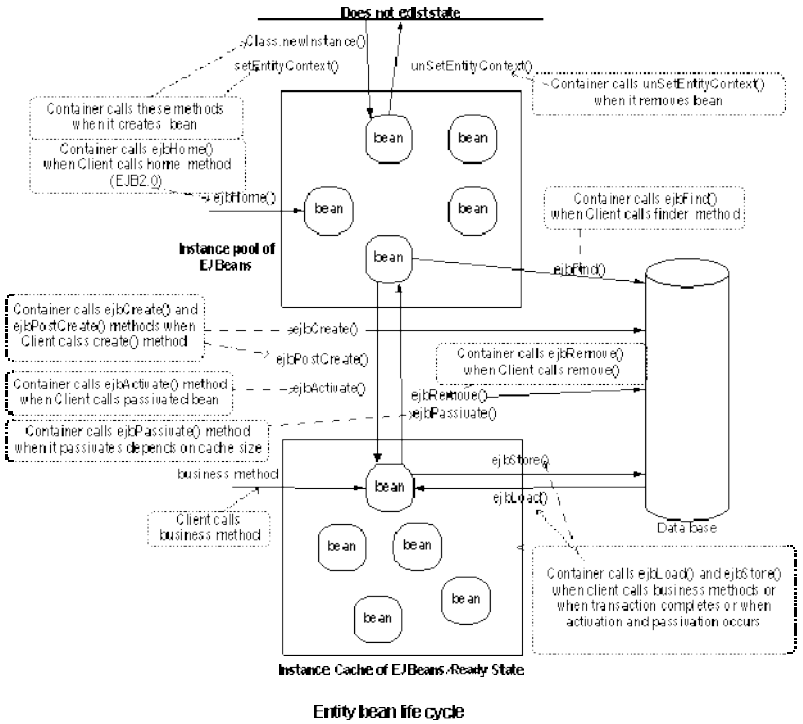
The common tuning practices that we discussed above are applicable for Entity beans also. Here we discuss specific practices for Entity beans. In order to get a clear picture, we will initially discuss Entity bean life cycle. Here we discuss the tuning practices for both CMP (Container managed persistence) and BMP (Bean managed persistence). Let us start with entity bean life cycle.

Entity bean life cycle

The life cycle of Entity beans is a combination of Stateless and Stateful bean life cycles. There is slight difference between Container managed persistence (CMP) and Bean managed persistence (BMP) entity bean's life cycle that is negligible. So here we will discuss a generalized life cycle that applies to both.

The following figure illustrates the life cycle of Entity beans.

Figure7: Entity session bean life cycle



Entity bean life cycle

Here you see both instance pool and instance cache. instance pool maintains entity beans without state

data and instance cache maintains entity beans with state data.

You can control life cycle in Entity beans by mentioning instance pool size and instance cache size in vendor specific deployment descriptor. For example weblogic's weblogic-ejb-jar.xml has element for instance pool size,

```
<pool>
    <max-beans-in-free-pool>100</max-beans-in-free-pool>
    <initial-beans-in-free-pool>50</initial-beans-in-free-pool>
</pool>
```

and instance cache size

```
<entity-cache>
    <max-beans-in-cache>100</max-beans-in-cache>
</entity-cache>
```

and in JBoss, jboss.xml has an element

<instance-pool> to mention pool size.

and for instance cache size

```
<instance-cache>
<container-cache-conf>
    <cache-policy>
        <cache-policy-conf>
            <min-capacity>5</min-capacity>
            <max-capacity>10</max-capacity>

        </cache-policy-conf>
    </cache-policy>
</container-cache-conf>
</instance-cache>
```

If you mention 50 beans as initial beans and 100 beans as maximum beans for instance pool, 50 (min) and 100(max) for instance cache, life cycle starts when the server starts up.

When the server starts up, First, the EJB Container/Server creates 50 beans using Class.newInstance() method and puts them in the pool and it calls setEntityContext() method on each bean. The Container can remove beans from the pool depending on clients accesses and idle time of beans in the pool. When it removes the bean from the pool it calls unSetEntityContext() method on the bean.

Next, When the client calls the create() method, the Container calls corresponding ejbCreate() method on one of the beans in the instance pool and creates a row in the database and populates the values to the variables and puts it in the instance cache after returning primary key. At this stage an EJBObject is assigned to the client that communicates to the bean in the instance cache. Next, the Container calls ejbPostCreate() method. At this stage, the bean is moved from pool to cache and is ready to serve clients business methods.

Next, When the client calls the business method, the Container calls ejbLoad() that updates the beans state, executes the business method, and calls ejbStore() method to store the data in the database. If the concurrent active clients are more than cache size then the container passivates a bean and calls ejbStore(), ejbPassivate() methods and puts it back in the instance pool. If the idle client calls again after some time, container calls ejbLoad() to get latest data, and calls ejbActivate() method and puts it in the instance cache.

Next, If the client calls remove() method, the Container calls ejbRemove() method that removes the data from the database and puts the bean back in the instance pool from instance cache.

With this discussion, we understand that

1. Client controls life cycle of a bean that involves creation of data in the database thus moving the bean from pool to the cache and removal of data from the database thus moving the bean from cache to the pool.

2. Container controls the life cycle in the pool and cache and also activation and passivation process in the cache.

3. Both client and container control ejbLoad() and ejbStore() methods depending upon client's method calls and Container activation and passivation process.

Finally the overall life cycle depends upon clients concurrent operations, instance pool size and instance cache size.

Now let us discuss how we can tune Entity beans.

Tune Entity beans instance pool size

As per Entity bean life cycle discussion, we understand that we can control creation and destruction of beans by describing pool size(min and max) in vendor specific deployment descriptor (or other vendor specific manner). If this size is less (if your default size is less or you configure a smaller size) then the Container has to put the clients in the queue when the number of concurrent clients accessing (create/finder/home methods) are more than the max pool size. And also instance cache depends up on instance pool because the instance cache needs to get beans from the instance pool. So if the pool size is

less, It degrades the performance and clients take more time to execute. For best performance, give maximum beans in pool as equal to number of maximum concurrent client accesses (create/finder/home methods), so that it reduces creation and destruction of beans.

You can configure your pool size in vendor specific deployment descriptor (or other vendor specific manner). In the above Entity bean life cycle section, We discussed how to configure this pool size in Weblogic and JBoss servers.

Tune Entity beans instance cache size

As per Entity bean life cycle discussion, we understand that we can control activation and passivation indirectly by describing instance cache size in vendor specific deployment descriptor file (or other vendor specific manner). Look at Entity bean life cycle section for configuring instance cache size in Weblogic and JBoss servers.

Activation and passivation are expensive. For every activation the Container calls ejbLoad() to get latest data from the database and calls ejbActivate() method. For every passivation the Container calls ejbStore () to store data in the database and calls ejbPassivate() method. ejbLoad() and ejbStore() methods communicate with the database to synchronize the latest data. If the concurrent active clients (when the client calls business methods) are more than instance cache size then activation and passivation occur often thus effecting performance. So in order to increase performance, give optimal cache size. Cache size must be equal to concurrent active clients accessing the bean.

Note: The instance cache size and pool size in entity beans are larger than session beans. The beans in the pool and cache should accommodate the entity beans requirements like finder methods that return large number of records and populate the data in the beans. So be careful when you configure entity bean pool size and cache size. If you are not sure about what the exact parameters are then use default pool size and cache size.

Use setEntityContext() method as cache

setEntityContext() is called only once in a bean's life time. Because Entity beans in the pool are reused by number of other clients, you can cache any bean specific resources like Entity home references and DataSource references in this method. You need to declare those resources as instance variables and acquire them in this method. It is similar to the technique that we already discussed in Use setSessionContext() as cache. These resources will be specific to a bean but not available globally. For global reuse, it is better to use the technique that we discussed in Cache EJBHome object references. You can use this technique to acquire other resources also. Remember that you should not acquire physical resources like database connections in these methods if the concurrent clients are more and pool size is more, it is better to acquire such resources in each method and release them in that method only. Use setEntityContext() method to cache bean specific resources that are needed by other clients as well.

Release resources in unSetEntityContext() method

The Container calls unSetEntityContext() method just before removing a bean from the pool. So whatever resources that were acquired (as we discussed above) need to be released in this method.

Use Lazy loading

Whenever you implement parent child relationships 1:1 (one to one), 1:M (one to many) and M:M (many to many) in your Entity beans, you need to be careful about when you are loading child (sub relationship) data.

For example, in BMP, you would write a 1:M (OrderBean:LineItem entity beans) relationship like this

```
public class OrderBean implements EntityBean {
    private long orderId;
    private String orderName;
    private long lineItemId;    // FK to the child table LineItems table
    private Collection lineItems;    // child data from LineItems table
  public void ejbLoad() {
      // step1: select orderId, orderName and lineItemId from Order table
      /* step2: get lineItems by looking up LineItem entity bean
            : look up LineItem home reference through JNDI
            lineItems = lineItemHome.findLineItems(lineItemId);
    */
  }
  public Collection getLineItems(){
      return lineItems;
  }
```

Here you are getting data for both parent and child (OrderBean and LineItem) in one call in ejbLoad() method. This is called as aggressive/eager loading. Here you are getting data from the database on client's every business method request or when transaction ends. But client may often be interested in orderId and orderName only but not LineItems. For example, if you use finder method to get orders, you

might get 100 orders and 1000 lineItems. You get 1000 lineItems records unnecessarily even though the client didn't want lineItems. Remember that ejbLoad() is called even when the Container wants to activate a bean, this is an extra overhead.

You can solve this problem using Lazy loading. In Lazy loading, you load the child data as and when required. The above example can be rewritten like this :

```
public class OrderBean implements EntityBean {
    private long orderId;
    private String orderName;
    private long lineItemId;    // FK to the child table LineItems table
    private Collection lineItems;    // child data from LineItems table
  public void ejbLoad() {
      // step1: select orderId, orderName and lineItemId from Order table
  }
  public Collection getLineItems(){
     /* step2: get lineItems by looking up LineItem entity bean
              : look up LineItem home reference through JNDI
              lineItems = lineItemHome.findLineItems(lineItemId);
     */
       return lineItems;
  }
```

Here you load the data as and when client calls getLineItems() but not when it calls other business methods. So Lazy loading is best to improve performance and use it when you implement relationships in Entity beans.

Choose optimal transaction isolation level

Isolation levels represent how a database maintains data integrity against the problems like dirty reads, phantom reads and non-repeatable reads which can occur due to concurrent transactions. You can avoid these problems by describing following isolation levels in vendor's deployment descriptor file(or other vendor specific manner)

TRANSACTION_READ_UNCOMMITED

TRANSACTION_READ_COMMITED

TRANSACTION_REPEATABLE_READ

TRANSACTION_SERIALIZABLE

The top most is least isolation level and bottom most is strict isolation level. Databases use read and write locks depending on above isolation levels to control transactions. Let us first discuss the problems related to concurrent transactions to the database and then the remedy to these problems.

Problems due to concurrent transactions

The following table describes isolation level against the problem that it prevents :

Transaction Level	Permitted Phenomena			Performance impact
	Dirty reads	Non Repeatable reads	Phantom reads	
TRANSACTION_READ_UNCOMMITED	YES	YES	YES	FASTEST
TRANSACTION_READ_COMMITED	NO	YES	YES	FAST
TRANSACTION_REPEATABLE_READ	NO	NO	YES	MEDIUM
TRANSACTION_SERIALIZABLE	NO	NO	NO	SLOW

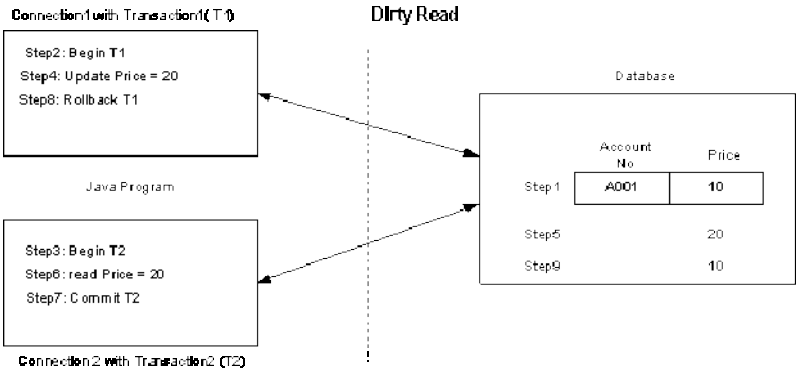
YES means the transaction level does not prevent the problem

NO means the transaction level prevents the problem

By setting isolation levels, you are having an impact on the performance as mentioned in the above table. Databases use read and write locks to control above isolation levels. Let us have a look at each of these problems, and then look at the impact on the performance.

Dirty read problem :

The following figure illustrates Dirty read problem :



Step 1: Database row has PRODUCT = A001 and PRICE = 10

Step 2: Connection1 starts Transaction1 (T1) .

Step 3: Connection2 starts Transaction2 (T2) .

Step 4: T1 updates PRICE =20 for PRODUCT = A001

Step 5: Database has now PRICE = 20 for PRODUCT = A001

Step 6: T2 reads PRICE = 20 for PRODUCT = A001

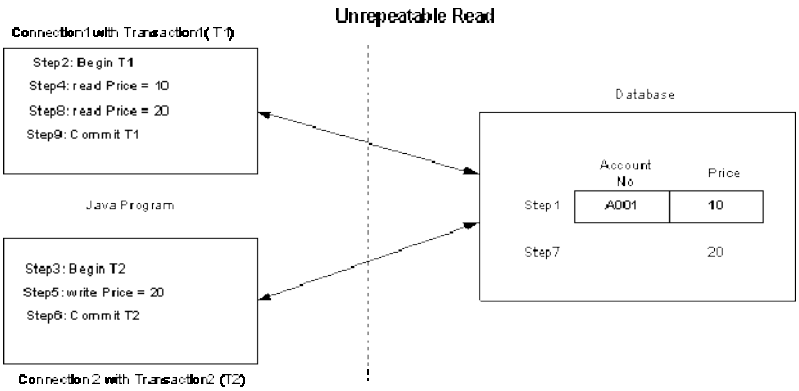
Step 7: T2 commits transaction

Step 8: T1 rollbacks the transaction because of some problem

The problem is that T2 gets wrong PRICE=20 for PRODUCT = A001 instead of 10 because of uncommitted read. Obviously it is very dangerous in critical transactions if you read inconsistent data. If you are sure about not accessing data concurrently then you can allow this problem by setting TRANSACTION_READ_UNCOMMITED or TRANSACTION_NONE elseTRANSACTION_READ_COMMITED to avoid this problem.

Unrepeatable read problem :

The following figure illustrates Unrepeatable read problem :



Step 1: Database row has PRODUCT = A001 and PRICE = 10

Step 2: Connection1 starts Transaction1 (T1) .

Step 3: Connection2 starts Transaction2 (T2) .

Step 4: T1 reads PRICE =10 for PRODUCT = A001

Step 5: T2 updates PRICE = 20 for PRODUCT = A001

Step 6: T2 commits transaction

Step 7: Database row has PRODUCT = A001 and PRICE = 20

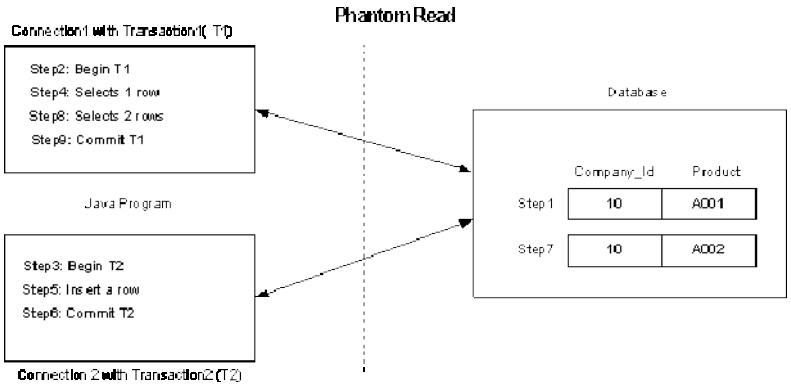
Step 8: T1 reads PRICE = 20 for PRODUCT = A001

Step 9: T1 commits transaction

Here the problem is that Transaction1 reads 10 first time and reads 20 second time but it is supposed to be 10 always whenever it reads a record in that transaction. You can control this problem by setting isolation level as TRANSACTION_REPEATABLE_READ.

Phantom read problem :

The following figure illustrates Phantom read problem :



Step 1: Database has a row PRODUCT = A001 and COMPANY_ID = 10

Step 2: Connection1 starts Transaction1 (T1) .

Step 3: Connection2 starts Transaction2 (T2) .

Step 4: T1 selects a row with a condition SELECT PRODUCT WHERE COMPANY_ID = 10

Step 5: T2 inserts a row with a condition INSERT PRODUCT=A002 WHERE COMPANY_ID= 10

Step 6: T2 commits transaction

Step 7: Database has 2 rows with that condition

Step 8: T1 select again with a condition SELECT PRODUCT WHERE COMPANY_ID=10 and gets 2 rows instead of 1 row

Step 9: T1 commits transaction

Here the problem is that T1 gets 2 rows instead of 1 row up on selecting the same condition second time. You can control this problem by setting isolation level as TRANSACTION_SERIALIZABLE

Choosing a right isolation level for your program:

Choosing a right isolation level for your program depends upon your application's requirement. In a single application itself the requirement generally changes, suppose if you write a program for searching a product catalog from your database then you can choose TRANSACTION_READ_ UNCOMMITED because you need not worry about the isolation problems , some other program can insert records at the same time, you don't have to bother much about that insertion. Obviously this improves performance significantly.

If you write a critical program like bank or stocks analysis program where you want to control all of the isolation problems, you can choose TRANSACTION_SERIALIZABLE for maximum safety. Here it is the tradeoff between the safety and performance.

Other two isolation levels need good understanding of your requirement. If your application needs only committed records, then TRANSACTION_READ_COMMITED isolation is the good choice. If your application needs to read a row exclusively till you finish your work, then TRANSACTION_REPEATABLE_READ is the best choice.

Note: Be aware of your database server's support for these isolation levels. Database servers may not support all of these isolation levels. Oracle server supports only two isolation levels, TRANSACTION_READ_COMMITED and TRANSACTION_SERIALIZABLE isolation level, default isolation level is TRANSACTION_READ_COMMITED.

Use proper locking strategy

We discussed above how a database can lock data depending on isolation level. The locking that happens at database level, is called as database locking strategy. Other than that, Your EJBServer/Container can support different locking strategies. Some EJB vendors such as Weblogic supports exclusive locking strategy that locks Entity beans at application server itself rather than at the database. Exclusive locking blocks other Entity beans till locked Entity bean completes it's transaction. This locking does not allow other beans to even read the data , it has an impact on performance. Your application server may have default locking strategy of either database locking or exclusive locking, it may have even other locking strategies. You need to configure your vendor's deployment descriptor file so that you can override default locking strategy depending on your application's requirement to improve performance.

Make read-only Entity beans

In your EJB application, your entity beans might be always getting data from the database but not updating data. But by default, your application server would read and write data using ejbLoad() and ejbStore() methods for every business method call or end of transaction. Why would you update database using ejbStore() method while your entity bean performs read only transactions? this effects performance

by calling unnecessary ejbStore() methods. In this situation, if your EJBServer/Container supports configuring for read only or read-write entity beans, it is better to configure as read only beans in vendor's deployment descriptor file to improve performance. If you configure your entity bean as read only then there will be no updates (no ejbStore() calls) to the database, so be careful when you configure this property. Some vendors support read-mostly property where you can configure when to read from database, you can also configure this feature if you need to reduce frequency of ejbLoad() calls. Look at your vendor's EJB server/container documentation on how to configure read only property.

Use dirty flag to avoid unchanged buffer data updation

EJB Container calls ejbStore() method when the business method execution completes or when the transaction completes irrespective of change in the bean's data. The entity bean's data may not change every time when the client calls the business method. So there will be lot of updates (calls to ejbStore()) to the database even though it doesn't require, thus degrading performance. To avoid this problem, you can configure dirty flag property in vendor specific deployment descriptor file for CMP1.1, for example is-modified-method-name in weblogic server's weblogic-ejb-jar.xml file. For EJB2.0 CMP, it does not require to configure because EJB2.0 CMP Container supports this feature implicitly. For BMP, you need to write this method in bean's class and call this method wherever necessary. This technique reduces calls to the database unless the bean's data changes, thus improving performance.

Commit the data after transaction

By default, Your EJB Container either commits the data after every method call or after completing the whole transaction. For example, in one transaction, if you call four methods of two entity beans from one session bean method, then the Container can either commit the data after each of the entity beans method's execution that calls ejbStore() method four times or it commits the data only once after execution of session bean method with four entity bean methods that completes whole transaction. Obviously if the transaction commits after it finishes completely, then it improves performance significantly. But other clients cannot see the data till it completes transaction. If your EJB Container/Server supports this feature, you can configure this feature in vendor's deployment descriptor or other vendor specific means. For example, weblogic server's weblogic-ejb-jar.xml has an element 'delay-updates-until-end-of-tx', that updates the database after completion of transaction. So commit the data after transaction by configuring vendor's file if you don't need other clients to read transactional data in between.

Do bulk updates

If you use relationships (1:1, 1:M, M:M) in CMP entity beans and want to update both parent and child data (bulk data), it is better to update the both parent and child data at the same time using vendor specific features. For example, weblogic supports this feature as field groups. Bulk updates reduces number of calls to the database and improves performance.

Use CMP instead of BMP

Before introduction of EJB2.0 specification often developers used BMP rather than CMP because the previous EJB spec does not support enough important features like relationships. But EJB2.0 spec for CMP has good relationship support as well as performance improvements.

One of the CMP performance improvement technique in EJB2.0 is that the Container can monitor bean's data (in-memory buffer) change and if any changes happens in that data then only Container will update the database. Because of this monitoring capability, CMP gives better performance than BMP. And another technique is when you call finder method in BMP, it initially retrieves primary key with first call to the database and then instance data by placing a second call to the database. It makes two calls to the database. But CMP gets the data with single call to the database for the finder methods. Thus CMP gives better performance than BMP because Container has good hold on CMP.

Use ejbHome() methods for global operations

EJB2.0 specification introduced ejbHome() methods in Entity beans. You can use these methods to perform global operations (for example, getting total number of persons) that do not relate to any specific entity bean instance data. When you call home methods from client, the Container calls the ejbHome() method on the bean that is in the instance pool before assigning any EJBObject to the client. This improves performance because the bean does not have state maintenance.

So use ejbHome() methods for global operations that do not have state (instance variables data) dependency.

Use connection pool

Creating a connection to the database server is expensive. It is even more expensive if the server is located on another machine. Connection pool contains a number of open database connections, and has open connections between minimum and maximum number that you specify in vendor specific manner. The pool expands and shrinks between minimum and maximum size depending on incremental capacity. You need to give minimum, maximum and incremental sizes as properties to the pool in order to maintain that functionality. You get the connection from the pool instead of getting it directly from the database. For example, if you give properties like min, max and incremental sizes as 3, 10 and 1 then pool is created with size 3 initially and if it reaches its capacity 3 and if a client requests a connection concurrently, it

increments its capacity by 1 till it reaches 10 and later on it puts all it's clients in a queue. You need to configure connection pool size in vendor specific manner and you need to to take care of properties like min, max and increment sizes. The maximum number of connections to be opened depend on your application's requirement and the maximum number of open connections your database can support.

In EJB deployment descriptor files, you need to configure DataSource reference that uses connection pool. Your EJB application improves performance significantly depending on connection pool size. So configure optimal connection pool to reduce expensive creation and destruction of database connections thus improving performance significantly.

Use JDBC tuning techniques

When you write BMP, you need to write JDBC code on your own. JDBC has lot of techniques to improve performance. There is a separate section for JDBC in this site. Those techniques are

- Choosing the right Driver
- Optimization with Connection
  - Set optimal row pre-fetch value
  - Use Connection pool
  - Control transaction
  - Choose optimal isolation level
  - Close Connection when finished
- Optimization with Statement
  - Choose right Statement interface
  - Do batch update
  - Do batch retrieval using Statement
  - Close Statement when finished
- Optimization with ResultSet
  - Do batch retrieval using ResultSet
  - Setup proper direction of processing rows
  - Use proper get methods
  - Close ResultSet when finished
- Optimization with SQL Query
- Cache the read-only and read-mostly data
- Fetch small amount of data iteratively rather than whole data at once
- JDBC Key Points

Use these techniques in BMP to boost your BMP performance.

Use direct JDBC when dealing with huge data

Generally developers prefer JDBC over entity beans due to performance considerations. Entity bean is a component that has an overhead about which we discussed in Choosing between EJB vs. non-EJB. Obviously Entity beans have advantages because of their component architecture. To get EJB architecture to some extent you can wrap JDBC with session beans. When it comes to choosing between entity beans versus JDBC calls, you can decide depending on quantity of data. For example, if you search data that retrieves 50000 records, JDBC is a better choice when compared to entity beans. Entity beans are not a problem with small data and you can even increase performance by the techniques which we discussed. So use JDBC with session beans when you deal with huge data to improve performance.

Use business logic in Entity beans

You should consider an entity bean not only as a data access object but also as a business object. Business logic depends purely on application business requirement or on data. If you write business methods that depend on data in entity bean, it reduces network calls on remote entity beans by reducing round trips for getting data. This works well when your entity beans are remote EJBs. So put business logic that depends on data, in remote entity beans there by reducing network calls.

Tuning Message driven beans

The session beans and entity beans can act as JMS producers using JMS API but can only consume

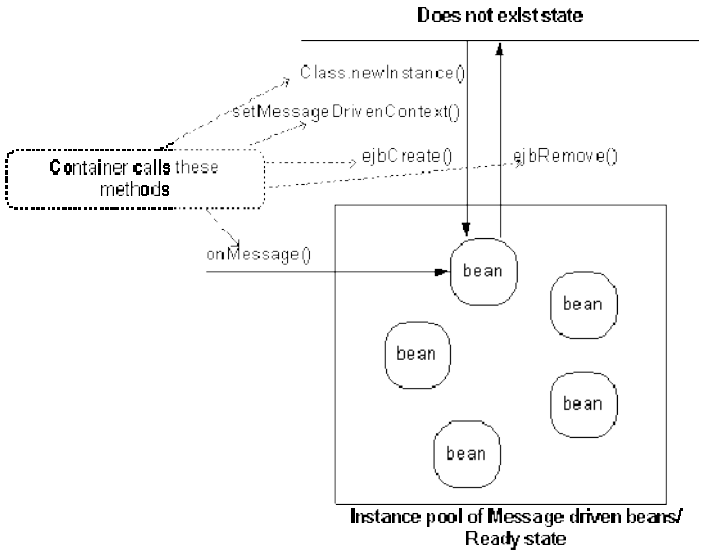
messages synchronously using MessageConsumer.receive() method. The reason for synchronous consumption of messages is that session and entity beans implement request-reply process synchronously but not asynchronously, that is why we have Message driven beans in EJB2.0. Message driven beans consume messages asynchronously from JMS Server.

We already discussed Common tuning practices practices in EJB in the above sections. Some of those practices such as Cache EJBHome object references, Use Clustering for scalability, Tune thread count, are applicable for Message driven beans also. Here we discuss about specific practices for Message driven beans. In order to get clear understanding of following tuning practices, we will discuss Message driven bean life cycle first.

Message driven bean life cycle

The life cycle of Message driven bean is analogous to the Stateless session bean because it also need not maintain client state. The main difference between them is that Stateless beans have business methods that are invoked by clients where as Message driven beans have onMessage() method of MessageListener interface that has business logic. The Message driven beans simply connect to the JMS server, consume messages from it and process those messages. A Message driven bean does not have a remote or home interface because it is not an RMI/IIOP component rather it uses JMS protocol to connect to the JMS server.

The following figure illustrates the life cycle of Message driven bean.



Message driven bean life cycle

You can control the number of message driven beans by mentioning instance pool size in vendor specific deployment descriptor. This controls when the beans are created, removed and call back methods are called. For example in weblogic's weblogic-ejb-jar.xml has element for instance pool size, that is

```
<pool>
    <max-beans-in-free-pool>100</max-beans-in-free-pool>
    <initial-beans-in-free-pool>50</initial-beans-in-free-pool>
</pool>
```

Here you can specify initial beans and maximum beans in the pool. If you mention for example 50 beans in initial beans and 100 for maximum beans in the pool, the life cycle starts when the server starts up.

When the server starts up, the EJB Container/Server creates 50 beans using Class.newInstance() method and puts it in the pool and it calls the following call back methods of the bean.

setMessageDrivenContext(ctx) and
ejbCreate() methods

those 50 beans are ready for consuming and processing 50 concurrent messages. If the concurrent messages are more than 50 then the Container creates more beans and calls the above methods till it reaches 100 beans (maximum beans in the pool). So at any time the Container can only have a maximum of 100 beans to consume messages concurrently. What will happen if the concurrent messages are more than 100? well, the container has to wait till beans are available in the pool.

When the message arrives, the container passes that message to onMessage(Message msg) method of one of the beans in the pool so that method can process business logic.

The Container removes the beans from the pool if the number of messages arriving are not many. How the Container removes the beans are depends upon it's specific algorithms. At this time Container calls ejbRemove() method of the bean and destroys the bean from the pool.

The creation and destruction of beans occurs depending on instance pool size and consumption (or arrival of messages to the Destination) of messages.

With this discussion, we understand the importance of pool size and when the call back methods are executed in Message driven beans. Now let us discuss how we can tune Message driven beans.

Tune Message driven beans instance pool size

The creation and destruction of beans is expensive. To reduce this cost, the EJB Container/Server creates pool of beans (that is vendor specific), you need to give optimal pool size for better performance. As we discussed above, you can give this pool size (initial beans and maximum beans) for example in weblogic's weblogic-ejb-jar.xml has an element <pool>. See your vendor documentation for more information.

The maximum number of beans in the pool effects performance. If this is less, then the Container has to put the messages in the wait mode in the JMS server when the number of messages arriving are more than the max pool size. It degrades the performance and takes more time to execute. For best performance, give maximum beans in the pool as equal to expected number of maximum concurrent messages.

Use setMessageDrivenContext() or ejbCreate() method as cache

In Message driven bean life cycle, the container invokes the setMessageDrivenContext() and ejbCreate() methods only once in it's life time when it creates the bean instance first time and puts it in the pool and later it will be used for the processing other messages till it is removed by the Container,You can use these methods to acquire resources like ConnectionFactory references, Destination references, home objects references of other session or entity beans or DataSource references and assign to the instance variables. Once you acquire these resources in these methods you need not create resources for processing other messages because those resources are already acquired and available for other message processes. Obviously these resources will be specific to a bean but will not be available globally. For global reuse, it is better to use the technique that we already discussed in Cache EJBHome object references, you can use this technique to acquire other resources also. Remember that you should not acquire physical resources like database connections in these methods if the concurrent arrival of messages are more and pool size is more, it is better to acquire those type of resources in onMessage() method and remove in the same method. Use setMessageDrivenContext() and ejbCreate() methods to cache bean specific resources that are needed for other clients.

Remove resources in ejbRemove()

In Message driven beans, the Container calls ejbRemove() method just before removing a bean from the pool like Stateless session beans. So whatever resources you acquired in your bean (as discussed above) need to be released in this method.

Use JMS tuning techniques

We have separate section for JMS, See Best practices to improve performance in JMS. Those practices are meant for both JMS producers and JMS consumers. You can specifically look at Consumer techniques (Message driven bean act as consumer) to improve overall performance of Message driven beans. See

- Optimization with Connection
  - Start the Connection when appropriate
  - Process messages concurrently
  - Close the Connection when finished
- Optimization with Session
  - Choose proper acknowledgement mode
  - Control transaction
  - Close the Session when finished
- Optimization with Destination
- Optimization with Message Producer/Consumer
  - Choose non-durable messages where appropriate
  - Set TimeToLive value properly
  - Receive messages asynchronously
  - Close Producer/Consumer when finished
- Optimization with Message
- Choosing right JMS Server

- JMS Key Points

Use these techniques to boost your Message driven bean's performance.

Key Points

Common tuning practices for EJBs

1. Use Local interfaces that are available in EJB2.0 if you deploy both EJB Client and EJB in the same EJB Server.
2. Use Vendor specific pass-by-reference implementation to make EJB1.1 remote EJBs as Local EJBs if you deploy both EJB Client and EJB in the same EJB Server.
3. Wrap entity beans with session beans to reduce network calls and to promote declarative transactions. Optionally, make entity beans as local beans where appropriate.
4. Make coarse grained session and entity beans to reduce network calls.
5. Control serialization by modifying unnecessary data variables with 'transient' key word to avoid unnecessary data transfer over network.
6. Cache EJBHome references to avoid JNDI lookup overhead.
7. Avoid transaction overhead for non-transactional methods of session beans by declaring 'NotSupported' or 'Never' transaction attributes that avoid further propagation of transactions.
8. Set proper transaction age(time-out) to avoid large transaction.
9. Use clustering for scalability.
10. Tune thread count for EJB Server to increase EJB Server capacity.
11. Choose servlet's HttpSession object rather than Stateful session bean to maintain client state if you don't require component architecture and services of Stateful session bean.
12. Choose best EJB Server by testing with ECperf tool kit.
13. Choose normal java object over EJB if you don't want built-in services such as RMI/IIOP, transactions, security, persistence, resource pooling, thread safe, client state etc..

Stateless session beans

1. Tune the Stateless session beans pool size to avoid overhead of creation and destruction of beans.
2. Use setSessionContext() or ejbCreate() method to cache bean specific resources.
3. Release acquired resources in ejbRemove() method

Stateful session beans

1. Tune Stateful session beans cache size to avoid overhead of activation and passivation process.
2. Set optimal Stateful session bean age(time-out) to avoid resource congestion.
3. Use 'transient' key word for unnecessary variables of Stateful session bean to avoid serialization overhead.
4. Remove Stateful session beans explicitly from client using remove() method.

Entity beans

1. Tune the entity beans pool size to avoid overhead of creation and destruction of beans.
2. Tune the entity beans cache size to avoid overhead of activation, passivation and database calls.
3. Use setEntityContext() method to cache bean specific resources.
4. Release acquired resources in unSetEntityContext() method
5. Use Lazy loading to avoid unnecessary pre-loading of child data.
6. Choose optimal transaction isolation level to avoid blocking of other transactional clients.
7. Use proper locking strategy.
8. Make read-only entity beans for read only operations.
9. Use dirty flag to avoid unchanged buffer data updation.
10. Commit the data after the transaction completes to reduce database calls in between transaction.
11. Do bulk updates to reduce database calls.
12. Use CMP rather than BMP to utilize built-in performance optimization facilities of CMP.
13. Use ejbHome() methods for global operations.
14. Tune connection pool size to reduce overhead of creation and destruction of database connections.
15. Use JDBC tuning techniques in BMP.
16. Use direct JDBC rather than using entity beans when dealing with huge data such as searching a large database.
17. Use business logic that is specific to entity bean data.

Message driven beans

1. Tune the Message driven beans pool size to promote concurrent processing of messages.
2. Use setMesssageDrivenContext() or ejbCreate() method to cache bean specific resources.
3. Release acquired resources in ejbRemove() method.

4. Use JMS tuning techniques in Message driven beans.

Feed back

We appreciate and welcome your comments on this section. Email commentsZZZ@precisejavaZZZ.com (remove ZZZ which is placed to prevent spam). Please note that we may not be able to reply to all the emails due to huge number of emails that we receive but we appreciate your comments and feedback.