



# **JBoss Enterprise Application Platform 5**

## **Performance Tuning Guide**

for use with JBoss Enterprise Application Platform 5.2.x

Edition 5.2.0



# JBoss Enterprise Application Platform 5 Performance Tuning Guide

---

for use with JBoss Enterprise Application Platform 5.2.x  
Edition 5.2.0

Andrig Miller

## **Edited by**

Eva Kopalova

Petr Penicka

Russell Dickenson

Scott Mumford

## Legal Notice

Copyright © 2012 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## Abstract

This Performance Tuning Guide provides guidance on optimizing the performance of the JBoss Enterprise Application Platform 5 and its patch releases.

## Table of Contents

<b>PREFACE</b> .....	<b>3</b>
1. FILE NAME CONVENTIONS	3
<b>CHAPTER 1. INTRODUCTION</b> .....	<b>4</b>
1.1. PLATFORM COMPONENTS	4
<b>CHAPTER 2. CONNECTORS</b> .....	<b>6</b>
2.1. JAVA I/O CONNECTOR	6
2.1.1. maxKeepAliveRequests	6
2.1.2. maxThreads	8
2.1.3. minSpareThreads	9
2.1.4. Tuning the thread pool	9
2.2. AJP CONNECTOR	9
<b>CHAPTER 3. SERVLET CONTAINER</b> .....	<b>11</b>
3.1. CACHED CONNECTION MANAGER	11
3.2. HTTP SESSION REPLICATION	12
3.2.1. Full Replication	12
3.2.2. Buddy Replication	13
3.2.3. Monitoring JGroups via JMX	14
<b>CHAPTER 4. EJB 3 CONTAINER</b> .....	<b>16</b>
4.1. STATELESS SESSION BEAN	16
4.1.1. ThreadLocalPool	16
4.1.2. StrictMaxPool	16
4.1.3. Pool Sizing	16
4.1.4. Local EJB3 Calls	17
4.2. STATEFUL SESSION BEAN	18
4.2.1. Stateful Configuration	18
4.2.2. Full Replication	19
4.2.3. Buddy Replication	19
4.3. REMOTE EJB CLIENTS	19
4.4. CACHEDCONNECTIONMANAGER	21
4.5. ENTITY BEANS	22
4.5.1. Second level cache	22
4.5.1.1. Marking entities to be cached	25
4.5.2. Prepared Statements	26
4.5.3. Batch Inserts	26
4.5.3.1. Batch Processing; Tests And Results	31
Conclusions	37
4.5.4. Batching Database Operations	38
4.6. MESSAGE DRIVEN BEANS	40
<b>CHAPTER 5. JAVA CONNECTOR ARCHITECTURE</b> .....	<b>44</b>
5.1. DATA SOURCES	44
5.2. DATA SOURCE CONNECTION POOLING	44
5.3. MINIMUM POOL SIZE	44
5.4. MAXIMUM POOL SIZE	45
5.5. BALANCING POOL SIZE VALUES	45
5.6. JMS INTEGRATION OR PROVIDER	48
<b>CHAPTER 6. JAVA MESSAGE SERVICE (JMS) PROVIDER</b> .....	<b>50</b>
6.1. SWITCHING THE DEFAULT MESSAGING PROVIDER TO HORNETQ	50

6.2. EXPLOITING HORNETQ'S SUPPORT OF LINUX'S NATIVE ASYNCHRONOUS I/O	50
6.3. HORNETQ'S JOURNAL	51
6.4. HORNETQ AND TRANSACTIONS	52
<b>CHAPTER 7. LOGGING PROVIDER</b> .....	<b>53</b>
7.1. CONSOLE LOGGING	53
7.2. APPENDERS	54
7.3. LOCATION OF LOG FILES	55
7.4. WRAPPING LOG STATEMENTS	55
<b>CHAPTER 8. PRODUCTION CONFIGURATION</b> .....	<b>57</b>
<b>CHAPTER 9. JAVA VIRTUAL MACHINE TUNING</b> .....	<b>58</b>
9.1. 32-BIT VS. 64-BIT JVM	58
9.2. LARGE PAGE MEMORY	59
9.3. GARBAGE COLLECTION AND PERFORMANCE TUNING	62
9.4. OTHER JVM OPTIONS TO CONSIDER	63
<b>CHAPTER 10. PROFILING</b> .....	<b>65</b>
10.1. INSTALL	65
10.2. ENABLE OPROFILE	65
10.3. CONTROLLING OPROFILE	66
10.4. DATA CAPTURE	66
<b>CHAPTER 11. PUTTING IT ALL TOGETHER</b> .....	<b>68</b>
11.1. TEST CONFIGURATION	68
11.2. TESTING STRATEGY	68
11.3. TEST RESULTS	68
<b>APPENDIX A. REVISION HISTORY</b> .....	<b>71</b>

# PREFACE

## 1. FILE NAME CONVENTIONS

The following naming conventions are used in file paths for readability. Each convention is styled so that it stands out from the rest of text:

### ***JBOSS\_EAP\_DIST***

The installation root of the JBoss Enterprise Application Platform instance. This folder contains the main folders that comprise the server such as `/jboss-as`, `/seam`, and `/resteasy`.

### ***PROFILE***

The name of the server profile you use as part of your testing or production configuration. The server profiles reside in `JBOSS_EAP_DIST/jboss-as/server`.

The conventions above use the Unix path separator character of the forward slash (/), which applies to Red Hat Enterprise Linux and Solaris. If the JBoss Enterprise Application Platform instance is installed on Windows Server, swap the forward slash for the backslash character (\) in all instructions.

## CHAPTER 1. INTRODUCTION

The *Performance Tuning Guide* is a comprehensive reference on the configuration and optimization of JBoss Enterprise Application Platform 5. Performance tuning the platform is unique to each installation because of internal and external factors. The optimization of external factors including: application design, scalability, database performance, network performance and storage performance are outside the scope of this book. If Red Hat Enterprise Linux is the underlying operating system, refer to the Red Hat Enterprise Linux *Performance Tuning Guide* and *Storage Administration Guide* for further optimization advice. Otherwise refer to the specific vendor's documentation.

Performance tuning is not a goal in itself but a cyclical process of performance monitoring, configuration changes and review. The objective should be to meet business requirements, with each change having a measurable objective. It's recommended to apply one change at a time and monitor the results so that you can judge what does and does not result in a performance improvement. Not all workloads are equal, and not all changes will result in a performance improvement. The JBoss Enterprise Application Platform's components are the main focus of this book but coverage also includes a chapter on the Java Virtual Machine as its performance is also critical.

### 1.1. PLATFORM COMPONENTS

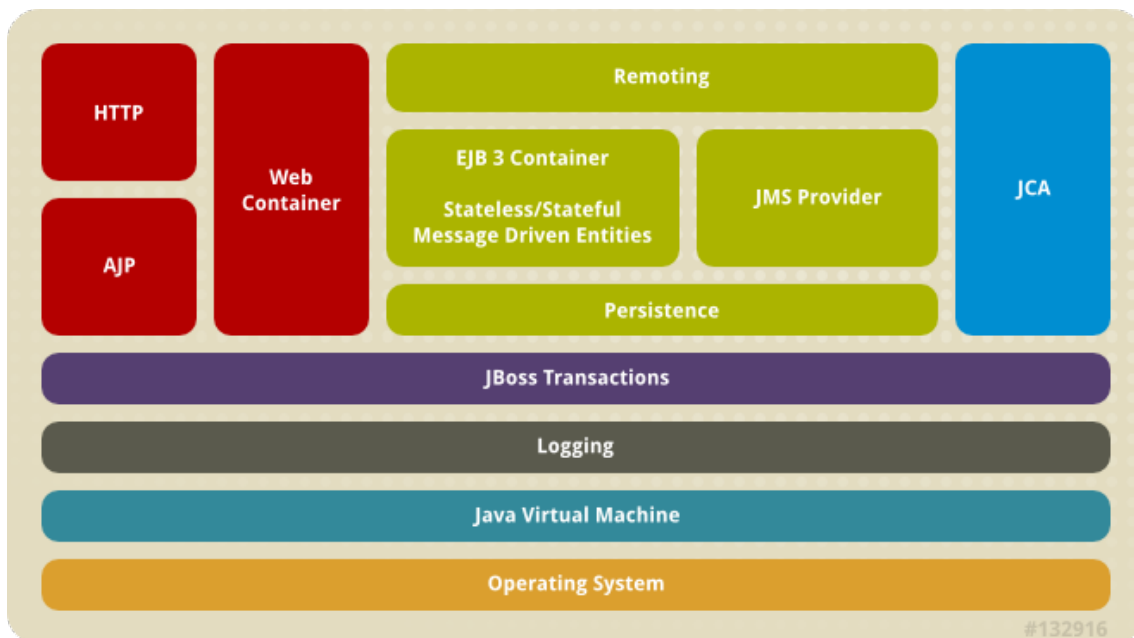


Figure 1.1. JBoss Enterprise Application Platform architecture

Figure 1.1, “JBoss Enterprise Application Platform architecture” illustrates the components of JBoss Enterprise Application Platform. Performance tuning of each component is covered in its own chapter but since each component interacts with one or more other components there is some overlapping material.

For each component various performance-related settings are discussed, including:

- Connection pooling
- Thread pooling
- Object and/or component pooling
- I/O configuration



- Logging and JMS provider
- Replication options
- Caching

## CHAPTER 2. CONNECTORS

Two main web connectors will be covered in this chapter: the Java I/O connector, using Java I/O to serve HTTP(s) connections directly to the platform and a native connector that uses Apache's Portable Runtime (APR) native code library. An Apache web server can be used to serve all HTTP(S) requests, using `mod_jk` or `mod_cluster` connect to the platform. In this scenario the tuning of the thread pool used for serving requests via the Apache Jserv Protocol (AJP) will be covered.

### 2.1. JAVA I/O CONNECTOR

The Java I/O Connector is used when clients send HTTP requests directly to the platform, and not to a front end HTTPD server such as an Apache web server. To use this connector the libraries of the native connector must be removed because they're used by default if detected by the platform. Many configuration options are available, including the following performance-related options:

- `maxKeepAliveRequests`
- `maxThreads`
- `minSpareThreads`

All of these parameters are configured in the `server.xml` file for the JBoss Web embedded servlet container: `JBOSS_EAP_DIST/jboss-as/server/PROFILE/deploy/jbossweb.sar`. Near the top of this file is the `Connector` section, where these parameters are configured. Note that the `minimal` configuration does *not* include JBoss Web.

```
<!-- A HTTP/1.1 Connector on port 8080 -->
<Connector protocol="HTTP/1.1" port="8080"
address="${jboss.bind.address}"
connectionTimeout="20000" redirectPort="8443" />
```

#### 2.1.1. maxKeepAliveRequests

The `maxKeepAliveRequests` parameter specifies how many pipe-lined requests a user agent can send over a persistent HTTP connection (which is the default in HTTP 1.1) before it will close the connection. A persistent connection is one which the client expects to remain open once established, instead of closing and reestablishing the connection with each request. Typically the client (a browser, for example) will send an HTTP header called a "Connection" with a token called "keep-alive" to specify a persistent connection. This was one of the major improvements of the HTTP protocol in version 1.1 over 1.0 because it improves scalability of the server considerably, especially as the number of clients increases. To close the connection, the client or the server can include the "Connection" attribute in its header, with a token called "close". This signals either side that after the request is completed, the connection is to be closed. The client, or user agent, can even specify the "close" token on its "Connection" header with its initial request, specifying that it does not want to use a persistent connection.

The default value for `maxKeepAliveRequests` is 100, so after an initial request from a client creates a persistent connection, it can send 100 requests over that connection before the server will close the connection.

Below is an extract from an actual client request which illustrates an HTTP POST with `Connection: keep-alive` content in the header, followed by an HTTP POST, again with `Connection: keep-alive` specified. This is the 100th client request so in the response to the HTTP POST is the instruction `Connection: close`.

```

POST /Order/NewOrder HTTP/1.1
Connection: keep-alive
Cookie: $Version=0; JSESSIONID=VzNmllbAmpDmInSJQ152bw__; $Path=/Order
Content-Type: application/x-www-form-urlencoded
Content-Length: 262
User-Agent: Jakarta Commons-HttpClient/3.1
Host: 192.168.1.22:8080

customerid=86570&productid=1570&quantity=6&productid=6570&quantity=19&prod
uctid=11570&quantity=29&productid=16570&quantity=39&productid=21570&quanti
ty=49&productid=26570&quantity=59&productid=&quantity=&productid=&quantity
=&productid=&quantity=&NewOrder=NewOrderGET /Order/OrderInquiry?
customerid=86570 HTTP/1.1
Connection: keep-alive
Cookie: $Version=0; JSESSIONID=VzNmllbAmpDmInSJQ152bw__; $Path=/Order
User-Agent: Jakarta Commons-HttpClient/3.1
Host: 192.168.1.22:8080

[1181 bytes missing in capture file]HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 1024
Date: Wed, 26 Jan 2011 17:10:40 GMT
Connection: close

```

To put this in real-life terms, imagine a test run during which 1,800 users process 3.6 million transactions (all via HTTP POST and GET requests) with `maxKeepAliveRequests` set to the default value of 100. With 1,800 clients, each performing 2,000 requests (3.6 million / 1,800 = 2,000), the number of times connections are closed is 36,000 (3.0 million / 100). Depending on the number of clients, type of clients (browsers tend to open multiple connections on multiple threads for performance reasons), and type of content being served to the client, this can be significantly higher. Besides the overhead of tearing down, and creating new connections, there is also the overhead, although small, of tracking how many requests have been made over each persistent connection.

There are three options to consider for `maxKeepAliveRequests`:

- set it to a specific value, higher than the default;
- disable persistent connections by setting the value 1;
- set persistent connections to unlimited by setting the value -1.

Setting `maxKeepAliveRequests` to the value 1 in effect disables persistent connections because this sets the limit to 1. The other options of raising the limit to a specific value and unlimited require further analysis of the situation. Setting the value to unlimited is easiest because the platform will use whatever it calculates to be the optimal value for the current workload. However it's possible to run out of operating system file descriptors, if there are more concurrent clients than there are file descriptors available. Choosing a specific value is a more conservative approach and is less risky than using unlimited. The best method of finding the most suitable value is to monitor the maximum number of connections used and if this is higher than what's available, set it higher then do performance testing.

The importance of tuning the `maxKeepAliveRequests` value can be revealed in conducting performance tests. In a repeat of the above test, `maxKeepAliveRequests` was set to 1 (disabled) and the tests re-run. After only a little over 58,000 transactions, processing stopped because of I/O errors at the client. With `maxKeepAliveRequests` set to unlimited and the same tests run again, a 1.2%

increase in throughput was gained. Although this may seem a small increase, it can be significant. For example, if you take into account the 1.2% increase in the number of transactions that can be processed in this test, over a 12 hour period (e.g. one business day), it adds up to over 1.5 million more transactions that can be processed.

Below is an extract from `server.xml` in which `maxKeepAliveRequests` is set to `-1` or unlimited.

```
<!-- A HTTP/1.1 Connector on port 8080 -->
<Connector protocol="HTTP/1.1" port="8080" address="{jboss.bind.address}"
  connectionTimeout="20000" redirectPort="8443"
  maxKeepAliveRequests="-1" />
```

## 2.1.2. maxThreads

The `maxThreads` parameter creates the thread pool that sits directly behind the connector, and actually processes the request. It's very important to set this for most workloads as the default is quite low, currently 200 threads. If no threads are available when the request is made, the request is refused so getting this value right is critical. In the event the maximum number of threads is reached, this will be noted in the log:

```
2011-01-27 16:18:08,881 INFO [org.apache.tomcat.util.net.JIoEndpoint]
(http-192.168.1.22-8080-Acceptor-0) Maximum number of threads (200)
created for connector with address /192.168.1.22 and port 8080
```

The thread pool can be monitored via the administration console, as per the following screenshot:

The screenshot shows the JBoss EAP Admin Console interface. On the left is a navigation tree with the connector `http://192.168.1.22:8080` selected. The main content area displays the 'Metrics' tab for this connector. The status is 'Available'. Below the status, there are sections for 'Traits' (none available), 'Numeric Metrics', and a table of metrics.

Name	Value	Description
<b>Category: utilization</b>		
Request Count	600,636	the total number of requests processed since the last restart
Error Count	0	the number of errors while processing requests since the last restart
Current Active Threads	1	the number of threads for this connector that are currently active
Current Thread Count	200	the number of threads for this connector that currently exist
<b>Category: performance</b>		
Maximum Request Time	11,233 s	the maximum time it took to process a request since the last restart

Figure 2.1. Connectors metrics

On the left hand navigation pane, you can clearly see the connector, listening on port 8080, and in the metrics tab you can see the current utilization. On initial start of the platform, the thread pool is empty, and it creates threads on demand, based on incoming requests. This needs to be taken into consideration operationally, not necessarily performance wise. On start up there is some overhead in creating the thread pool but afterward there is no further overhead of servicing requests. On Linux, the overhead is quite low because the thread library can create and destroy native threads very quickly, but other operating systems have significant overhead, so the load will vary depending on which operating system the platform is hosted on.

### 2.1.3. minSpareThreads

The `minSpareThreads` parameter specifies the minimum number of threads that must be maintained in the thread pool. If sufficient resources are available set `minSpareThreads` to the same value as `maxThreads`. If `maxThreads` is set to represent a peak, but that peak does not last very long, set `minSpareThreads` to a lower value. That way, the platform can reclaim those resources as threads sit idle. There are a couple of ways to look at the interaction between these two parameters. If resource consumption is not an issue, set `minSpareThreads` to what you need to process at peak loads, but allow for an increase by setting `maxThreads` between 10% and 25% higher. If resource consumption is a concern, then set the `maxThreads` to only what you need for peak, and set the `minSpareThreads` to a value that won't hurt overall throughput after a drop in load occurs. Most applications have large swings in load over the course of a given day.



#### NOTE

From a performance testing perspective, it's also important to consider this thread creation overhead. It's best practice to allow an initial warm-up period, on start up of the platform, before making performance measurements. The measurement period should also be long enough to allow garbage collection to occur for all heap spaces, including Eden, Survivor, and Old generation. If this advice is not followed, results will be skewed by start up activity instead of reflecting sustained throughput and response times.

### 2.1.4. Tuning the thread pool

To size the thread pool correctly, you need to consider how many concurrent clients you will be serving through this connector. The number could vary greatly depending on what executes behind the HTTP request, and what response times are required. Setting the pool large is a reasonable option because threads are only created on demand. It's best to monitor the pool through the administration console, as the metrics tab shows the number of active threads, and how many it created. If the number of active threads reaches the maximum defined by the `maxThreads` parameter, it needs to be increased and again monitored.

Below is an extract from the `server.xml` configuration file in which all three parameters are assigned specific values.

```
<!-- A HTTP/1.1 Connector on port 8080 -->
<Connector protocol="HTTP/1.1" port="8080" address="{jboss.bind.address}"
  connectionTimeout="20000" redirectPort="8443" maxThreads="3000"
  minSpareThreads="2000" maxKeepAliveRequests="-1" />
```

## 2.2. AJP CONNECTOR

The AJP (Apache JServ Protocol) connector is typically used when a web server, such as Apache web server, is placed in front of the platform. Since AJP is its own binary protocol, configuration is different, but shares quite a bit of commonality in what can be configured with the Java I/O Endpoint. The following configuration parameters will be discussed:

- `maxThreads`
- `minSpareThreads`

Two of these parameters - `maxThreads` and `minSpareThreads` - are common with the Java I/O Connector. The AJP protocol also has a keep alive parameter but its default behavior is to always keep connections open, which avoids the overhead of setting up and tearing down connections. A common

configuration is to have the front-end web server in a DMZ, with a firewall allowing only incoming HTTP(S) to the web server, and another firewall between the web server and the platform, allowing only AJP traffic. Maintaining persistent connections in such a configuration is more efficient than the cost of constantly connecting and disconnecting the various components. Firewall configuration is another factor to consider because some firewalls close connections when no traffic has passed over a connection in a certain period.

The same considerations for the AJP connector apply as for the Java I/O connector. The thread pool is monitored in the same way in the administration console. The values set are also set in the same configuration file: `server.xml`. Below is an extract from `server.xml` in which both `maxThreads` and `minSpareThreads` have been configured:

```
<!-- A AJP 1.3 Connector on port 8009 -->
    redirectPort="8443" maxThreads="3000" minSpareThreads="2000" />
    <Connector protocol="AJP/1.3" port="8009"
address="${jboss.bind.address}"
```

## CHAPTER 3. SERVLET CONTAINER

There are two main configuration parameters that have a direct effect on performance and scalability: cached connection manager and HTTP session replication.

### 3.1. CACHED CONNECTION MANAGER

The Cached Connection Manager is used for debugging data source connections and supporting lazy enlistment of a data source connection in a transaction, tracking whether they are used and released properly by the application. At the cost of some overhead, it can provide tracing of the usage, and make sure that connections from a data source are not leaked by your application. Although that seems like an advantage, in some instances it's considered an anti-pattern and so to be avoided. If you are using bean managed transactions (BMT), it allows you to do the following (shown in pseudo-code):

```
Connection connection = dataSource.getConnection();
transaction.begin();
<Do some work>
transaction.commit();
```

Instead of:

```
transaction.begin();
Connection connection = datasource.getConnection();
<Do some work>
transaction.commit();
```

The first option is very useful for debugging purposes but should not be used in a production environment. In the **default**, **standard** and **all** configurations, the `CachedConnectionManager` is configured to be in the servlet container in debug mode. It's also configured in the **production** configuration but with debug mode off. If you do not use BMT, and/or you do not have the anti-pattern, described earlier, it's best to remove the `CachedConnectionManager`. The configuration is in the file `server.xml` in the directory `JBOSS_EAP_DIST/jboss-as/server/<PROFILE>/deploy/jbossweb.sar`. Note that the `minimal` configuration does *not* include JBoss Web.

Below is an extract from `server.xml` in which the `CachedConnectionManager` is enabled.

```
<!-- Check for unclosed connections and transaction terminated checks in
servlets/jsps.
Important: The dependency on the CachedConnectionManager in META-
INF/jboss-service.xml must be uncommented, too -->
<Valve className="org.jboss.web.tomcat.service.jca.CachedConnectionValve"

cachedConnectionManagerObjectName="jboss.jca:service=CachedConnectionManag
er"
    transactionManagerObjectName="jboss:service=TransactionManager" />
```

To disable the `CachedConnectionManager`, comment the last three lines, as per the following example:

```
<!-- Check for unclosed connections and transaction terminated checks in
servlets/jsps.
Important: The dependency on the CachedConnectionManager in META-
INF/jboss-service.xml must be uncommented, too
<Valve className="org.jboss.web.tomcat.service.jca.CachedConnectionValve"
```

```
cachedConnectionManagerObjectName="jboss.jca:service=CachedConnectionManager"

transactionManagerObjectName="jboss:service=TransactionManager" />
-->
```

Another configuration file also needs to be edited: `jboss-beans.xml` in the `JBoss_EAP_DIST/jboss-as/server/<PROFILE>/deploy/jbossweb.sar/META-INF` directory. Note that the minimal configuration does *not* include JBoss Web. This file is used by the micro-container for JBoss Web's integration with it, and it specifies the connections between the dependent components. In this case, the `CachedConnectionManager`'s valve is dependent on the transaction manager. So, in order to get rid of the valve properly, we have to remove the dependency information from this configuration file. The pertinent information is at the top of the file, and it looks like the following:

```
<!-- Only needed if the
org.jboss.web.tomcat.service.jca.CachedConnectionValve
    is enabled in the tomcat server.xml file.
-->
    <depends>jboss.jca:service=CachedConnectionManager</depends>
<!-- Transaction manager for unfinished transaction checking in the
CachedConnectionValve -->
<depends>jboss:service=TransactionManager</depends>
```

Comment these lines as in the following example:

```
<!-- Only needed if the
org.jboss.web.tomcat.service.jca.CachedConnectionValve
    is enabled in the tomcat server.xml file.
-->
    <!--<depends>jboss.jca:service=CachedConnectionManager</depends> -->
    <!-- Transaction manager for unfinished transaction checking in the
CachedConnectionValve -->
<!--<depends>jboss:service=TransactionManager</depends>-->
```

When editing XML, comments cannot be nested so it's important to get this correct. Refer to the section on the EJB 3 container for instructions on removing the `CachedConnectionManager` from there.

## 3.2. HTTP SESSION REPLICATION

For the servlet container, when deployed using clustering and a cluster aware application, HTTP session replication becomes a key aspect of performance and scalability across the cluster. There are two main methods for HTTP session replication: full replication, in which HTTP sessions are replicated to all nodes in the cluster, and buddy replication, where each node has at least one buddy, with one being the default. Each replication method has the ability to do full object replication, and fined grained replication, where only changes in the HTTP session are replicated.

### 3.2.1. Full Replication

Full replication is the default configuration because with most clusters having only a few nodes it provides maximum benefit with minimum configuration overhead. When there are more than two nodes, the method of replication needs to be considered as each option offers different benefits.



Although full replication requires the least configuration, as the number of nodes increases the overhead of maintaining inter-node connections increases. Factors which need to be considered include: how often new sessions are created, how often old sessions are removed, how often the data in the HTTP session changes, and the session's size. Making a decision requires answers to these questions and relies on knowledge of each applications' usage pattern of HTTP sessions.

To illustrate this with a practical example, consider a clustered application which uses the HTTP session to store a reference to a single stateful session bean. The bean is reused by the client to scroll through a result set derived from a database query. Even when the result set is no longer needed, and replaced within the stateful session bean with another result set, the stateful session bean session is not canceled, but instead a method is called that resets the state. So the HTTP session stays the same for the duration of the clients' requests, no matter how many are made and how long they are active. This usage pattern is very static, and involves a very small amount of data. It will scale very well with full replication, as the default network protocol is a reliable multicast over UDP. One packet (even assuming a 1,500 byte Ethernet frame size) will be sent over the network for all nodes to retrieve when a new HTTP session is created, and when one is removed for each user. In this case it's an efficient operation even across a fairly large cluster because the memory overhead is very small since the HTTP session is just holding a single reference. The size of the HTTP session is just one factor to consider. Even if the HTTP session is moderate in size, the amount of memory used calculated as:

```

HTTP session size
x active sessions
x number of nodes

```

As the number of clients and/or cluster nodes increases, the amount of memory in use across all nodes rapidly increases. The amount of memory in use is the same across all cluster nodes because they each need to maintain the same information, despite the fact that each client is communicating with only one node at a time.

In summary, whether you should use the default configuration of full replication is really up to your particular workload. If you have multiple applications deployed at the same time, the complexity increases, as you might have applications that behave quite differently between each other, where the HTTP session is concerned. If you have a cluster larger than two nodes, and you are doing HTTP session replication, then consider buddy replication as your starting point.

### 3.2.2. Buddy Replication

Buddy replication is a configuration in which state replication is limited to two or more cluster nodes, although the default is paired nodes. When failover occurs it's not limited to that node's buddy but can occur to any node and session information migrated to the new node if necessary. This method of replication is very scalable because the overhead in memory usage and network traffic is significantly decreased. Buddy replication is configured in a file called `jboss-cache-manager-jboss-beans.xml`, in the directory: `JBOSS_EAP_DIST/jboss-as/server/<PROFILE>/deploy/cluster/jboss-cache-manager.sar/META-INF`. Note that the `default`, `standard` and `minimal` configurations do not have the clustering configuration, nor do they have clustering code deployed.

```

<!-- Standard cache used for web sessions -->
<entry><key>standard-session-cache</key>
<value>
  <bean name="StandardSessionCacheConfig"
class="org.jboss.cache.config.Configuration">
    ...
    <property name="buddyReplicationConfig">
      <bean

```

```

class="org.jboss.cache.config.BuddyReplicationConfig">
    <!-- Just set to true to turn on buddy
replication -->
    <property name="enabled">true</property>

```

For buddy replication to work as advertised, it is highly recommended that you use sticky sessions. In other words, once a session is created in a node on the cluster, repeated requests will continue to go to the same node in the cluster. If you don't do this, you will defeat the advantages of buddy replication, as requests going to another node, let's say in a round robin fashion, will not have the state information, and the state will have to be migrated, similar to a fail-over scenario.

### 3.2.3. Monitoring JGroups via JMX

When the Enterprise Application Platform clustering services create a JGroups Channel to use for intra-cluster communication, they also register with the JMX server a number of MBeans related to that channel; one for the channel itself and one for each of its constituent protocols. For users interested in monitoring the performance-related behavior of a channel, a number of MBean attributes may prove useful.

#### **jboss.jgroups:cluster=<cluster\_name>,protocol=UDP,type=protocol**

Provides statistical information on the sending and receipt of messages over the network, along with statistics on the behavior of the two thread pools used to carry incoming messages up the channel's protocol stack.

Useful attributes directly related to the rate of transmission and receipt include **MessagesSent**, **BytesSent**, **MessagesReceived** and **BytesReceived**.

Useful attributes related to the behavior of the thread pool used to carry ordinary incoming messages up the protocol stack include **IncomingPoolSize** and **IncomingQueueSize**. Equivalent attributes for the pool of threads used to carry special, unordered "out-of-band" messages up the protocol stack include **OOBPoolSize** and **OOBQueueSize**. Note that **OOBQueueSize** will typically be 0 as the standard JGroups configurations do not use a queue for OOB messages.

#### **jboss.jgroups:cluster=<cluster\_name>,protocol=UNICAST,type=protocol**

Provides statistical information on the behavior of the protocol responsible for ensuring lossless, ordered delivery of unicast (i.e. point-to-point) messages.

The ratio of **NumRetransmissions** to **MessagesSent** can be tracked to see how frequently messages are not being received by peers and need to be retransmitted. The **NumberOfMessagesInReceiveWindows** attribute can be monitored to track how many messages are queuing up on a recipient node waiting for a message with an earlier sequence number to be received. A high number indicates messages are being dropped and need to be retransmitted.

#### **jboss.jgroups:cluster=<cluster\_name>,protocol=NAKACK,type=protocol**

Provides statistical information on the behavior of the protocol responsible for ensuring lossless, ordered delivery of multicast (i.e. point-to-multipoint) messages.

Use the **XmitRequestsReceived** attribute to track how often a node is being asked to re-transmit a messages it sent; use **XmitRequestsSent** to track how often a node is needing to request retransmission of a message.

#### **jboss.jgroups:cluster=<cluster\_name>,protocol=FC,type=protocol**

Provides statistical information on the behavior of the protocol responsible for ensuring fast message senders do not overwhelm slow receivers.

Attributes useful for monitoring whether threads seeking to send messages are having to block while waiting for credits from receivers include **Blockings**, **AverageTimeBlocked** and **TotalTimeBlocked**.

## CHAPTER 4. EJB 3 CONTAINER

The EJB 3 container is the JBoss Enterprise Application Platform's implementation of the Java EE 5 specifications for EJB 3.0. It implements all the standard bean types described by the specification e.g. stateless session beans, stateful session beans, message driven beans, and entity beans. While these all share the same names as EJB 2.x components, they are now based on a POJO development model using annotations, or optionally deployment descriptors. There are various configuration parameters that will affect the throughput of your application depending on which of these beans you use. Each bean type will be discussed in this chapter, including configuration parameters that affect overall throughput and the inter-relationships between them.

### 4.1. STATELESS SESSION BEAN

With stateless session beans, our implementation uses a pooling strategy to offer a “method ready” instance for use by the application. There are two types of pools that can be configured for use with stateless session beans:

- ThreadLocalPool
- StrictMaxPool

#### 4.1.1. ThreadLocalPool

The ThreadLocalPool is a pool of instances in a thread local variable and is the default pool type. It's local to each execution thread within the platform that executes methods on a stateless session bean. The ThreadLocalPool is derived from an unlimited pool implementation, so there is no limit to the number of instances that can be in the pool. The actual number of instances is dependent on other parameters. For each thread that executes methods on a stateless session bean within your application, the pool size of the ThreadLocalPool is the number of unique stateless sessions. Since there is a pool on each thread, the actual number of instances is the number of stateless session beans in your deployments multiplied by the number of threads that execute methods on those beans. This approach has a configuration advantage because the size of the pool is changed dynamically with changes to the application - e.g. the adding or removing of stateless session beans over time.

#### 4.1.2. StrictMaxPool

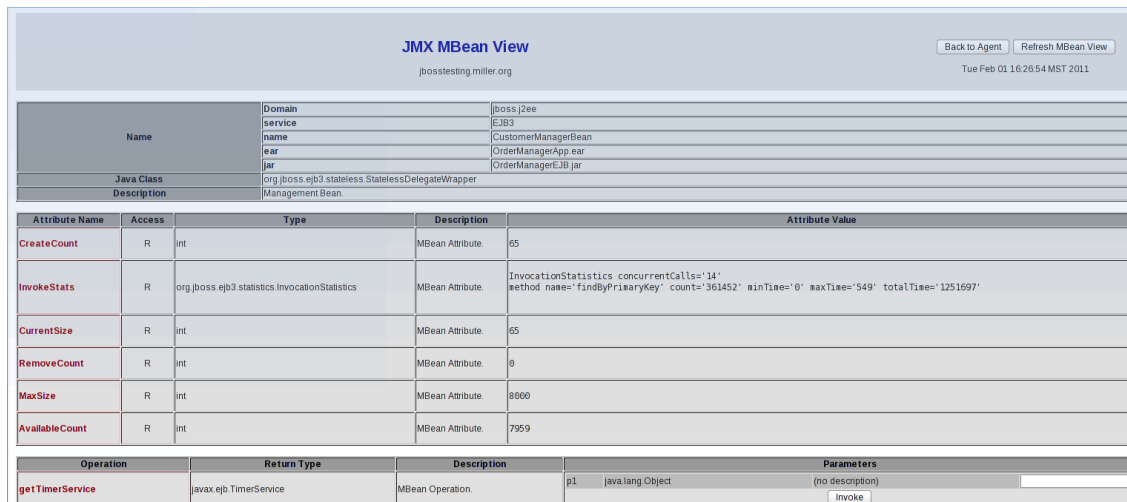
StrictMaxPool is the other pool implementation, a single pool used across all execution threads, that contains instances of stateless session beans. It has a maximum size so if there are no available instances in the pool for a thread to execute, the calling thread will block and wait until there is an available instance. If a StrictMaxPool is used for stateless session beans and it's not sized correctly, this creates a bottleneck. In some circumstances this might be used deliberately to limit the rate of transactions but it's not recommended.

If there are no available instances in the pool at the time a call is made, the calling thread will be blocked but only for a certain amount of time (which can vary depending on the configuration). If that time period is exceeded, the error message below will be logged, confirming that the pool is too small. This situation may not be obvious but can result in significant performance degradation. When using the StrictMaxPool, correct sizing is critical and this is covered in the next section.

```
javax.ejb.EJBException: Failed to acquire the pool semaphore,  
strictTimeout=10000
```

#### 4.1.3. Pool Sizing

It's recommended to set the size of the pool equal to the maximum number of concurrent requests. Confirming the maximum can best be done by monitoring statistics using the JMX console. In this example the maximum size of the pool is 8,000, which is quite large based on the other data that is shown. The CreateCount metric shows that the platform created 65 beans for the pool, and that 7,959 is the current available count. This means at that moment, there are 41 instances of the bean in use.



The screenshot shows the JMX MBean View for a StatelessDelegateWrapper MBean. The interface includes a header with the title 'JMX MBean View', a user identifier 'jbossstesting miller.org', and buttons for 'Back to Agent' and 'Refresh MBean View'. The timestamp is 'Tue Feb 01 16:26:54 MST 2011'. Below the header is a table with the following data:

Name	Domain	Value
service	jboss:j2ee	
name	EJB3	
ear	CustomerManagerBean	
jar	OrderManagerApp.ear	
jar	OrderManagerEJB.jar	
Java Class	org.jboss.ejb3.stateless.StatelessDelegateWrapper	
Description	Management Bean	

Below this is a table of attributes:

Attribute Name	Access	Type	Description	Attribute Value
CreateCount	R	int	MBean Attribute	65
InvokeStats	R	org.jboss.ejb3.statistics.InvocationStatistics	MBean Attribute	InvocationStatistics concurrentCalls='14' method name='findByPrimaryKey' count='361452' minTime='0' maxTime='549' totalTime='1251697'
CurrentSize	R	int	MBean Attribute	65
RemoveCount	R	int	MBean Attribute	0
MaxSize	R	int	MBean Attribute	8000
AvailableCount	R	int	MBean Attribute	7959

At the bottom is a table of operations:

Operation	Return Type	Description	Parameters
getTimerService	javax.ejb.TimerService	MBean Operation	p1 java.lang.Object (no description) <input type="text"/> <input type="button" value="Invoke"/>

Figure 4.1. JMX Console - JMX MBean View

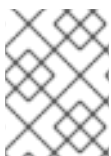


## NOTE

The statistics presented in the JMX console must be manually refreshed and, because of its implementation, are not guaranteed to be accurate but should be used as a guide only.

There is no absolute "best" pool implementation because much depends on the application and its deployment. The recommended method of deciding between the two is to test both with your application and infrastructure. For StrictMaxPool it's critical that the pool's usage be monitored to ensure that it's large enough, otherwise performance statistics will be skewed.

Stateless session beans are configured in the file `ejb3-interceptors-aop.xml`, which is located in the directory: `JBOSS_EAP_DIST/jboss-as/server/<PROFILE>/deploy`. The relevant sections are titled "Stateless Bean" and "JACC Stateless Bean". Stateless Bean is a basic, stateless bean while JACC Stateless Bean provides for permission classes to satisfy the Java EE authorization model. Note that the `minimal` configuration does *not* include the EJB 3 container.



## NOTE

Since for ThreadLocalPool there is no limit to the number of instances that can be in the pool, the `maxSize` and `timeout` parameters are irrelevant.

### 4.1.4. Local EJB3 Calls

When an application uses a remote EJB3 interface the call is serialized and deserialized by default because this is a requirement of network communication. If the EJB is running in the same JVM as the client calling it, all communications are local so there's no need for serialization or deserialization. To avoid the CPU load involved and so improve the application's responsiveness, enable the system property `org.jboss.ejb3.IsLocalInterceptor.passByRef` to true, for example: add `-Dorg.jboss.ejb3.remoting.IsLocalInterceptor.passByRef=true` to `JAVA_OPTS` in the server's `run.conf` configuration file.

## 4.2. STATEFUL SESSION BEAN

As a stateful component, stateful session beans require a different implementation from the container. First, because they maintain state, that state is associated exclusively with one client, so there is only one instance per client. Second, because they maintain state and are tied to one client, the container must prevent any concurrent modification to that state. Third, because they maintain state, that state has to participate in replication for a clustered environment. Finally, if the instance is not accessed in a period of time, and the bean is not canceled, the state may be passivated to disk. All these factors play a role in the throughput that can be achieved when using stateful session beans. It's important to choose the right state management solution for your application. Stateful session beans can be useful, especially where the number of concurrent users is not too large- i.e. into millions.

To help with understanding the difference that a stateless vs. stateful approach can have, consider a real-life situation where a stateless approach was the imposed standard. In this situation there was an entire set of services in the application that looked-up data for users for use in maintenance tasks, or placing orders, etc. In almost all cases, these “look-ups” would return more data than the user could effectively deal with, and there was logic in the client to page through the list. As it paged through the list, if it needed to get more data than was on the client at that time, it called the server again. When it did this it actually passed a row number to the server to say, starting at this row number, return back the next set of rows. This made the queries very complex, as the queries themselves had to scroll through the data and return the correct set. Of course the user could page down through, but also back up again, and this process was repeated many times in certain cases. So, the design was stateless, but the load it actually exacted on the database was extraordinarily high, and the response time of the services themselves suffered. Users complained of poor response times as well. To solve this issue the services were made stateful (with one instance per client), the paging logic was removed from the queries themselves, and scrolled through the data in memory as the client paged through the rows. All queries were limited to a maximum number of rows, because returning more than that meant that the search criteria for the look-up should be expanded to narrow down the result set to make it usable for the end user anyway. The results of the change to a stateful approach were a significant decrease in database load and an improvement in response times. This is a good example of how stateful session beans can be used, and used effectively.

### 4.2.1. Stateful Configuration

There are two aspects of stateful session beans to be considered: the underlying cache that supports the life-cycle of the bean and replication of the bean's state.

In the same file that configures HTTP session replication is the configuration for stateful session bean replication. The configuration file is `jboss-cache-manager.sar`, in the directory `JBOSS_EAP_DIST/jboss-as/server/PROFILE/deploy/cluster/jboss-cache-manager.sar/META-INF`. Note that the `default`, `standard` and `minimal` configurations do not have the clustering configuration, nor the clustering code deployed. In this file is an entry called `sfsb-cache`, as seen below:

```
<!-- Standard cache used for EJB3 SFSB caching -->
<entry><key>sfsb-cache</key>
<value>
  <bean name="StandardSFSBCacheConfig"
class="org.jboss.cache.config.Configuration">
  <!-- No transaction manager lookup -->
  <!-- Name of cluster. Needs to be the same for all members -->
  <property name="clusterName">${jboss.partition.name:DefaultPartition}-
SFSBCache</property>
  <!-- Use a UDP (multicast) based stack. Need JGroups flow control (FC)
because we are using asynchronous replication. -->
```

```

<property
name="multiplexerStack">${jboss.default.jgroups.stack:udp}</property>
<property name="fetchInMemoryState">true</property>
<property name="nodeLockingScheme">PESSIMISTIC</property>
<property name="isolationLevel">REPEATABLE_READ</property>

```

The two most important configuration parameters are the node locking scheme and isolation level, which are analogous to databases. Pessimistic locking assumes that multiple clients should not own the lock on the data, in this case the data in the cache, at the same time. The isolation level is similar to the concept in database management systems. Since the container above this cache already prevents more than one client modifying the state of a bean, changing the node locking scheme provides no advantage.

### 4.2.2. Full Replication

As we discussed in HTTP session replication, full replication is the default, and it will perform just fine in two-node clusters. State in stateful session beans tends to be even more complex than state in the HTTP session, so use caution in using full replication for stateful session beans than for the HTTP session. If buddy replication is used for either HTTP sessions or stateful session beans, and the web tier of the platform (servlet container) is not separated from the rest of the platform, it is good practice to use the same method for both. Even if you can get away with full replication on HTTP sessions, your use of stateful session beans may drive you towards buddy replication.

### 4.2.3. Buddy Replication

Again, as we discussed with HTTP session replication, buddy replication is available for stateful session beans as well. The configuration file for buddy replication is `jboss-cache-manager-jboss-beans.xml`, in the directory `JBOSS_EAP_DIST/jboss-as/server/PROFILE/deploy/cluster/jboss-cache-manager.sar/META-INF`. Note that the `default`, `standard` and `minimal` configurations do not have the clustering configuration, nor the clustering code deployed. To configure buddy replication for stateful session beans, change the `buddyReplicationConfig`'s `property` to `true`, as in the following extract.

```

<!-- Standard cache used for EJB3 SFSB caching -->
<entry><key>sfsb-cache</key>
  <value>
    <bean name="StandardSFSBCacheConfig"
class="org.jboss.cache.config.Configuration">
      ...
      <property name="buddyReplicationConfig">
        <bean class="org.jboss.cache.config.BuddyReplicationConfig">
          <!-- Just set to true to turn on buddy replication -->
          <property name="enabled">true</property>

```

## 4.3. REMOTE EJB CLIENTS

The configuration options discussed so far apply to both local and remote clients, but there are additional configuration options that needs to be considered where remote clients are concerned: `maxPoolSize` and `clientMaxPoolSize`. Local clients use the local interface of stateless and stateful session beans, so the execution of those methods is done on the incoming connectors' thread pool. Remote clients call the remote interfaces of stateless and stateful session beans, interacting with the platform's remoting solution. It accepts connections from remote clients, marshals the arguments and calls the remote interface of the corresponding bean, whether that is a stateless or stateful session bean. Note that the `minimal` configuration does *not* have the remoting service deployed.

The configuration file is `remoting-jboss-beans.xml`, in the directory `JBOSS_EAP_DIST/jboss-as/server/<PROFILE>/deploy`, an extract of which is below:

```
<!-- Parameters visible only to server -->
<property name="serverParameters">
  <map keyClass="java.lang.String" valueClass="java.lang.String">
    <!-- Selected optional parameters: -->
    <!-- Maximum number of worker threads on the -->
    <!-- server (socket transport). Defaults to 300. -->
    <!--entry><key>maxPoolSize</key> <value>500</value></entry-->
      <!-- Number of seconds after which an idle worker thread will be
-->
      <!-- purged (socket transport). By default purging is not enabled.
-->
      <!--entry><key>idleTimeout</key> <value>60</value></entry-->
    </map>
  </property>
```

The most important parameter here is the `maxPoolSize` parameter which specifies the number of worker threads that will be used to execute the remote calls. The comments state that the default is 300, which is quite large, but that depends on the number of remote clients, their request rate, and the corresponding response times. Just like any other connector, the number of clients calling the server, their request rate, and the corresponding response times need to be taken into account to size the pool correctly. One way to determine this is to monitor the statistics on the client to see how many invocations occur. For example, if the remote client is a stateless session bean on another server, the JMX console contains invocation statistics that could be examined to determine how many concurrent calls are coming from that client. If there are many remote servers acting as clients then statistics have to be obtained from each remote client. To change the pool size, uncomment the line and set the value.

Consider the opposite case, where the client is the local server, calling another remote server. There is a matching pool parameter called `clientMaxPoolSize`, specified in the same configuration file as above:

```
<!-- Maximum number of connections in client invoker's -->
  <!-- connection pool (socket transport). Defaults to 50. -->
  <!--entry><key>clientMaxPoolSize</key> <value>20</value></entry-->
```

In this example the parameter is present but commented so the default value of 50 applies. To change the value, uncomment the line and change the value to the desired value.

To determine the required value it's necessary to understand how many concurrent calls are being made to remote beans. Depending on the method used, there is some monitoring you can do. For example, if the remote bean invocations are being made from a local stateless session bean the JMX console statistics can be used. Included are the statistics for that stateless session's bean and `concurrentCalls` value to see how many concurrent invocations are being made, as in the following example:



**JMX MBean View**  
jbossstesting miller.org

Back to Agent | Refresh MBean View  
Tue Feb 01 16:26:54 MST 2011

Name	Domain	Value
service	jboss.j2ee	EJB3
name		CustomerManagerBean
ear		OrderManagerApp.ear
jar		OrderManagerEJB.jar
Java Class	org.jboss.ejb3.stateless.StatelessDelegateWrapper	
Description	Management Bean	

Attribute Name	Access	Type	Description	Attribute Value
CreateCount	R	int	MBean Attribute	65
InvokeStats	R	org.jboss.ejb3.statistics.InvocationStatistics	MBean Attribute	InvocationStatistics concurrentCalls='14' method name='findByPrimaryKey' count='361452' minTime='0' maxTime='549' totalTime='1251697'
CurrentSize	R	int	MBean Attribute	65
RemoveCount	R	int	MBean Attribute	8
MaxSize	R	int	MBean Attribute	8000
AvailableCount	R	int	MBean Attribute	7959

Operation	Return Type	Description	Parameters
getTimerService	javax.ejb.TimerService	MBean Operation	pt java.lang.Object (no description) Invoke

**Figure 4.2. JMX Console - JMX MBean View**

The screenshot shows the attribute **InvokeStats** and it has an attribute value that shows the **InvocationStatistics** class, and the value **concurrentCalls** equals 14. This could give you a hint about how many client connections need to be available in the pool.

## 4.4. CACHEDCONNECTIONMANAGER

The servlet container chapter described the cached connection manager, what it does, and how you can remove it. This section shows how to remove it from the EJB 3 container, and a short note about how you can remove it from the old EJB 2.x container as well. Note that the **minimal** configuration does *not* contain the EJB 3 container.

The configuration for the cached connection manager, in the EJB 3 container, is **ejb3-interceptors-aop.xml**, in the directory: **JBOSS\_EAP\_DIST/jboss-as/server/<PROFILE>/deploy**.

To remove the cached connection manager, comment or remove the following two lines from the configuration file.

```
<interceptor
  factory="org.jboss.ejb3.connectionmanager.CachedConnectionInterceptorFactory"
  scope="PER_CLASS" />
<interceptor-ref
  name="org.jboss.ejb3.connectionmanager.CachedConnectionInterceptorFactory"
  />
```

The configuration of the EJB 2.x container is in **standardjboss.xml**, in the directory: **JBOSS\_EAP\_DIST/jboss-as/server/<PROFILE>/conf**. Note that the **minimal** configuration, does *not* contain the EJB 2.x container.

To remove the cached connection manager, comment or remove the following line from each container configuration (there are many in the file).

```
<interceptor>org.jboss.resource.connectionmanager.CachedConnectionInterceptor</interceptor>
```

## 4.5. ENTITY BEANS

Entity beans are avoided by some because of historical issues with EJB 1 and EJB 2 but with EJB 3 their use is rising. In discussing how to get the best possible throughout while using entity beans, there are four topics to cover:

- second-level cache
- prepared statements
- batch inserts
- batching database operations

### 4.5.1. Second level cache

As Hibernate is the JPA provider, entity beans in EJB 3 sit on top of Hibernate. This is in stark contrast to the old EJB 2.x entities, which had their own complete implementation apart from Hibernate. In fact, Hibernate, and other object relational mapping (ORM) frameworks were the inspiration for JPA and EJB 3 entities. Since Hibernate is the underlying implementation, we have a second level cache that we can utilize, just as if we were using Hibernate directly instead of entities. Hibernate's second level cache has gone through some evolution over the years, and has improved with each release. The JBoss Cache implementation as the second level cache, has some very useful features, and the default configuration for EJB 3 entities is very good. To enable the use of the second level cache for entities in an EJB3 application, persistence units are defined in the `persistence.xml` file that is packaged with an application. Here is an extract from `persistence.xml`:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="services" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/MySqlDS</jta-data-source>
    <properties>
      <property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory"/>
      <property name="hibernate.cache.region.jbc2.cachefactory"
value="java:CacheManager"/>
      <property name="hibernate.cache.use_second_level_cache"
value="true"/>
      <property name="hibernate.cache.use_query_cache"
value="false"/>
      <property name="hibernate.cache.use_minimal_puts"
value="true"/>
      <property name="hibernate.cache.region.jbc2.cfg.entity"
value="mvcc-entity"/>
      <property name="hibernate.cache.region_prefix"
value="services"/>
    </properties>
  </persistence-unit>
</persistence>
```

The configuration parameters relevant to the second level cache are:

`hibernate.cache.region.factory_class` specifies the cache factory to be used by the underlying Hibernate session factory, in this example `JndiMultiplexedJBossCacheRegionFactory`. This factory implementation creates a single cache for all types of data that can be cached (entities, collections, query results and timestamps). With other options you can create caches that are tailored to each type of data, in separate cache instances. In this example, there is only one cache instance, and only entities and collections are cached. The second important parameter above is the `hibernate.cache.use_second_level_cache`, which is set to true, enabling the cache. The query cache is disabled with `hibernate.cache.use_query_cache` set to false.

`hibernate.cache.use_minimal_puts`, set to true, specifies the behavior of writes to the cache. It minimizes the writes to the cache at the expense of more reads, the default for a clustered cache.

`hibernate.cache.region.jbc2.cfg.entity` specifies the underlying JBoss Cache configuration, in this case the multiversion concurrency control (MVCC) entity cache (mvcc-entity).

`hibernate.cache.region_prefix` is set to the same name as the persistent unit itself. Specifying a name here is optional, but if you do not specify a name, a long default name is generated. The mvcc-entity configuration is in the file `jboss-cache-manager-jboss-beans.xml`, in the directory: `JBOSS_EAP_DIST/jboss-as/server/<PROFILE>/deploy/cluster/jboss-cache-manager.sar/META-INF`. Note that the default, standard and minimal configurations do not have the JBoss Cache configured or deployed.

Below is an extract from the configuration of the MVCC cache:

```
<!-- A config appropriate for entity/collection caching that uses MVCC
locking -->
  <entry><key>mvcc-entity</key>
  <value>
    <bean name="MVCCEntityCache"
class="org.jboss.cache.config.Configuration">
<!-- Node locking scheme -->
<property name="nodeLockingScheme">MVCC</property>
<!-- READ_COMMITTED is as strong as necessary for most
2nd Level Cache use cases. -->
<property name="isolationLevel">READ_COMMITTED</property>
<property name="useLockStriping">>false</property>
<!-- Mode of communication with peer caches.
INVALIDATION_SYNC is highly recommended as the mode for use
with entity and collection caches. -->
<property name="cacheMode">INVALIDATION_SYNC</property>
  <property name="evictionConfig">
    <bean class="org.jboss.cache.config.EvictionConfig">
      <property name="wakeUpInterval">5000</property>
      <!-- Overall default -->
      <property name="defaultEvictionRegionConfig">
        <bean class="org.jboss.cache.config.EvictionRegionConfig">
          <property name="regionName"></property>
          <property name="evictionAlgorithmConfig">
            <bean
class="org.jboss.cache.eviction.LRUAlgorithmConfig">
              <!-- Evict LRU node once we have more than this
number of nodes -->
                <property name="maxNodes">500000</property>
                <!-- And, evict any node that hasn't been accessed
in this many seconds -->
                <property name="timeToLiveSeconds">7200</property>
```

```
        <!-- Do not evict a node that's been accessed within
this many seconds.
        Set this to a value greater than your max
expected transaction length. -->
        <property name="minTimeToLiveSeconds">300</property>
    </bean>
</property>
</bean>
</property>
```

In the configuration above, the following parameters are of particular interest in tuning the cache:

- `isolationLevel`
- `cacheMode`
- `maxNodes`
- `timeToLiveSeconds`
- `minTimeToLiveSeconds`

`isolationLevel` is similar to database isolation level for transactions. JBoss Cache is fully transactional and can participate as a full resource in transactions, so that stale data is not stored in the cache. Some applications may not be affected by stale data in a cache so configuration can vary accordingly. The default is `READ_COMMITTED`, which is the same as in the example data source configuration for the database connection pool. It's recommended to set this the same as in the data source to avoid odd behavior in the application. JBoss Cache supports the following isolation levels:

- `NONE`
- `READ_UNCOMMITTED`
- `READ_COMMITTED`
- `REPEATABLE_READ`
- `SERIALIZABLE`

The default is `REPEATABLE_READ`, which is used in the example configuration.

`cacheMode` specifies that across the cluster, cached entities will be invalidated on other nodes, so that another node does not return a different value. Invalidation is done in a synchronous manner, which ensures that the cache is in a correct state when the invalidation request completes. This is very important for caches when in a cluster, and is the recommended setting. Replication, instead of invalidation, is an option, but is much more expensive and limits scalability, possibly preventing caching from being effective in providing increased throughput. In this example, `cacheMode` is set to `INVALIDATION_SYNC`.

The following three parameters - `maxNodes`, `timeToLiveSeconds`, and `minTimeToLiveSeconds` - define the size of the cache and how long things live in the cache.

`maxNodes` specifies the maximum number of nodes that can be in the cache at any one time. The default for `maxNodes` is 10,000, which is quite small, and in the example configuration it was set to 500,000. Deciding how large to make this value depends on the entities being cached, the access

pattern of those entities, and how much memory is available to use. If the cache uses too much memory, other platform components could be starved of resources and so performance may be degraded. If the cache is too small, not enough entities may be stored in the cache to be of benefit.

`timeToLiveSeconds` specifies how long something remains in the cache before it becomes eligible to be evicted. The default value is 1,000 seconds or about 17 minutes, which is a quite short duration. Understanding the access and load pattern is important. Some applications have very predictable load patterns, where the majority of the load occurs at certain times of day, and lasts a known duration. Tailoring the time that entities stay in the cache towards that pattern helps tune performance.

`minTimeToLive` sets the minimum amount of time an entity will remain in the cache, the default being 120 seconds, or two minutes. This parameter should be set to equal or greater than the maximum transaction timeout value, otherwise it's possible for a cached entity to be evicted from the cache before the transaction completes.

#### 4.5.1.1. Marking entities to be cached

The `@Cache` annotation is added on an entity bean you want to cache and it takes one argument: `CacheConcurrencyStrategy`. The `@Cache` annotation requires the following two imports:

- `import org.hibernate.annotations.Cache;`
- `import org.hibernate.annotations.CacheConcurrencyStrategy;`

The `@Cache` annotation looks like the following in the entity code: `@Cache(usage = CacheConcurrencyStrategy.READ_ONLY)` where the `CacheConcurrencyStrategy` can be:

- `NONE`
- `NONSTRICT_READ_WRITE`
- `READ_ONLY`
- `TRANSACTIONAL`

Of these options, only two strategies are relevant to JBoss Cache as the second level cache provider: `READ_ONLY`, and `TRANSACTIONAL`.

**READ\_ONLY** guarantees that the entity will never change while the application is running. This allows the use of read only semantics, which is by far the most optimal performing cache concurrency strategy.

**TRANSACTIONAL** allows the use of database ACID semantics on entities in the cache. Anything to be cached while the application is running should be marked `TRANSACTIONAL`. Avoid caching entities that are likely to be updated frequently. If an entity is updated too frequently, caching can actually increase overhead and so slow throughput. Each update sends an invalidation request across the cluster, so that the state is correctly maintained as well, so the overhead affects every node, not only the node where the update occurs.

Starting with the JBoss Cache 3.x series, which made its debut in EAP 5.0.0, we have a transactional cache that uses MVCC. Also, the `mvcc-entity` configuration we looked at earlier is the default for entities with the platform. MVCC is a well-known algorithm that allows updates to be in process, but not blocking other transactions from reading the data. So writers (updates) do not block readers from reading a consistent image of the data. This is very important for concurrency, as it's not a pessimistic lock that will block anyone from reading the data. For the window of time that a transaction may be updating a particular entity, other transactions in flight that are readers of that entity will not block, and get a consistent image of the data for their use, until, of course, the transaction commits. This

provides a level of scalability that was non-existent in any second level cache providers until JBoss Cache 3.x introduced it (at least for updates). Of course, multiple updates in different transactions to the same entity will still block. Once again the read/write ratios are extremely important to getting good throughput with any entities that are cached and can be updated while the application is running.

### 4.5.2. Prepared Statements

When using the JPA annotations for queries, the result is prepared statements that will be executed against the database. Prepared statements have two phases for execution: preparation of the statement, and execution of the statement. Statement preparation involves significant CPU load so to improve throughput prepared statements can be cached. Statements can be cached either via the JDBC driver or configuration of the data source. The method recommended here is to configure the data source because it works whether or not the JDBC driver provides for caching of prepared statements. For example, the MySQL JDBC driver's prepared statement caching is done on a connection by connection basis. In the case of many connections, this implementation takes up considerably more memory, also each prepared statement must be cached for each connection in the pool.

To enable caching of prepared statement, add the following two lines to the data source configuration file, a file with the pattern `*-ds.xml` (where the `*` is usually your database, such as `oracle`, `mysql`, `db2`, etc.) in the directory: `JBOSS_EAP_DIST/jboss-as/server/PROFILE/deploy`. Note that the `minimal` configuration does *not* support data sources.

```
<prepared-statement-cache-size>100</prepared-statement-cache-size>  
<shared-prepared-statements>true</shared-prepared-statements>
```

The first line enables the prepared statement cache and sets the size of the cache. This should be large enough to hold all the prepared statements across any and all deployed applications using this particular data source (multiple data sources can be configured). The second line states that if two requests are executed in the same transaction the prepared statement cache should return the same statement. This will avoid the use of CPU cycles in preparing the same statements over and over again, increasing throughput and decreasing response times. The actual improvement will vary according to specific circumstances but is well worth the effort.

### 4.5.3. Batch Inserts

Batch inserts is the ability to send a set of inserts to a single table, once to the database as a single insert statement instead of individual statements. This method improves latency, and data insert times. These improvements occur not only with batch processing, loading large quantities of data, but also with OLTP workloads.

The following is an example of regular inserts:

```
INSERT INTO EJB3.OrderLine (OrderId, LineNumber, ProductId, Quantity,  
Price, ExtendedPrice) VALUES("67ef590kalk4568901thbn7190akioe1", 1, 25,  
10, 1.00, 10.00);  
INSERT INTO EJB3.OrderLine (OrderId, LineNumber, ProductId, Quantity,  
Price, ExtendedPrice) VALUES("67ef590kalk4568901thbn7190akioe1", 2, 16, 1,  
1.59, 1.59);  
INSERT INTO EJB3.OrderLine (OrderId, LineNumber, ProductId, Quantity,  
Price, ExtendedPrice) VALUES("67ef590kalk4568901thbn7190akioe1", 3, 55, 5,  
25.00, 125.00);  
INSERT INTO EJB3.OrderLine (OrderId, LineNumber, ProductId, Quantity,  
Price, ExtendedPrice) VALUES("67ef590kalk4568901thbn7190akioe1", 4, 109,  
1, 29.98, 29.98);
```

The following is an example of batch inserts:

```
INSERT INTO EJB3.OrderLine (OrderId, LineNumber, ProductId, Quantity,
Price, ExtendedPrice) VALUES("67ef590kalk4568901thbn7190akioe1", 1, 25,
10, 1.00, 10.00)
, ("67ef590kalk4568901thbn7190akioe1", 2, 16, 1, 1.59, 1.59)
, ("67ef590kalk4568901thbn7190akioe1", 3, 55, 5, 25.00, 125.00)
, ("67ef590kalk4568901thbn7190akioe1", 4, 109, 1, 29.98, 29.98);
```

Before discussing how to enable this behavior from an EJB 3 application using Hibernate, the example data model above needs further explanation. The primary key of the EJB3.OrderLine table is the OrderId, and the LineNumber. Originally, the entity Order and OrderLine had a MySQL auto increment column as the OrderId, and the primary key of Order, and the first half of the primary key of the OrderLine entity. The SQL syntax for normal inserts is unaffected by this, as Hibernate does not need to know the primary key value ahead of time. In fact, the order in which those SQL statements are sent to the database does not matter. With batch inserts, since we want to send them all over as one set, and eventually as one SQL statement, Hibernate needs to know the primary key ahead of time. This is for the simple reason that Hibernate pushes each "persist" request from an EJB 3 application into an action queue, and there may be inserts to various tables interleaved in a single transaction. Hibernate has to sort the inserts by primary key in order to batch them at all. An auto increment column value is only known after the insert, so this will fail. Therefore the model has to use a primary key that can be known prior to being inserted into the database. A generated value can be used as the primary key, but the generation strategy used must allow it to be retrieved via JDBC before the inserts are flushed to the database. IDENTITY cannot be used as a primary key generation strategy, but TABLE, SEQUENCE and UUID can be used. It's important to know that when Hibernate performs sorting to get the insert statements batched together it uses the hash code value of the entity so it's imperative that the hash code method is properly defined.



#### NOTE

UUID is a Hibernate-specific key generation technique and is not in the JPA or JPA 2 specification so code will not be portable if it is used.

The following code sample illustrates batch processing. It's an extract of the Order performance application, and inserts reference data that is needed for the test runs.

```
@TransactionTimeout(4800)
public void createInventories(int batchSize) throws CreateDataException {
    if(numberOfInventoryRecords() > 0) {
        throw new CreateDataException("Inventory already exists!");
    }
    int rowsToFlush = 0, totalRows = 0;
    Random quantity = new Random(System.currentTimeMillis());
    List<Product> products = productManager.findAllProducts();
    List<DistributionCenter> distributionCenters =
distributionCenterManager.findAllDistributionCenters();
    InventoryPK inventoryPk = null;
    Inventory inventory = null;
    for(Product product: products) {
        for(DistributionCenter distributionCenter: distributionCenters) {
            inventoryPk = new InventoryPK();
            inventory = new Inventory();
            inventoryPk.setProductId(product.getProductId());
            inventoryPk.setDistributionCenterId(distributionCenter.getDistributionCent
```



```

erId());
inventory.setPrimaryKey(inventoryPk);
inventory.setQuantityOnHand(quantity.nextInt(25000));
inventory.setBackorderQuantity(0);
inventory.setVersion(1);
batchEntityManager.persist(inventory);
rowsToFlush++;
totalRows++;
    if(rowsToFlush == batchSize) {
        batchEntityManager.flush();
        rowsToFlush = 0;
        batchEntityManager.clear();
        if(log.isTraceEnabled()) {
            log.trace("Just flushed " + batchSize + " rows to the
database.");
            log.trace("Total rows flushed is " + totalRows);
        }
    }
}
return;
}

```

The method is annotated with a `TransactionTimeout` annotation that specifies a longer transaction timeout value, because the default of 300 seconds is too short in this instance. The method takes a single parameter called `batchSize` which allows tuning of the batch insert size. This is good practice because it allows different batch insert sizes to be tested to optimize the results. After creating the inventory entity, `persist` is called on the entity manager. The code that follows is significant in demonstrating the batch processing method.

After calling `persist` a check is made to confirm that the number of entities or rows persisted is equal to the `batchSize` parameter. If so, the entity manager's `flush` method is called. Since this is a single large transaction, the normal Hibernate behavior is to flush everything on the transaction commit, or when the JDBC batch size has been reached. To flush a specific number of rows, instead of letting Hibernate decide, an explicit call to flush is made. Note that Hibernate will flush at two intervals: at the commit of the transaction and when it gets to the JDBC batch size parameter. So there's no need to explicitly call `flush` at all, instead set the JDBC batch size parameter and avoid the batch size parameter altogether. In this case the batch size parameter was explicitly set to the maximum once, then different batch sizes less than or equal to the JDBC batch size parameter were tested. This could be simpler, but the alternative would be to change the Hibernate configuration and redeploy the application in between tests. The Hibernate JDBC batch size parameter (`hibernate.jdbc.batch_size`) is specified in the properties values of `persistence.xml` for the persistence unit of the application entities.

The parameter `hibernate.order_inserts` tells Hibernate to order the inserts by the primary key (using the entities hash code value actually). If entities use a generated primary key, you need to make sure that fetching generated key values is enabled, but this is enabled in Hibernate by default.

Below is an extract from the configuration file with both these parameters set:

```

<persistence-unit name="batch-services" transaction-type="JTA">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <jta-data-source>java:/MySqlDS</jta-data-source>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="none"/>
    <property name="hibernate.default_catalog" value="EJB3"/>
  </properties>
</persistence-unit>

```



```

        <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
        <property name="hibernate.order_updates" value="true"/>
        <property name="hibernate.order_inserts" value="true"/>
        <property name="hibernate.jdbc.batch_versioned_data"
value="true"/>
        <property name="hibernate.jdbc.fetch_size" value="50000"/>
        <property name="hibernate.jdbc.batch_size" value="50000"/>
        <property name="hibernate.default_batch_fetch_size"
value="50000"/>
        <property name="hibernate.connection.release_mode"
value="auto"/>
    </properties>
</persistence-unit>

```

The following code sample illustrates OLTP. It creates an Order which contains OrderLines (has-a relationship in the object model). The OrderLine entities can be batch inserted, as what is passed into this method is the Order object with its OrderLine collection.

```

public Order createOrder(Customer customer
    , List<OrderLine> orderLines
    , BigDecimal totalOrderAmount) {
    String addressLine2 = customer.getAddressLine2();
    if (addressLine2 == null) {
        addressLine2 = "";
    }
    Address shippingAddress = new Address();
    shippingAddress.setAddressLine1(customer.getAddressLine1());
    shippingAddress.setAddressLine2(addressLine2);
    shippingAddress.setCity(customer.getCity());
    shippingAddress.setState(customer.getState());
    shippingAddress.setZipCode(customer.getZipCode());
    shippingAddress.setZipCodePlusFour(customer.getZipCodePlusFour());
    Order newOrder = new Order();
    newOrder.setCustomerId(customer.getCustomerId());
    newOrder.setDistributionCenterId(customer.getDistributionCenterId());
    newOrder.setShippingAddressLine1(shippingAddress.getAddressLine1());
    newOrder.setShippingAddressLine2(shippingAddress.getAddressLine2());
    newOrder.setShippingCity(shippingAddress.getCity());
    newOrder.setShippingState(shippingAddress.getState());
    newOrder.setShippingZipCode(shippingAddress.getZipCode());

    newOrder.setShippingZipCodePlusFour(shippingAddress.getZipCodePlusFour());
    newOrder.setTotalOrderAmount(totalOrderAmount);
    newOrder.setOrderDate(new Date());
    newOrder.setCustomer(customer);
    entityManager.persist(newOrder);
    String orderId = newOrder.getOrderId();
    for (OrderLine orderLine: orderLines) {
        orderLine.getOrderLinePK().setOrderId(orderId);
    }
    newOrder.setOrderLines(orderLines);
    entityManager.persist(newOrder);
    return newOrder;
}

```

Note there is no call to the entity manager's flush method. Everything is accomplished through the persistence unit's Hibernate configuration, which is as follows:

```
<persistence-unit name="services" transaction-type="JTA">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <jta-data-source>java:/MySqlDS</jta-data-source>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="none"/>
    <property name="hibernate.default_catalog" value="EJB3"/>
    <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
    <property name="hibernate.order_updates" value="true"/>
    <property name="hibernate.order_inserts" value="true"/>
    <property name="hibernate.jdbc.batch_versioned_data"
value="true"/>
    <property name="hibernate.jdbc.fetch_size" value="500"/>
    <property name="hibernate.jdbc.batch_size" value="500"/>
    <property name="hibernate.default_batch_fetch_size" value="16"/>
    <property name="hibernate.connection.release_mode"
value="auto"/>
    <property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory"/>
    <property name="hibernate.cache.region.jbc2.cachefactory"
value="java:CacheManager"/>
    <property name="hibernate.cache.use_second_level_cache"
value="true"/>
    <property name="hibernate.cache.use_query_cache"
value="false"/>
    <property name="hibernate.cache.use_minimal_puts"
value="true"/>
    <property name="hibernate.cache.region.jbc2.cfg.entity"
value="mvcc-entity"/>
    <property name="hibernate.cache.region_prefix"
value="services"/>
  </properties>
</persistence-unit>
```

The same two configuration parameters for Hibernate are used, where the inserts are ordered, and the JDBC batch size is set. In this case, an order that has up to 500 line items could be batched. Because OrderLine entities are persisted and the transaction will commit upon exiting the createOrder method, there is no need to do anything further. Of course, the data model changes to use a generated key strategy that can be known prior to the inserts being executed is important. Finally, there's a database-specific configuration parameter required to make this work just as you want.

With the MySQL JDBC driver a connection property must be set to enable the driver to detect inserts to the same table that can be combined into a single insert statement: `rewriteBatchedStatements`. Since Hibernate will send the inserts it's batching using the JDBC method `executeBatch`, the JDBC driver can rewrite the multiple insert statements into a single insert statement as illustrated at the top of the article. This property must be specified with a value of `true`, and you can do this through setting the connection property in the data source XML file for MySQL as such:

```
<datasources>
  <local-tx-datasource>
    <jndi-name>MySqlDS</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/EJB3</connection-
url>
```

```

        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <user-name>username</user-name>
        <password>password</password>
        <exception-sorter-class-
name>org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter</exceptio
n-sorter-class-name>
        <min-pool-size>400</min-pool-size>
        <max-pool-size>450</max-pool-size>
        <!-- The transaction isolation level must be read committed for
optimistic locking to work properly -->
        <transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-
isolation>
        <connection-property
name="rewriteBatchedStatements">true</connection-property>
        <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml
(optional) -->
        <metadata>
            <type-mapping>mySQL</type-mapping>
        </metadata>
    </local-tx-datasource>
</datasources>

```



### WARNING

MySQL has a default maximum for SQL statement size, and depending on the number of columns, column types, and values, you could exceed that limit. The limit is 1MB, but can be made larger if necessary by setting `max_allowed_packet` variable in `my.cnf`. It takes as its value the number of megabytes such as 2MB, or 16MB, etc. If you are dealing with BLOB, or other large data for the values of the columns, keep this limit in mind.

#### 4.5.3.1. Batch Processing; Tests And Results

Below are the parameters and results of a series of performance test conducted to illustrate the effectiveness of the examples above.

Each set of tests has an in-context result comment and there is a *Conclusions* overview at the end of the section.

- [Example 4.1, “Batch Processing Test”](#)
- [Example 4.2, “OLTP Baseline Testing”](#)
- [Example 4.3, “OLTP Hibernate Batch Processing Test”](#)
- [Example 4.4, “OLTP Hibernate and MySQL Batch Processing Test”](#)
- [Example 4.5, “Higher Line Items Baseline Test”](#)
- [Example 4.6, “Higher Line Items Hibernate Batch Processing Test”](#)

- [Example 4.7, “Higher Line Items Hibernate And MySQL Batch Processing Test”](#)

### Example 4.1. Batch Processing Test

Testing performed with the reference data load function of the Order performance application.

#### Test Parameters

- Environment:
  - Quad-core Intel i7
  - 8 GB of RAM
  - MySQL database
  - JBoss Enterprise Application Platform **production** profile.
- Table Contents:
  - 41,106 rows
  - 250,000 products
  - 6.25 million inventory records
  - 2 thousand suppliers
  - 1.25 million supplier inventory records
  - 2 million rows for customers and pricing methods.
- Batching enabled in Hibernate.
- Batch insert size of 10,000 rows.
- *MySQL Batching* refers to the *rewriteBatchedStatements* parameter (which is set in `mysql-ds.xml`).

**Table 4.1. Batch Processing Test Results**

MySQL Batching	Throughput (rows per second)
Off	6,734.01
On	39,575.75
<b>Result</b>	<b>487.7% increase in throughput.</b>

### Example 4.2. OLTP Baseline Testing

Three tests were run against the OLTP example. This test does not use any batching features and thus sets the baseline for further tests.

### Test Parameters

- Environment:
  - Quad-core Intel i7
  - 8 GB of RAM
  - MySQL database
  - JBoss Enterprise Application Platform **production** profile.
- The composite benchmark is run with randomly created orders with 1 to 9 order line items. The average is around 4.5 to 5 order line items per order. This mimics a real world scenario.
- Batch processing **off**.
- *rewriteBatchedStatements* parameter **unset** in `mysql-ds.xml`.

**Table 4.2. OLTP Baseline Results**

Order Inserts	Throughput (orders per second).
10,000 order inserts with a single order line item	45.80
10,000 order inserts with five order line items	43.96
10,000 order inserts with nine order line items	43.14
<b>Result</b>	<b>Single item orders have the highest throughput, with five and nine following respectively.</b>

### Example 4.3. OLTP Hibernate Batch Processing Test

The second OLTP test was run with Hibernate batching switched on. All other parameters stayed the same as the previous test.

#### Test Parameters

- Environment:
  - Quad-core Intel i7
  - 8 GB of RAM
  - MySQL database
  - JBoss Enterprise Application Platform **production** profile.
- The composite benchmark is run with randomly created orders with 1 to 9 order line items. The average is around 4.5 to 5 order line items per order. This mimics a real world scenario.
- Hibernate batch processing **on**.

- MySQL batching (*rewriteBatchedStatements*) unset in `mysql-ds.xml`.

Table 4.3. OLTP Hibernate Batch Processing Results

Order Inserts	Throughput (orders per second).
10,000 order inserts with a single order line item	45.64
10,000 order inserts with five order line items	43.62
10,000 order inserts with nine order line items	42.78
<b>Result</b>	<b>Ordering the inserts lowered throughput slightly; differences range from .24 to .36 orders per second.</b>

#### Example 4.4. OLTP Hibernate and MySQL Batch Processing Test

The third OLTP test enabled both Hibernate and MySQL batching features.

##### Test Parameters

- Environment:
  - Quad-core Intel i7
  - 8 GB of RAM
  - MySQL database
  - JBoss Enterprise Application Platform **production** profile.
- The composite benchmark is run with randomly created orders with 1 to 9 order line items. The average is around 4.5 to 5 order line items per order. This mimics a real world scenario.
- Hibernate batch processing on.
- MySQL batching (*rewriteBatchedStatements*) on in `mysql-ds.xml`.

Table 4.4. OLTP Hibernate and MySQL Batch Processing Results

Order Inserts	Throughput (orders per second).
10,000 order inserts with a single order line item	46.11
10,000 order inserts with five order line items	44.25
10,000 order inserts with nine order line items	43.77
<b>Result</b>	<b>An increase in throughput ranging from .21 to .63 orders per second, with nine line item orders showing the highest improvement.</b>
<b>Order Inserts</b>	<b>Throughput (orders per second).</b>
	Five line item orders show a .66% improvement over the baseline and nine line item orders show a 1.46% improvement.

#### Example 4.5. Higher Line Items Baseline Test

Some addition testing was performed using higher line item benchmarks. This test does not use any batching features and thus sets the baseline for further tests.

##### Test Parameters

- Environment:
  - Quad-core Intel i7
  - 8 GB of RAM
  - MySQL database
  - JBoss Enterprise Application Platform production profile.
- The composite benchmark is run with randomly created orders with 1 to 9 order line items. The average is around 4.5 to 5 order line items per order. This mimics a real world scenario.
- Hibernate batch processing off.
- MySQL batching (*rewriteBatchedStatements*) unset in `mysql-ds.xml`.

Table 4.5. Higher Line Items Baseline Results

Order Inserts	Throughput (orders per second).
10,000 order inserts with a single order line item	53.34
<b>Result</b>	<b>Because 15 and 20 line items orders take much longer, the server can process even more single line item orders.</b>

Order Inserts	Throughput (orders per second).
10,000 order inserts with 15 order line items	31.25
10,000 order inserts with 20 order line items	30.09
<b>Result</b>	<b>Because 15 and 20 line items orders take much longer, the server can process even more single line item orders.</b>

#### Example 4.6. Higher Line Items Hibernate Batch Processing Test

In this test, Hibernate's batching feature is enabled. All other parameters (including higher line items) remain the same.

##### Test Parameters

- Environment:
  - Quad-core Intel i7
  - 8 GB of RAM
  - MySQL database
  - JBoss Enterprise Application Platform **production** profile.
- The composite benchmark is run with randomly created orders with 1 to 9 order line items. The average is around 4.5 to 5 order line items per order. This mimics a real world scenario.
- Hibernate batch processing on.
- MySQL batching (*rewriteBatchedStatements*) unset in `mysql-ds.xml`.

**Table 4.6. Higher Line Items Hibernate Batch Processing Results**

Order Inserts	Throughput (orders per second).
10,000 order inserts with a single order line item	54.83
10,000 order inserts with fifteen order line items	31.31
10,000 order inserts with twenty order line items	30.06
<b>Result</b>	<p><b>Single line item orders showed a 2.79% increase (an extra 1.49 orders per second).</b></p> <p><b>15 line item orders increased by .06 orders per second and the twenty line item orders decreased by .03 orders per second. These results are within the expected margin of test variations.</b></p>



### Example 4.7. Higher Line Items Hibernate And MySQL Batch Processing Test

In this last test, both the Hibernate and MySQL batching features are tested using higher line items.

#### Test Parameters

- Environment:
  - Quad-core Intel i7
  - 8 GB of RAM
  - MySQL database
  - JBoss Enterprise Application Platform **production** profile.
- The composite benchmark is run with randomly created orders with 1 to 9 order line items. The average is around 4.5 to 5 order line items per order. This mimics a real world scenario.
- Hibernate batch processing on.
- MySQL batching (*rewriteBatchedStatements*) on in `mysql-ds.xml`.

**Table 4.7. Higher Line Items Hibernate And MySQL Batch Processing Results**

Order Inserts	Throughput (orders per second).
10,000 order inserts with a single order line item	60.79
10,000 order inserts with fifteen order line items	37.39
10,000 order inserts with twenty order line items	35.52
<b>Result</b>	<b>Single line orders showed a 13.97% increase, 15 line item orders showed an increase of 19.65% and 20 line item orders increased by 18.05%.</b>

#### Conclusions

For OLTP workloads, if you have situations where you are regularly going to insert tens of rows that could be batch inserted, you should consider batch insertion. Of course, this requires the JDBC driver to have the same capability as the MySQL JDBC driver to convert the multiple inserts into a single insert.

The fact that throughput improvement was found in the smaller test is significant, considering that there were other queries being executed. The customer must be found for each order to be processed, which requires a three-way join from customer to pricing method to distribution center. Each line item looks up its product cost information as well, and then the line items are priced, and a total is calculated for the entire order.

So, the insert does not dominate the time, which is a significant factor in any decision to use these settings for an application. Batch inserts clearly offer an advantage for batch operations but it is not clear if it can also improve OLTP.

It is necessary to understand the typical cases for how many times you have enough rows involved in an insert that can be converted to a batch insert, and how much of the overall processing time the inserts are versus the rest of the processing.

Additionally, batch inserts are much more efficient in database operations, so while you may not get direct benefit in the response time of transactions, that gained efficiency on the database may allow other processing to be better served.

#### 4.5.4. Batching Database Operations

Hibernate has long had the ability to batch database operations, within the context of each transaction. The concept is to reduce the latency between the application platform and the database server. If each and every query is sent to the database individually, latency is increased because of the sheer number of round trips to and from the database. Batching of database operations can be used in any application that uses Hibernate as the persistence mechanism.

There are two Hibernate parameters that control the behavior of batching database operations:

- `hibernate.jdbc.fetch_size`
- `hibernate.jdbc.batch_size`

Both of these options set properties within the JDBC driver. In the first case, `hibernate.jdbc.fetch_size` sets the statement's fetch size within the JDBC driver, that is the number of rows fetched when there is more than a one row result on select statements. In the second case, `hibernate.jdbc.batch_size` determines the number of updates (inserts, updates and deletes) that are sent to the database at one time for execution. This parameter is necessary to do batch inserts, but must be coupled with the `ordered_inserts` parameter and the JDBC driver's capability to rewrite the inserts into a batch insert statement.



#### WARNING

Ordering inserts and updates only helps improve performance if the JDBC driver can rewrite the statements and should not be used unless the JDBC driver has that capability.

Determining the correct values to be assigned to each of these parameters requires analysis of each application's behavior when transacting with its database. The following sections examine differences in performance of the test application with differing values for each parameter.

The first parameter examined is `hibernate.jdbc.fetch_size`. In the example application, there are queries that return a single row, queries that return 3 rows, and one query that can return up to 500 rows. In the last case, the query itself is actually limited through the API to 500 rows, although there can be many more rows than that that could satisfy the query. The other interesting thing to consider is the fact that the query that could return 500 rows is executed in a stateful session bean. When the query is executed, only 20 rows of data, of the total matching rows, are returned at one time to the client. If the client found what they wanted, and does not page further through the result set, only those first 20 rows (or fewer rows, if the query returns less than 20) will ever be used from the result set. Taking that into account, `fetch_size` values of 20 and 500 were tested. In tests, 20 gave the best result, with 500 actually being slightly slower. Twenty was also faster than the default of 1. In the case of 20 versus 500, 20 provided 1.73% more throughput while in the case of 20 vs. 1, 20 provided

2.33% more throughput. Aligning the value with the query that can return upwards of 500 rows, and paginating the results 20 rows at a time was the most effective. Thorough analysis of the queries in the application and the frequency with which those queries are executed is necessary to size this correctly.

The application referred to throughout this book consists of two parts: a batch processing piece that creates reference data and an OLTP piece that processes business transactions using the reference data. In regards to these two parameters, they have very distinct needs so the challenge is how to meet the needs of both. One solution is to have multiple persistence units within a single application, which in turn means the application can have multiple entity managers, each with a different configuration. The norm is to have different entities within the different persistence units but while this is the norm, it is not prohibited to have the same entities in different persistence units. However you cannot use the same entities in multiple persistence units and their respective entities' managers at the same time. Because the batch processing of the application has to be run first to create the reference data, and the OLTP part run second (and independently of the batch processing) two persistence units could be defined with their respective entities managers and the same entities in both. That was highly convenient for the example application but in the real world that may not be possible. Instead, you may have to divide up batch processing and OLTP processing into different application deployments if they have the same entities. To illustrate this, here is the application's persistence.xml:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="services" transaction-type="JTA">
<provider>org.hibernate.ejb.HibernatePersistence</provider>
  <jta-data-source>java:/MySqlDS</jta-data-source>
  <properties>
    <property name="hibernate.hbm2ddl.auto"
value="none"/>
    <property name="hibernate.default_catalog"
value="EJB3"/>
    <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
    <property name="hibernate.order_inserts"
value="true"/>
    <property
name="hibernate.default_batch_fetch_size" value="20"/>
    <property name="hibernate.jdbc.fetch_size"
value="20"/>
    <property name="hibernate.jdbc.batch_size"
value="20"/>
    <property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory"/>
    <property
name="hibernate.cache.region.jbc2.cachefactory"
value="java:CacheManager"/>
    <property
name="hibernate.cache.use_second_level_cache" value="true"/>
    <property name="hibernate.cache.use_query_cache"
value="false"/>
    <property name="hibernate.cache.use_minimal_puts"
value="true"/>
    <property
```

```

name="hibernate.cache.region.jdbc2.cfg.entity" value="mvcc-entity"/>
    <property name="hibernate.cache.region_prefix"
value="services"/>
    </properties>
</persistence-unit>
<persistence-unit name="batch-services" transaction-
type="JTA">

<provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/MySqlDS</jta-data-source>
    <properties>
        <property name="hibernate.hbm2ddl.auto"
value="none"/>
        <property name="hibernate.default_catalog"
value="EJB3"/>
        <property name="hibernate.order_inserts"
value="true"/>
        <property name="hibernate.jdbc.fetch_size"
value="50000"/>
        <property name="hibernate.jdbc.batch_size"
value="50000"/>
        <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
        <property
name="hibernate.cache.use_second_level_cache" value="false"/>
    </properties>
</persistence-unit>
</persistence>

```

In this example, two persistence units are created: `services` and `batch-services`. The OLTP processing, "services", is configured with relatively small fetch and batch sizes. The batch processing, "batch-services", is configured with large fetch and batch sizes. There is no entity caching in the batch processing because its unnecessary, while there is an entity cache defined for the OLTP side of things. This is a very convenient and powerful feature of JPA.

For the second parameter, `hibernate.jdbc.batch_size`, the Hibernate documentation recommends a value of between 5 and 30 but this value depends upon the application's needs. The Hibernate documentation's recommendation is suitable for most OLTP-like applications. If you examine insert and update queries within the application, you can make a determination of how many rows are inserted and updated across the set. If inserts or updates occur no more than one row at a time, then leaving it unspecified is suitable. If inserts and/or updates affects many rows at a time, in a single transaction, then setting this is very important. If batch inserts are not required, and the aim is to send all the updates in as few trips to the database as possible, you should size it according to what is really happening in the application. If the largest transaction you have is inserting and updating 20 rows, then set it 20, or perhaps a little larger to accommodate any future changes that might take place.

## 4.6. MESSAGE DRIVEN BEANS

Message driven beans (MDB) are an EJB 3 bean type that is invoked from a message, specifically from a JMS message. There are three areas of concern when tuning for message driven beans. The first is the number of sessions to be run in parallel. The second is the pool size for the beans themselves. This pool is similar to the pool for stateless session beans, but is related to the session parameter. The third is thread pooling, discussion of which is deferred to the JCA chapter.

The number of sessions is the number of actual message driven beans which will be waiting on the

same queue and/or topic. Assume a queue is defined, and ten messages are to be processed from that queue at the same time, requiring ten sessions. In the EJB 3 world, this can be achieved through an annotation on the bean implementation itself. In fact, there are two parameters within the annotation: `maxSessions` and `minSessions`. The annotation itself is the standard EJB 3 MDB of: `@ActivationConfigProperty(propertyName, propertyValue)`

So, within the MDB, the minimum and maximum number of sessions can be specified, giving the opportunity to control how many instances will process messages from a given queue at one time. Refer to the following example:

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName="destinationType"
        , propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="destination"
        , propertyValue="queue/replenish")
    @ActivationConfigProperty(propertyName="minSessions"
        , propertyValue="25")
    @ActivationConfigProperty(propertyName="maxSessions"
        , propertyValue="50")
})
```

In this example two activation configuration properties are defined, specifying the minimum and maximum number of sessions. The minimum defines that there should always be that number of instances of the MDB in the pool, in the method ready state (ready to receive messages). The maximum specifies there can be up to that many processing messages, so if there are messages waiting and all of the instances are busy, the pool can grow from the minimum to the maximum. In this example there can be up to fifty instances of the MDB in the pool, processing messages. The “pool” parameter mentioned here is highly interrelated to the minimum and maximum number of sessions.

Just as there are pools for stateless sessions beans, there is also an instance pool for message driven beans. In this case though, there is really only one pool type that can be used: `StrictMaxPool`. The reason why `StrictMaxPool` is the only option for message driven beans is simply the nature of the bean type. Having an unbounded number of message driven beans processing messages could present problems. Many systems that use messaging process millions upon millions of messages, and it's not scalable to keep increasing the number of instances to process everything in parallel. Also it's really not possible because the default for `maxSessions` parameter is 15.

There's a relationship between the pool and the sessions parameters: `maxSessions` and `minSessions`. Assume that the number of `maxSessions` is set to a number larger than the default of 15, perhaps 50, but the default pool size to hold the instances of the MDB is only 15. Only 15 instances will process messages at once, not the 50 as might be expected. The reason for only getting 15 is that the default pool size (as defined in the configuration file) is 15, like the default `maxSessions`. So changing the `maxSessions` does not actually help unless the pool's configuration is changed to match. The pool's configuration is specified in the same file as for stateless sessions beans: `ejb3-interceptors-aop.xml`, in the directory `JBOSS_EAP_DIST/jboss-as/server/PROFILE/deploy`. Note that the `minimal` configuration does not contain the EJB 3 container.

Below is an example of this file:

```
<domain name="Message Driven Bean" extends="Intercepted Bean"
inheritBindings="true">
    <bind pointcut="execution(public * *->*(..))">
        <interceptor-ref
name="org.jboss.ejb3.security.AuthenticationInterceptorFactory"/>
        <interceptor-ref
name="org.jboss.ejb3.security.RunAsSecurityInterceptorFactory"/>
```

```

        </bind>
        <bind pointcut="execution(public * *->*(..))">
            <interceptor-ref
name="org.jboss.ejb3.tx.CMTTxInterceptorFactory"/>
            <interceptor-ref
name="org.jboss.ejb3.stateless.StatelessInstanceInterceptor"/>
            <interceptor-ref
name="org.jboss.ejb3.tx.BMTTxInterceptorFactory"/>
            <interceptor-ref
name="org.jboss.ejb3.AllowedOperationsInterceptor"/>
            <interceptor-ref
name="org.jboss.ejb3.entity.TransactionScopedEntityManagerInterceptor"/>
            <!-- interceptor-ref
name="org.jboss.ejb3.interceptor.EJB3InterceptorsFactory"/ -->
            <stack-ref name="EJBInterceptors"/>
        </bind>
        <annotation expr="class(*) AND
!class(@org.jboss.ejb3.annotation.Pool)">
            @org.jboss.ejb3.annotation.Pool (value="StrictMaxPool",
maxSize=15, timeout=10000)
        </annotation>
    </domain>
    <domain name="Consumer Bean" extends="Intercepted Bean"
inheritBindings="true">
        <bind pointcut="execution(public * *->*(..))">
            <interceptor-ref
name="org.jboss.ejb3.security.RunAsSecurityInterceptorFactory"/>
        </bind>
        <bind pointcut="execution(public * *->*(..))">
            <interceptor-ref
name="org.jboss.ejb3.tx.CMTTxInterceptorFactory"/>
            <interceptor-ref
name="org.jboss.ejb3.stateless.StatelessInstanceInterceptor"/>
            <interceptor-ref
name="org.jboss.ejb3.tx.BMTTxInterceptorFactory"/>
            <interceptor-ref
name="org.jboss.ejb3.AllowedOperationsInterceptor"/>
            <interceptor-ref
name="org.jboss.ejb3.entity.TransactionScopedEntityManagerInterceptor"/>
        </bind>
        <bind pointcut="execution(public * *->*(..)) AND (has(* *-
>@org.jboss.ejb3.annotation.CurrentMessage(..)) OR hasfield(* *-
>@org.jboss.ejb3.annotation.CurrentMessage))">
            <interceptor-ref
name="org.jboss.ejb3.mdb.CurrentMessageInjectorInterceptorFactory"/>
        </bind>
        <bind pointcut="execution(public * *->*(..))">
            <!-- interceptor-ref
name="org.jboss.ejb3.interceptor.EJB3InterceptorsFactory"/ -->
            <stack-ref name="EJBInterceptors"/>
        </bind>
        <annotation expr="class(*) AND
!class(@org.jboss.ejb3.annotation.Pool)">
            @org.jboss.ejb3.annotation.Pool (value="StrictMaxPool",

```

```
maxSize=15, timeout=10000)  
    </annotation>  
</domain>
```

In this example are four important settings. First is the definition of the message driven bean. Second is the `StrictMaxPool` configuration, with the default value of 15, the same default value for the `maxSessions` parameter. The next two are the same, except for something called a consumer bean. Consumer beans are a JBoss-specific implementation, not Java EE compliant, that gives all the functionality of the message driven bean, but does so without having to implement the `MessageListener` interface, and is a pure POJO model like stateless and stateful session beans. It brings the EJB 3 POJO model to asynchronous processing, without the old EJB 2.x model that was carried over for message driven beans. Of course, if this option is used, the application will not be portable, so use at your own risk.

The `maxSize` parameter on the `StrictMaxPool` needs to be the same as the `maxSessions` set on the bean itself. If they are *not* set to the same value, strange behavior occurs. When the queue has more messages than either the `maxSessions` or `maxSize` value, messages up to the `maxSessions`' value will be processed, then the number of messages next processed is the difference between `maxSessions` and `maxSize`, with this cycle repeating. To illustrate this, `maxSessions` is set to 25, the pool size left at the default of 15. When messages are being processed the number of instances will alternate between 15 and 10, which is the difference between the `maxSessions` and the `maxSize` parameters. So 15 messages will be processed, then 10, then 15, then 10, repeating until the number of messages drops to 15 or below. Keeping these two parameters in sync is important to get the desired behavior.

Determining the "right" value for number of sessions is dependent on the rate of incoming messages, processing time it takes to process each message, and the requirement for how long it should take to process messages. It may be best for some applications to throttle the message processing rate and allow the queue to grow over time, or at least during peak times, allowing the processing to catch up, and eventually drain the queue when the incoming rate drops. For other applications it may be best to match the processing rate to the incoming rate. In either case calculating the right value is based on the number of incoming messages over a given period of time and how long does it take to process each message. Unfortunately there's no strict mathematical formula as there will not be perfectly linear scalability when processing in parallel, and scalability will vary by hardware, OS, persistence mechanism (in the case of persistent messages), JVM version and even the version of EAP, and which messaging provider you would use. As mentioned earlier, EAP 5.1.x has two messaging providers which, although they provide the same JMS API work with the same MDB implementation, are very different under the covers. One uses a database for persistent messages and the other a file journal, which is by far the biggest difference. That difference is huge in regards to scalability, and can have a profound effect on the number of MDB instances required. There is no substitute for load testing to get this parameter right for your environment.

## CHAPTER 5. JAVA CONNECTOR ARCHITECTURE

Within the platform, Java Connector Architecture (JCA) is used in two main areas: integration with data sources, such as relational databases, and integration with JMS providers. In EAP 5.1.2 and above, two JMS providers are available: JBoss Messaging technology, and the newer HornetQ technology. External JMS providers can be integrated through their provided JCA resource adapters. This chapter will cover data sources and their respective parameters, also JMS integration and its relevant parameters.

### 5.1. DATA SOURCES

Data sources are the means of defining the properties of a relational database. This is done with a `*-ds.xml` file in the `deploy` directory of a specific configuration. The platform ships with examples for quite a few relational databases, and there is a complete list of certified relational databases on the JBoss Enterprise Application Platform product pages of the Red Hat website. A data source definition has many parameters but the focus here is on two specific parameters: `min-pool-size` and `max-pool-size`.

### 5.2. DATA SOURCE CONNECTION POOLING

When connecting to a data source, JBoss Enterprise Application Platform must allocate resources and de-allocate resources for every connection. This is quite expensive in terms of time and system resources. Connection pooling reduces the cost of data source connections by creating a number ("pool") of data source connections available to be shared by applications. Pooling data source connections is much more efficient than allocating and de-allocating resources on demand. There are third-party alternatives for data source pooling but the native method is already included and avoids an extra layer of complexity.

### 5.3. MINIMUM POOL SIZE

The `min-pool-size` data source parameter defines the minimum size of the connection pool. The default minimum is zero connections, so if a minimum pool size is not specified, no connections will be created in the connection pool when the platform starts. As data source transactions occur, connections will be requested but because the pool defaults to zero on start up, none will be available. The connection pool examines the minimum and maximum parameters, creates connections and places them in the pool. Users of any affected application will experience a delay while this occurs. During periods of inactivity the connection pool will shrink, possibly to the minimum value, and later when transactions later occur application performance will again suffer.



The screenshot shows the JBoss EAP Admin Console interface. On the left is a navigation tree with categories like JBossAS Servers, Applications, and Resources. The main content area is titled 'MySQLDS' and has tabs for Summary, Configuration, Metrics, Control, and Content. The 'Metrics' tab is active, showing a 'Status: Available' indicator. Below the status, there are sections for 'Traits' (Run State: RUNNING, Local Transaction: true) and 'Numeric Metrics'. A table lists several performance metrics with their values and descriptions. A 'Refresh' button is located at the bottom of the metrics section.

Name	Value	Description
<b>Category: performance</b>		
Available Connection Count	500	the maximum number of connections that are available
Connection Count	1	the number of connections that are currently in the pool
Connection Created Count	1	the number of connections that have been created since the datasource was last started
Connection Destroyed Count	0	the number of connections that have been destroyed since the datasource was last started
In Use Connection Count	0	the number of connections that are currently in use
Max Connections In Use Count	1	the most connections that have been simultaneously in use since this datasource was started
Max Size	500	Max Size
Min Size	0	Min Size

Figure 5.1. Data Source Metrics

## 5.4. MAXIMUM POOL SIZE

The `max-pool-size` parameter data source parameter defines the maximum size of the connection pool. It's more important that the `min-pool-size` parameter because it limits the number of active connections to the data source and so the concurrent activity on the data source. If this value is set too low it's likely that the platform's hardware resources will be underutilized.



### NOTE

Setting the `max-pool-size` too low is a common configuration error.

## 5.5. BALANCING POOL SIZE VALUES

The aim of adjusting the `min-pool-size` and `max-pool-size` pool values is to allow all possible concurrent database activity to execute, and so maximize throughput. To do this requires an understanding of how much concurrent activity in the server is occurring. As a guide of concurrent activity, look at the platform's thread pools to see how many threads are active. This is available through the administration console, as in the following screenshot:

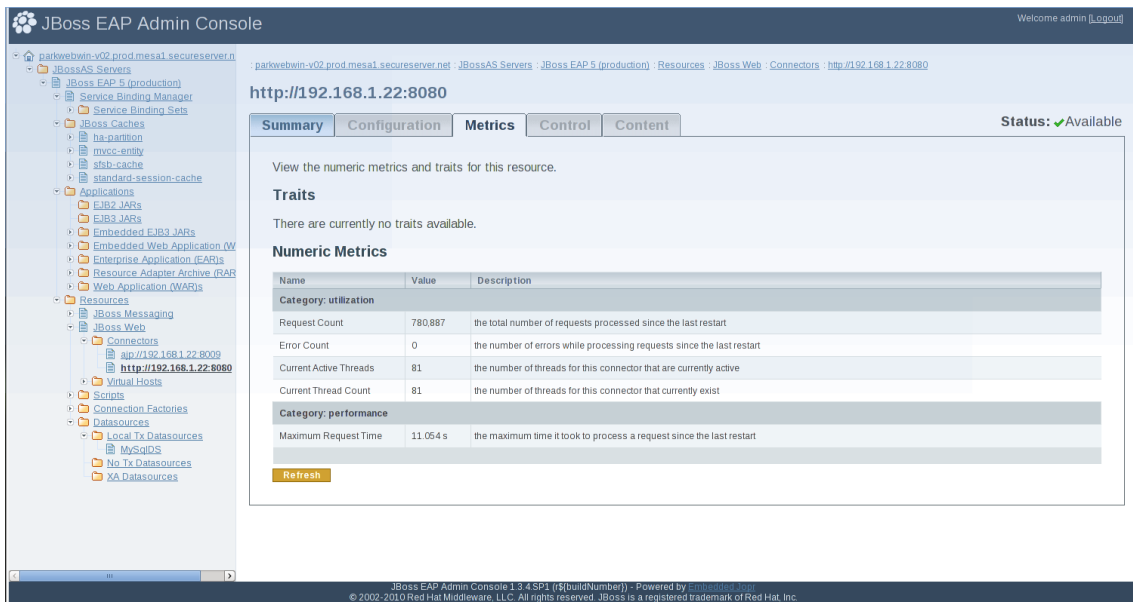


Figure 5.2. JMX Console Datasource Metrics

In this example there were 81 active threads in the JBoss Web HTTP connector, which is the primary thread pool in the test used for this illustration. That number is to be expected because there were 80 users, with no think times between requests, and response times were between 18 and 20 ms on average. Having 81 threads active accounts for the continued use of the 80 users, plus the one user for the administration console itself. Most of the time the actual number of active threads was actually only 80, but when the refresh button is clicked, the administration console consumed a thread during its collection of the data. In this example, you may presume that the maximum size of the connection pool should at least be 80 but it may not be as simple as it appears.

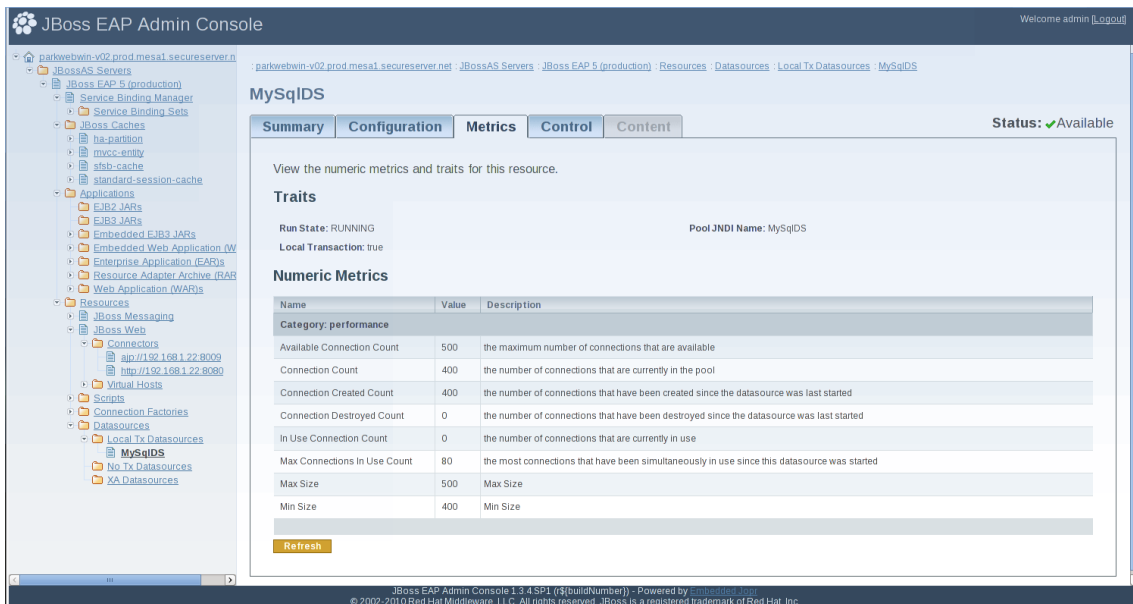


Figure 5.3. Admin Console Datasource Metrics

In looking at the data source metrics in the administration console, the “Max Connections In Use Count” is reported as 80. With 80 concurrent users and each transaction performing database transactions, a value of 80 seems obvious. In various situations, though, this figure may be misleading. If “Max Connections in Use Count” equals the max-pool-size parameter then this value is meaningless. It's also possible that the thread pools have many more active threads than threads that will eventually hit the database. Also there may be multiple thread pools in use within the platform, each with their own data access characteristics. Finally, it's possible for an EJB 3 application to skew the number of connections needed.

It's a requirement of the EJB 3 specifications that each bean that has an entity manager injected into it also have a unique instance of the entity manager. This has an indirect influence on the database connection pool sizing. Since Hibernate is the JPA provider, a unique instance of an entity manager and a Hibernate session, this requires a database connection for the queries within that entity manager and Hibernate session. In the case of a simple application, where stateless session beans execute queries and never call other stateless session beans, a single database connection is consumed for that processing. In a more complex application, assume there's a stateless session bean that executes queries but also calls on other stateless sessions beans. Because that also execute queries, there are multiple entity managers, potentially consuming multiple database connections and all participating in the same transaction. In a more complex example again, assume there's a stateless session bean that puts a message on a JMS queue, then does some further processing that executes queries and asynchronously, a message driven bean de-queues that message and does some processing that executes queries. From these examples it's evident that the more complex the application structure, the harder it will be to determine just how many connections may be used. This is even further complicated by the speed at which connections can be returned to the pool for reuse, and how fast they can be enlisted in a transaction when reused from the pool. One technique you can use is to continue increasing the maximum pool size until the "Max Connections In Use Count" stays below the maximum size parameter of the pool. This may not be possible in a production environment so a test environment is recommended.

There's one more check that can be made to determine if the connection pool is too small and so might be throttling throughput. During peak loads you can use the JMX console to take a thread dump of the running platform. If in the output you see the following the connection pool is too small:

```
Thread: http-192.168.1.22-8080-2 : priority:5, demon:true, threadId:145,
threadState:TIMED_WAITING
- waiting on <0x2222b715> (a java.util.concurrent.Semaphore$FairSync)
sun.misc.Unsafe.park(Native Method)
java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:226)
java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireSharedNanos
(AbstractQueuedSynchronizer.java:1037)
java.util.concurrent.locks.AbstractQueuedSynchronizer.tryAcquireSharedNano
s(AbstractQueuedSynchronizer.java:1326)
java.util.concurrent.Semaphore.tryAcquire(Semaphore.java:410)
org.jboss.resource.connectionmanager.InternalManagedConnectionPool.getConn
ection(InternalManagedConnectionPool.java:193)
org.jboss.resource.connectionmanager.JBossManagedConnectionPool$BasePool.g
etConnection(JBossManagedConnectionPool.java:747)
org.jboss.resource.connectionmanager.BaseConnectionManager2.getManagedConn
ection(BaseConnectionManager2.java:404)
org.jboss.resource.connectionmanager.TxConnectionManager.getManagedConnect
ion(TxConnectionManager.java:424)
org.jboss.resource.connectionmanager.BaseConnectionManager2.allocateConnec
tion(BaseConnectionManager2.java:496)
org.jboss.resource.connectionmanager.BaseConnectionManager2$ConnectionMana
gerProxy.allocateConnection(BaseConnectionManager2.java:941)
org.jboss.resource.adapter.jdbc WrapperDataSource.getConnection(WrapperDat
aSource.java:89)
org.hibernate.ejb.connection.InjectedDataSourceConnectionProvider.getConne
ction(InjectedDataSourceConnectionProvider.java:47)
org.hibernate.jdbc.ConnectionManager.openConnection(ConnectionManager.java
:446)
org.hibernate.jdbc.ConnectionManager.getConnection(ConnectionManager.java:
167)
```

In this example thread dump from the ServerInfo section of the JMX Console's jboss.system category

(where you can invoke a `listThreadDump` operation) you can see that the thread is in a `TIMED_WAITING` state and waiting on a semaphore. If you look closely at the stack trace you also see that the `InternalManagedConnectionPool` is executing its `getConnection()` method. In the connection pool, if no connection is available for use it waits on a semaphore, which has the same number of tickets available as the `max-pool-size` parameter. If a transaction is waiting for a connection to become available in the pool, it waits until one is returned to the pool. This has an advantage over continuing to increase the pool size because it's possible to see just how many threads are waiting on a connection from the pool. More accurate information means it takes fewer iterations to get the pool's size right.

In closing discussion on the pool size, a rule of thumb is to set the minimum to the size which allows for maximum throughput and the maximum to 10% to 20% higher than the minimum. Setting the maximum higher than the minimum allows a buffer for situations in which load unexpectedly increases, a situation almost guaranteed to occur in real life.

## 5.6. JMS INTEGRATION OR PROVIDER

JMS integration in all Java EE servers is provided through JCA since it's a requirement of the specifications. As mentioned in Message Driven Beans section, there are two JMS providers: the original JBoss Messaging and the newer HornetQ technology. In either case, the focus in tuning each is the JCA thread pool.

The JCA container has its own thread pool, and this pool needs to be sized appropriately to handle the messaging load. It's highly related to the `maxSession` parameter that was discussed with message driven beans. Besides message driven beans, you can use the JMS API directly within the container, and not be tied to message driven beans at all. The JCA thread pool's configuration is in the file `jca-jboss-beans.xml` in the directory `JBOSS_EAP_DIST/jboss-as/server/PROFILE/deploy`. Note that the `minimal` configuration does *not* contain the JCA container.



### NOTE

Below is an example of this configuration file.

```
<!-- THREAD POOL -->
  <bean name="WorkManagerThreadPool"
class="org.jboss.util.threadpool.BasicThreadPool">
<!-- Expose via JMX -->
<annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(name="jboss.jca:
service=WorkManagerThreadPool",
exposedInterface=org.jboss.util.threadpool.BasicThreadPoolMBean.class)</an
notation>
<!-- The name that appears in thread names -->
<property name="name">WorkManager</property>
<!-- The maximum amount of work in the queue -->
<property name="maximumQueueSize">1024</property>
<!-- The maximum number of active threads -->
<property name="maximumPoolSize">100</property>
<!-- How long to keep threads alive after their last work (default one
minute) -->
<property name="keepAliveTime">60000</property>
</bean>
```

In tuning the thread pool the focus is on just two parameters: `maximumPoolSize` and `maximumQueueSize`.

The `maximumPoolSize` parameter specifies the number of threads in the thread pool for use by the JMS provider. The default size is 100, which may or may not be appropriate. If message driven beans are used in the application, make sure that there are enough threads in the thread pool to handle the maximum number of sessions. If only one message driven bean is used, ensure that the thread pool is at least the size of the number of sessions for that one bean. If there are many different message driven beans, the calculations are more complex.

When you have many message-driven beans, each with their own `maxSessions`, start by adding all of the `maxSession` values together. This assumes that they all will hit those maximums at the same time, and different components are likely to have different usage patterns. If you understand the usage pattern, you may be able to reduce the thread pool accordingly, to just take into account the maximums that overlap each other. In fact, reducing it if you do not need 100 is a good step because it will save some memory.

If you are not sure of the application's behavior, you can monitor the usage of the thread pool through the JMX console. Below is a screen shot from the JMX console that shows the attributes of the thread pool:

Attribute Name	Access	Type	Description	Attribute Value
ThreadGroupName	RW	java.lang.String	Attribute exposed for management	JBoss Pooled Threads
QueueSize	R	int	Attribute exposed for management	0
MaximumQueueSize	RW	int	Attribute exposed for management	1024
KeepAliveTime	RW	long	Attribute exposed for management	60000
BlockingMode	RW	org.jboss.util.threadpool.BlockingMode	Attribute exposed for management	abort
ClassLoaderSource	RW	org.jboss.util.loading.ClassLoaderSource	Attribute exposed for management	null
Name	RW	java.lang.String	Attribute exposed for management	WorkManager
Instance	R	org.jboss.util.threadpool.ThreadPool	Attribute exposed for management	WorkManager (2)
PoolNumber	R	int	Attribute exposed for management	2
MinimumPoolSize	RW	int	Attribute exposed for management	100

Figure 5.4. JMX Console Thread Pool Statistics

On the second line of the table in the screenshot you see the attribute name called `QueueSize`, a read-only attribute which records any requests for threads from the thread pool that could not be satisfied because all the threads were currently in use. If this value is above zero, then the thread pool is too small to service all incoming requests. This value will not update automatically so it's necessary to refresh the page manually to see the value updated while the system is running.

The `maximumQueueSize` parameter specifies how many requests will wait for a thread to become available before an exception is thrown and processing is aborted. The default value is 1024, which is a safe value. You may want to fail fast, versus wait for an extended period of time, or have more than 1024 requests backed up before failing. Setting this parameter to a lower value means you'll know earlier of exceptions but at the expense of application uptime. It's recommended to leave it at the default value as it provides greater reliability. If a request is waiting in the queue it's because there are none available in the pool. While sizing the pool, monitor the number of requests waiting in the queue to confirm whether or not the pool is too small.

## CHAPTER 6. JAVA MESSAGE SERVICE (JMS) PROVIDER

As discussed in the JCA chapter, JBoss Enterprise Application Platform offers two JMS providers: the default JBoss Messaging and the new HornetQ. From a performance perspective, HornetQ is far superior to JBoss Messaging.

### 6.1. SWITCHING THE DEFAULT MESSAGING PROVIDER TO HORNETQ

To change the default JMS provider from JBoss Messaging to HornetQ, a script is provided in the EAP distribution which moves JBoss Messaging out of the deployment folders and deploys HornetQ. The script is called `switch.sh` (`switch.bat` on Microsoft Windows Server) and can be found in the directory `JBOSS_EAP_DIST/jboss-as/extras/hornetq`. This script requires the open source build tool Ant be in the path in order to work.



#### WARNING

There is no "undo" option except to reinstall the binaries, so be sure you want to do this before proceeding.

### 6.2. EXPLOITING HORNETQ'S SUPPORT OF LINUX'S NATIVE ASYNCHRONOUS I/O

If JBoss Enterprise Application Platform is running on Red Hat Enterprise Linux (RHEL), you can take advantage of HornetQ's ability to leverage Linux's native asynchronous I/O for its journal. HornetQ has moved from the database approach of JBoss Messaging for persistent messages to a new high performance journal that exists on the file system. Since this journal lives on the file system, HornetQ uses Java's New Input/Output (NIO) routines as the mechanism for reading and writing the journal. There is also a native library, that if detected, will enable HornetQ to leverage Linux's native asynchronous I/O capabilities. This has to be done in "native" code, as Java does not directly support OS asynchronous I/O. Using the operating system's own asynchronous I/O provides the best possible performance and scalability. To enable this option two steps are required.

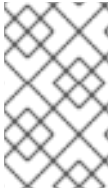
The first step is to ensure that the `libaio` package is installed on your instance of RHEL. This package provides the API for HornetQ's native code that allows it to use asynchronous I/O. With the `libaio` package installed, the next step is to ensure the native code for HornetQ is available. That has to be placed at the same directory level as your EAP instance. For example, if you have EAP installed in `/opt/redhat/jboss-eap-5.1`, then the native code must be put in the same directory. This is because, being native code, it must be the right version for your RHEL version, and the right libraries for the JVM. Both the 32-bit and 64-bit JVMs are supported so there are 32-bit and 64-bit versions of the libraries. The `run.sh` script will automatically detect the directory relative to the EAP installation directory and resulting `JBOSS_HOME` that is set within that script (`run.bat` for Windows won't have this, as this is a Linux only feature). If you have everything installed correctly, you should see this message in the log after start up.

```
10:50:05,113 INFO [JournalStorageManager] Using AIO Journal
```

If you do not have everything installed correctly, you will see a message like the following:

```
2011-02-18 10:05:11,485 WARNING
```

```
[org.hornetq.core.deployers.impl.FileConfigurationParser] (main) AIO
wasn't located on this platform, it will fall back to using pure Java NIO.
If your platform is Linux, install LibAIO to enable the AIO journal
```



## NOTE

These installation instructions are only for zip based installations. If you are using the EAP RPM installation method then everything should be placed in the correct locations by the RPM.

## 6.3. HORNETQ'S JOURNAL

Aside from enabling native asynchronous I/O (if installed on RHEL), the next configuration parameter to optimize is the journal's location, relative to the file system configuration. The default configuration places the journal in `${jboss.server.data.dir}/${hornetq.data.dir}:hornetq/journal`, which expands to: `JBOSS_EAP_DIST/jboss-as/server/PROFILE/data/hornetq/journal`. Note that the `minimal` configuration does *not* contain the JMS provider.

Depending on the underlying storage configuration, this may not be the optimal location. A common configuration is to locate rarely-accessed file (e.g. executables/binaries) on relatively slow, cheaper storage and often-accessed files (e.g. log files, journaling files) on high-performance, more expensive storage. If you have very high messaging loads, the journal should be placed on a high-performance file system, with a high-performance disk configuration. In general, HornetQ's performance is bound by the speed of the file system and network so the configuration of these are most likely be governing factors of performance. To change the location of the journal, the HornetQ configuration file needs to be modified. The configuration file is called `hornetq-configuration.xml`, in the directory `JBOSS_EAP_DIST/jboss-as/server/PROFILE/deploy/hornetq/`.

Below is an extract from the configuration file:

```
<configuration xmlns="urn:hornetq"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:hornetq /schema/hornetq-configuration.xsd">
<!-- Do not change this name.
  This is used by the dependency framework on the deployers,
  to make sure this deployment is done before any other deployment -->
<name>HornetQ.main.config</name>
<clustered>true</clustered>
<log-delegate-factory-class-
name>org.hornetq.integration.logging.Log4jLogDelegateFactory</log-
delegate-factory-class-name>
<bindings-
directory>${jboss.server.data.dir}/${hornetq.data.dir:hornetq}/bindings</b
indings-directory>
<journal-
directory>${jboss.server.data.dir}/${hornetq.data.dir:hornetq}/journal</jo
urnal-directory>
<journal-min-files>10</journal-min-files>
<large-messages-
directory>${jboss.server.data.dir}/${hornetq.data.dir:hornetq}/largemessag
es</large-messages-directory>
<paging-
```



```
directory>${jboss.server.data.dir}/${hornetq.data.dir:hornetq}/paging</paging-directory>
</configuration>
```

From the above extract, the following directories are of interest:

- journal-directory
- large-messages-directory
- paging-directory

The location specified for **journal-directory** is the most critical with respect to performance but the others can also have a significant impact.

The **journal-directory** is where persistent messages are stored.

The **large-message** directory is used to store messages that are larger than can fit into memory. If the application processes large messages, this is also a performance critical directory.

The **paging-directory** is similar to the large message directory, except that is an accumulation of many messages, that no longer fit into memory or are swapped/paged to disk, which is analogous to operating system paging to swap. Just like the journal and large message directory, this is also a performance critical directory, if swapping/paging occurs.

There are two ways to change the locations to be used by these directories. First, you can change the `${jboss.server.data.dir}` variable, by passing `-Djboss.server.data.dir=<path>` where `path` is the new directory location (higher performance file system and disk configuration). This will have the effect of moving everything that uses that directory to the new location, not just the HornetQ directories. This approach has an added bonus because there is another performance critical directory under that path, which contains the transaction journal. Transactions are logged for crash recovery purposes within this same path and this can also benefit from being located on higher-performing storage. The other option is to change the configuration directly, remove the `${jboss.server.data.dir}` and replace it with the new location. In either case, moving this to faster storage will improve performance. Using the command-line option method is easiest because it uses the variable, so there's no need to change the configuration file.

## 6.4. HORNETQ AND TRANSACTIONS

With the older JBoss Messaging technology, persistent messages were stored in a database, with several choices. The JBoss Messaging tables could be combined with all other tables into one unified data source, or separated into its own schema, catalog or database instance (whichever terminology is relevant to your particular database). If messaging schema is combined with other tables, all transactions that span both the messaging system and the application's tables would participate in the same transaction, and commit and rollback as a group. This would be true, even if you defined the data source to be an XA data source, because the XA protocol specification allows a single phase commit optimization for just this circumstance. If you kept them separate, then you would have to define your application data source and the messaging data source to both be XA data sources, so you would get the proper transaction semantics between your messages and your application. Now that HornetQ uses its own file system journal there is no way to unify the other persistent parts of the application with the journal from a transaction perspective, as was possible with JBoss Messaging. If there's a need for messaging and application persistence to participate in the same transaction, the data source must be setup as an XA data source.



## CHAPTER 7. LOGGING PROVIDER

In discussing logging and its impact on performance, there are four topics to be covered:

- console logging
- type of appender to use
- location of log files
- wrapping of log statements

### 7.1. CONSOLE LOGGING

Console logging is by far the most expensive form of logging in terms of time because console output is not buffered. All configurations, except for the production configuration, have console logging enabled because it's very convenient during development. It's especially convenient when using an IDE, such as JBoss Developer Studio, and any other IDE for that matter. With console logging on, all messages that go to the log show up in the IDE, which is an advantage when doing development work. However it's a distinct disadvantage in a production environment because it's slow and expensive.

The simplest method to avoid console logging is to use the production configuration. If a new, custom configuration is to be created, base that configuration on the production configuration. To disable console logging for an existing configuration, it's necessary to edit the logging configuration file `jboss-log4j.xml` in the directory `JBOSS_EAP_DIST/jboss-as/server/PROFILE/conf`. Note that the `production` configuration is preconfigured with console logging disabled.

The following is an extract of the relevant sections from the production configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<!-- =====
-->
<!--
-->
<!-- Log4j Configuration
-->
<!--
-->
<!-- =====
-->
<!-- ===== -->
<!-- More Appender examples -->
<!-- ===== -->
<!-- Buffer events and log them asynchronously -->
<appender name="ASYNC" class="org.apache.log4j.AsyncAppender">
  <errorHandler class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
  <appender-ref ref="FILE"/>
  <appender-ref ref="CONSOLE"/>
  <appender-ref ref="SMTP"/>
</appender>
<!-- ===== -->
<!-- Setup the Root category -->
<!-- ===== --><root>
  <!--
```

```

    Set the root logger priority via a system property. Note this is
    parsed by log4j,
    so the full JBoss system property format is not supported; e.g.
    setting a default via ${jboss.server.log.threshold:WARN} will not
    work.
    -->
    <priority value="${jboss.server.log.threshold}"/>
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE"/>
</root>
</log4j:configuration>

```

Console logging can be disabled either by deleting the entire Appender section from the configuration file or deleting the line containing `<appender-ref ref="CONSOLE"/>`

## 7.2. APPENDERS

Console logging is a form of appender and there are two other main types of file based appenders, one that writes synchronously and the other asynchronously. The amount of data logged by an application can vary greatly according to what the application does and how it behaves. If a problem occurs in the background, for instance, there may be a sudden increase in logging occurrences, resulting in performance degradation of the application. The asynchronous appender is recommended over the synchronous appender because it will generally perform better. The difference in performance varies on the amount of logging that is typical for the application.

The easiest method of using the asynchronous appender is to use the production configuration or base a new configuration on it because it (as noted earlier) has console logging disabled but the asynchronous appender enabled. The configuration file to be modified is `jboss-log4.xml` in the directory `JBOSS_EAP_DIST/jboss-as/server/<PROFILE>/conf`.

The following is an extract of the two relevant sections from the production configuration file:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<!-- =====>
-->
<!--
-->
<!-- Log4j Configuration
-->
<!--
-->
<!-- =====>
-->
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/"
debug="false">
<!-- ===== -->
<!-- More Appender examples -->
<!-- ===== -->
<!-- Buffer events and log them asynchronously -->
<appender name="ASYNC" class="org.apache.log4j.AsyncAppender">
  <errorHandler class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
  <appender-ref ref="FILE"/>
<!--
  <appender-ref ref="CONSOLE"/>

```

```

    <appender-ref ref="SMTP"/>
    -->
</appender>
<!-- ===== -->
<!-- Setup the Root category -->
<!-- ===== -->
<root>
    <!--
        Set the root logger priority via a system property. Note this is
        parsed by log4j,
        so the full JBoss system property format is not supported; e.g.
        setting a default via ${jboss.server.log.threshold:WARN} will not
        work.
    -->
    <priority value="${jboss.server.log.threshold}"/>
    <!-- appender-ref ref="CONSOLE"/ -->
    <appender-ref ref="ASYNC"/>
</root>
</log4j:configuration>

```

In the ASYNC appender configuration, the FILE-based appender is active but the CONSOLE and SMTP appenders are explicitly excluded. In the Root category, the Appender reference is ASYNC, enabling the asynchronous appender.

### 7.3. LOCATION OF LOG FILES

The storage location of log files should be considered as a potential performance issue. Logging could perform poorly if on a file system and disk configuration that is poor for I/O throughput, degrading the whole platform's performance. The likely performance impact is heavily dependent on the rate at which messages are logged. Even if the logging rate is normally quite low, there can be an impact if debug/trace logging needs to be enabled to troubleshoot an issue. With this form of logging enabled the logs can grow very fast and impact performance dramatically, also making it more difficult to get the information needed.

To change the storage location of log files choose one of the following options, with option 1 recommended because it's easier to maintain:

1. Specify the preferred value for the variable from the command-line when starting JBoss Enterprise Application Platform, such as: `-Djboss.server.log.dir=<path_for_log_files>`
2. Edit the configuration file, replacing the variable `${jboss.server.log.dir}` with the preferred path.

### 7.4. WRAPPING LOG STATEMENTS

Here the discussion is not about the logging of JBoss Enterprise Application Platform but the application's own logging. It's common practice for developers to make liberal use of the code like the following example:

```
log.debug("Some text..." + Object);
```

There does not seem to be anything obviously wrong with the code. It states that when debug logging is turned on, the message should be written to the log. Since an application is only run with debug logging turned on when debugging is being done, you would not expect any impact on performance. In fact there is a likely impact and it occurs because of the way statements like this are evaluated by the platform. Before the statement itself is evaluated, a lot of activity occurs in preparation: creation of

temporary string objects, calling the object's `toString()` method, creation of an internal `LoggingEvent` object, formatting data, getting the current date, getting the thread ID of the calling thread and so on. Once all this is complete the statement is finally evaluated and if debug logging is not enabled, the created objects are again destroyed and processing continues to the next statement. The creation of these objects takes system resources, mainly CPU cycles and memory, with the performance impact multiplied by the number of debug statements processed.

There's a method which can reduce the performance impact of using debug statements, in which they're wrapped in a boolean statement. As the following example shows, whether or not debug logging is enabled is first evaluated and only then does the processing of the debug statement occur. There is a trade-off between processing of the boolean statement versus that of the debug statement but fewer system resources are used and so the performance impact is reduced.

```
if (debugEnabled()) {  
    log.debug("Some text..." + Object);  
}
```

It's not necessary to wrap every debug statement, only those which are often encountered. Take the example of a debug statement placed in a `try . . catch` structure, where the exception is rare. Since the debug statement will only rarely be evaluated, the performance difference between wrapping and not wrapping the statement in a boolean expression is likely to be minimal. As usual, there should be a measurable gain from the effort put into performance improvements.

## CHAPTER 8. PRODUCTION CONFIGURATION

In several chapters of this book it's been mentioned that the suggested optimizations are already present in the production configuration. Understanding just what this configuration is and how it was created can be useful in optimizing the JBoss Enterprise Application Platform. The production configuration was created with the aim of making the platform production-ready quickly and easily. By default, clustering and caching is enabled so that customers immediately have the benefit of these high-availability features. Although it's a good starting point for a real-life configuration, it should be tuned to best meet the application's requirements.

The main differences between the production and other configurations are:

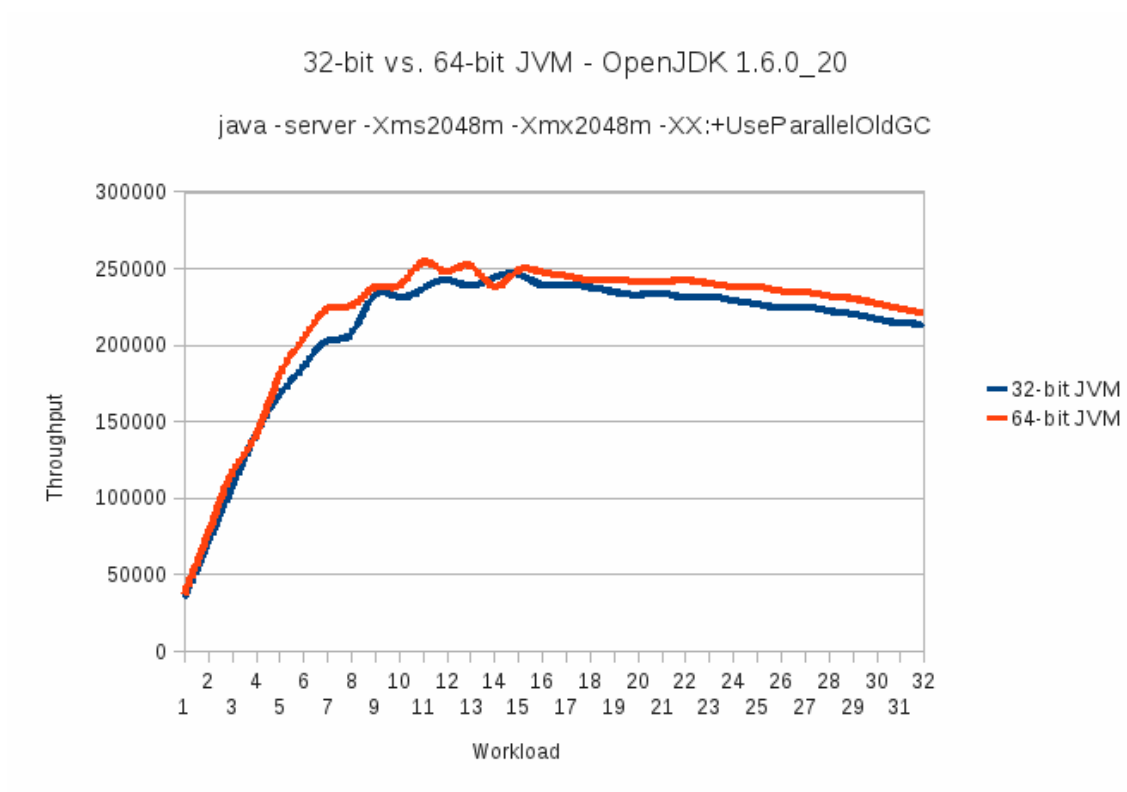
- Console logging is disabled.
- Asynchronous appender for logging is enabled.
- Logging verbosity is turned down in many cases.
  - Additional categories are defined to limit logging.
  - Default logging level is debug, but is overridden by the limit categories that were added.
    - This enables support to make things easier for customers to turn on debug level logging for various components, since the customer can just comment out the limiting category, and they will have debug logging for it.
- The cached connection manager is set to not be in debug mode.
- Clustering is properly configured to work with supported databases.
  - In the “all” configuration, the configuration is created to support the HSQL database, which is bundled for convenience with the server, but does not support clustering properly.
  - This is specifically the UUID generator.
- The heap size is set higher than in the community edition.
  - Typically the heap size for the JVM is set to 512 megabytes in the community edition. For the platform it's set to the largest value possible for a 32-bit JVM across all platforms.
  - The largest 32-bit setting that works across all supported JVM and operating systems is 1.3 gigabytes (Windows sets the limit).
- All administration is secured by default.
  - This was recently done in the community releases as well, so its now the same as the rest.
- Hot deployment is still turned on by default, but the scan rate is changed from every 5 seconds to once per minute.
  - It could be disabled altogether but many customers do use it in production and if there a large number of deployments, scanning every 5 seconds can affect performance badly.

## CHAPTER 9. JAVA VIRTUAL MACHINE TUNING

Java Virtual Machine tuning can be a complex task, given the number of configuration options and changes with each new release. As in similar situations, the best advice is to keep the configuration as simple as possible rather than trying to tweak every available setting. In this chapter several general options which apply to all workloads will be discussed, also the performance impact of a 32-bit versus 64-bit Java Virtual Machine.

### 9.1. 32-BIT VS. 64-BIT JVM

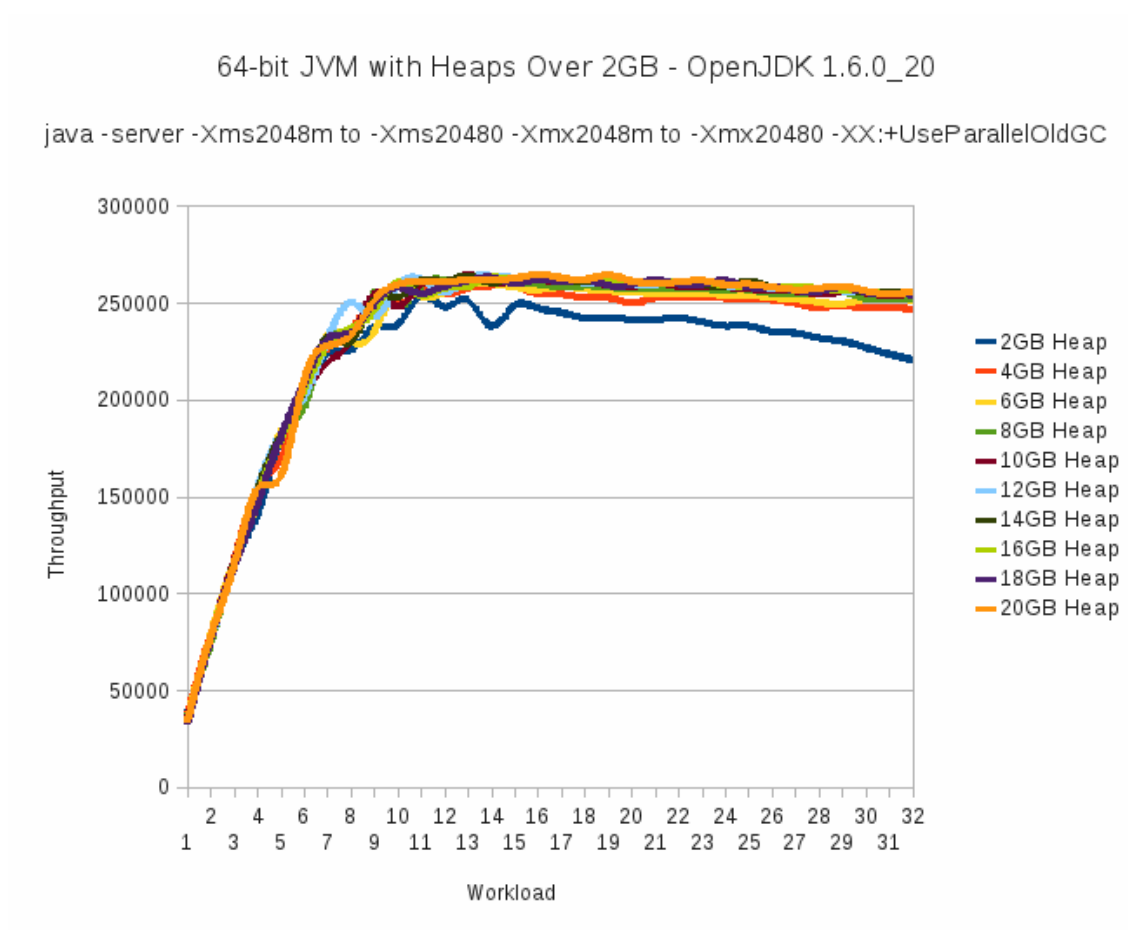
A question commonly raised when discussing performance is which gives better overall performance: 32-bit or 64-bit JVM? It would seem that a 64-bit JVM hosted by a 64-bit operating system should perform better than a 32-bit JVM on modern, 64-bit capable hardware. To try and provide some quantitative data on the topic, testing was performed with an industry standard workload. All tests were run on the same system, with the same operating system, JVM version and settings with one exception described below.



**Figure 9.1. JVM Throughput - 32-bit versus 64-bit**

In the graph above you can see that the two JVMs are quite similar in throughput, with the 64-bit JVM approximately 3.9% faster than the 32-bit JVM using a 2GB heap size. The 2GB heap size was used as it's the common limit for a 32-bit JVM on many operating systems. Having said that, the 32-bit JVM cannot be that large on Windows 2008, and can be larger on Red Hat Enterprise Linux (around 2.4 to 2.6GB depending on version). The 64-bit JVM has improved markedly over the last couple of years, relative to the 32-bit JVM, mainly because of the introduction of compressed ordinary object pointers (OOPs). OOPs implements a compression of the internal memory pointers within the JVM for objects, reducing the size of the heap. Instead of using the native machine word size for pointers, it compresses them to be competitive with the 32-bit JVM. The following article is recommended reading on the topic of compressed OOPs: <http://wikis.sun.com/display/HotSpotInternals/CompressedOops>

The real advantage of the 64-bit JVM is that heap sizes much larger than 2GB can be used. Large page memory with the 64-bit JVM give further optimizations. The following graph shows the results running from 4GB heap sizes, in two gigabyte increments, up to 20GB heap.



**Figure 9.2. JVM Throughput - comparison of incrementally larger heap sizes**

The difference in performance between 2GB and 4GB heap memory is a significant 6.11% higher throughput. For the remainder of the heap sizes increment, the incremental throughput improvements are smaller and between 18GB and 20GB heap sizes throughput drops slightly. With a 16GB heap the throughput is 9.64% higher, just a few percentage points higher than 4GB. It may not seem significant but those few percentage points in throughput equates to almost 23,000 operations per second faster. Since memory is relatively cheap, the increased throughput is more than justified. A larger heap size contributes to improved performance but further improvement requires use of a JVM and operating system feature called large page memory.

## 9.2. LARGE PAGE MEMORY

The default memory page size in most operating systems is 4 kilobytes (kb). For a 32-bit operating system the maximum amount of memory is 4 GB, which equates to 1,048,576  $((1024*1024*1024*4)/4096)$  memory pages. A 64-bit operating system can address 18 Exabytes of memory in theory which equates to a huge number of memory pages. The overhead of managing such a large number of memory pages is significant, regardless of the operating system. The largest heap size used for tests covered in this book was 20 GB, which equates to 5,242,880 memory pages, a five-fold increase over 4 GB.

Large memory pages are pages of memory which are significantly larger than 4 kb, usually 2 Mb. In some instances it's configurable, from 2MB to 256MB. For the systems used in the tests for this book, the page size is 2MB. With 2MB pages, the number of pages for a 20GB heap memory drops from

5,242,880 to 10,240! A 99.8% reduction in the number of memory pages means significantly less management overhead for the underlying operating system.

Large memory pages are locked in memory, and cannot be swapped to disk like regular memory pages which has both advantages and disadvantages. The advantage is that if the heap is using large page memory it can not be paged or swapped to disk so it's always readily available. For Linux the disadvantage is that for applications to use it they have to attach to it using the correct flag for the `shmget()` system call, also they need to have the proper security permissions for the `memlock()` system call. For any application that does not have the ability to use large page memory, the server will look and behave as if the large page memory does not exist, which could be a major problem. Care must be taken when configuring large page memory, depending on what else is running on your server besides the JVM.

To enable large page memory, add the following option to the command-line used to start the platform:

**-XX:+UseLargePages**

This option applies to OpenJDK, and the Oracle proprietary HotSpot-based JVM but there are similar options for IBM's and Oracle's JRockit JVMs. Refer to their documentation for further details. It's also necessary to change the following parameters:

**kernel.shmmax = n**

Where *n* is equal to the number of bytes of the maximum shared memory segment allowed on the system. You should set it at least to the size of the largest heap size you want to use for the JVM, or alternatively you can set it to the total amount of memory in the system.

**vm.nr\_hugepages = n**

Where *n* is equal to the number of large pages. You will need to look up the large page size in `/proc/meminfo`.

**vm.huge\_tlb\_shm\_group = gid**

Where *gid* is a shared group id for the users you want to have access to the large pages.

The next step is adding the following in `/etc/security/limits.conf`:

```
<username>    soft    memlock    n
<username>    hard    memlock    n
```

Where `<username>` is the runtime user of the JVM and *n* is the number of pages from `vm.nr_hugepages` \* the page size in KB from `/proc/meminfo`.

Instead of setting *n* to a specific value, this can be set to **unlimited**, which reduces maintenance.

Enter the command `sysctl -p` and these settings will be persistent. To confirm that this had taken effect, check that in the statistics available via `/proc/meminfo`, `HugePages_Total` is greater than 0. If `HugePages_Total` is either zero (0) or less than the value you configured, there are one of two things that could be wrong:

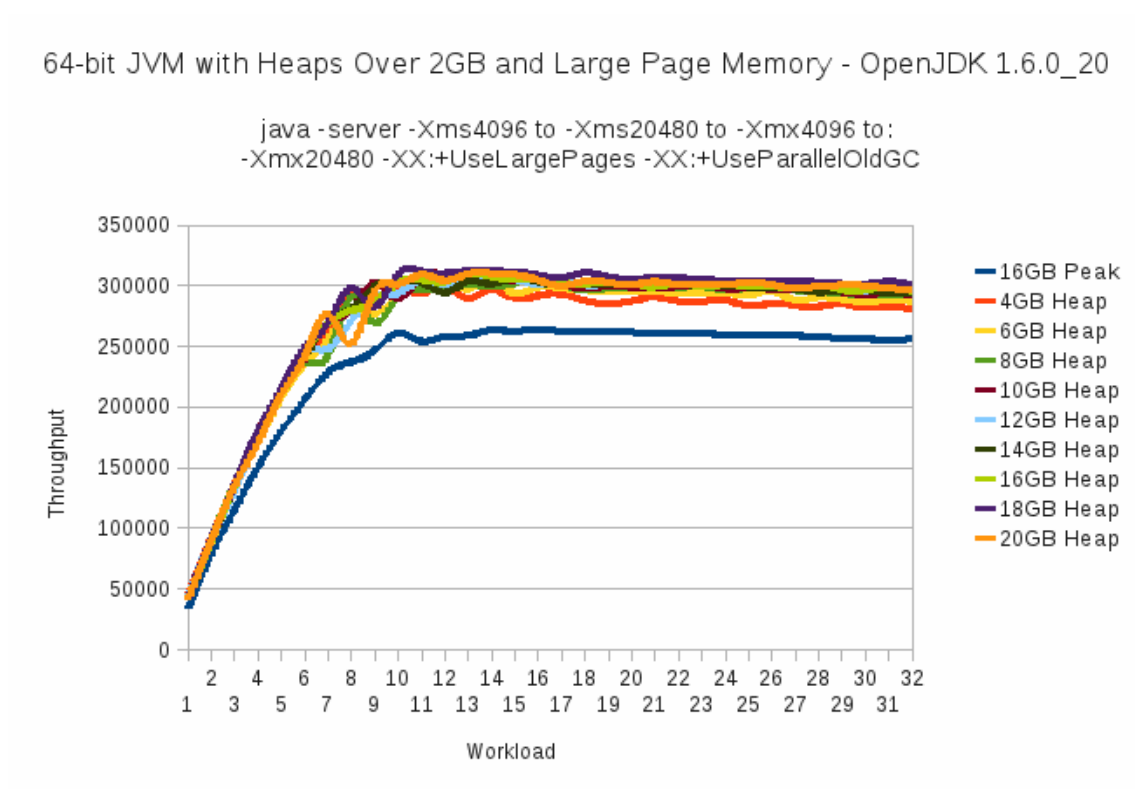
- the specified number of memory pages was greater than was available;
- there were not enough contiguous memory pages available.



When large page memory is allocated by the operating system, it must be in contiguous space. While the operating system is running, memory pages get fragmented. If the request failed because of this it may be necessary to reboot, so that the allocation of memory pages occurs before applications are started.

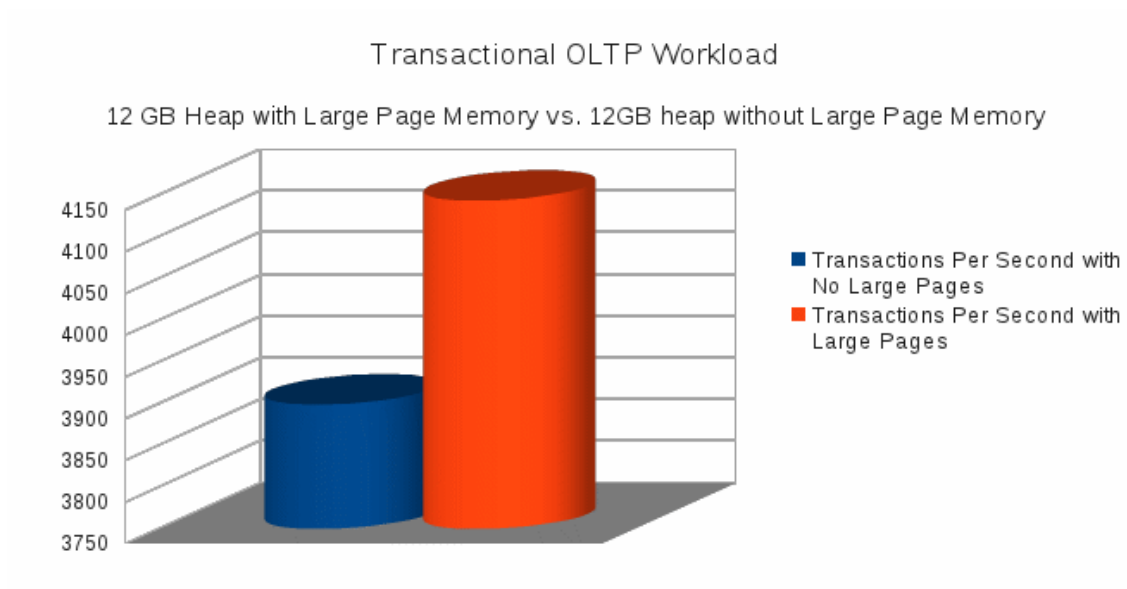
With the release of Red Hat Enterprise Linux 6, a new operating system capability called transparent huge pages (huge pages are the same as large pages) is available. This feature gives the operating system the ability to combine standard memory pages and make them large pages dynamically and without configuration. It enables the option of using large page memory for any application, even if it's not directly supported. Consider using this option since it reduces the configuration effort at a relatively small performance cost. Consult the Red Hat Enterprise Linux 6 documentation for specific configuration instructions.

The graph below shows the performance difference of the standard workload used in the 32-bit vs. 64-bit JVM comparison section.



**Figure 9.3. JVM Throughput - 32-bit versus 64-bit**

The peak line that was created with the 16GB heap is included to illustrate the difference. All heap sizes, even down to 4GB were substantially faster than the best without large page memory. The peak was actually the 18GB heap size run, which had 6.58% higher throughput than the 4GB result. This result was also 17.48% more throughput than the 16GB test run without large page memory. In these results it's evident that using large page memory is worthwhile, even for rather small heap sizes.

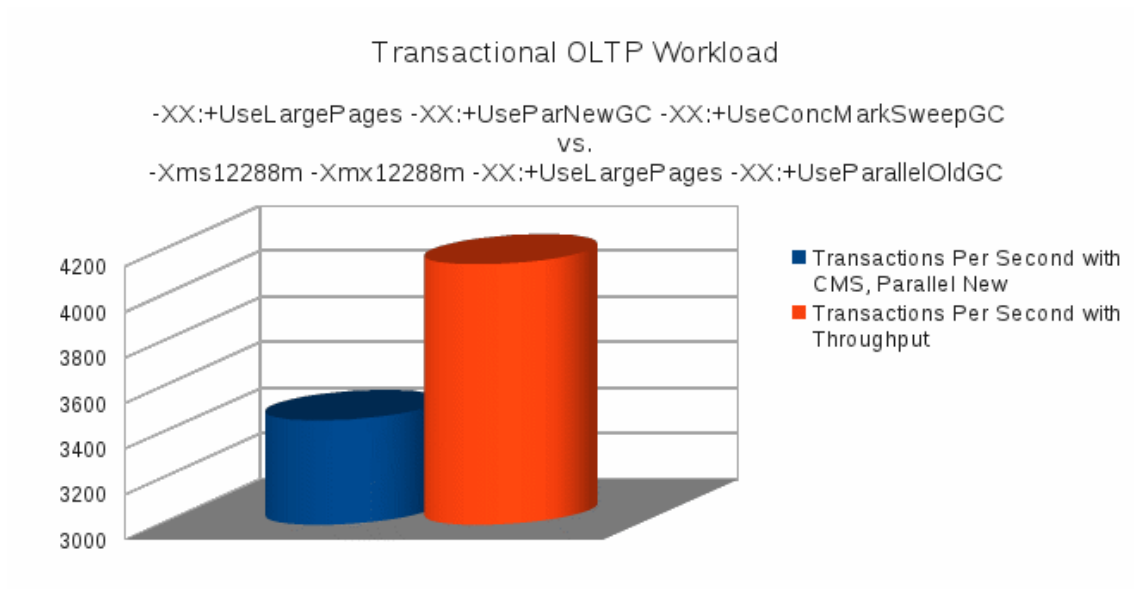


**Figure 9.4. JVM Throughput - comparison of with and without large pages enabled**

This graph compares two runs, with and without large page memory, using the EJB 3 OLTP application that has been referenced throughout this book. The results are similar to what we saw in the Java only workload. When executing this test, using the same 12GB heap size but without large page memory, the result was just under 3,900 transactions per second (3,899.02 to be exact). With large page memory enabled the result was over 4,100 transactions per second (4,143.25 to be exact). This represents a 6.26% throughput improvement which is not as large as the Java-only workload but this is to be expected, as this is a more complex workload, with a fairly large percentage of the time not even in the JVM itself. It's still significant however, because it equates to 244+ transactions per second more or 14,000+ extra transactions per minute. In real terms this means more work (processing) can be done in the same amount of time.

### 9.3. GARBAGE COLLECTION AND PERFORMANCE TUNING

For all the tests referred to in this book the JVM garbage collection option used was: - *XX:+UseParallelOldGC*. Many people choose the Concurrent Mark and Sweep (CMS) collector, collection is slower on both the Eden space and Old generation. The following graph shows the difference between using the throughput collector with parallel collection on the Old generation in comparison to using the CMS collector, which works on the Old generation and parallel collection on the Eden space.



**Figure 9.5. Transactional OLTP Workload**

The difference in performance between the Parallel Garbage Collector and the Concurrent Mark and Sweep Garbage collector can be explained by their underlying design. If you do not specify the option `-XX:+UseParallelOldGC`, the throughput collector defaults to parallel collection on the Eden space, and single threaded collection on the Old generation. With a 12GB heap, the Old generation is 8GB in size, which is a lot of memory to garbage collect in a single thread fashion. By specifying that the Old generation should also be collected in parallel, the collection algorithms designed for the highest throughput is used, hence the name "throughput collector". When the option `-XX:+UseParallelOldGC` is specified it also enables the option `-XX:+UseParNewGC`. In comparison, the CMS collector is not optimized for throughput but instead for more predictable response times. The focus of this book is tuning for throughput, not response time. The choice of garbage collector depends on whether higher throughput or more predictable response times benefits the application most. For real-time systems, the trade-off is usually lower throughput for more deterministic results in response times.

## 9.4. OTHER JVM OPTIONS TO CONSIDER

This section covers some additional JVM options:

- `-XX:+CompressedOops`
- `-XX:+AggressiveOpts`
- `-XX:+DoEscapeAnalysis`
- `-XX:+UseBiasedLocking`
- `-XX:+EliminateLocks`

**WARNING**

When considering the JVM options covered in this section, be aware that their behavior is subject to change depending on the version, therefore the effect of each should be tested before being implemented in a production environment.

Compressed OOPs, covered briefly in [Section 9.1, “32-bit vs. 64-bit JVM”](#), implements a compression of the internal memory pointers within the JVM for objects and so reduces the heap. From JVM revision 1.6.0\_20 and above compressed OOPs is on by default but if an earlier revision of the JDK is being used this option should be used for heap sizes 32GB and lower.

Aggressive optimizations, `-XX:+AggressiveOpts`, enables additional Hotspot JVM optimizations that are not enabled by default. The behavior enabled by this option is subject to change with each JVM release and so its effects should be tested prior to implementing it in production.

The final options are locking based options, `-XX:+DoEscapeAnalysis`, `-XX:+UseBiasedLocking` and `-XX:+EliminateLocks`. They are designed to work together to eliminate locking overhead. Their effect on performance is unpredictable for specific workloads and so require testing prior to being implemented. Reduced locking should improve concurrency and, on current multi-core hardware, improve throughput.

In summary, all these options should be considered on their merits. To accurately judge their impact on performance they should be tested independently of each other.

## CHAPTER 10. PROFILING

Profiling is monitoring the resource usage of an application with the aim of detecting areas for improvement. One of the difficulties in profiling is choosing a tool which provides useful data without having too great an impact on performance itself. On Red Hat Enterprise Linux, the OProfile tool is available for profiling the JBoss Enterprise Application Platform, requiring less system resources than other profiling tools.

### 10.1. INSTALL

#### Task: Install OProfile

Complete this task to install OProfile and its dependencies.

1. In a terminal, enter the following command:

```
yum install oprofile oprofile-jit
```

2. For all enabled yum repositories (`/etc/yum.repos.d/*.repo`), replace `enabled=0` with `enabled=1` in each file's `debuginfo` section. If there is no `debug` section, add it, using the example below as a template.

```
[rhel-debuginfo]
name=Red Hat Enterprise Linux $releasever - $basearch - Debug
baseurl=ftp://ftp.redhat.com/pub/redhat/linux/enterprise/$releasever
/en/os/$basearch/Debuginfo/
enabled=1
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release
```

3. In a terminal, enter the following command, modifying the JDK version number to match the version installed:

```
yum install yum-plugin-auto-update-debug-info java-1.6.0-openjdk-
debuginfo
```

### 10.2. ENABLE OPROFILE

Add the following to the JVM's start parameters, usually `standalone.conf` or `domain.conf` in the platform's `JBOSS_EAP_DIST/bin` directory.

```
# 64-bit JVM
-agentpath:/usr/lib64/oprofile/libjvmti_oprofile.so
# 32-bit JVM
-agentpath:/usr/lib/oprofile/libjvmti_oprofile.so
```

For example:

```
# Specify options to pass to the Java VM.
#
if [ "$JAVA_OPTS" = "x" ]; then
    JAVA_OPTS="-Xms10240m -Xmx10240m -XX:+UseLargePages -
```

```
XX:+UseParallelOldGC"
  JAVA_OPTS="$JAVA_OPTS -Djava.net.preferIPv4Stack=true -
Dorg.jboss.resolver.warning=true"
  JAVA_OPTS="$JAVA_OPTS -Dsun.rmi.dgc.client.gcInterval=3600000 -
Dsun.rmi.dgc.server.gcInterval=3600000"
  JAVA_OPTS="$JAVA_OPTS -
agentpath:/usr/lib64/oprofile/libjvmti_oprofile.so"
fi
```

## 10.3. CONTROLLING OPROFILE

OProfile runs as a daemon and must be running while profiling data is collected. The command-line utility `opcontrol` is used to manage OProfile's state.

```
opcontrol --start-daemon
```

Starts the OProfile daemon. This can be done at any time prior to collecting profile data.

```
opcontrol --start
```

Starts the collection of profiling data. This command must be given before starting the workload to be profiled.

```
opcontrol --stop
```

Stops the collection of profiling data.

```
opcontrol --dump
```

Dumps the profiling data to the default file.

```
opcontrol --shutdown
```

Shuts down the OProfile daemon.

## 10.4. DATA CAPTURE

Before starting any profiling, start the OProfile daemon. Once it is running, each profiling session typically consists of the following steps:

1. Start capturing profiling data
2. Start/run the workload
3. Stop capturing profiling data
4. Dump profiling data
5. Generate the profiling report

The simplest profiling report is produced by the command:

```
opreport -l --output-file=<filename>
```

-

This command produces a list of processes and their CPU cycles, from highest to lowest. Note that OProfile monitors the entire system, not just the Java processes. For additional information on OProfile, including additional reporting options, refer to the man page and the home page - <http://oprofile.sourceforge.net>.

## CHAPTER 11. PUTTING IT ALL TOGETHER

This book has covered many aspects of JBoss Enterprise Application Platform and opportunities for performance optimization. In presenting example performance improvements an OLTP application has been used as the test case. Before presenting the results of optimizing the performance of this application, here are details of the underlying infrastructure:

### 11.1. TEST CONFIGURATION

#### Server Configuration

- CPU: Intel Nehalem-based server with two, four-core Xeon E5520 processors, running at 2.27GHz, with hyper-threading enabled, giving 16 virtual cores
- Memory: 24GB of RAM
- Local storage: two 80GB solid state drives, in a RAID-0 configuration
- Operating system: Linux

A RAID 0 configuration was chosen for local storage to avoid I/O contention with the database

#### Database

Hosted on MySQL 5.5.x with the new asynchronous I/O capabilities and the transaction limit removed.

#### Load Driver

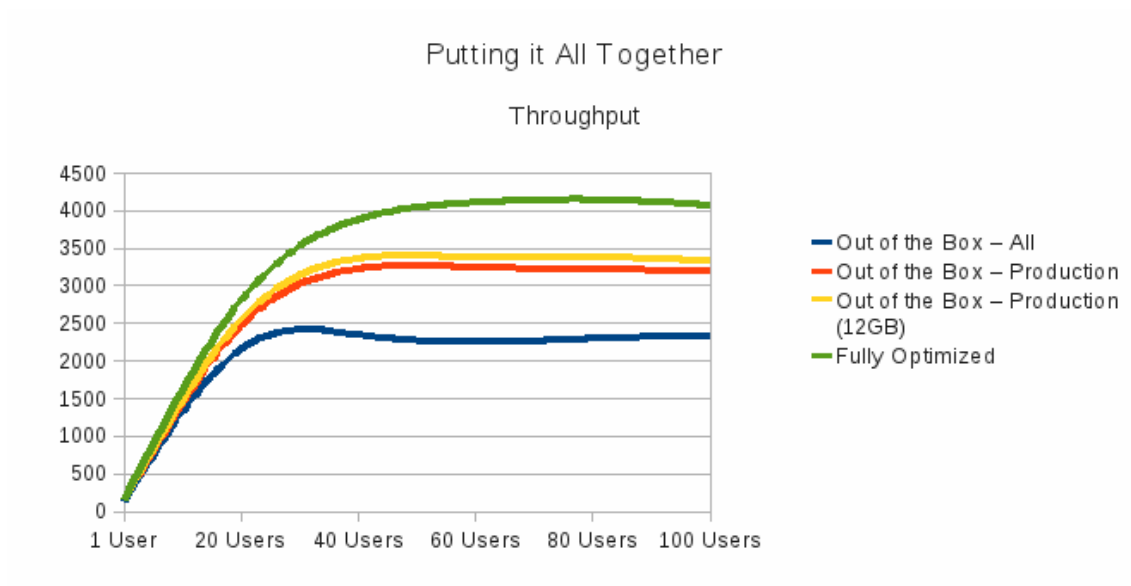
Laptop with four-core CPU and hyper-threading enabled, dedicated gigabit network connection between the laptop and the server.

### 11.2. TESTING STRATEGY

The tests conducted with the number of users increased from 1 to 100 users, stepping up by 20 users at a time, with no think time between requests from the load driver. This allowed quick turnaround time per test run, and created results that were easier to present. The same loads were run using 1 second think times, but it scales to so many users that it makes it difficult to present the data. The results were very similar anyway, from a throughput improvement perspective. Before each test run the order transactions in the workload were deleted and the InnoDB buffer pool was checked for dirty pages before starting the next run. Each test was run five times to ensure reproducibility of the results. A maximum of 100 users was used because at was at that point which throughput plateaued.

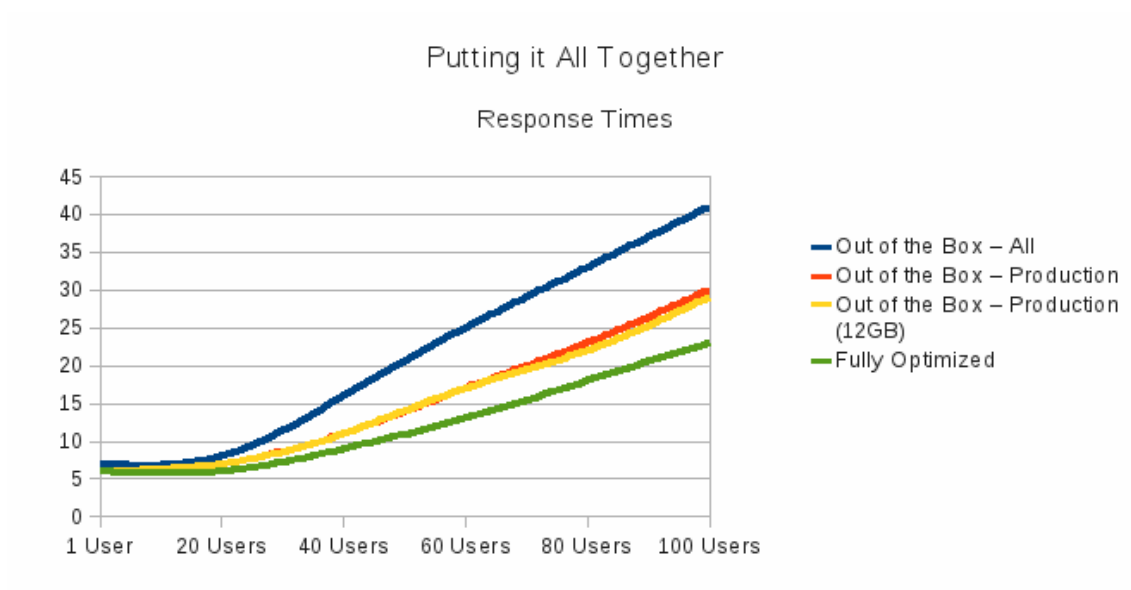
### 11.3. TEST RESULTS





**Figure 11.1.**

The above graph compares the throughput of four configurations (transactions per second). Running the same workload, tests were run with the “all” configuration unmodified, except for deploying the application and its data source, as the baseline with no optimizations. As discussed in the book, the “all” configuration is the basis for the production configuration. The next test was the “production” configuration, with nothing modified but the application and data source deployed, just like the “all” configuration. As you can see from the results, the changes made to the production configuration had a very positive effect on throughput. In fact, throughput is 39.13% higher at the peaks and that was achieved simply by deploying the application into a different configuration. The next result is using the “production” configuration again, but with the heap size set to match the fully optimized test run, which was 12GB. This test was run to demonstrate that making the heap size larger does not necessarily account for much of a performance gains since the throughput in this test was only 4.14% higher. An increase in heap size must be matched with changes in garbage collection and large page memory. Without those optimizations a larger heap may help, but not by as much as might be expected. The fully optimized result is the very best with throughput 77.82% higher.



**Figure 11.2.**

Optimizations made a significant difference in response times. Not only is the fully optimized configuration the fastest by a wide margin, its slope is also less steep, showing the improved scalability of the fully optimized result. At the peak throughput level, the fully optimized result has a response

time that is 45.45% lower than the baseline. At the 100 user level, the response time is 78.26% lower!

## APPENDIX A. REVISION HISTORY

<b>Revision 5.2.0-100.400</b> Rebuild with publican 4.0.0	<b>2013-10-31</b>	<b>Rüdiger Landmann</b>
<b>Revision 5.2.0-100</b> Incorporated changes for JBoss Enterprise Application Platform 5.2.0 GA. For information about documentation changes to this guide, refer to <i>Release Notes 5.2.0</i> .	<b>Wed 23 Jan 2013</b>	<b>Russell Dickenson</b>
<b>Revision 5.1.2-133</b> Rebuild for Publican 3.	<b>Wed 18 Jul 2012</b>	<b>Anthony Towns</b>
<b>Revision 5.1.2-100</b> First release.	<b>Thu 8 Dec 2011</b>	<b>Russell Dickenson</b>