Oracle® Cloud

Tuning Performance of Oracle WebLogic Server

12*c* (12.1.3)

E60584-01

June 2016

This document is for people who monitor performance and tune the components in a WebLogic Server environment.

Oracle Cloud Tuning Performance of Oracle WebLogic Server, 12c (12.1.3)

E60584-01

Contents

Preface

This preface describes the document accessibility features and conventions used in this guide—*Tuning Performance of Oracle WebLogic Server*.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction and Roadmap

This chapter describes the contents and organization of this guide—*Tuning Performance of Oracle WebLogic Server*.

This chapter includes the following sections:

- Document Scope and Audience
- Guide to this Document
- Performance Features of this Release

Document Scope and Audience

This document is written for people who monitor performance and tune the components in a WebLogic Server environment. It is assumed that readers know server administration and hardware performance tuning fundamentals, WebLogic Server, XML, and the Java programming language.

Guide to this Document

- This chapter, Introduction and Roadmap introduces the organization of this guide.
- Top Tuning Recommendations for WebLogic Server discusses the most frequently recommended steps for achieving optimal performance tuning for applications running on WebLogic Server.
- Performance Tuning Roadmap provides a roadmap to help tune your application environment to optimize performance.
- Tuning Java Virtual Machines (JVMs) discusses JVM tuning considerations.
- Tuning WebLogic Diagnostic Framework and Java Flight Recorder Integration provides information on how WebLogic Diagnostic Framework (WLDF) works with Java Flight Recorder.
- Tuning WebLogic Server contains information on how to tune WebLogic Server to match your application needs.
- Tuning the WebLogic Persistent Store provides information on how to tune a persistent store.
- DataBase Tuning provides information on how to tune your data base.
- Tuning WebLogic Server EJBs provides information on how to tune applications that use EJBs.

- Tuning Message-Driven Beans provides information on how to tune Message-Driven beans.
- Tuning Data Sources provides information on how to tune JDBC applications.
- Tuning Transactions provides information on how to tune Logging Last Resource transaction optimization.
- Tuning WebLogic JMS provides information on how to tune applications that use WebLogic JMS.
- Tuning WebLogic JMS Store-and-Forward provides information on how to tune applications that use JMS Store-and-Forward.
- Tuning WebLogic Message Bridge provides information on how to tune applications that use the WebLogic Message Bridge.
- Tuning Resource Adapters provides information on how to tune applications that use resource adaptors.
- Tuning Web Applications provides best practices for tuning WebLogic Web applications and application resources:
- Tuning Web Services provides information on how to tune applications that use Web services.

Performance Features of this Release

WebLogic Server introduces the following performance enhancements:

- Improving Cluster Throughput using XA Transaction Cluster Affinity
- Configure XA Transactions without TLogs.

2 Top Tuning Recommendations for WebLogic Server

Tuning WebLogic Server and your WebLogic Server application is a complex and iterative process. To get you started, we recommend the following ways optimize your application's performance. These tuning techniques are applicable to nearly all WebLogic applications.

This chapter includes the following sections:

- Tune Pool Sizes
- Use the Prepared Statement Cache
- Use Logging Last Resource Optimization
- Tune Connection Backlog Buffering
- Use Optimistic or Read-only Concurrency
- Use Local Interfaces
- Use eager-relationship-caching
- Tune HTTP Sessions
- Tune Messaging Applications

Tune Pool Sizes

Provide pool sizes (such as pools for JDBC connections, Stateless Session EJBs, and MDBs) that maximize concurrency for the expected thread utilization.

- For WebLogic Server releases 9.0 and higher—A server instance uses a self-tuned thread-pool. The best way to determine the appropriate pool size is to monitor the pool's current size, shrink counts, grow counts, and wait counts. See Thread Management . Tuning MDBs are a special case, please see Tuning Message-Driven Beans.

Use the Prepared Statement Cache

The prepared statement cache keeps compiled SQL statements in memory, thus avoiding a round-trip to the database when the same statement is used later. See Tuning Data Sources.

Use Logging Last Resource Optimization

When using transactional database applications, consider using the JDBC data source Logging Last Resource (LLR) transaction policy instead of XA. The LLR optimization can significantly improve transaction performance by safely eliminating some of the 2PC XA overhead for database processing, especially for two-phase commit database insert, update, and delete operations. For more information, see Tuning Data Sources.

Tune Connection Backlog Buffering

You can tune the number of connection requests that a WebLogic Server instance accepts before refusing additional requests. This tunable applies primarily for Web applications. See Tuning Connection Backlog Buffering.

Use Optimistic or Read-only Concurrency

Use optimistic concurrency with cache-between-transactions or read-only concurrency with query-caching for CMP EJBs wherever possible. Both of these two options leverage the Entity Bean cache provided by the EJB container.

- Optimistic-concurrency with cache-between-transactions work best with read-mostly beans. Using verify-reads in combination with these provides high data consistency guarantees with the performance gain of caching. See Tuning WebLogic Server EJBs.
- Query-caching is a WebLogic Server 9.0 feature that allows the EJB container to cache results for arbitrary non-primary-key finders defined on read-only EJBs. All of these parameters can be set in the application/module deployment descriptors. See Concurrency Strategy.

Use Local Interfaces

Use local-interfaces or use call-by-reference semantics to avoid the overhead of serialization when one EJB calls another or an EJB is called by a servlet/JSP in the same application. Note the following:

- In release prior to WebLogic Server 8.1, call-by-reference is turned on by default. For releases of WebLogic Server 8.1 and higher, call-by-reference is turned off by default. Older applications migrating to WebLogic Server 8.1 and higher that do not explicitly turn on call-by-reference may experience a drop in performance.
- This optimization does not apply to calls across different applications.

Use eager-relationship-caching

Use eager-relationship-caching wherever possible. This feature allows the EJB container to load related beans using a single SQL statement. It improves performance by reducing the number of database calls to load related beans in transactions when a bean and it's related beans are expected to be used in that transaction. See Tuning WebLogic Server EJBs.

Tune HTTP Sessions

Optimize your application so that it does as little work as possible when handling session persistence and sessions. You should also design a session management strategy that suits your environment and application. See Session Management.

Tune Messaging Applications

Oracle provides messaging users a rich set of performance tunables. In general, you should always configure quotas and paging. See:

- Tuning the WebLogic Persistent Store
- Tuning WebLogic JMS
- Tuning WebLogic JMS Store-and-Forward
- Tuning WebLogic Message Bridge

3 Performance Tuning Roadmap

Use the performance tuning roadmap and tuning tips described in this chapter as a methodical approach to optimizing your system's performance.

This chapter includes the following sections:

- Performance Tuning Roadmap
- Tuning Tips

Performance Tuning Roadmap

The following steps provide a roadmap to help tune your application environment to optimize performance:

1. Understand Your Performance Objectives
2. Measure Your Performance Metrics
3. Locate Bottlenecks in Your System
4. Minimize Impact of Bottlenecks
5. Achieve Performance Objectives

Understand Your Performance Objectives

To determine your performance objectives, you need to understand the application deployed and the environmental constraints placed on the system. Gather information about the levels of activity that components of the application are expected to meet, such as:

- The anticipated number of users.
- The number and size of requests.
- The amount of data and its consistency.
- Determining your target CPU utilization.

  Your target CPU usage should not be 100%, you should determine a target CPU utilization based on your application needs, including CPU cycles for peak usage. If your CPU utilization is optimized at 100% during normal load hours, you have no capacity to handle a peak load. In applications that are latency sensitive and maintaining the ability for a fast response time is important, high CPU usage (approaching 100% utilization) can reduce response times while throughput stays constant or even increases because of work queuing up in the server. For such applications, a 70% - 80% CPU utilization recommended. A good target for non-latency sensitive applications is about 90%.

Performance objectives are limited by constraints, such as

- The ability to interoperate between domains, use legacy systems, support legacy data.
- Development, implementation, and maintenance costs.

You will use this information to set realistic performance objectives for your application environment, such as response times, throughput, and load on specific hardware.

Measure Your Performance Metrics

After you have determined your performance criteria in Understand Your Performance Objectives, take measurements of the metrics you will use to quantify your performance objectives.

Monitor Disk and CPU Utilization

Run your application under a high load while monitoring the:

- Application server (disk and CPU utilization)
- Database server (disk and CPU utilization)

The goal is to get to a point where the application server achieves your target CPU utilization. If you find that the application server CPU is under utilized, confirm whether the database is bottle necked. If the database CPU is 100 percent utilized, then check your application SQL calls query plans. For example, are your SQL calls using indexes or doing linear searches? Also, confirm whether there are too many `ORDER BY` clauses used in your application that are affecting the database CPU.

If you discover that the database disk is the bottleneck (for example, if the disk is 100 percent utilized), try moving to faster disks or to a RAID (redundant array of independent disks) configuration, assuming the application is not doing more writes then required.

Once you know the database server is not the bottleneck, determine whether the application server disk is the bottleneck. Some of the disk bottlenecks for application server disks are:

- Persistent Store writes
- Transaction logging (tlogs)
- HTTP logging
- Server logging

The disk I/O on an application server can be optimized using faster disks, disabling synchronous JMS writes, using JTA direct writes for tlogs, or increasing the HTTP log buffer.

Locate Bottlenecks in Your System

If you determine that neither the network nor the database server is the bottleneck, start looking at your operating system, JVM, and WebLogic Server configurations. Most importantly, is the machine running WebLogic Server able to get your target CPU utilization with a high client load? If the answer is no, then check if there is any locking taking place in the application. You should profile your application to pinpoint bottlenecks and improve application performance, see "Java Mission Control."

Tip:

Even if you find that the CPU is 100 percent utilized, you should profile your application for performance improvements.

Minimize Impact of Bottlenecks

In this step, you tune your environment to minimize the impact of bottlenecks on your performance objectives. It is important to realize that in this step you are minimizing the impact of bottlenecks, not eliminating them. Tuning allows you to adjust resources to achieve your performance objectives. For the scope of this document, this includes (from most important to least important):

- Tune Your Application
- Tune your DB
- Tune WebLogic Server Performance Parameters
- Tune Your JVM
- Tuning the WebLogic Persistent Store

Tune Your Application

To quote the authors of *Oracle WebLogic Server: Optimizing WebLogic Server Performance:* "Good application performance starts with good application design. Overly-complex or poorly-designed applications will perform poorly regardless of the system-level tuning and best practices employed to improve performance." In other words, a poorly designed application can create unnecessary bottlenecks. For example, resource contention could be a case of poor design, rather than inherent to the application domain.

For more information, see:

- Tuning WebLogic Server EJBs
- Tuning Message-Driven Beans
- Tuning Data Sources
- Tuning Transactions
- Tuning WebLogic JMS
- Tuning WebLogic JMS Store-and-Forward
- Tuning WebLogic Message Bridge
- Tuning Resource Adapters
- Tuning Web Applications
- Tuning Web Services

Tune your DB

You can tune your Oracle Database Cloud Service instance. For information about how to do this, see Database Tuning.

Tune WebLogic Server Performance Parameters

The WebLogic Server uses a number of OOTB (out-of-the-box) performance-related parameters that can be fine-tuned depending on your environment and applications. Tuning these parameters based on your system requirements (rather than running with default settings) can greatly improve both single-node performance and the scalability characteristics of an application. See Tuning WebLogic Server.

Tune Your JVM

The Java virtual machine (JVM) is a virtual "execution engine" instance that executes the bytecodes in Java class files on a microprocessor. See Tuning Java Virtual Machines (JVMs).

Achieve Performance Objectives

Performance tuning is an iterative process. After you have minimized the impact of bottlenecks on your system, go to Step 2, Measure Your Performance Metrics and determine if you have met your performance objectives.

Tuning Tips

This section provides tips and guidelines when tuning overall system performance:

- Performance tuning is not a silver bullet. Simply put, good system performance depends on: good design, good implementation, defined performance objectives, and performance tuning.
- Performance tuning is ongoing process. Implement mechanisms that provide performance metrics which you can compare against your performance objectives, allowing you to schedule a tuning phase before your system fails.
- The object is to meet your performance objectives, not eliminate all bottlenecks. Resources within a system are finite. By definition, at least one resource (CPU, memory, or I/O) will be a bottleneck in the system. Tuning allows you minimize the impact of bottlenecks on your performance objectives.
- Design your applications with performance in mind:
  - Keep things simple - avoid inappropriate use of published patterns.
  - Apply Java EE performance patterns.
  - Optimize your Java code.

4

Tuning Java Virtual Machines (JVMs)

Configure the JVM tuning options for WebLogic Server. The Java virtual machine (JVM) is a virtual "execution engine" instance that executes the bytecodes in Java class files on a microprocessor. How you tune your JVM affects the performance of WebLogic Server and your applications.

This chapter includes the following sections:

- JVM Tuning Considerations
- Garbage Collection
- Increasing Java Heap Size for Managed Servers

JVM Tuning Considerations

The following table presents general JVM tuning considerations for WebLogic Server.

Table 4-1 General JVM Tuning Considerations

Tuning Factor	Information Reference
Tuning heap size and garbage collection	For WebLogic Server heap size tuning details, see Garbage Collection.
Mixed client/server JVMs	Deployments using different JVM versions for the client and server are supported in WebLogic Server. See "Supported Configurations" in *What's New in Oracle WebLogic Server* for links to the latest supported mixed client/server JVMs.
UNIX threading models	Choices you make about Solaris threading models can have a large impact on the performance of your JVM on Solaris. You can choose from multiple threading models and different methods of synchronization within the model, but this varies from JVM to JVM. See *"Thread Priority on the Solaris Platform"* at `http://docs.oracle.com/javase/7/docs/technotes/guides/vm/thread-priorities.html`.

Garbage Collection

Garbage collection is the VM's process of freeing up unused Java objects in the Java heap. The following sections provide information on tuning your VM's garbage collection:

- VM Heap Size and Garbage Collection
- Using Verbose Garbage Collection to Determine Heap Size

- Specifying Heap Size Values
- Automatically Logging Low Memory Conditions
- Manually Requesting Garbage Collection

VM Heap Size and Garbage Collection

The Java heap is where the objects of a Java program live. It is a repository for live objects, dead objects, and free memory. When an object can no longer be reached from any pointer in the running program, it is considered "garbage" and ready for collection. A best practice is to tune the time spent doing garbage collection to within 5% of execution time.

The JVM heap size determines how often and how long the VM spends collecting garbage. An acceptable rate for garbage collection is application-specific and should be adjusted after analyzing the actual time and frequency of garbage collections. If you set a large heap size, full garbage collection is slower, but it occurs less frequently. If you set your heap size in accordance with your memory needs, full garbage collection is faster, but occurs more frequently.

The goal of tuning your heap size is to minimize the time that your JVM spends doing garbage collection while maximizing the number of clients that WebLogic Server can handle at a given time. To ensure maximum performance during benchmarking, you might set high heap size values to ensure that garbage collection does not occur during the entire run of the benchmark.

You might see the following Java error if you are running out of heap space:

```
java.lang.OutOfMemoryError <<no stack trace available>>
java.lang.OutOfMemoryError <<no stack trace available>>
Exception in thread "main"
```

To modify heap space values, see Specifying Heap Size Values.

To configure WebLogic Server to detect automatically when you are running out of heap space and to address low memory conditions in the server, see Automatically Logging Low Memory Conditions and Specifying Heap Size Values.

Using Verbose Garbage Collection to Determine Heap Size

The verbose garbage collection option (`verbosegc`) enables you to measure exactly how much time and resources are put into garbage collection. To determine the most effective heap size, turn on verbose garbage collection and redirect the output to a log file for diagnostic purposes.

The following steps outline this procedure:

1. Monitor the performance of WebLogic Server under maximum load while running your application.

2. Use the `-verbosegc` option to turn on verbose garbage collection output for your JVM and redirect both the standard error and standard output to a log file.

   This places thread dump information in the proper context with WebLogic Server informational and error messages, and provides a more useful log for diagnostic purposes.

   For example, on Windows and Solaris, enter the following:

```
% java -ms32m -mx200m -verbosegc -classpath $CLASSPATH
-Dweblogic.Name=%SERVER_NAME% -Dbea.home="C:\Oracle\Middleware"
-Dweblogic.management.username=%WLS_USER%
-Dweblogic.management.password=%WLS_PW%
-Dweblogic.management.server=%ADMIN_URL%
-Dweblogic.ProductionModeEnabled=%STARTMODE%
-Djava.security.policy="%WL_HOME%\server\lib\weblogic.policy" weblogic.Server >>
logfile.txt 2>&1
```

where the `logfile.txt 2>&1` command redirects both the standard error and standard output to a log file.

3. Analyze the following data points:

   a. How often is garbage collection taking place? In the weblogic.log file, compare the time stamps around the garbage collection.

   b. How long is garbage collection taking? Full garbage collection should not take longer than 3 to 5 seconds.

   c. What is your average memory footprint? In other words, what does the heap settle back down to after each full garbage collection? If the heap always settles to 85 percent free, you might set the heap size smaller.

4. Make sure that the heap size is not larger than the available free RAM on your system.

   Use as large a heap size as possible without causing your system to "swap" pages to disk. The amount of free RAM on your system depends on your hardware configuration and the memory requirements of running processes on your machine. See your system administrator for help in determining the amount of free RAM on your system.

5. If you find that your system is spending too much time collecting garbage (your allocated virtual memory is more than your RAM can handle), lower your heap size.

   Typically, you should use 80 percent of the available RAM (not taken by the operating system or other processes) for your JVM.

6. If you find that you have a large amount of available free RAM remaining, run more instances of WebLogic Server on your machine.

   Remember, the goal of tuning your heap size is to minimize the time that your JVM spends doing garbage collection while maximizing the number of clients that WebLogic Server can handle at a given time.

Specifying Heap Size Values

System performance is greatly influenced by the size of the Java heap available to the JVM. This section describes the command line options you use to define the heap sizes values.You must specify Java heap size values each time you start an instance of WebLogic Server. This can be done either from the `java` command line or by modifying the default values in the sample startup scripts that are provided with the WebLogic distribution for starting WebLogic Server.

Tuning Tips for Heap Sizes

The following section provides general guidelines for tuning VM heap sizes:

- The heap sizes should be set to values such that the maximum amount of memory used by the VM does not exceed the amount of available physical RAM. If this value is exceeded, the OS starts paging and performance degrades significantly. The VM always uses more memory than the heap size. The memory required for internal VM functionality, native libraries outside of the VM, and permanent generation memory (for the Sun VM only: memory required to store classes and methods) is allocated in addition to the heap size settings.
- When using a generational garbage collection scheme, the nursery size should not exceed more than half the total Java heap size. Typically, 25% to 40% of the heap size is adequate.
- In production environments, set the minimum heap size and the maximum heap size to the same value to prevent wasting VM resources used to constantly grow and shrink the heap. This also applies to the New generation heap sizes.

JRockit JVM Heap Size Options

Although JRockit provides automatic heap resizing heuristics, they are not optimal for all applications. In most situations, best performance is achieved by tuning the VM for each application by adjusting the heaps size options shown in the following table.

Table 4-2 JRockit JVM Heap Size Options

Task	Option	Description
Setting the Nursery	`-Xns`	Optimally, you should try to make the nursery as large as possible while still keeping the garbage collection pause times acceptably low. This is particularly important if your application is creating a lot of temporary objects. The maximum size of a nursery cannot exceed 95% of the maximum heap size.
Setting initial and minimum heap size	`-Xms`	Oracle recommends setting the minimum heap size (`-Xms`) equal to the maximum heap size (`-Xmx`) to minimize garbage collections.
Setting maximum heap size	`-Xmx`	Setting a low maximum heap value compared to the amount of live data decrease performance by forcing frequent garbage collections.
Setting garbage collection	`-Xgc: parallel`	
Performs adaptive optimizations as early as possible in the Java application run.	`-XXaggressive:memory`	To do this, the bottleneck detector will run with a higher frequency from the start and then gradually lower its frequency. This options also tells JRockit to use the available memory aggressively.

For example, when you start a WebLogic Server instance from a java command line, you could specify the JRockit VM heap size values as follows:

```
$ java -Xns10m -Xms512m -Xmx512m
```

The default size for these values is measured in bytes. Append the letter 'k' or 'K' to the value to indicate kilobytes, 'm' or 'M' to indicate megabytes, and 'g' or 'G' to indicate gigabytes. The example above allocates 10 megabytes of memory to the Nursery heap sizes and 512 megabytes of memory to the minimum and maximum heap sizes for the WebLogic Server instance running in the JVM.

Automatically Logging Low Memory Conditions

WebLogic Server enables you to automatically log low memory conditions observed by the server. WebLogic Server detects low memory by sampling the available free memory a set number of times during a time interval. At the end of each interval, an average of the free memory is recorded and compared to the average obtained at the next interval. If the average drops by a user-configured amount after any sample interval, the server logs a low memory warning message in the log file and sets the server health state to "warning."

Manually Requesting Garbage Collection

You may find it necessary to manually request full garbage collection from the WebLogic Server Administration Console. When you do, remember that garbage collection is costly as the JVM often examines every living object in the heap.

Increasing Java Heap Size for Managed Servers

For better performance, you may need to increase the heap size for each Managed Server in your environment.

The following sections provide information about how to modify the Java heap size for Managed Servers:

- Using the Administration Console to Set Java Heap Size
- Modify the startManagedWebLogic Script to Set Java Heap Size
- Using the Command Line to Set Java Heap Size
- Determining the Memory Values Used by a Managed Server

See "Configuring Remote Startup Arguments" in *Administering Node Manager for Oracle WebLogic Server*.

Using the Administration Console to Set Java Heap Size

If you use Node Manager to start the Managed Servers, you can specify a heap size as a Java argument on the **Server Start** tab for each Managed Server. Your heap size values are then persisted in the `startup.properties` file for the server.

Modify the startManagedWebLogic Script to Set Java Heap Size

You can update the `startManagedWebLogic` script with the required heap size in `JAVA_OPTIONS`. For example:

```
JAVA_OPTIONS="-Xms2g -Xmx2g" ${JAVA_OPTIONS}
```

See "Starting Managed Servers with a Startup Script" in *Administering Server Startup and Shutdown for Oracle WebLogic Server*.

Using the Command Line to Set Java Heap Size

You can pass JVM parameters when starting a managed server by invoking `weblogic.Server` class in a Java command. See "weblogic.Server Command-Line Reference" in the *Command Reference for Oracle WebLogic Server*.

Determining the Memory Values Used by a Managed Server

Start scripts and the Administration Console (the `startup.properties` file) are common ways to configure memory arguments in managed servers. Often, they are set in multiple places and with different values. How do you determine which values are actually used by a running managed server?

A running managed server always uses the last set of memory arguments passed to the server during startup. You can verify this by looking through the log files. If you see the memory arguments listed multiple times, the last set in the output contains the values used by the server. If you used the Administration Console to set the values, see Using the Administration Console to Set Java Heap Size, the `startup.properties` file is always processed last, after all scripts. This provides a convenient method to tune memory size for an individual managed server when using common scripts. If you prefer not to include memory arguments that are not actually used in the environment, you will need to remove any extraneous memory arguments such as `MEM_ARGS` and `JAVA_OPTONS` from the scripts used to start a managed server.

5

Tuning WebLogic Diagnostic Framework and Java Flight Recorder Integration

Tune WLDF and Java Flight Recorder using the tips and guidelines described in this chapter.

This chapter includes the following sections:

- Using Java Flight Recorder
- Using WLDF
- Tuning Considerations

Using Java Flight Recorder

Java Flight Recorder is a performance monitoring and profiling tool that records diagnostic information on a continuous basis, making it always available, even in the wake of catastrophic failure such as a system crash. Java Flight Recorder is available in Oracle HotSpot. When WebLogic Server is configured with HotSpot, Java Flight Recorder is not enabled by default. See "Using Java Flight Recorder with Oracle HotSpot" in *Configuring and Using the Diagnostics Framework for Oracle WebLogic Server*, for information about how to enable Java Flight Recorder with WebLogic Server.

Using WLDF

If WebLogic Server is configured with Oracle HotSpot, and the Java Flight Recorder is enabled, the Java Flight Recorder data is automatically also captured in the diagnostic image capture. This data can be extracted from the diagnostic image capture and viewed in Java Mission Control. If Java Flight Recorder is not enabled, or if WebLogic Server is configured with a different JVM, the Java Flight Recorder data is not captured in the diagnostics image capture.

The volume of Java Flight Recorder data that is captured can be configured using the `Diagnostic Volume` attribute in the WebLogic Server Administration Console, see "Configuring WLDF Diagnostic Volume" in *Configuring and Using the Diagnostics Framework for Oracle WebLogic Server*. You can also set the volume using WLST.

Tuning Considerations

In most environments, there is little performance impact when the `Diagnostic Volume` is set to `Low` and the most performance impact if `Diagnostic Volume` is set to `High`. The volume of diagnostic data produced by WebLogic Server needs to be weighed against potential performance loss.

6 Tuning WebLogic Server

Tune WebLogic Server to match your application using the procedures described in this chapter.

This chapter includes the following sections:

- Setting Java Parameters for Starting WebLogic Server
- Development vs. Production Mode Default Tuning Values
- Deployment
- Thread Management
- Tuning Network I/O
- Optimize Java Expressions
- Using WebLogic Server Clusters to Improve Performance
- Monitoring a WebLogic Server Domain
- Tuning Class and Resource Loading
- SSL Considerations

Setting Java Parameters for Starting WebLogic Server

Java parameters must be specified whenever you start WebLogic Server. For simple invocations, this can be done from the command line with the `weblogic.Server` command. However, because the arguments needed to start WebLogic Server from the command line can be lengthy and prone to error, Oracle recommends that you incorporate the command into a script. To simplify this process, you can modify the default values in the sample scripts that are provided with the WebLogic Server distribution, as described in "Specifying Java Options for a WebLogic Server Instance" in *Administering Server Startup and Shutdown for Oracle WebLogic Server*.

If you used the Configuration Wizard to create your domain, the WebLogic startup scripts are located in the *domain-name* directory where you specified your domain. By default, this directory is *`ORACLE_HOME`*`\user_projects\domain\`*`domain-name`*, where *`ORACLE_HOME`* is the directory you specified as the Oracle Home when you installed Oracle WebLogic Server, and *`domain-name`* is the name of the domain directory defined by the selected configuration template.

You need to modify some default Java values in these scripts to fit your environment and applications. The important performance tuning parameters in these files are the `JAVA_HOME` parameter and the Java heap size parameters:

- Change the value of the variable `JAVA_HOME` to the location of your `JDK`. For example:

  ```
  set JAVA_HOME=myjdk_location
  ```

  where *myjdk_location* is the path to your supported JDK for this release. See "Oracle Fusion Middleware Supported System Configurations."

- For higher performance throughput, set the minimum Java heap size equal to the maximum heap size. For example:

  ```
  "%JAVA_HOME%\bin\java" -server -Xms512m -Xmx512m -classpath %CLASSPATH% -
  ```

See Specifying Heap Size Values for details about setting heap size options.

Development vs. Production Mode Default Tuning Values

You can indicate whether a domain is to be used in a development environment or a production environment. WebLogic Server uses different default values for various services depending on the type of environment you specify. Specify the startup mode for your domain as shown in the following table.

Note: Oracle Java Cloud Service instances are created for production mode by default.

Table 6-1 Startup Modes

Choose this mode	when . . .
Development	You are creating your applications. In this mode, the configuration of security is relatively relaxed, allowing you to auto-deploy applications.
Production	Your application is running in its final form. In this mode, security is fully configured.

The following table lists the performance-related configuration parameters that differ when switching from development to production startup mode.

Table 6-2 Differences Between Development and Production Modes

Tuning Parameter	In development mode . . .	In production mode . . .
SSL	You can use the demonstration digital certificates and the demonstration keystores provided by the WebLogic Server security services. With these certificates, you can design your application to work within environments secured by SSL. For more information about managing security, see "Configuring SSL" in *Securing WebLogic Server*.	You should not use the demonstration digital certificates and the demonstration keystores. If you do so, a warning message is displayed.

Table 6-2 (Cont.) Differences Between Development and Production Modes

Tuning Parameter	In development mode . . .	In production mode . . .
Deploying Applications	WebLogic Server instances can automatically deploy and update applications that reside in the domain_name/autodeploy directory (where domain_name is the name of a domain). It is recommended that this method be used only in a single-server development environment. For more information, see "Auto-Deploying Applications in Development Domains" in *Deploying Applications to Oracle WebLogic Server*.	The auto-deployment feature is disabled, so you must use the WebLogic Server Administration Console, the weblogic.Deployer tool, or the WebLogic Scripting Tool (WLST). For more information, see "Understanding WebLogic Server Deployment" in *Deploying Applications to Oracle WebLogic Server*.
Web Services Test Client	Is enabled by default.	Is disabled (and undeployed), by default. See "Enabling and Disabling the Web Services Test Client" in *Administering Web Services*.

For information on switching the startup mode from development to production, see “Domain Modes” in *Administering Server Environments for Oracle WebLogic Server*.

Deployment

The following sections provide information on how to improve deployment performance:

- On-demand Deployment of Internal Applications
- Use FastSwap Deployment to Minimize Redeployment Time
- Generic Overrides

On-demand Deployment of Internal Applications

WebLogic Server deploys many internal applications during startup. Many of these internal applications are not needed by every user. You can configure WebLogic Server to wait and deploy these applications on the first access (on-demand) instead of always deploying them during server startup. This can conserve memory and CPU time during deployment as well as improving startup time and decreasing the base memory footprint for the server. For a development domain, the default is for WLS to deploy internal applications on-demand. For a production-mode domain, the default is for WLS to deploy internal applications as part of server startup. For more information on how to use and configure this feature, see "On-demand Deployment of Internal Applications" in *Deploying Applications to Oracle WebLogic Server*.

Use FastSwap Deployment to Minimize Redeployment Time

In deployment mode, you can set WebLogic Server to redefine Java classes in-place without reloading the ClassLoader. This means that you do not have to wait for an application to redeploy and then navigate back to wherever you were in the Web page flow. Instead, you can make your changes, auto compile, and then see the effects immediately. For more information on how to use and configure this feature, see "Using FastSwap Deployment to Minimize Redeployment"in *Deploying Applications to WebLogic Server*.

Generic Overrides

Generic overrides allow you to override application specific property files without having to crack a jar file by placing application specific files to be overridden into the `AppFileOverrides` optional subdirectory. For more information on how to use and configure this feature, see "Generic File Loading Overrides" in *Deploying Applications to WebLogic Server*.

Thread Management

WebLogic Server provides the following mechanisms to manage threads to perform work.

- Tuning a Work Manager
- Understanding the Differences Between Work Managers and Execute Queues
- Tuning the Stuck Thread Detection Behavior

Tuning a Work Manager

In this release, WebLogic Server allows you to configure how your application prioritizes the execution of its work. Based on rules you define and by monitoring actual runtime performance, WebLogic Server can optimize the performance of your application and maintain service level agreements (SLA).

You tune the thread utilization of a server instance by defining rules and constraints for your application by defining a Work Manger and applying it either globally to WebLogic Server domain or to a specific application component. The primary tuning considerations are:

- How Many Work Managers are Needed?
- What are the SLA Requirements for Each Work Manager?

See "Using Work Managers to Optimize Scheduled Work" in *Administering Server Environments for Oracle WebLogic Server*.

How Many Work Managers are Needed?

Each distinct SLA requirement needs a unique work manager.

What are the SLA Requirements for Each Work Manager?

Service level agreement (SLA) requirements are defined by instances of request classes. A request class expresses a scheduling guideline that a server instance uses to allocate threads. See "Understanding Work Managers" in *Administering Server Environments for Oracle WebLogic Server*.

Understanding the Differences Between Work Managers and Execute Queues

The easiest way to conceptually visualize the difference between the execute queues of previous releases with work managers is to correlate execute queues (or rather, execute-queue managers) with work managers and decouple the one-to-one relationship between execute queues and thread pools.

For releases prior to WebLogic Server 9.0, incoming requests are put into a default execute queue or a user-defined execute queue. Each execute queue has an associated execute queue manager that controls an exclusive, dedicated thread-pool with a fixed number of threads in it. Requests are added to the queue on a first-come-first-served basis. The execute-queue manager then picks the first request from the queue and an available thread from the associated thread pool and dispatches the request to be executed by that thread.

For releases of WebLogic Server 9.0 and higher, there is a single priority-based execute queue in the server. Incoming requests are assigned an internal priority based on the configuration of work managers you create to manage the work performed by your applications. The server increases or decreases threads available for the execute queue depending on the demand from the various work-managers. The position of a request in the execute queue is determined by its internal priority:

- The higher the priority, closer it is placed to the head of the execute queue.
- The closer to the head of the queue, more quickly the request will be dispatched a thread to use.

Work managers provide you the ability to better control thread utilization (server performance) than execute-queues, primarily due to the many ways that you can specify scheduling guidelines for the priority-based thread pool. These scheduling guidelines can be set either as numeric values or as the capacity of a server-managed resource, like a JDBC connection pool.

Tuning the Stuck Thread Detection Behavior

WebLogic Server automatically detects when a thread in an execute queue becomes "stuck." Because a stuck thread cannot complete its current work or accept new work, the server logs a message each time it diagnoses a stuck thread.

WebLogic Server diagnoses a thread as stuck if it is continually working (not idle) for a set period of time. You can tune a server's thread detection behavior by changing the length of time before a thread is diagnosed as stuck, and by changing the frequency with which the server checks for stuck threads. Although you can change the criteria WebLogic Server uses to determine whether a thread is stuck, you cannot change the default behavior of setting the "warning" and "critical" health states when all threads in a particular execute queue become stuck. For more information, see "Configuring WebLogic Server to Avoid Overload Conditions" in *Administering Server Environments for Oracle WebLogic Server*.

Tuning Network I/O

The following sections provide information on network communication between clients and servers (including T3 and IIOP protocols, and their secure versions):

- Tuning Muxers
- Network Channels

- Reducing the Potential for Denial of Service Attacks
- Tuning Connection Backlog Buffering
- Tuning Cached Connections

Tuning Muxers

WebLogic Server uses software modules called muxers to read incoming requests on the server and incoming responses on the client. WebLogic Server supports the following muxer types:

- Non-Blocking IO Muxer
- Other Muxers

Non-Blocking IO Muxer

WebLogic Server provides a non-blocking IO muxer implementation as the default muxer configuration. In the default configuration, `MuxerClass` is set to `weblogic.socket.NIOSocketMuxer.`

Other Muxers

Native Muxers and the Java Muxer are not recommended for most environments. If you must enable these muxers, the value of the `MuxerClass` attribute must be explicitly set:

- Solaris/HP-UX Native Muxer: `weblogic.socket.DevPollSocketMuxer`
- POSIX Native Muxer: `weblogic.socket.PosixSocketMuxer`
- Windows Native Muxer: `weblogic.socket.NTSocketMuxer`
- Java Muxer: `weblogic.socket.JavaSocketMuxer`

For example, switching to the native NT Socket Muxer on Windows platforms may improve performance for larger messages/payloads when there is one socket connection to the WebLogic Server instance.

```
-Dweblogic.MuxerClass=weblogic.socket.NTSocketMux
```

The POSIX Native Muxer provides similar performance improvements for larger messages/payloads in UNIX-like systems that support poll system calls, such as Solaris and HP-UX:

```
-Dweblogic.MuxerClass=weblogic.socket.PosixSocketMuxer
```

Native Muxers

Native muxers use platform-specific native binaries to read data from sockets. The majority of all platforms provide some mechanism to poll a socket for data. For example, Unix systems use the poll system call and the Windows architecture uses completion ports. Native muxers provide superior scalability because they implement a non-blocking thread model. When a native muxer is used, the server creates a fixed number of threads dedicated to reading incoming requests.

With native muxers, you may be able to improve throughput for some CPU-bound applications (for example, SpecJAppServer) by using the following:

```
-Dweblogic.socket.SocketMuxer.DELAY_POLL_WAKEUP=xx
```

where *xx* is the amount of time, in microseconds, to delay before checking if data is available. The default value is 0, which corresponds to no delay.

Java Muxer

A Java muxer has the following characteristics:

- Uses pure Java to read data from sockets.
- It is also the only muxer available for RMI clients.
- Blocks on reads until there is data to be read from a socket. This behavior does not scale well when there are a large number of sockets and/or when data arrives infrequently at sockets. This is typically not an issue for clients, but it can create a huge bottleneck for a server.

These characteristics may be acceptable if there are a small number of clients and the rate at which requests arrive at the server is fairly high. Under these conditions, the Java muxer performs as well as a native muxer and eliminates Java Native Interface (JNI) overhead. Unlike native muxers, the number of threads used to read requests is not fixed and is tunable for Java muxers by configuring the `Percent Socket Readers` parameter setting in the WebLogic Server Administration Console. Ideally, you should configure this parameter so the number of threads roughly equals the number of remote concurrently connected clients up to 50 percent of the total thread pool size. Each thread waits for a fixed amount of time for data to become available at a socket. If no data arrives, the thread moves to the next socket.

Network Channels

Network channels, also called network access points, allow you to specify different quality of service (QOS) parameters for network communication. Each network channel is associated with its own exclusive socket using a unique IP address and port. By default, T3 requests from a multi-threaded client are multiplexed over the same remote connection and the server instance reads requests from the socket one at a time. If the request size is large, this becomes a bottleneck.

Although the primary role of a network channel is to control the network traffic for a server instance, you can leverage the ability to create multiple custom channels to allow a multi-threaded client to communicate with server instance over multiple connections, reducing the potential for a bottleneck. To configure custom multi-channel communication, use the following steps:

1. Configure multiple network channels using different IP and port settings.

2. In your client-side code, use a JNDI URL pattern similar to the pattern used in clustered environments. The following is an example for a client using two network channels:

   ```
   t3://<ip1>:<port1>,<ip2>:<port2>
   ```

See "Understanding Network Channels" in *Administering Server Environments for Oracle WebLogic Server*.

Reducing the Potential for Denial of Service Attacks

To reduce the potential for Denial of Service (DoS) attacks while simultaneously optimizing system availability, WebLogic Server allows you to specify the following settings:

- Maximum incoming message size
- Complete message timeout
- Number of file descriptors (UNIX systems)

For optimal system performance, each of these settings should be appropriate for the particular system that hosts WebLogic Server and should be in balance with each other, as explained in the sections that follow.

Tuning Message Size

WebLogic Server allows you to specify a maximum incoming request size to prevent server from being bombarded by a series of large requests. You can set a global value or set specific values for different protocols and network channels. Although it does not directly impact performance, JMS applications that aggregate messages before sending to a destination may be refused if the aggregated size is greater than specified value. See Tuning Applications Using Unit-of-Order.

Tuning Complete Message Timeout

Make sure that the complete message timeout parameter is configured properly for your system. This parameter sets the maximum number of seconds that a server waits for a complete message to be received.

The default value is 60 seconds, which applies to all connection protocols for the default network channel. This setting might be appropriate if the server has a number of high-latency clients. However, you should tune this to the smallest possible value without compromising system availability.

If you need a complete message timeout setting for a specific protocol, you can alternatively configure a new network channel for that protocol.

Tuning Number of File Descriptors

On UNIX systems, each socket connection to WebLogic Server consumes a file descriptor. To optimize availability, the number of file descriptors for WebLogic Server should be appropriate for the host machine. By default, WebLogic Server configures 1024 file descriptors. However, this setting may be low, particularly for production systems.

Note that when you tune the number of file descriptors for WebLogic Server, your changes should be in balance with any changes made to the complete message timeout parameter. A higher complete message timeout setting results in a socket not closing until the message timeout occurs, which therefore results in a longer hold on the file descriptor. So if the complete message timeout setting is high, the file descriptor limit should also be set high. This balance provides optimal system availability with reduced potential for denial-of-service attacks.

For information about how to tune the number of available file descriptors, consult your UNIX vendor's documentation.

Tuning Connection Backlog Buffering

You can tune the number of connection requests that a WebLogic Server instance will accept before refusing additional requests. The `Accept Backlog` parameter specifies how many Transmission Control Protocol (TCP) connections can be buffered in a wait queue. This fixed-size queue is populated with requests for connections that the TCP stack has received, but the application has not accepted yet.

You can tune the number of connection requests that a WebLogic Server instance will accept before refusing additional requests.

Tuning Cached Connections

Use the `http.keepAliveCache.socketHealthCheckTimeout` system property for tuning the behavior of how a socket connection is returned from the cache when keep-alive is enabled when using HTTP 1.1 protocol. By default, the cache does not check the health condition before returning the cached connection to the client for use. Under some conditions, such as due to an unstable network connection, the system needs to check the connection's health condition before returning it to the client. To enable this behavior (checking the health condition), set `http.keepAliveCache.socketHealthCheckTimeout` to a value greater than 0.

Optimize Java Expressions

Set the optimize-java-expression element to optimize Java expressions to improve runtime performance. See `jsp-descriptor` in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server.*

Using WebLogic Server Clusters to Improve Performance

A WebLogic Server cluster is a group of WebLogic Servers instances that together provide fail-over and replicated services to support scalable high-availability operations for clients within a domain. A cluster appears to its clients as a single server but is in fact a group of servers acting as one to provide increased scalability and reliability.

A domain can include multiple WebLogic Server clusters and non-clustered WebLogic Server instances. Clustered WebLogic Server instances within a domain behave similarly to non-clustered instances, except that they provide failover and load balancing. The Administration Server for the domain manages all the configuration parameters for the clustered and non-clustered instances.

For more information about clusters, see "Understanding WebLogic Server Clustering" in *Administering Clusters for Oracle WebLogic Server*.

Scalability and High Availability

Scalability is the ability of a system to grow in one or more dimensions as more resources are added to the system. Typically, these dimensions include (among other things), the number of concurrent users that can be supported and the number of transactions that can be processed in a given unit of time.

Given a well-designed application, it is entirely possible to increase performance by simply adding more resources. To increase the load handling capabilities of WebLogic Server, add another WebLogic Server instance to your cluster—without changing your application. Clusters provide two key benefits that are not provided by a single server: scalability and availability.

WebLogic Server clusters bring scalability and high-availability to Java EE applications in a way that is transparent to application developers. Scalability expands the capacity of the middle tier beyond that of a single WebLogic Server or a single computer. The only limitation on cluster membership is that all WebLogic Servers must be able to communicate by IP multicast. New WebLogic Servers can be added to a cluster dynamically to increase capacity.

A WebLogic Server cluster guarantees high-availability by using the redundancy of multiple servers to insulate clients from failures. The same service can be provided on multiple servers in a cluster. If one server fails, another can take over. The ability to have a functioning server take over from a failed server increases the availability of the application to clients.

Note:

Provided that you have resolved all application and environment bottleneck issues, adding additional servers to a cluster should provide linear scalability. When doing benchmark or initial configuration test runs, isolate issues in a single server environment before moving to a clustered environment.

Clustering in the Messaging Service is provided through distributed destinations; connection concentrators, and connection load-balancing (determined by connection factory targeting); and clustered Store-and-Forward (SAF). Client load-balancing with respect to distributed destinations is tunable on connection factories. Distributed destination Message Driven Beans (MDBs) that are targeted to the same cluster that hosts the distributed destination automatically deploy only on cluster servers that host the distributed destination members and only process messages from their local destination. Distributed queue MDBs that are targeted to a different server or cluster than the host of the distributed destination automatically create consumers for every distributed destination member. For example, each running MDB has a consumer for each distributed destination queue member.

How to Ensure Scalability for WebLogic Clusters

In general, any operation that requires communication between the servers in a cluster is a potential scalability hindrance. The following sections provide information on issues that impact the ability to linearly scale clustered WebLogic servers:

- Database Bottlenecks
- Session Replication
- Asynchronous HTTP Session Replication
- Invalidation of Entity EJBs
- Invalidation of HTTP sessions
- JNDI Binding_ Unbinding and Rebinding

Database Bottlenecks

In many cases where a cluster of WebLogic servers fails to scale, the database is the bottleneck. In such situations, the only solutions are to tune the database or reduce load on the database by exploring other options. See DataBase Tuning and Tuning Data Sources.

Session Replication

User session data can be stored in two standard ways in a Java EE application: stateful session EJBs or HTTP sessions. By themselves, they are rarely a impact cluster scalability. However, when coupled with a session replication mechanism required to provide high-availability, bottlenecks are introduced. If a Java EE application has Web and EJB components, you should store user session data in HTTP sessions:

- HTTP session management provides more options for handling fail-over, such as replication, a shared DB or file.
- Superior scalability.
- Replication of the HTTP session state occurs outside of any transactions. Stateful session bean replication occurs in a transaction which is more resource intensive.
- The HTTP session replication mechanism is more sophisticated and provides optimizations a wider variety of situations than stateful session bean replication.

See Session Management.

Asynchronous HTTP Session Replication

Asynchronous replication of http sessions provides the option of choosing asynchronous session replication using:

- Asynchronous HTTP Session Replication using a Secondary Server
- Asynchronous HTTP Session Replication using a Database

Asynchronous HTTP Session Replication using a Secondary Server

Set the PersistentStoreType to async-replicated or async-replicated-if-clustered to specify asynchronous replication of data between a primary server and a secondary server. See session-descriptor section of Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server. To tune batched replication, adjust the SessionFlushThreshold parameter.

Replication behavior depends on cluster type. The following table describes how asynchronous replication occurs for a given cluster topology.

Table 6-3 Asynchronous Replication Behavior by Cluster Topology

Topology	Behavior
LAN	Replication to a secondary server within the same cluster occurs asynchronously with the "async-replication" setting in the webapp.
MAN	Replication to a secondary server in a remote cluster. This happens asynchronously with the "async-replication" setting in the webapp.
WAN	Replication to a secondary server within the cluster happens asynchronously with the "async-replication" setting in the webapp. Persistence to a database through a remote cluster happens asynchronously regardless of whether "async-replication" or "replication" is chosen.

The following section outlines asynchronous replication session behavior:

- During undeployment or redeployment:
  - The session is unregistered and removed from the update queue.
  - The session on the secondary server is unregistered.

- If the application is moved to admin mode, the sessions are flushed and replicated to the secondary server. If secondary server is down, the system attempts to failover to another server.
- A server shutdown or failure state triggers the replication of any batched sessions to minimize the potential loss of session information.

Asynchronous HTTP Session Replication using a Database

Set the PersistentStoreType to async-jdbc to specify asynchronous replication of data to a database. See session-descriptor section of Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server. To tune batched replication, adjust the SessionFlushThreshold and the SessionFlushInterval parameters.

The following section outlines asynchronous replication session behavior:

- During undeployment or redeployment:
  - The session is unregistered and removed from the update queue.
  - The session is removed from the database.
- If the application is moved to admin mode, the sessions are flushed and replicated to the database.

Invalidation of Entity EJBs

This applies to entity EJBs that use a concurrency strategy of `Optimistic` or `ReadOnly` with a read-write pattern.

`Optimistic`—When an `Optimistic` concurrency bean is updated, the EJB container sends a multicast message to other cluster members to invalidate their local copies of the bean. This is done to avoid optimistic concurrency exceptions being thrown by the other servers and hence the need to retry transactions. If updates to the EJBs are frequent, the work done by the servers to invalidate each other's local caches become a serious bottleneck. A flag called `cluster-invalidation-disabled` (default false) is used to turn off such invalidations. This is set in the `rdbms` descriptor file.

`ReadOnly` with a read-write pattern—In this pattern, persistent data that would otherwise be represented by a single EJB are actually represented by two EJBs: one read-only and the other updatable. When the state of the updateable bean changes, the container automatically invalidates corresponding read-only EJB instance. If updates to the EJBs are frequent, the work done by the servers to invalidate the read-only EJBs becomes a serious bottleneck.

Invalidation of HTTP sessions

Similar to Invalidation of Entity EJBs, HTTP sessions can also be invalidated. This is not as expensive as entity EJB invalidation, since only the session data stored in the secondary server needs to be invalidated. HTTP sessions should be invalidated if they are no longer in use.

JNDI Binding, Unbinding and Rebinding

In general, JNDI binds, unbinds and rebinds are expensive operations. However, these operations become a bigger bottleneck in clustered environments because JNDI tree changes have to be propagated to all members of a cluster. If such operations are performed too frequently, they can reduce cluster scalability significantly.

Improving Cluster Throughput using XA Transaction Cluster Affinity

XA transaction cluster affinity allows server instances that are participating in a global transactions to service related requests rather than load-balancing these requests to other member servers. When `Enable Transaction Affinity=true`, cluster throughput is increased by:

- Reducing inter-server transaction coordination traffic
- Improving resource utilization, such as reducing JDBC connections
- Simplifying asynchronous processing of transactions

See "XA Transaction Affinity" in *Administering Clusters for Oracle WebLogic Server*.

Monitoring a WebLogic Server Domain

The following sections provide information on how to monitor WebLogic Server domains:

- Using the Administration Console to Monitor WebLogic Server
- Using the WebLogic Diagnostic Framework
- Using JMX to Monitor WebLogic Server
- Using WLST to Monitor WebLogic Server

Using the Administration Console to Monitor WebLogic Server

The tool for monitoring the health and performance of your WebLogic Server domain is the Administration Console.

Using the WebLogic Diagnostic Framework

The WebLogic Diagnostic Framework (WLDF) is a monitoring and diagnostic framework that defines and implements a set of services that run within WebLogic Server processes and participate in the standard server life cycle. See "Overview of the WLDF Architecture" in *Configuring and Using the Diagnostics Framework for Oracle WebLogic Server*.

Using JMX to Monitor WebLogic Server

WebLogic Server provides its own set of MBeans that you can use to configure, monitor, and manage WebLogic Server resources. See "Understanding WebLogic Server MBeans" in *Developing Custom Management Utilities Using JMX for Oracle WebLogic Server*.

Using WLST to Monitor WebLogic Server

The WebLogic Scripting Tool (WLST) is a command-line scripting interface that system administrators and operators use to monitor and manage WebLogic Server instances and domains. See "Understanding WebLogic Server MBeans" in *Developing Custom Management Utilities Using JMX for Oracle WebLogic Server*.

Resources to Monitor WebLogic Server

The Oracle Technology Network at `http://www.oracle.com/technetwork/index.html` provides product downloads, articles, sample code, product documentation, tutorials, white papers, news groups, and other key content for WebLogic Server.

Tuning Class and Resource Loading

The default class and resource loading default behavior in WebLogic Server is to search the classloader hierarchy beginning with the root. As a result, the full system `classpath` is searched for every class or resource loading request, even if the class or resource belongs to the application. For classes and resources that are only looked up once (for example: classloading during deployment), the cost of the full `classpath` search is typically not a serious problem. For classes and resources that are requested repeatedly by an application at runtime (explicit application calls to `loadClass` or `getResource`) the CPU and memory overhead of repeatedly searching a long system and application `classpath` can be significant. The worst case scenario is when the requested class or resource is missing. A missing class or resource results in the cost of a full scan of the `classpath` and is compounded by the fact that if an application fails to find the class/resource it is likely to request it repeatedly. This problem is more common for resources than for classes.

Ideally, application code is optimized to avoid requests for missing classes and resources and frequent repeated calls to load the same class/resource. While it is not always possible to fix the application code (for example, a third party library), an alternative is to use WebLogic Server's Filtering Loader Mechanism.

Filtering Loader Mechanism

WebLogic Server provides a filtering loader mechanism that allows the system `classpath` search to be bypassed when looking for specific application classes and resources that are on the application `classpath`. This mechanism requires a user configuration that specifies the specific classes and resources that bypass the system `classpath` search. See "Using a Filtering Classloader" in *Developing Applications for Oracle WebLogic Server*.

New for this release is the ability to filter resource loading requests. The basic configuration of resource filtering is specified in `META-INF/weblogic-application.xml` file and is similar to the class filtering. The the syntax for filtering resources is shown in the following example:

```
<prefer-application-resources>
<resource-name>x/y</resource-name>
<resource-name>z*</resource-name>
</prefer-application-resources>
```

In this example, resource filtering has been configured for the exact resource name "x/y" and for any resource whose name starts with "z". '*' is the only wild card pattern allowed. Resources with names matching these patterns are searched for only on the application `classpath`, the system `classpath` search is skipped.

Note:

If you add a class or resource to the filtering configuration and subsequently get exceptions indicating the class or resource isn't found, the most likely cause is that the class or resource is on the system `classpath`, not on the application `classpath`.

Class Caching

WebLogic Server allows you to enable class caching for faster start ups. Once you enable caching, the server records all the classes loaded until a specific criterion is reached and persists the class definitions in an invisible file. When the server restarts, the cache is checked for validity with the existing code sources and the server uses the cache file to bulk load the same sequence of classes recorded in the previous run. If any change is made to the system classpath or its contents, the cache will be invalidated and re-built on server restart.

The advantages of using class caching are:

- Reduces server startup time.
- The package level index reduces search time for all classes and resources.

For more information, see "Configuring Class Caching" in *Developing Applications for Oracle WebLogic Server*.

Note:

Class caching is supported in development mode when starting the server using a `startWebLogic` script. Class caching is disabled by default and is not supported in production mode. The decrease in startup time varies among different JRE vendors.

SSL Considerations

If WebLogic Server is configured with JDK 7, you may find that the out-of-the-box SSL performance slower than in previous WebLogic Server releases. This performance change is due to the stronger cipher and MAC algorithm used by default when JDK 7 is used with the JSSE-based SSL provider in WebLogic Server. See "SSL Performance Considerations" in *Administering Security for Oracle WebLogic Server*.

7

Tuning the WebLogic Persistent Store

The persistent store provides a built-in, high-performance storage solution for WebLogic Server subsystems and services that require persistence. Tune the persistent store by following the procedures described in this chapter.

This chapter includes the following sections:

Before reading this chapter, Oracle recommends becoming familiar with WebLogic store administration and monitoring. See "Using the WebLogic Persistent Store" in *Administering Server Environments for Oracle WebLogic Server*.

Overview of Persistent Stores

The following sections provide information on using persistent stores.

Using the Default Persistent Store

Each server instance, including the administration server, has a default persistent store that requires no configuration. The default store is a file-based store that maintains its data in a group of files in a server instance's `data\store\default` directory. A directory for the default store is automatically created if one does not already exist. This default store is available to subsystems that do not require explicit selection of a particular store and function best by using the system's default storage mechanism. For example, a JMS Server with no persistent store configured will use the default store for its Managed Server and will support persistent messaging. See "Using the WebLogic Persistent Store" in *Administering Server Environments for Oracle WebLogic Server*.

Using Custom File Stores and JDBC Stores

In addition to using the default file store, you can also configure a file store or JDBC store to suit your specific needs. A custom file store, like the default file store, maintains its data in a group of files in a directory. However, you may want to create a custom file store so that the file store's data is persisted to a particular storage device. When configuring a file store directory, the directory must be accessible to the server instance on which the file store is located.

A JDBC store can be configured when a relational database is used for storage. A JDBC store enables you to store persistent messages in a standard JDBC-capable database, which is accessed through a designated JDBC data source. The data is stored in the JDBC store's database table, which has a logical name of `WLStore`. It is up to the database administrator to configure the database for high availability and performance. See:

- "When to Use a Custom Persistent Store" in *Administering Server Environments for Oracle WebLogic Server*.
- "Comparing File Stores and JDBC Stores" in *Administering Server Environments for Oracle WebLogic Server*.
- "Creating a Custom (User-Defined) File Store" in *Administering Server Environments for Oracle WebLogic Server*.
- "Creating a JDBC Store" in *Administering Server Environments for Oracle WebLogic Server*.

Using a JDBC TLOG Store

You can configure a JDBC TLOG store to persist transaction logs to a database, which allows you to leverage replication and HA characteristics of the underlying database, simplify disaster recovery, and improve Transaction Recovery service migration. See "Using a JDBC TLog Store" in *Administering Server Environments for Oracle WebLogic Server*.

Using JMS Paging Stores

Each JMS server implicitly creates a file based paging store. When the WebLogic Server JVM runs low on memory, this store is used to page non-persistent messages as well as persistent messages. Depending on the application, paging stores may generate heavy disk activity.

You can optionally change the directory location and the threshold settings at which paging begins. You can improve performance by locating paging store directories on a local file system, preferably in a temporary directory. Paging store files do not need to be backed up, replicated, or located in universally accessible location as they are automatically repopulated each time a JMS server restarts. See "JMS Server Behavior in WebLogic Server 9.x and Later" in *Administering JMS Resources for Oracle WebLogic Server*.

Note:

Paged persistent messages are potentially physical stored in two different places:

- Always in a recoverable default or custom store.
- Potentially in a paging directory.

Using Diagnostic Stores

The Diagnostics store is a file store that implicitly always uses the `Disabled` synchronous write policy. It is dedicated to storing WebLogic server diagnostics information. One diagnostic store is configured per WebLogic Server instance. See "Configuring Diagnostic Archives" in *Configuring and Using the Diagnostics Framework for Oracle WebLogic Server*.

Best Practices When Using Persistent Stores

- For subsystems that share the same server instance, share one store between multiple subsystems rather than using a store per subsystem. Sharing a store is more efficient for the following reasons:
  - A single store batches concurrent requests into single I/Os which reduces overall disk usage.
  - Transactions in which only one resource participates are lightweight one-phase transactions. Conversely, transactions in which multiple stores participate become heavier weight two-phase transactions.

  For example, configure all SAF agents and JMS servers that run on the same server instance so that they share the same store.
- Add a new store only when the old store(s) no longer scale.

Tuning JDBC Stores

The following section provides information on tuning JDBC Stores:

- Under heavy JDBC store I/O loads, you can improve performance by configuring a JDBC store to use multiple JDBC connections to concurrently process I/O operations. See "Enabling I/O Multithreading for JDBC Stores" in *Administering Server Environments for Oracle WebLogic Server*.
- The location of the JDBC store DDL that is used to initialize empty stores is now configurable. This simplifies the use of custom DDL for database table creation, which is sometimes used for database specific performance tuning. See "Using the WebLogic Persistent Store" in *Administering Server Environments for Oracle WebLogic Server*.

Tuning File Stores

The following section provides information on tuning File Stores:

- Basic Tuning Information

- Tuning a File Store Direct-Write-With-Cache Policy

Basic Tuning Information

The following section provides general information on tuning File Stores:

- For custom and default file stores, tune the Synchronous Write Policy.
  - There are three transactionally safe synchronous write policies: `Direct-Write-With-Cache`, `Direct-Write`, and `Cache-Flush`. `Direct-Write-With-Cache` is generally has the best performance of these policies, `Cache-Flush` generally has the lowest performance, and `Direct-Write` is the default. Unlike other policies, `Direct-Write-With-Cache` creates cache files in addition to primary files.
  - The `Disabled` synchronous write policy is transactionally unsafe. The `Disabled` write-policy can dramatically improve performance, especially at low client loads. However, it is unsafe because writes become asynchronous and data can be lost in the event of Operating System or power failure.
  - See "Guidelines for Configuring a Synchronous Write Policy" in *Administering the WebLogic Persistent Store*.

Note:

Certain older versions of Microsoft Windows may incorrectly report storage device synchronous write completion if the Windows default `Write Cache Enabled` setting is used. This violates the transactional semantics of transactional products (not specific to Oracle), including file stores configured with a `Direct-Write` (default) or `Direct-Write-With-Cache` policy, as a system crash or power failure can lead to a loss or a duplication of records/messages. One of the visible symptoms is that this problem may manifest itself in high persistent message/transaction throughput exceeding the physical capabilities of your storage device. You can address the problem by applying a Microsoft supplied patch, disabling the Windows `Write Cache Enabled` setting, or by using a power-protected storage device. See `http://support.microsoft.com/kb/281672` and `http://support.microsoft.com/kb/332023`.

- When performing head-to-head vendor comparisons, make sure all the write policies for the persistent store are equivalent. Some non-WebLogic vendors default to the equivalent of *`Disabled`*.
- If disk performance continues to be a bottleneck, consider purchasing disk or RAID controller hardware that has a built-in write-back cache. These caches significantly improve performance by temporarily storing persistent data in volatile memory. To ensure transactionally safe write-back caches, they must be protected against power outages, host machine failure, and operating system failure. Typically, such protection is provided by a battery-backed write-back cache.

Tuning a File Store Direct-Write-With-Cache Policy

The `Direct-Write-With-Cache` synchronous write policy is commonly the highest performance option that still provides transactionally safe disk writes. It is typically not as high-performing as the `Disabled` synchronous write policy, but the `Disabled` policy is not a safe option for production systems unless you have some means to prevent loss of buffered writes during a system failure.

`Direct-Write-With-Cache` file stores write synchronously to a primary set of files in the location defined by the `Directory` attribute of the file store configuration. They also asynchronously write to a corresponding temporary cache file in the location defined by the `CacheDirectory` attribute of the file store configuration. The cache directory and the primary file serve different purposes and require different locations. In many cases, primary files should be stored in remote storage for high availability, whereas cache files are strictly for performance and not for high availability and can be stored locally.

When the `Direct-Write-With-Cache` synchronous write policy is selected, there are several additional tuning options that you should consider:

- Setting the `CacheDirectory`. For performance reasons, the cache directory should be located on a local file system. It is placed in the operating system temp directory by default.
- Increasing the `MaxWindowBufferSize` and `IOBufferSize` attributes. These tune native memory usage of the file store.
- Increasing the `InitialSize` and `MaxFileSize` tuning attributes. These tune the initial size of a store, and the maximum file size of a particular file in the store respectively.

Additional Considerations

Consider the following when tuning the `Direct-Write-With-Cache` policy:

- There may be additional security and file locking considerations when using the `Direct-Write-With-Cache` synchronous write policy. See *Securing a Production Environment for Oracle WebLogic Server* and the `CacheDirectory` and `LockingEnabled` attributes of `JMSFileStoreMBean`.
  – The `JMSFileStoreMBean` is deprecated, but the individual bean attributes apply to the non-deprecated beans for custom and default file stores.
- It is safe to delete a cache directory while the store is not running, but this may slow down the next store boot. Cache files are re-used to speed up the file store boot and recovery process, but only if the store's host WebLogic server has been shut down cleanly prior to the current boot (not after `kill -9`, nor after an OS/JVM crash) and there was no off-line change to the primary files (such as a store admin compaction). If the existing cache files cannot be safely used at boot time, they are automatically discarded and new files are created. In addition, a `Warning` log 280102 is generated. After a migration or failover event, this same `Warning` message is generated, but can be ignored.
- If the a `Direct-Write-With-Cache` file store fails to load a `wlfileio` native driver, the synchronous write policy automatically changes to the equivalent of `Direct-Write` with `AvoidDirectIO=true`. To view a running custom or default file store's configured and actual synchronous write policy and driver, examine the server log for WL-280008 and WL-280009 messages.
- To prevent unused cache files from consuming disk space, test and development environments may need to be modified to periodically delete cache files that are left over from temporarily created domains. In production environments, cache files are managed automatically by the file store.

8

Database Tuning

You can tune your Oracle Database Cloud Service instance much as you would tune your on-premises database.

For information about tuning performance of your database, see Tuning Oracle Database Performance on Database as a Service in *Using Oracle Database Cloud - Database as a Service*.

For general information about good database tuning practices, see General Suggestions.

General Suggestions

This section provides general database tuning suggestions:

- Good database design — Distribute the database workload across multiple disks to avoid or reduce disk overloading. Good design also includes proper sizing and organization of tables, indexes, and logs.
- Disk I/O optimization — Disk I/O optimization is related directly to throughput and scalability. Access to even the fastest disk is orders of magnitude slower than memory access. Whenever possible, optimize the number of disk accesses. In general, selecting a larger block/buffer size for I/O reduces the number of disk accesses and might substantially increase throughput in a heavily loaded production environment.
- Checkpointing — This mechanism periodically flushes all dirty cache data to disk, which increases the I/O activity and system resource usage for the duration of the checkpoint. Although frequent checkpointing can increase the consistency of on-disk data, it can also slow database performance. Most database systems have checkpointing capability, but not all database systems provide user-level controls. Oracle, for example, allows administrators to set the frequency of checkpoints while users have no control over SQLServer 7.x checkpoints. For recommended settings, see the product documentation for the database you are using.
- Disk and database overhead can sometimes be dramatically reduced by batching multiple operations together and/or increasing the number of operations that run in parallel (increasing concurrency). Examples:
  - Increasing the value of the Message bridge `BatchSize` or the Store-and-Forward `WindowSize` can improve performance as larger batch sizes produce fewer but larger I/Os.
  - Programmatically leveraging JDBC's batch APIs.
  - Use the MDB transaction batching feature. See Tuning Message-Driven Beans.

  - Increasing concurrency by increasing `max-beans-in-free-pool` and thread pool size for MDBs (or decreasing it if batching can be leveraged).

9 Tuning WebLogic Server EJBs

Tune the WebLogic Server EJBs for your application environment by following the procedures described in this chapter.

This chapter includes the following sections:

General EJB Tuning Tips

- Deployment descriptors are schema-based. Descriptors that are new in this release of WebLogic Server are not available as DTD-based descriptors.
- Avoid using the `RequiresNew` transaction parameter. Using `RequiresNew` causes the EJB container to start a new transaction after suspending any current transactions. This means additional resources, including a separate data base connection are allocated.
- Use local-interfaces or set call-by-reference to true to avoid the overhead of serialization when one EJB calls another or an EJB is called by a servlet/JSP in the same application. Note the following:
  - In release prior to WebLogic Server 8.1, call-by-reference is turned on by default. For releases of WebLogic Server 8.1 and higher, call-by-reference is turned off by default. Older applications migrating to WebLogic Server 8.1 and higher that do not explicitly turn on call-by-reference may experience a drop in performance.
  - This optimization does not apply to calls across different applications.
- Use Stateless session beans over Stateful session beans whenever possible. Stateless session beans scale better than stateful session beans because there is no state information to be maintained.
- WebLogic Server provides additional transaction performance benefits for EJBs that reside in a WebLogic Server cluster. When a single transaction uses multiple EJBs, WebLogic Server attempts to use EJB instances from a single WebLogic Server instance, rather than using EJBs from different servers. This approach minimizes network traffic for the transaction. In some cases, a transaction can use EJBs that reside on multiple WebLogic Server instances in a cluster. This can occur

in heterogeneous clusters, where all EJBs have not been deployed to all WebLogic Server instances. In these cases, WebLogic Server uses a multitier connection to access the datastore, rather than multiple direct connections. This approach uses fewer resources, and yields better performance for the transaction. However, for best performance, the cluster should be homogeneous — all EJBs should reside on all available WebLogic Server instances.

Tuning EJB Caches

The following sections provide information on how to tune EJB caches:

- Tuning the Stateful Session Bean Cache
- Tuning the Entity Bean Cache
- Tuning the Query Cache

Tuning the Stateful Session Bean Cache

The EJB Container caches stateful session beans in memory up to a count specified by the `max-beans-in-cache` parameter specified in `weblogic-ejb-jar.xml`. This parameter should be set equal to the number of concurrent users. This ensures minimum passivation of stateful session beans to disk and subsequent activation from disk which yields better performance.

Tuning the Entity Bean Cache

Entity beans are cached at two levels by the EJB container:

- Transaction-Level Caching
- Caching between Transactions
- Ready Bean Caching

Transaction-Level Caching

Once an entity bean has been loaded from the database, it is always retrieved from the cache whenever it is requested when using the `findByPrimaryKey` or invoked from a cached reference in that transaction. Getting an entity bean using a non-primary key finder always retrieves the persistent state of the bean from the data base.

Caching between Transactions

Entity bean instances are also cached between transactions. However, by default, the persistent state of the entity beans are not cached between transactions. To enable caching between transactions, set the value of the `cache-between-transactions` parameter to true.

Is it safe to cache the state? This depends on the concurrency-strategy for that bean. The entity-bean cache is really only useful when `cache-between-transactions` can be safely set to true. In cases where `ejbActivate()` and `ejbPassivate()` callbacks are expensive, it is still a good idea to ensure the entity-cache size is large enough. Even though the persistent state may be reloaded at least once per transaction, the beans in the cache are already activated. The value of the cache-size is set by the deployment descriptor parameter `max-beans-in-cache` and should be set to maximize cache-hits. In most situations, the value need not be larger than the product of the number of rows in the table associated with the entity bean and the number of threads expected to access the bean concurrently.

Ready Bean Caching

For entity beans with a high cache miss ratio, maintaining ready bean instances can adversely affect performance.

If you can set `disable-ready-instances` in the `entity-cache element` of an `entity-descriptor`, the container does not maintain the ready instances in cache. If the feature is enabled in the deployment descriptor, the cache only keeps the active instances. Once the involved transaction is committed or rolled back, the bean instance is moved from active cache to the pool immediately.

Tuning the Query Cache

Query Caching is a new feature in WebLogic Server 9.0 that allows read-only CMP entity beans to cache the results of arbitrary finders. Query caching is supported for all finders except `prepared-query` finders. The query cache can be an application-level cache as well as a bean-level cache. The size of the cache is limited by the `weblogic-ejb-jar.xml` parameter `max-queries-in-cache`. The `finder-level` flag in the `weblogic-cmp-rdbms` descriptor file, `enable-query-caching` is used to specify whether the results of that finder are to be cached. A flag with the same name has the same purpose for internal relationship finders when applied to the `weblogic-relationship-role` element. Queries are evicted from the query-cache under the following circumstances:

- The query is least recently used and the `query-cache` has hit its size limit.
- At least one of the EJBs that satisfy the query has been evicted from the entity bean cache, regardless of the reason.
- The query corresponds to a finder that has `eager-relationship-caching` enabled and the query for the associated internal relationship finder has been evicted from the related bean's query cache.

It is possible to let the size of the entity-bean cache limit the size of the query-cache by setting the `max-queries-in-cache` parameter to 0, since queries are evicted from the cache when the corresponding EJB is evicted. This may avoid some lock contention in the query cache, but the performance gain may not be significant.

Tuning EJB Pools

The following section provides information on how to tune EJB pools:

- Tuning the Stateless Session Bean Pool
- Tuning the MDB Pool
- Tuning the Entity Bean Pool

Tuning the Stateless Session Bean Pool

The EJB container maintains a pool of stateless session beans to avoid creating and destroying instances. Though generally useful, this pooling is even more important for performance when the `ejbCreate()` and the `setSessionContext()` methods are expensive. The pool has a lower as well as an upper bound. The upper bound is the more important of the two.

- The upper bound is specified by the `max-beans-in-free-pool` parameter. It should be set equal to the number of threads expected to invoke the EJB concurrently. Using too small of a value impacts concurrency.
- The lower bound is specified by the `initial-beans-in-free-pool` parameter. Increasing the value of `initial-beans-in-free-pool` increases the time it takes to deploy the application containing the EJB and contributes to startup time for the server. The advantage is the cost of creating EJB instances is not incurred at run time. Setting this value too high wastes memory.

Tuning the MDB Pool

The life cycle of MDBs is very similar to stateless session beans. The MDB pool has the same tuning parameters as stateless session beans and the same factors apply when tuning them. In general, most users will find that the default values are adequate for most applications. See Tuning Message-Driven Beans.

Tuning the Entity Bean Pool

The entity bean pool serves two purposes:

- A target objects for invocation of finders via reflection.
- A pool of bean instances the container can recruit if it cannot find an instance for a particular primary key in the cache.

The entity pool contains anonymous instances (instances that do not have a primary key). These beans are not yet active (meaning `ejbActivate()` has not been invoked on them yet), though the EJB context has been set. Entity bean instances evicted from the entity cache are passivated and put into the pool. The tunables are the `initial-beans-in-free-pool` and `max-beans-in-free-pool`. Unlike stateless session beans and MDBs, the `max-beans-in-free-pool` has no relation with the thread count. You should increase the value of `max-beans-in-free-pool` if the entity bean constructor or `setEnityContext()` methods are expensive.

CMP Entity Bean Tuning

The largest performance gains in entity beans are achieved by using caching to minimize the number of interactions with the data base. However, in most situations, it is not realistic to be able to cache entity beans beyond the scope of a transaction. The following sections provide information on WebLogic Server EJB container features, most of which are configurable, that you can use to minimize database interaction safely:

- Use Eager Relationship Caching
- Use JDBC Batch Operations
- Tuned Updates
- Using Field Groups
- include-updates
- call-by-reference
- Bean-level Pessimistic Locking

- Concurrency Strategy

Use Eager Relationship Caching

Using eager relationship caching allows the EJB container to load related entity beans using a single SQL join. Use only when the same transaction accesses related beans. See "Relationship Caching" in *Developing Enterprise JavaBeans, Version 2.1, for Oracle WebLogic Server*.

In this release of WebLogic Server, if a CMR field has specified both `relationship-caching` and cascade-delete, the owner bean and related bean are loaded to SQL which can provide an additional performance benefit.

Using Inner Joins

The EJB container always uses an outer join in a CMP bean finder when eager `relationship-caching` is turned on. Typically, inner joins are faster to execute than outer joins with the drawback that inner joins do not return rows which do not have data in the corresponding joined table. Where applicable, using an inner join on very large databases may help to free CPU resources.

In WLS 10.3, `use-inner-join` has been added in `weblogic-cmp-rdbms-jar.xml`, as an attribute of the weblogic-rdbms-bean, as shown here:

```
<weblogic-rdbms-bean>

<ejb-name>exampleBean</ejb-name>

...

<use-inner-join>true</use-inner-join>

</weblogic-rdbms-bean>
```

This element should only be set to `true` if the CMP bean's related beans can never be null or an empty set.

The default value is `false`. If you specify its value as true, all relationship cache query on the entity bean use an inner join instead of a left outer join to execute a select query clause.

Use JDBC Batch Operations

JDBC batch operations are turned on by default in the EJB container. The EJB container automatically re-orders and executes similar data base operations in a single batch which increases performance by eliminating the number of data base round trips. Oracle recommends using batch operations.

Tuned Updates

When an entity EJB is updated, the EJB container automatically updates in the data base only those fields that have actually changed. As a result the update statements are simpler and if a bean has not been modified, no data base call is made. Because different transactions may modify different sets of fields, more than one form of update statements may be used to store the bean in the data base. It is important that you account for the types of update statements that may be used when setting the size of the prepared statement cache in the JDBC connection pool. See Cache Prepared and Callable Statements.

Using Field Groups

Field groups allow the user to segregate commonly used fields into a single group. If any of the fields in the group is accessed by application/bean code, the entire group is loaded using a single SQL statement. This group can also be associated with a finder. When the finder is invoked and `finders-load-bean` is true, it loads only those fields from the data base that are included in the field group. This means that if most transactions do not use a particular field that is slow to load, such as a BLOB, it can be excluded from a field-group. Similarly, if an entity bean has a lot of fields, but a transaction uses only a small number of them, the unused fields can be excluded.

Note:

Be careful to ensure that fields that are accessed in the same transaction are not configured into separate field-groups. If that happens, multiple data base calls occur to load the same bean, when one would have been enough.

include-updates

This flag causes the EJB container to flush all modified entity beans to the data base before executing a finder. If the application modifies the same entity bean more than once and executes a non-pk finder in-between in the same transaction, multiple updates to the data base are issued. This flag is turned on by default to comply with the EJB specification.

If the application has transactions where two invocations of the same or different finders could return the same bean instance and that bean instance could have been modified between the finder invocations, it makes sense leaving `include-updates` turned on. If not, this flag may be safely turned off. This eliminates an unnecessary flush to the data base if the bean is modified again after executing the second finder. This flag is specified for each finder in the `cmp-rdbms` descriptor.

call-by-reference

When it is turned off, method parameters to an EJB are passed by value, which involves serialization. For mutable, complex types, this can be significantly expensive. Consider using for better performance when:

- The application does not require call-by-value semantics, such as method parameters are not modified by the EJB.

or

- If modified by the EJB, the changes need not be invisible to the caller of the method.

This flag applies to all EJBs, not just entity EJBs. It also applies to EJB invocations between servlets/JSPs and EJBs in the same application. The flag is turned off by default to comply with the EJB specification. This flag is specified at the bean-level in the WebLogic-specific deployment descriptor.

Bean-level Pessimistic Locking

Bean-level pessimistic locking is implemented in the EJB container by acquiring a data base lock when loading the bean. When implemented, each entity bean can only be accessed by a single transaction in a single server at a time. All other transactions are blocked, waiting for the owning transaction to complete. This is a useful alternative to using a higher data base isolation level, which can be expensive at the RDBMS level. This flag is specified at the bean level in the `cmp-rdbms` deployment descriptor.

Note:

If the lock is not exclusive lock, you man encounter deadlock conditions. If the data base lock is a shared lock, there is potential for deadlocks when using that RDBMS.

Concurrency Strategy

The `concurrency-strategy` deployment descriptor tells the EJB container how to handle concurrent access of the same entity bean by multiple threads in the same server instance. Set this parameter to one of four values:

- `Exclusive`—The EJB container ensures there is only one instance of an EJB for a given primary key and this instance is shared among all concurrent transactions in the server with the container serializing access to it. This concurrency setting generally does not provide good performance unless the EJB is used infrequently and chances of concurrent access is small.
- `Database`—This is the default value and most commonly used concurrency strategy. The EJB container defers concurrency control to the database. The container maintains multiple instances of an EJB for a given primary-key and each transaction gets it's own copy. In combination with this strategy, the database isolation-level and bean level pessimistic locking play a major role in determining if concurrent access to the persistent state should be allowed. It is possible for multiple transactions to access the bean concurrently so long as it does not need to go to the database, as would happen when the value of `cache-between-transactions` is true. However, setting the value of `cache-between-transactions` to true unsafe and not recommended with the `Dababase` concurrency strategy.
- `Optimistic`—The goal of the optimistic concurrency strategy is to minimize locking at the data base and while continuing to provide data consistency. The basic assumption is that the persistent state of the EJB is changed very rarely. The container attempts to load the bean in a nested transaction so that the isolation-level settings of the outer transaction does not cause locks to be acquired at the data base. At commit-time, if the bean has been modified, a predicated update is used to ensure it's persistent state has not been changed by some other transaction. If so, an `OptimisticConcurrencyException` is thrown and must be handled by the application.

  Since EJBs that can use this concurrency strategy are rarely modified, using `cache-between-transactions` on can boost performance significantly. This strategy also allows commit-time verification of beans that have been read, but not changed. This is done by setting the `verify-rows` parameter to `Read` in the `cmp-rdbms` descriptor. This provides very high data-consistency while at the same time minimizing locks at the data base. However, it does slow performance somewhat. It is recommended that the optimistic verification be performed using a version column: it is faster, followed closely by timestamp, and more distantly by modified and read. The modified value does not apply if verify-rows is set to `Read`.

  When an optimistic concurrency bean is modified in a server that is part of a cluster, the server attempts to invalidate all instances of that bean cluster-wide in the expectation that it will prevent `OptimisticConcurrencyExceptions`. In some cases, it may be more cost effective to simply let other servers throw an `OptimisticConcurrencyException`. in this case, turn off the cluster-wide invalidation by setting the `cluster-invalidation-disabled` flag in the `cmp-rdbms` descriptor.

- `ReadOnly`—The ReadOnly value is the most performant. When selected, the container assumes the EJB is non-transactional and automatically turns on `cache-between-transactions`. Bean states are updated from the data base at periodic, configurable intervals or when the bean has been programmatically invalidated. The interval between updates can cause the persistent state of the bean to become stale. This is the only concurrency-strategy for which `query-caching` can be used. See Caching between Transactions.

Tuning In Response to Monitoring Statistics

The WebLogic Server Administration Console reports a wide variety of EJB runtime monitoring statistics, many of which are useful for tuning your EJBs. This section discusses how some of these statistics can help you tune the performance of EJBs.

Cache Miss Ratio

The cache miss ratio is a ratio of the number of times a container cannot find a bean in the cache (cache miss) to the number of times it attempts to find a bean in the cache (cache access):

```
Cache Miss Ratio = (Cache Total Miss Count / Cache Total Access Count) * 100
```

A high cache miss ratio could be indicative of an improperly sized cache. If your application uses a certain subset of beans (read primary keys) more frequently than others, it would be ideal to size your cache large enough so that the commonly used beans can remain in the cache as less commonly used beans are cycled in and out upon demand. If this is the nature of your application, you may be able to decrease your cache miss ratio significantly by increasing the maximum size of your cache.

If your application doesn't necessarily use a subset of beans more frequently than others, increasing your maximum cache size may not affect your cache miss ratio. We recommend testing your application with different maximum cache sizes to determine which give the lowest cache miss ratio. It is also important to keep in mind that your server has a finite amount of memory and therefore there is always a trade-off to increasing your cache size.

Lock Waiter Ratio

When using the `Exclusive` concurrency strategy, the lock waiter ratio is the ratio of the number of times a thread had to wait to obtain a lock on a bean to the total amount of lock requests issued:

```
Lock Waiter Ratio = (Current Waiter Count / Current Lock Entry Count) * 100
```

A high lock waiter ratio can indicate a suboptimal concurrency strategy for the bean. If acceptable for your application, a concurrency strategy of Database or Optimistic will allow for more parallelism than an Exclusive strategy and remove the need for locking at the EJB container level.

Because locks are generally held for the duration of a transaction, reducing the duration of your transactions will free up beans more quickly and may help reduce your lock waiter ratio. To reduce transaction duration, avoid grouping large amounts of work into a single transaction unless absolutely necessary.

Lock Timeout Ratio

When using the `Exclusive` concurrency strategy, the lock timeout ratio is the ratio of timeouts to accesses for the lock manager:

```
Lock Timeout Ratio =(Lock Manager Timeout Total Count / Lock Manager Total Access
Count) * 100
```

The lock timeout ratio is closely related to the lock waiter ratio. If you are concerned about the lock timeout ratio for your bean, first take a look at the lock waiter ratio and our recommendations for reducing it (including possibly changing your concurrency strategy). If you can reduce or eliminate the number of times a thread has to wait for a lock on a bean, you will also reduce or eliminate the amount of timeouts that occur while waiting.

A high lock timeout ratio may also be indicative of an improper transaction timeout value. The maximum amount of time a thread will wait for a lock is equal to the current transaction timeout value.

If the transaction timeout value is set too low, threads may not be waiting long enough to obtain access to a bean and timing out prematurely. If this is the case, increasing the trans-timeout-seconds value for the bean may help reduce the lock timeout ratio.

Take care when increasing the trans-timeout-seconds, however, because doing so can cause threads to wait longer for a bean and threads are a valuable server resource. Also, doing so may increase the request time, as a request ma wait longer before timing out.

Pool Miss Ratio

The pool miss ratio is a ratio of the number of times a request was made to get a bean from the pool when no beans were available, to the total number of requests for a bean made to the pool:

```
Pool Miss Ratio = (Pool Total Miss Count / Pool Total Access Count) * 100
```

If your pool miss ratio is high, you must determine what is happening to your bean instances. There are three things that can happen to your beans.

- They are in use.
- They were destroyed.
- They were removed.

Follow these steps to diagnose the problem:

1. Check your destroyed bean ratio to verify that bean instances are not being destroyed.
2. Investigate the cause and try to remedy the situation.
3. Examine the demand for the EJB, perhaps over a period of time.

One way to check this is via the Beans in Use Current Count and Idle Beans Count displayed in the WebLogic Server Administration Console. If demand for your EJB spikes during a certain period of time, you may see a lot of pool misses as your pool is emptied and unable to fill additional requests.

As the demand for the EJB drops and beans are returned to the pool, many of the beans created to satisfy requests may be unable to fit in the pool and are therefore removed. If this is the case, you may be able to reduce the number of pool misses by increasing the maximum size of your free pool. This may allow beans that were created to satisfy demand during peak periods to remain in the pool so they can be used again when demand once again increases.

Destroyed Bean Ratio

The destroyed bean ratio is a ratio of the number of beans destroyed to the total number of requests for a bean.

```
Destroyed Bean Ratio = (Total Destroyed Count / Total Access Count) * 100
```

To reduce the number of destroyed beans, Oracle recommends against throwing non-application exceptions from your bean code except in cases where you want the bean instance to be destroyed. A non-application exception is an exception that is either a java.rmi.RemoteException (including exceptions that inherit from RemoteException) or is not defined in the throws clause of a method of an EJB's home or component interface.

In general, you should investigate which exceptions are causing your beans to be destroyed as they may be hurting performance and may indicate problem with the EJB or a resource used by the EJB.

Pool Timeout Ratio

The pool timeout ratio is a ratio of requests that have timed out waiting for a bean from the pool to the total number of requests made:

```
Pool Timeout Ratio = (Pool Total Timeout Count / Pool Total Access Count) * 100
```

A high pool timeout ratio could be indicative of an improperly sized free pool. Increasing the maximum size of your free pool via the `max-beans-in-free-pool` setting will increase the number of bean instances available to service requests and may reduce your pool timeout ratio.

Another factor affecting the number of pool timeouts is the configured transaction timeout for your bean. The maximum amount of time a thread will wait for a bean from the pool is equal to the default transaction timeout for the bean. Increasing the `trans-timeout-seconds` setting in your `weblogic-ejb-jar.xml` file will give threads more time to wait for a bean instance to become available.

Users should exercise caution when increasing this value, however, since doing so may cause threads to wait longer for a bean and threads are a valuable server resource. Also, request time might increase because a request will wait longer before timing out.

Transaction Rollback Ratio

The transaction rollback ratio is the ratio of transactions that have rolled back to the number of total transactions involving the EJB:

```
Transaction Rollback Ratio = (Transaction Total Rollback Count / Transaction Total
Count) * 100
```

Begin investigating a high transaction rollback ratio by examining the Transaction Timeout Ratio reported in the WebLogic Server Administration Console. If the transaction timeout ratio is higher than you expect, try to address the timeout problem first.

An unexpectedly high transaction rollback ratio could be caused by a number of things. We recommend investigating the cause of transaction rollbacks to find potential problems with your application or a resource used by your application.

Transaction Timeout Ratio

The transaction timeout ratio is the ratio of transactions that have timed out to the total number of transactions involving an EJB:

```
Transaction Timeout Ratio = (Transaction Total Timeout Count / Transaction Total
Count) * 100
```

A high transaction timeout ratio could be caused by the wrong transaction timeout value. For example, if your transaction timeout is set too low, you may be timing out transactions before the thread is able to complete the necessary work. Increasing your transaction timeout value may reduce the number of transaction timeouts.

You should exercise caution when increasing this value, however, since doing so can cause threads to wait longer for a resource before timing out. Also, request time might increase because a request will wait longer before timing out.

A high transaction timeout ratio could be caused by a number of things such as a bottleneck for a server resource. We recommend tracing through your transactions to investigate what is causing the timeouts so the problem can be addressed.

10 Tuning Message-Driven Beans

Use the tuning and best practice information for Message-Driven Beans (MDBs), as described in this chapter.

This chapter includes the following sections:

- Use Transaction Batching
- MDB Thread Management
- Best Practices for Configuring and Deploying MDBs Using Distributed Topics
- Using MDBs with Foreign Destinations
- Token-based Message Polling for Transactional MDBs Listening on Queues/Topics
- Compatibility for WLS 10.0 and Earlier-style Polling

Use Transaction Batching

MDB transaction batching allows several JMS messages to be processed in one container managed transaction. Batching amortizes the cost of transactions over multiple messages and when used appropriately, can reduce or even eliminate the throughput difference between 2PC and 1PC processing. See "Transaction Batching of MDBs" in *Developing Message-Driven Beans for Oracle WebLogic Server*.

- Using batching may require reducing the number of concurrent MDB instances. If too many MDB instances are available, messages may be processed in parallel rather than in a batch. See MDB Thread Management.
- While batching generally increases throughput, it may also increase latency (the time it takes for an individual message to complete its MDB processing).

MDB Thread Management

Thread management for MDBs is described in terms of concurrency—the number of MDB instances that can be active at the same time. The following sections provide information on MDB concurrency:

- Determining the Number of Concurrent MDBs
- Selecting a Concurrency Strategy
- Thread Utilization When Using WebLogic Destinations
- Limitations for Multi-threaded Topic MDBs

Determining the Number of Concurrent MDBs

Table 10-1 provides information on how to determine the number of concurrently running MDB instances for a server instance.

Table 10-1 Determining Concurrency for WebLogic Server MDBs

Type of work manager or execute queue	Threads
Default work manager or unconstrained work manager	varies due to self-tuning, up to Min(`max-beans-in-free-pool,16`)
Default work manager with self-tuning disabled	Min(`default-thread-pool-size`/2+1, `max-beans-in-free-pool`) This is also the default thread pool concurrency algorithm for WebLogic Server 8.1
Custom execute queue	Min(`execute-queue-size`, `max-beans-in-free-pool`)
Custom work manager with constraint	varies due to self-tuning, between `min-thread-constraint` and Min(`max-threads-constraint`, `max-beans-in-free-pool`)

Transactional WebLogic MDBs use a synchronous polling mechanism to retrieve messages from JMS destinations if they are either: A) listening to non-WebLogic queues; or B) listening to a WebLogic queue and transaction batching is enabled. See Token-based Message Polling for Transactional MDBs Listening on Queues/Topics.

Selecting a Concurrency Strategy

The following section provides general information on selecting a concurrency strategy for your applications:

Note:

Every application is unique, select a concurrency strategy based on how your application performs in its environment.

- In most situations, if the message stream has bursts of messages, using an unconstrained work manager with a high fair share is adequate. Once the messages in a burst are handled, the threads are returned to the self-tuning pool.
- In most situations, if the message arrival rate is high and constant or if low latency is required, it makes sense to reserve threads for MDBs. You can reserve threads by either specifying a work manager with a `min-threads-constraint` or by using a custom execute queue.
- If you migrate WebLogic Server 8.1 applications that have custom MDB execute queues, you can convert the MDB execute queue to a custom work manager that has a configured `max-threads-constraint` parameter and a high fair share setting.

Note:

You must configure the `max-threads-constraint` parameter to override the default concurrency of 16.

- In WebLogic Server 8.1, you could increase the size of the default execute queue knowing that a larger default pool means a larger maximum MDB concurrency. Default thread pool MDBs upgraded to WebLogic Server 9.0 will have a fixed maximum of 16. To achieve MDB concurrency numbers higher than 16, you will need to create a custom work manager or custom execute queue. See Table 10-1.

Thread Utilization When Using WebLogic Destinations

The following section provides information on how threads are allocated when WebLogic Server interoperates with WebLogic destinations.

- Non-transactional WebLogic MDBs allocate threads from the thread-pool designated by the `dispatch-policy` as needed when there are new messages to be processed. If the MDB has successfully connected to its source destination, but there are no messages to be processed, then the MDB will use no threads.

- Transactional WebLogic MDBs with transaction batching *disabled* work the same as non-transactional MDBs except for Topic MDBs with a Topic Messages Distribution Mode of `Compatibility` (the default), in which case the MDB always limits the thread pool size to 1.

- The behavior of transactional MDBs with transaction batching *enabled* depends on whether the MDB is listening on a topic or a queue:

  – *MDBs listening on topics*: — Each deployed MDB uses a dedicated daemon polling thread that is created in Non-Pooled Threads thread group.

    ◆ Topic Messages Distribution Mode = Compatibility: Each deployed MDB uses a dedicated daemon polling thread that is created in the Non-Pooled Threads thread group.

    ◆ Topic Messages Distribution Mode = One-Copy-Per-Server or One-Copy-Per-Application: Same as queues.

  – *MDBs listening on queues* — Instead of a dedicated thread, each deployed MDB uses a token-based, synchronous polling mechanism that always uses at least one thread from the `dispatch-policy`. See Token-based Message Polling for Transactional MDBs Listening on Queues/Topics.

For information on how threads are allocated when WebLogic Server interoperates with MDBs that consume from Foreign destinations, see Thread Utilization for MDBs that Process Messages from Foreign Destinations.

Limitations for Multi-threaded Topic MDBs

When the `topicMessagesDistributionMode` is `Compatibility`, the default behavior for non-transactional topic MDBs is to multi-thread the message processing. In this situation, the MDB container fails to provide reproducible behavior when the topic is not a WebLogic JMS Topic, such as unexpected exceptions and acknowledgement of messages that have not yet been processed. For example, if an application throws a `RuntimeException` from `onmessage`, the container may still acknowledge the message. Oracle recommends setting `max-beans-in-free-pool` to a value of 1 in the deployment descriptor to prevent multi-threading in topic MDBs when the topic is a foreign vendor topic (not a WebLogic JMS topic).

Caution:

Non-transactional Foreign Topics: Oracle recommends explicitly setting `max-beans-in-free-pool` to `1` for non-transactional MDBs that work with foreign (non-WebLogic) topics. Failure to do so may result in lost messages in the event of certain failures, such as the MDB application throwing `Runtime` or `Error` exceptions.

Unit-of-Order: Oracle recommends explicitly setting `max-beans-in-free-pool` to `1` for non-transactional `Compatibility` mode MDBs that consume from a WebLogic JMS topic and process messages that have a WebLogic JMS Unit-of-Order value. Unit-of-Order messages in this use case may not be processed in order unless `max-beans-in-free-pool` is set to `1`.

Transactional MDBs automatically force concurrency to 1 regardless of the max-beans-in-free-pool setting.

Best Practices for Configuring and Deploying MDBs Using Distributed Topics

Message-driven beans provide a number of application design and deployment options that offer scalability and high availability when using distributed topics. For more detailed information, see "Configuring and Deploying MDBs Using Distributed Topics" in *Developing Message-Driven Beans for Oracle WebLogic Server*.

Using MDBs with Foreign Destinations

Note:

The term "Foreign destination" in this context refers to destinations that are hosted by a non-WebLogic JMS provider. It does not refer to remote WebLogic destinations.

The following sections provide information on the behavior of WebLogic Server when using MDBs that consume messages from Foreign destinations:

- Concurrency for MDBs that Process Messages from Foreign Destinations
- Thread Utilization for MDBs that Process Messages from Foreign Destinations

Concurrency for MDBs that Process Messages from Foreign Destinations

The concurrency of MDBs that consume from destinations hosted by foreign providers (non-WebLogic JMS destinations) is determined using the same algorithm that is used for WebLogic JMS destinations.

Thread Utilization for MDBs that Process Messages from Foreign Destinations

The following section provides information on how threads are allocated when WebLogic Server interoperates with MDBs that process messages from foreign destinations:

- Non-transactional MDBs use a foreign vendor's thread, not a WebLogic Server thread. In this situation, the `dispatch-policy` is ignored except for determining concurrency.
- Transactional MDBs run in WebLogic Server threads, as follow:
  - *MDBs listening on topics* — Each deployed MDB uses a dedicated daemon polling thread that is created in Non-Pooled Threads thread group.
  - *MDBs listening on queues* — Instead of a dedicated thread, each deployed MDB uses a token-based, synchronous polling mechanism that always uses at least one thread from the `dispatch-policy`. See Token-based Message Polling for Transactional MDBs Listening on Queues/Topics

Token-based Message Polling for Transactional MDBs Listening on Queues/Topics

Transactional WebLogic MDBs use a synchronous polling mechanism to retrieve messages from JMS destinations if they are:

- Listening to non-WebLogic queues
- Listening to a WebLogic queue and transaction batching is enabled
- Listening to a WebLogic Topic where:
  - Topic Messages Distribution Mode = One-Copy-Per-Server and transaction batching is enabled
  - Topic Messages Distribution Mode = One-Copy-Per-Application and transaction batching is enabled

With synchronous polling, one or more WebLogic polling threads synchronously receive messages from the MDB's source destination and then invoke the MDB application's `onMessage` callback.

As of WebLogic 10.3, the polling mechanism changed to a token-based approach to provide better control of the concurrent poller thread count under changing message loads. In previous releases, the thread count ramp-up could be too gradual in certain use cases. Additionally, child pollers, once awoken, could not be ramped down and returned back to the pool for certain foreign JMS providers.

When a thread is returned to the thread pool with token-based polling, the thread's internal JMS consumer is closed rather than cached. This assures that messages will not be implicitly pre-fetched by certain foreign JMS Providers while there is no polling thread servicing the consumer.

In addition, each MDB maintains a single token that provides permission for a given poller thread to create another thread.

- *On receipt of a message* — A poller thread that already has the token or that is able to acquire the token because the token is not owned, wakes up an additional poller thread and gives the token to the new poller if the maximum concurrency has not yet been reached. If maximum concurrency has been reached, the poller thread simply releases the token (leaving it available to any other poller).

- *On finding an empty queue/Topic* — A poller tries to acquire the token and if successful will try to poll the queue periodically. If it fails to acquire the token, it returns itself back to the pool. This ensures that with an empty queue or topic, there is still at least one poller checking for messages.

Compatibility for WLS 10.0 and Earlier-style Polling

In WLS 10.0 and earlier, transactional MDBs with batching enabled created a dedicated polling thread for each deployed MDB. This polling thread was not allocated from the pool specified by `dispatch-policy`, it was an entirely new thread in addition to the all other threads running on the system. See Use Transaction Batching.

To override the token-based polling behavior and implement the WLS 10.0 and earlier behavior, you can either:

- At the server level, set the `weblogic.mdb.message.81StylePolling` system property to `True` to override the token-based polling behavior.
- At the MDB level, set the `use81-style-polling` element under `message-driven-descriptor` to override the token-based polling behavior. When using foreign transactional MDBs with the WLS 8.1-style polling flag, some foreign vendors require a permanently allocated thread per concurrent MDB instance. These threads are drawn from the pool specified by `dispatch-policy` and are not returned to the pool until the MDB is undeployed. Since these threads are not shared, the MDB can starve other resources in the same pool. In this situation, you may need to increase the number of threads in the pool. With the token-based polling approach for such foreign vendors, the thread's internal JMS message consumer is closed rather than cached to assure that messages will not be reserved by the destination for the specific consumer.

11 Tuning Data Sources

To get the best performance from your WebLogic data sources, use the tuning tips provided in this chapter.

This chapter includes the following sections:

Tune the Number of Database Connections

A straightforward and easy way to boost performance of a data source in WebLogic Server applications is to set the value of `Initial Capacity` equal to the value for `Maximum Capacity` when configuring connection pools in your data source.

Creating a database connection is a relatively expensive process in any environment. Typically, a connection pool starts with a small number of connections. As client demand for more connections grow, there may not be enough in the pool to satisfy the requests. WebLogic Server creates additional connections and adds them to the pool until the maximum pool size is reached.

One way to avoid connection creation delays for clients using the server is to initialize all connections at server startup, rather than on-demand as clients need them. Set the initial number of connections equal to the maximum number of connections in the Connection Pool tab of your data source configuration. You will still need to determine the optimal value for the `Maximum Capacity` as part of your pre-production performance testing.

For more tuning information, see "Tuning Data Source Connection Pool Options" in *Administering JDBC Data Sources for Oracle WebLogic Server*.

Waste Not

Another simple way to boost performance is to avoid wasting resources. Here are some situations where you can avoid wasting JDBC related resources:

- JNDI lookups are relatively expensive, so caching an object that required a looked-up in client code or application code avoids incurring this performance hit more than once.
- Once client or application code has a connection, maximize the reuse of this connection rather than closing and reacquiring a new connection. While acquiring and returning an existing creation is much less expensive than creating a new one, excessive acquisitions and returns to pools creates contention in the connection pool and degrades application performance.
- Don't hold connections any longer than is necessary to achieve the work needed. Getting a connection once, completing all necessary work, and returning it as soon as possible provides the best balance for overall performance.

Use Test Connections on Reserve with Care

When `Test Connections on Reserve` is enabled, the server instance checks a database connection prior to returning the connection to a client. This helps reduce the risk of passing invalid connections to clients.

However, it is a fairly expensive operation. Typically, a server instance performs the test by executing a full-fledged SQL query with each connection prior to returning it. If the SQL query fails, the connection is destroyed and a new one is created in its place. A new and optional performance tunable has been provided in WebLogic Server 9.x within this "test connection on reserve" feature. The new optional performance tunable in 9.x allows WebLogic Server to skip this SQL-query test within a configured time window of a prior successful client use (default is 10 seconds). When a connection is returned to the pool by a client, the connection is timestamped by WebLogic Server. WebLogic Server will then skip the SQL-query test if this particular connection is returned to a client within the time window. Once the time window expires, WebLogic Server will execute the SQL-query test. This feature can provide significant performance boosts for busy systems using "test connection on reserve".

Cache Prepared and Callable Statements

When you use a prepared statement or callable statement in an application or EJB, there is considerable processing overhead for the communication between the application server and the database server and on the database server itself. To minimize the processing costs, WebLogic Server can cache prepared and callable statements used in your applications. When an application or EJB calls any of the statements stored in the cache, WebLogic Server reuses the statement stored in the cache. Reusing prepared and callable statements reduces CPU usage on the database server, improving performance for the current statement and leaving CPU cycles for other tasks. For more details, see "Increasing Performance with the Statement Cache" in *Administering JDBC Data Sources for Oracle WebLogic Server*.

Using the statement cache can dramatically increase performance, but you must consider its limitations before you decide to use it. For more details, see "Usage Restrictions for the Statement Cache" in *Administering JDBC Data Sources for Oracle WebLogic Server*.

Using Pinned-To-Thread Property to Increase Performance

To minimize the time it takes for an application to reserve a database connection from a data source and to eliminate contention between threads for a database connection, you can add the `Pinned-To-Thread` property in the connection Properties list for the data source, and set its value to `true`.

In this release, the Pinned-To-Thread feature does not work with multi data sources, Oracle RAC, and IdentityPool. These features rely on the ability to return a connection to the connection pool and reacquire it if there is a connection failure or connection identity does not match

Database Listener Timeout under Heavy Server Loads

In some situations where WebLogic Server is under heavy loads (high CPU utilization), the database listener may timeout and throw an exception while creating a new connection. To workaround this issue, increase the listener timeout on the database server. The following example is for an Oracle driver and database:

- The exception thrown is a `ResourceDeadException` and the driver exception was `Socket read timed out`.
- The workaround is to increase the timeout of the database server using the following:

  ```
  sqlnet.ora: SQLNET.INBOUND_CONNECT_TIMEOUT=180
  ```

  ```
  listener.ora: INBOUND_CONNECT_TIMEOUT_listener_name=180
  ```

Disable Wrapping of Data Type Objects

By default, data type objects for Array, Blob, Clob, NClob, Ref, SQLXML, and Struct, plus ParameterMetaData and ResultSetMetaData objects are wrapped with a WebLogic wrapper. You can disable wrapping, which can improve performance and allow applications to use native driver objects directly. See "Using Unwrapped Data Type Objects" in *Administering JDBC Data Sources for Oracle WebLogic Server*.

Advanced Configurations for Oracle Drivers and Databases

Oracle provides advanced configuration options that can provide improved data source and driver performance when using Oracle drivers and databases. Options include proxy authentication, setting credentials on a connection, connection harvesting, and labeling connections. See "Advanced Configurations for Oracle Drivers and Databases" in *Administering JDBC Data Sources for Oracle WebLogic Server*.

Use Best Design Practices

Most performance gains or losses in a database application is not determined by the application language, but by how the application is designed. The number and location of clients, size and structure of DBMS tables and indexes, and the number and types of queries all affect application performance. See "Designing Your Application for Best Performance" in *Developing JDBC Applications for Oracle WebLogic Server*.

12 Tuning Transactions

To optimize transaction performance, use the tuning guidelines described in this chapter.

This chapter includes the following sections:

Global Transaction Tuning

XA transaction cluster affinity allows server instances that are participating in a global transactions to service related requests rather than load-balancing these requests to other member servers. When `Enable Transaction Affinity=true`, cluster throughput is increased by:

- Reducing inter-server transaction coordination traffic
- Improving resource utilization, such as reducing JDBC connections
- Simplifying asynchronous processing of transactions

See "XA Transaction Affinity" in *Administering Clusters for Oracle WebLogic Server*.

XA Transaction Cluster Affinity

XA transaction cluster affinity allows server instances that are participating in a global transactions to service related requests rather than load-balancing these requests to other member servers. When `Enable Transaction Affinity=true`, cluster throughput is increased by:

- Reducing inter-server transaction coordination traffic
- Improving resource utilization, such as reducing JDBC connections
- Simplifying asynchronous processing of transactions

See "XA Transaction Affinity" in *Administering Clusters for Oracle WebLogic Server*.

Logging Last Resource Transaction Optimization

The Logging Last Resource (LLR) transaction optimization through JDBC data sources safely reduces the overhead of two-phase transactions involving database inserts, updates, and deletes. Two phase transactions occur when two different resources participate in the same global transaction (global transactions are often referred to as "XA" or "JTA" transactions). Consider the following:

- Typical two-phase transactions in JMS applications usually involve both a JMS server and a database server. The LLR option can as much as double performance compared to XA.
- The safety of the JDBC LLR option contrasts with well known but less-safe XA optimizations such as "last-agent", "last-participant", and "emulate-two-phase-commit" that are available from other vendors as well as WebLogic.
- JDBC LLR works by storing two-phase transaction records in a database table rather than in the transaction manager log (the TLOG).

See "Logging Last Resource Transaction Optimization" in *Developing JTA Applications for Oracle WebLogic Server*.

LLR Tuning Guidelines

The following section provides tuning guidelines for LLR:

- Oracle recommends that you read and understand "Logging Last Resource Transaction Optimization" in *Developing JTA Applications for Oracle WebLogic Server* and "Programming Considerations and Limitations for LLR Data Sources" in *Administering JDBC Data Sources for Oracle WebLogic Server*. LLR has a number of important administration and design implications.
- JDBC LLR generally improves performance of two-phase transactions that involve SQL updates, deletes, or inserts.
- LLR generally reduces the performance of two-phase transactions where all SQL operations are read-only (just selects).
- JDBC LLR pools provide no performance benefit to WebLogic JDBC stores. WebLogic JDBC stores are fully transactional but do not use JTA (XA) transactions on their internal JDBC connections.
- Consider using LLR instead of the less safe "last-agent" optimization for connectors, and the less safe "emulate-two-phase-commit" option for JDBC connection pools (formerly known as the "enable two-phase commit" option for pools that use non-XA drivers).
- On Oracle databases, heavily used LLR tables may become fragmented over time, which can lead to unused extents. This is likely due to the highly transient nature of the LLR table's data. To help avoid the issue, set `PCT_FREE` to 5 and `PCT_USED` to 95 on the LLR table. Also periodically defragment using the `ALTER TABLESPACE [tablespace-name] COALESCE` command.

Read-only, One-Phase Commit Optimizations

When resource managers, such as the Oracle Database, provide read-only optimizations, Oracle WebLogic can provide a read-only, one-phase commit optimization that provides a number of benefits – even when enabling multiple connections of the same XA transactions – such as eliminating `XAResource.prepare` network calls and transaction log writes, both in Oracle WebLogic and in the resource manager.

See "Read-only, One-Phase Commit Optimizations" in *Developing JTA Applications for Oracle WebLogic Server*.

Configure XA Transactions without TLogs

Improves XA transaction performance by eliminating TLogs when XA transactions span a single Transaction Manager (TM). XA transaction resources (Determiners) are used during transaction recovery when a TLog is not present. See "Transactions without TLogs" in *Developing JTA Applications for Oracle WebLogic Server*.

13

Tuning WebLogic JMS

Get the most out of your applications by implementing the administrative performance tuning features available with WebLogic JMS, described in this chapter.

This chapter includes the following sections:

JMS Performance & Tuning Check List

The following section provides a checklist of items to consider when tuning WebLogic JMS:

- Always configure quotas, see Defining Quota.

- Verify that default paging settings apply to your needs, see Paging Out Messages To Free Up Memory. Paging lowers performance but may be required if JVM memory is insufficient.
- Avoid large message backlogs. See Handling Large Message Backlogs.
- Create and use custom connection factories with all applications instead of using default connection factories, including when using MDBs. Default connection factories are not tunable, while custom connection factories provide many options for performance tuning.
- Write applications so that they cache and re-use JMS client resources, including JNDI contexts and lookups, and JMS connections, sessions, consumers, or producers. These resources are relatively expensive to create. For information on detecting when caching is needed, as well as on built-in pooling features, see Cache and Re-use Client Resources.
- For asynchronous consumers and MDBs, tune `MessagesMaximum` on the connection factory. Increasing `MessagesMaximum` can improve performance, decreasing `MessagesMaximum` to its minimum value can lower performance, but helps ensure that messages do not end up waiting for a consumer that's already processing a message. See Tuning MessageMaximum.
- Avoid single threaded processing when possible. Use multiple concurrent producers and consumers and ensure that enough threads are available to service them.
- Tune server-side applications so that they have enough instances. Consider creating dedicated thread pools for these applications. See Tuning Message-Driven Beans.
- For client-side applications with asynchronous consumers, tune client-side thread pools using Client-side Thread Pools.
- Tune persistence as described in Tuning the WebLogic Persistent Store. In particular, it's normally best for multiple JMS servers, destinations, and other services to share the same store so that the store can aggregate concurrent requests into single physical I/O requests, and to reduce the chance that a JTA transaction spans more than one store. Multiple stores should only be considered once it's been established that the a single store is not scaling to handle the current load.
- If you have large messages, see Tuning for Large Messages.
- Prevent unnecessary message routing in a cluster by carefully configuring connection factory targets. Messages potentially route through two servers, as they flow from a client, through the client's connection host, and then on to a final destination. For server-side applications, target connection factories to the cluster. For client-side applications that work with a distributed destination, target connection factories only to servers that host the distributed destinations members. For client-side applications that work with a singleton destination, target the connection factory to the same server that hosts the destination.
- If JTA transactions include both JMS and JDBC operations, consider enabling the JDBC LLR optimization. LLR is a commonly used safe "ACID" optimization that can lead to significant performance improvements, with some drawbacks. See Tuning Transactions.

- If you are using Java clients, avoid thin Java clients except when a small jar size is more important than performance. Thin clients use the slower IIOP protocol even when T3 is specified so use a full java client instead. See *Developing Stand-alone Clients for Oracle WebLogic Server*.
- Tune JMS Store-and-Forward according to Tuning WebLogic JMS Store-and-Forward.
- Tune a WebLogic Messaging Bridge according Tuning WebLogic Message Bridge.
- If you are using non-persistent non-transactional remote producer clients, then consider enabling one-way calls. See Using One-Way Message Sends.
- Consider using JMS distributed queues. See "Using Distributed Queues" in *Developing JMS Applications for Oracle WebLogic Server*.
- If you are already using distributed queues, see Tuning Distributed Queues.
- Consider using advanced distributed topic features (PDTs). See "Developing Advanced Pub/Sub Applications" in *Developing JMS Applications for Oracle WebLogic Server*.
- If your applications use Topics, see Tuning Topics.
- Avoid configuring sorted destinations, including priority sorted destinations. FIFO or LIFO destinations are the most efficient. Destination sorting can be expensive when there are large message backlogs, even a backlog of a few hundred messages can lower performance.
- Use careful selector design. See "Filtering Messages" in *Developing JMS Applications for Oracle WebLogic Server*.
- Run applications on the same WebLogic Servers that are also hosting destinations. This eliminates networking and some or all marshalling overhead, and can heavily reduce network and CPU usage. It also helps ensure that transactions are local to a single server. This is one of the major advantages of using an application server's embedded messaging.

Handling Large Message Backlogs

When message senders inject messages faster than consumers, messages accumulate into a message *backlog*. Large backlogs can be problematic for a number of reasons, for example:

- Indicates consumers may not be capable of handling the incoming message load, are failing, or are not properly load balanced across a distributed queue.
- Can lead to out-of-memory on the server, which in turn prevents the server from doing any work.
- Can lead to high garbage collection (GC) overhead. A JVM's GC overhead is partially proportional to the number of live objects in the JVM.

Improving Message Processing Performance

One area for investigation is to improve overall message processing performance. Here are some suggestions:

- Follow the JMS tuning recommendations as described in JMS Performance & Tuning Check List.
- Check for programming errors in newly developed applications. In particular, ensure that non-transactional consumers are acknowledging messages, that transactional consumers are committing transactions, that plain `javax.jms` applications called `javax.jms.Connection.start()`, and that transaction timeouts are tuned to reflect the needs of your particular application. Here are some symptoms of programming errors: consumers are not receiving any messages (make sure they called `start()`), high "pending" counts for queues, already processed persistent messages re-appearing after a shutdown and restart, and already processed transactional messages re-appearing after a delay (the default JTA timeout is 30 seconds, default transacted session timeout is one hour).
- Check WebLogic statistics for queues that are not being serviced by consumers. If you're having a problem with distributed queues, see Tuning Distributed Queues.
- Check WebLogic statistics for topics with high pending counts. This usually indicates that there are topic subscriptions that are not being serviced. There may be a slow or unresponsive consumer client that's responsible for processing the messages, or it's possible that a durable subscription may no longer be needed and should be deleted, or the messages may be accumulating due to delayed distributed topic forwarding. You can check statistics for individual durable subscriptions on the WebLogic Server Administration Console. A durable subscription with a large backlog may have been created by an application but never deleted. Unserviced durable subscriptions continue to accumulate topic messages until they are either administratively destroyed, or unsubscribed by a standard JMS client.
- Understand distributed topic behavior when not all members are active. In distributed topics, each produced message to a particular topic member is forwarded to each remote topic member. If a remote topic member is unavailable then the local topic member will store each produced message for later forwarding. Therefore, if a topic member is unavailable for a long period of time, then large backlogs can develop on the active members. In some applications, this backlog can be addressed by setting expiration times on the messages. See Defining a Message Expiration Policy.
- In certain applications it may be fine to automatically delete old unprocessed messages. See Handling Expired Messages.
- For transactional MDBs, consider using MDB transaction batching as this can yield a 5 fold improvement in some use cases.
- Leverage distributed queues and add more JVMs to the cluster (in order to add more distributed queue member instances). For example, split a 200,000 message backlog across 4 JVMs at 50,000 messages per JVM, instead of 100,000 messages per JVM.
- For client applications, use asynchronous consumers instead of synchronous consumers when possible. Asynchronous consumers can have a significantly lower network overhead, lower latency, and do not block a thread while waiting for a message.
- For synchronous consumer client applications, consider: enabling `prefetch`, using `CLIENT_ACKNOWLEDGE` to enable acknowledging multiple consumed messages at a time, and using `DUPS_OK_ACKNOWLEDGE` instead of `AUTO_ACKNOWLEDGE`.

- For asynchronous consumer client applications, consider using `DUPS_OK_ACKNOWLEDGE` instead of `AUTO_ACKNOWLEDGE`.
- Leverage batching. For example, include multiple messages in each transaction, or send one larger message instead of many smaller messages.
- For non-durable subscriber client-side applications handling missing ("dropped") messages, investigate `MULTICAST_NO_ACKNOWLEDGE`. This mode broadcasts messages concurrently to subscribers over UDP multicast.

Controlling Message Production

Another area for investigation is to slow down or even stop message production. Here are some suggestions:

- Set lower quotas. See Defining Quota.
- Use fewer producer threads.
- Tune a sender blocking timeout that occurs during a quota condition, as described Blocking Senders During Quota Conditions. The timeout is tunable on connection factory.
- Tune producer flow control, which automatically slows down producer calls under threshold conditions. See Controlling the Flow of Messages on JMS Servers and Destinations.
- Consider modifying the application to implement flow-control. For example, some applications do not allow producers to inject more messages until a consumer has successfully processed the previous batch of produced messages (a windowing protocol). Other applications might implement a request/reply algorithm where a new request isn't submitted until the previous reply is received (essentially a windowing protocol with a window size of 1). In some cases, JMS tuning is not required as the synchronous flow from the RMI/EJB/Servlet is adequate.

Drawbacks to Controlling Message Production

Slowing down or stopping message processing has at least two potential drawbacks:

- It puts back-pressure on the down-stream flow that is calling the producer. Sometimes the down-stream flow cannot handle this back-pressure, and a hard-to-handle backlog develops behind the producer. The location of the backlog depends on what's calling the producer. For example, if the producer is being called by a servlet, the backlog might manifest as packets accumulating on the incoming network socket or network card.
- Blocking calls on server threads can lead to thread-starvation, too many active threads, or even dead-locks. Usually the key to address this problem is to ensure that the producer threads are running in a size limited dedicated thread pool, as this ensures that the blocking threads do not interfere with activity in other thread pools. For example, if an EJB or servlet is calling a "send" that might block for a significant time: configure a custom work manager with a `max threads` constraint, and set the `dispatch-policy` of the EJB/servlet to reference this work-manager.

Cache and Re-use Client Resources

JMS client resources are relatively expensive to create in comparison to sending and receiving messages. These resources should be cached or pooled for re-use rather than recreating them with each message. They include contexts, destinations, connection factories, connections, sessions, consumers, or producers.

In addition, it is important for applications to close contexts, connections, sessions, consumers, or producers once they are completely done with these resources. Failing to close unused resources leads to a memory leak, which lowers overall JVM performance and eventually may cause the JVM to fail with an out-of-memory error. Be aware that JNDI contexts have `close()` method, and that closing a JMS connection automatically efficiently closes all sessions, consumers, and producers created using the connection.

For server-side applications, WebLogic automatically wraps and pools JMS resources that are accessed using a resource reference. See "Enhanced Support for Using WebLogic JMS with EJBs and Servlets" in *Developing JMS Applications for Oracle WebLogic Server*. This pooling code can be inefficient at pooling producers if the target destination changes frequently, but there's a simple work-around: use anonymous producers by passing `null` for the destination when the application calls `createProducer()` and then instead pass the desired destination into each send call.

- To check for heavy JMS resource allocation or leaks, you can monitor mbean stats and/or use your particular JVM's built in facilities. You can monitor mbean stats using the console, WLST, or java code.
- Check JVM heap statistics for memory leaks or unexpectedly high allocation counts.
- Similarly, check WebLogic statistics for memory leaks or unexpectedly high allocation counts.

Tuning Distributed Queues

If produced messages are failing to load balance evenly across all distributed queue members, you may wish to change the configuration of your producer connection factories to disable *server affinity* (enabled by default).

Once created, a JMS consumer remains pinned to a particular queue member. This can lead to situations where consumers are not evenly load balanced across all distributed queue members, particularly if new members become available after all consumers have been initialized. If consumers fail to load balance evenly across all distributed queue members, the best option is to use an MDB that's targeted to a cluster designed to process the messages. WebLogic MDBs automatically ensure that all distributed queue members are serviced. If MDBs are not an option, here are some suggestions to improve consumer load balancing:

- Ensure that your application is creating enough consumers and the consumer's connection factory is tuned using the available load balancing options. In particular, consider disabling the default *server affinity* setting.)
- Change applications to periodically close and recreate consumers. This forces consumers to re-load balance.

- Consume from individual queue members instead of from the distributed queues logical name. Each distributed queue member is individually advertised in JNDI as *`jms-server-name@distributed-destination-jndi-name`*.
- Configure the distributed queue to enable forwarding. Distributed queue forwarding automatically internally forwards messages that have been idled on a member destination without consumers to a member that has consumers. This approach may not be practical for high message load applications.

> **Note:**
>
> Queue forwarding is not compatible with the WebLogic JMS Unit-of-Order feature, as it can cause messages to be delivered out of order.

See "Using Distributed Destinations" in *Developing JMS Applications for Oracle WebLogic Server* and "Configuring Advanced JMS System Resources" in *Administering JMS Resources for Oracle WebLogic Server*.

Tuning Topics

The following section provides information on how to tune WebLogic Topics:

- You may want to convert singleton topics to distributed topics. A distributed topic with a `Partitioned` policy generally outperforms the `Replicated` policy choice.
- Oracle highly recommends leveraging MDBs to process Topic messages, especially when working with Distributed Topics. MDBs automate the creation and servicing of multiple subscriptions and also provide high scalability options to automatically distribute the messages for a single subscription across multiple Distributed Topic members.
- There is a `Sharable` subscription extension that allows messages on a single topic subscription to be processed in parallel by multiple subscribers on multiple JVMs. WebLogic MDBs leverage this feature when they are not in `Compatibility` mode.
- If produced messages are failing to load balance evenly across the members of a Partitioned Distributed Topic, you may need to change the configuration of your producer connection factories to disable server affinity (enabled by default).
- Before using any of these previously mentioned advanced features, Oracle recommends fully reviewing the following related documentation:
  – "Configuring and Deploying MDBs Using Distributed Topics" in *Developing Message-Driven Beans for Oracle WebLogic Server*
  – "Developing Advanced Pub/Sub Applications" in *Administering JMS Resources for Oracle WebLogic Server*
  – "Advanced Programming with Distributed Destinations Using the JMS Destination Availability Helper API" in *Administering JMS Resources for Oracle WebLogic Server*

Tuning Non-durable Topic Publishers

Since WebLogic Server 9.0, a non-durable topic message publish request may block until the message is pushed to all consumers that are currently ready to process the message. This may cause non-durable topic publishers with large numbers of consumers to take longer to publish a message than expected. To revert to a publish that does not wait for consumers and waits only until it's confirmed the message arrived on a JMS server, use the following property:

```
-Dweblogic.messaging.DisableTopicMultiSender=true
```

Tuning for Large Messages

The following sections provide information on how to improve JMS performance when handling large messages:

- Tuning MessageMaximum
- Setting Maximum Message Size for Network Protocols
- Compressing Messages
- Paging Out Messages To Free Up Memory

Defining Quota

It is highly recommended to always configure message count quotas. Quotas help prevent large message backlogs from causing out-of-memory errors, and WebLogic JMS does not set quotas by default.

There are many options for setting quotas, but in most cases it is enough to simply set a `Messages Maximum` quota on each JMS Server rather than using destination level quotas. Keep in mind that each current JMS message consumes JVM memory even when the message has been paged out, because paging pages out only the message bodies but not message headers. A good rule of thumb for queues is to assume that each current JMS message consumes 512 bytes of memory. A good rule of thumb for topics is to assume that each current JMS message consumes 256 bytes of memory plus an additional 256 bytes of memory for each subscriber that hasn't acknowledged the message yet. For example, if there are 3 subscribers on a topic, then a single published message that hasn't been processed by any of the subscribers consumes 256 + 256*3 = 1024 bytes even when the message is paged out. Although message header memory usage is typically significantly less than these rules of thumb indicate, it is a best practice to make conservative estimates on memory utilization.

In prior releases, there were multiple levels of quotas: destinations had their own quotas and would also have to compete for quota within a JMS server. In this release, there is only one level of quota: destinations can have their own private quota or they can compete with other destinations using a shared quota.

In addition, a destination that defines its own quota no longer also shares space in the JMS server's quota. Although JMS servers still allow the direct configuration of message and byte quotas, these options are only used to provide quota for destinations that do not refer to a quota resource.

Quota Resources

A quota is a named configurable JMS module resource. It defines a maximum number of messages and bytes, and is then associated with one or more destinations and is responsible for enforcing the defined maximums. Multiple destinations referring to the same quota share available quota according to the sharing policy for that quota resource.

Quota resources include the following configuration parameters:

Table 13-1 Quota Parameters

Attribute	Description
Bytes Maximum and Messages Maximum	The Messages Maximum/Bytes Maximum parameters for a quota resource defines the maximum number of messages and/or bytes allowed for that quota resource. No consideration is given to messages that are pending; that is, messages that are in-flight, delayed, or otherwise inhibited from delivery still count against the message and/or bytes quota.
Quota Sharing	The Shared parameter for a quota resource defines whether multiple destinations referring to the same quota resource compete for resources with each other.
Quota Policy	The Policy parameter defines how individual clients compete for quota when no quota is available. It affects the order in which send requests are unblocked when the Send Timeout feature is enabled on the connection factory, as described in Tuning for Large Messages.

Destination-Level Quota

Destinations no longer define byte and messages maximums for quota, but can use a quota resource that defines these values, along with quota policies on sharing and competition.

The Quota parameter of a destination defines which quota resource is used to enforce quota for the destination. This value is dynamic, so it can be changed at any time. However, if there are unsatisfied requests for quota when the quota resource is changed, then those requests will fail with a `javax.jms.ResourceAllocationException`.

Note:

Outstanding requests for quota will fail at such time that the quota resource is changed. This does not mean changes to the message and byte attributes for the quota resource, but when a destination switches to a different quota.

JMS Server-Level Quota

In some cases, there will be destinations that do not configure quotas. JMS Server quotas allow JMS servers to limit the resources used by these *quota-less* destinations. All destinations that do not explicitly set a value for the Quota attribute share the quota of the JMS server where they are deployed. The behavior is exactly the same as if there were a special Quota resource defined for each JMS server with the Shared parameter enabled.

The interfaces for the JMS server quota are unchanged from prior releases. The JMS server quota is entirely controlled using methods on the JMSServerMBean. The quota policy for the JMS server quota is set by the Blocking Send Policy parameter on a JMS server, as explained in Specifying a Blocking Send Policy on JMS Servers. It behaves just like the Policy setting of any other quota.

Blocking Senders During Quota Conditions

- Defining a Send Timeout on Connection Factories
- Specifying a Blocking Send Policy on JMS Servers

Defining a Send Timeout on Connection Factories

Blocking producers during quota conditions (by defining a send timeout) can dramatically improve the performance of applications and benchmarks that continuously retry message sends on quota failures. The Send Timeout feature provides more control over message send operations by giving message produces the option of waiting a specified length of time until space becomes available on a destination. For example, if a producer makes a request and there is insufficient space, then the producer is blocked until space becomes available, or the operation times out. See Controlling the Flow of Messages on JMS Servers and Destinations for another method of flow control.

To use the WebLogic Server Administration Console to define how long a JMS connection factory will block message requests when a destination exceeds its maximum quota.

1. Navigate to the JMS Connection Factory: Configuration: Flow Control page.
2. In the Send Timeout field, enter the amount of time, in milliseconds, a sender will block messages when there is insufficient space on the message destination. Once the specified waiting period ends, one of the following results will occur:
   - If sufficient space becomes available before the timeout period ends, the operation continues.
   - If sufficient space does not become available before the timeout period ends, you receive a *resource allocation* exception.

     If you choose not to enable the blocking send policy by setting this value to 0, then you will receive a resource allocation exception whenever sufficient space is not available on the destination.
3. Click Save.

Specifying a Blocking Send Policy on JMS Servers

The Blocking Send policies enable you to define the JMS server's blocking behavior on whether to deliver smaller messages before larger ones when multiple message producers are competing for space on a destination that has exceeded its message quota.

To use the WebLogic Server Administration Console to define how a JMS server will block message requests when its destinations are at maximum quota.

1. Navigate to the JMS Server: Configuration: Thresholds and Quotas page of the WebLogic Server Administration Console.
2. From the Blocking Send Policy list box, select one of the following options:
   - FIFO — All send requests for the same destination are queued up one behind the other until space is available. No send request is permitted to complete when there another send request is waiting for space before it.
   - Preemptive — A send operation can preempt other blocking send operations if space is available. That is, if there is sufficient space for the current request, then that space is used even if there are previous requests waiting for space.
3. Click Save.

Tuning MessageMaximum

WebLogic JMS pipelines messages that are delivered to asynchronous consumers (otherwise known as message listeners) or prefetch-enabled synchronous consumers. This action aids performance because messages are aggregated when they are internally pushed from the server to the client. The messages backlog (the size of the pipeline) between the JMS server and the client is tunable by configuring the `MessagesMaximum` setting on the connection factory. See "Asynchronous Message Pipeline" in *Developing JMS Applications for Oracle WebLogic Server*.

In some circumstances, tuning the `MessagesMaximum` parameter may improve performance dramatically, such as when the JMS application defers acknowledges or commits. In this case, Oracle suggests setting the `MessagesMaximum` value to:

```
2 * (ack or commit interval) + 1
```

For example, if the JMS application acknowledges 50 messages at a time, set the `MessagesMaximum` value to 101.

Tuning MessageMaximum Limitations

Tuning the `MessagesMaximum` value too high can cause:

- Increased memory usage on the client.
- Affinity to an existing client as its pipeline fills with messages. For example: If `MessagesMaximum` has a value of 10,000,000, the first consumer client to connect will get all messages that have already arrived at the destination. This condition leaves other consumers without any messages and creates an unnecessary backlog of messages in the first consumer that may cause the system to run out of memory.
- Packet is too large exceptions and stalled consumers. If the aggregate size of the messages pushed to a consumer is larger than the current protocol's maximum message size (default size is 10 MB and is configured on a per WebLogic Server instance basis using the console and on a per client basis using the `-Dweblogic.MaxMessageSize` command line property), the message delivery fails.

Setting Maximum Message Size for Network Protocols

You may need to configure WebLogic clients in addition to the WebLogic Server instances, when sending and receiving large messages.

For most protocols, including T3, WLS limits the size of a network call to 10MB by default. If individual JMS message sizes exceed this limit, or if a set of JMS messages that is batched into the same network call exceeds this limit, this can lead to either "packet too large exceptions" and/or stalled consumers. Asynchronous consumers can cause multiple JMS messages to batch into the same network call, to control this batch size, see Tuning MessageMaximum Limitations.

To set the maximum message size on a server instance, tune the maximum message size for each supported protocol on a per protocol basis for each involved default channel or custom channel. In this context the word 'message' refers to all network calls over the given protocol, not just JMS calls.

To set the maximum message size on a client, use the following command line property:

```
-Dweblogic.MaxMessageSize
```

Note:

This setting applies to all WebLogic Server network packets delivered to the client, not just JMS related packets.

Compressing Messages

A message compression threshold can be set administratively by either specifying a Default Compression Threshold value on a connection factory or on a JMS SAF remote context. Compressed messages may actually inadvertently affect destination quotas since some message types actually grow larger when compressed

Once configured, message compression is triggered on producers for client sends, on connection factories for message receives and message browsing, or through SAF forwarding. Messages are compressed using GZIP. Compression only occurs when message producers and consumers are located on separate server instances where messages must cross a JVM boundary, typically across a network connection when WebLogic domains reside on different machines. Decompression automatically occurs on the client side and only when the message content is accessed, except for the following situations:

- Using message selectors on compressed XML messages can cause decompression, since the message body must be accessed in order to filter them. For more information on defining XML message selectors, see "Filtering Messages" in *Developing JMS Applications for Oracle WebLogic Server*.
- Interoperating with earlier versions of WebLogic Server can cause decompression. For example, when using the Messaging Bridge, messages are decompressed when sent from the current release of WebLogic Server to a receiving side that is an earlier version of WebLogic Server.

On the server side, messages always remains compressed, even when they are written to disk.

Paging Out Messages To Free Up Memory

With the *message paging* feature, JMS servers automatically attempt to free up virtual memory during peak message load periods. This feature can greatly benefit applications with large message spaces. Message paging is always enabled on JMS servers, and so a message paging directory is automatically created without having to configure one. You can, however, specify a directory using the Paging Directory option, then paged-out messages are written to files in this directory.

In addition to the paging directory, a JMS server uses either a file store or a JDBC store for persistent message storage. The file store can be user-defined or the server's default store. Paged JDBC store persistent messages are copied to both the JDBC store as well as the JMS Server's paging directory. Paged file store persistent messages that are small are copied to both the file store as well as the JMS Server's paging directory. Paged larger file store messages are not copied into the paging directory. See Best Practices When Using Persistent Stores.

However, a paged-out message does not free all of the memory that it consumes, since the message header with the exception of any user properties, which are paged out along with the message body, remains in memory for use with searching, sorting, and filtering. Queuing applications that use selectors to select paged messages may show severely degraded performance as the paged out messages must be paged back in. This does not apply to topics or to applications that select based only on message header fields (such as `CorrelationID`). A good rule of thumb is to conservatively assume that messages each use 512 bytes of JVM memory even when paged out.

Specifying a Message Paging Directory

If a paging directory is not specified, then paged-out message bodies are written to the default `\tmp` directory inside the *servername* subdirectory of a domain's root directory. For example, if no directory name is specified for the default paging directory, it defaults to:

```
oracle_home\user_projects\domains\domainname\servers\servername\tmp
```

where `domainname` is the root directory of your domain, typically `c:\Oracle\Middleware\Oracle_Home\user_projects\domains\domainname`, which is parallel to the directory in which WebLogic Server program files are stored, typically `c:\Oracle\Middleware\Oracle_Home\wlserver`.

To configure the Message Paging Directory attribute, see "Configure general JMS server properties" in Oracle WebLogic Server Administration Console Online Help.

Tuning the Message Buffer Size Option

The Message Buffer Size option specifies the amount of memory that will be used to store message bodies in memory before they are paged out to disk. The default value of Message Buffer Size is approximately one-third of the maximum heap size for the JVM, or a maximum of 512 megabytes. The larger this parameter is set, the more memory JMS will consume when many messages are waiting on queues or topics. Once this threshold is crossed, JMS may write message bodies to the directory specified by the Paging Directory option in an effort to reduce memory usage below this threshold.

It is important to remember that this parameter is not a quota. If the number of messages on the server passes the threshold, the server writes the messages to disk and evicts the messages from memory as fast as it can to reduce memory usage, but it will not stop accepting new messages. It is still possible to run out of memory if messages are arriving faster than they can be paged out. Users with high messaging loads who wish to support the highest possible availability should consider setting a quota, or setting a threshold and enabling flow control to reduce memory usage on the server.

Controlling the Flow of Messages on JMS Servers and Destinations

With the Flow Control feature, you can direct a JMS server or destination to slow down message producers when it determines that it is becoming overloaded. See Compressing Messages.

The following sections describe how flow control feature works and how to configure flow control on a connection factory.

- How Flow Control Works
- Configuring Flow Control
- Flow Control Thresholds

How Flow Control Works

Specifically, when either a JMS server or it's destinations exceeds its specified byte or message threshold, it becomes *armed* and instructs producers to limit their message flow (messages per second).

Producers will limit their production rate based on a set of flow control attributes configured for producers via the JMS connection factory. Starting at a specified `flow maximum` number of messages, a producer evaluates whether the server/destination is still armed at prescribed intervals (for example, every 10 seconds for 60 seconds). If at each interval, the server/destination is still armed, then the producer continues to move its rate down to its prescribed *flow minimum* amount.

As producers slow themselves down, the threshold condition gradually corrects itself until the server/destination is *unarmed*. At this point, a producer is allowed to increase its production rate, but not necessarily to the maximum possible rate. In fact, its message flow continues to be controlled (even though the server/destination is no longer armed) until it reaches its prescribed *flow maximum*, at which point it is no longer flow controlled.

Configuring Flow Control

Producers receive a set of flow control attributes from their session, which receives the attributes from the connection, and which receives the attributes from the connection factory. These attributes allow the producer to adjust its message flow.

Specifically, the producer receives attributes that limit its flow within a minimum and maximum range. As conditions worsen, the producer moves toward the minimum; as conditions improve; the producer moves toward the maximum. Movement toward the minimum and maximum are defined by two additional attributes that specify the rate of movement toward the minimum and maximum. Also, the need for movement toward the minimum and maximum is evaluated at a configured interval.

Flow Control options are described in following table:

Table 13-2 Flow Control Parameters

Attribute	Description
Flow Control Enabled	Determines whether a producer can be flow controlled by the JMS server.

Table 13-2 (Cont.) Flow Control Parameters

Attribute	Description
Flow Maximum	The maximum number of messages per second for a producer that is experiencing a threshold condition. If a producer is not currently limiting its flow when a threshold condition is reached, the initial flow limit for that producer is set to Flow Maximum. If a producer is already limiting its flow when a threshold condition is reached (the flow limit is less than Flow Maximum), then the producer will continue at its current flow limit until the next time the flow is evaluated. Once a threshold condition has subsided, the producer is not permitted to ignore its flow limit. If its flow limit is less than the Flow Maximum, then the producer must gradually increase its flow to the Flow Maximum each time the flow is evaluated. When the producer finally reaches the Flow Maximum, it can then ignore its flow limit and send without limiting its flow.
Flow Minimum	The minimum number of messages per second for a producer that is experiencing a threshold condition. This is the lower boundary of a producer's flow limit. That is, WebLogic JMS will not further slow down a producer whose message flow limit is at its Flow Minimum.
Flow Interval	An adjustment period of time, defined in seconds, when a producer adjusts its flow from the Flow Maximum number of messages to the Flow Minimum amount, or vice versa.
Flow Steps	The number of steps used when a producer is adjusting its flow from the Flow Minimum amount of messages to the Flow Maximum amount, or vice versa. Specifically, the Flow Interval adjustment period is divided into the number of Flow Steps (for example, 60 seconds divided by 6 steps is 10 seconds per step). Also, the movement (that is, the rate of adjustment) is calculated by dividing the difference between the Flow Maximum and the Flow Minimum into steps. At each Flow Step, the flow is adjusted upward or downward, as necessary, based on the current conditions, as follows: The downward movement (the decay) is geometric over the specified period of time (Flow Interval) and according to the specified number of Flow Steps. (For example, 100, 50, 25, 12.5). The movement upward is linear. The difference is simply divided by the number of Flow Steps.

Flow Control Thresholds

The attributes used for configuring bytes/messages thresholds are defined as part of the JMS server and/or its destination.Table 13-3 defines how the upper and lower thresholds start and stop flow control on a JMS server and/or JMS destination.

Table 13-3 Flow Control Threshold Parameters

Attribute	Description
Bytes/Messages Threshold High	When the number of bytes/messages exceeds this threshold, the JMS server/destination becomes armed and instructs producers to limit their message flow.

Table 13-3 (Cont.) Flow Control Threshold Parameters

Attribute	Description
Bytes/Messages Threshold Low	When the number of bytes/messages falls below this threshold, the JMS server/destination becomes unarmed and instructs producers to begin increasing their message flow. Flow control is still in effect for producers that are below their message flow maximum. Producers can move their rate upward until they reach their flow maximum, at which point they are no longer flow controlled.

Handling Expired Messages

The following sections describe two message expiration features, the message Expiration Policy and the Active Expiration of message, which provide more control over how the system searches for expired messages and how it handles them when they are encountered.

Active message expiration ensures that expired messages are cleaned up immediately. Moreover, expired message auditing gives you the option of tracking expired messages, either by logging when a message expires or by redirecting expired messages to a defined *error* destination.

- Defining a Message Expiration Policy
- Tuning Active Message Expiration

Defining a Message Expiration Policy

Use the message Expiration Policy feature to define an alternate action to take when messages expire. Using the Expiration Policy attribute on the Destinations node, an expiration policy can be set on a per destination basis. The Expiration Policy attribute defines the action that a destination should take when an expired message is encountered: discard the message, discard the message and log its removal, or redirect the message to an error destination.

Also, if you use JMS templates to configure multiple destinations, you can use the Expiration Policy field to quickly configure an expiration policy on all your destinations. To override a template's expiration policy for specific destinations, you can modify the expiration policy on any destination.

For instructions on configuring the Expiration Policy, click one of the following links:

- Configuring an Expiration Policy on Topics
- Configuring an Expiration Policy on Queues
- Configuring an Expiration Policy on Templates
- Defining an Expiration Logging Policy

Configuring an Expiration Policy on Topics

Follow these directions if you are configuring an expiration policy on topics without using a JMS template. Expiration policies that are set on specific topics will override the settings defined on a JMS template.

1. Navigate to the JMS Topic: Configuration: Delivery Failure page.
2. From the Expiration Policy list box, select an expiration policy option.
   - Discard — Expired messages are removed from the system. The removal is not logged and the message is not redirected to another location.
   - Log — Removes expired messages and writes an entry to the server log file indicating that the messages were removed from the system. You define the actual information that will be logged in the Expiration Logging Policy field in next step.
   - Redirect — Moves expired messages from their current location into the Error Destination defined for the topic.
3. If you selected the Log expiration policy in previous step, use the Expiration Logging Policy field to define what information about the message is logged.

   For more information about valid Expiration Logging Policy values, see Defining an Expiration Logging Policy.
4. Click Save.

Configuring an Expiration Policy on Queues

Follow these directions if you are configuring an expiration policy on queues without using a JMS template. Expiration policies that are set on specific queues will override the settings defined on a JMS template.

1. Navigate to the JMS Queue: Configuration: Delivery Failure page of the Oracle WebLogic Server Console.
2. From the Expiration Policy list box, select an expiration policy option.
   - Discard — Expired messages are removed from the system. The removal is not logged and the message is not redirected to another location.
   - Log — Removes expired messages from the queue and writes an entry to the server log file indicating that the messages were removed from the system. You define the actual information that will be logged in the Expiration Logging Policy field described in the next step.
   - Redirect — Moves expired messages from the queue and into the Error Destination defined for the queue.
3. If you selected the Log expiration policy in the previous step, use the Expiration Logging Policy field to define what information about the message is logged.

   For more information about valid Expiration Logging Policy values, see Defining an Expiration Logging Policy.
4. Click Save

Configuring an Expiration Policy on Templates

Since JMS templates provide an efficient way to define multiple destinations (topics or queues) with similar attribute settings, you can configure a message expiration policy on an existing template (or templates) for your destinations.

1. Navigate to the JMS Template: Configuration: Delivery Failure page on the Oracle WebLogic Server Console.
2. In the Expiration Policy list box, select an expiration policy option.
   - Discard — Expired messages are removed from the messaging system. The removal is not logged and the message is not redirected to another location.
   - Log — Removes expired messages and writes an entry to the server log file indicating that the messages were removed from the system. The actual information that is logged is defined by the Expiration Logging Policy field described in the next step.
   - Redirect — Moves expired messages from their current location into the Error Destination defined for the destination.
3. If you selected the Log expiration policy in Step 4, use the Expiration Logging Policy field to define what information about the message is logged.

   For more information about valid Expiration Logging Policy values, see Defining an Expiration Logging Policy.
4. Click Save.

Defining an Expiration Logging Policy

The following section provides information on the expiration policy.

The Expiration Logging Policy parameter has been deprecated in this release of WebLogic Server. In its place, Oracle recommends using the Message Life Cycle Logging feature, which provide a more comprehensive view of the basic events that JMS messages will traverse through once they are accepted by a JMS server, including detailed message expiration data. For more information about message life cycle logging options, see "Message Life Cycle Logging" in *Administering JMS Resources for Oracle WebLogic Server*.

For example, you could specify one of the following values:

- `JMSPriority, Name, Address, City, State, Zip`
- `%header%, Name, Address, City, State, Zip`
- `JMSCorrelationID, %properties%`

The `JMSMessageID` field is always logged and cannot be turned off. Therefore, if the Expiration Policy is not defined (that is, none) or is defined as an empty string, then the output to the log file contains only the `JMSMessageID` of the message.

Expiration Log Output Format

When an expired message is logged, the text portion of the message (not including timestamps, severity, thread information, security identity, etc.) conforms to the following format:

```
<ExpiredJMSMessage JMSMessageId='$MESSAGEID' >
 <HeaderFields Field1='Value1' [Field2='Value2'] … ] />
 <UserProperties Property1='Value1' [Property='Value2'] … ] />
</ExpiredJMSMessage>
```

where $`MESSAGEID` is the exact string returned by `Message.getJMSMessageID()`.

For example:

```
<ExpiredJMSMessage JMSMessageID='ID:P<851839.1022176920343.0' >
 <HeaderFields JMSPriority='7' JMSRedelivered='false' />
 <UserProperties Make='Honda' Model='Civic' Color='White'Weight='2680' />
</ExpiredJMSMessage>
```

If no header fields are displayed, the line for header fields is not be displayed. If no user properties are displayed, that line is not be displayed. If there are no header fields and no properties, the closing `</ExpiredJMSMessage>` tag is not necessary as the opening tag can be terminated with a closing bracket (`/>`).

For example:

```
<ExpiredJMSMessage JMSMessageID='ID:N<223476.1022177121567.1' />
```

All values are delimited with double quotes. All string values are limited to 32 characters in length. Requested fields and/or properties that do not exist are not displayed. Requested fields and/or properties that exist but have no value (a null value) are displayed as null (without single quotes). Requested fields and/or properties that are empty strings are displayed as a pair of single quotes with no space between them.

For example:

```
<ExpiredJMSMessage JMSMessageID='ID:N<851839.1022176920344.0' >
 <UserProperties First='Any string longer than 32 char ...' Second=null Third='' />
</ExpiredJMSMessage>
```

Tuning Active Message Expiration

Use the Active Expiration feature to define the timeliness in which expired messages are removed from the destination to which they were sent or published. Messages are not necessarily removed from the system at their expiration time, but they are removed within a user-defined number of seconds. The smaller the window, the closer the message removal is to the actual expiration time.

Configuring a JMS Server to Actively Scan Destinations for Expired Messages

Follow these directions to define how often a JMS server will actively scan its destinations for expired messages. The default value is 30 seconds, which means the JMS server waits 30 seconds between each scan interval.

1. Navigate to the JMS Server: Configuration: General page of the WebLogic Server Administration Console.
2. In the Scan Expiration Interval field, enter the amount of time, in seconds, that you want the JMS server to pause between its cycles of scanning its destinations for expired messages to process.

   To disable active scanning, enter a value of 0 seconds. Expired messages are passively removed from the system as they are discovered.
3. Click Save.

There are a number of design choices that impact performance of JMS applications. Some others include reliability, scalability, manageability, monitoring, user transactions, message driven bean support, and integration with an application server. In addition, there are WebLogic JMS extensions and features have a direct impact on performance.

For more information on designing your applications for JMS, see "Best Practices for Application Design" in *Developing JMS Applications for Oracle WebLogic Server*.

Tuning Applications Using Unit-of-Order

Message Unit-of-Order is a WebLogic Server value-added feature that enables a stand-alone message producer, or a group of producers acting as one, to group messages into a single unit with respect to the processing order (a sub-ordering). This single unit is called a Unit-of-Order (or UOO) and requires that all messages from that unit be processed sequentially in the order they were created. UOO replaces the following complex design patterns:

- A dedicated consumer with a unique selector per each sub-ordering
- A new destination per sub-ordering, one consumer per destination.

See "Using Message Unit-of-Order" in *Developing JMS Applications for Oracle WebLogic Server*.

Using UOO and Distributed Destinations

To ensure strict ordering when using distributed destinations, each different UOO is pinned to a specific physical destination instance. There are two options for automatically determining the correct physical destination for a given UOO:

- Hashing – Is generally faster and the UOO setting. Hashing works by using a hash function on the UOO name to determine the physical destination. It has the following drawbacks:
  - It doesn't correctly handle the administrative deleting or adding physical destinations to a distributed destination.
  - If a UOO hashes to an unavailable destination, the message send fails.
- Path Service – Is a single server UOO directory service that maps the physical destination for each UOO. The Path Service is generally slower than hashing if there are many differently named UOO created per second. In this situation, each new UOO name implicitly forces a check of the path service before sending the message. If the number of UOOs created per second is limited, Path Service performance is not an issue as the UOO paths are cached throughout the cluster.

Migrating Old Applications to Use UOO

For releases prior to WebLogic Server 9.0, applications that had strict message ordering requirements were required to do the following:

- Use a single physical destination with a single consumer
- Ensure the maximum asynchronous consumer message backlog (The `MessagesMaximum` parameter on the connection factory) was set to a value of 1.

UOO relaxes these requirements significantly as it allows for multiple consumers and allows for a asynchronous consumer message backlog of any size. To migrate older applications to take advantage of UOO, simply configure a default UOO name on the physical destination. See "Ordered Redelivery of Messages" in *Developing JMS Applications for Oracle WebLogic Server*.

Using One-Way Message Sends

One-way message sends can greatly improve the performance of applications that are bottle-necked by senders, but do so at the risk of introducing a lower QOS (quality-of-service). Typical message sends from a JMS producer are termed *two-way sends* because they include both an internal request *and* an internal response. When an producer application calls `send()`, the call generates a request that contains the application's message and then waits for a response from the JMS server to confirm its receipt of the message. This call-and-response mechanism regulates the producer, since the producer is forced to wait for the JMS server's response before the application can make another send call. Eliminating the response message eliminates this wait, and yields a *one-way send*. WebLogic Server supports a configurable one-way send option for non-persistent, non-transactional messaging; no application code changes are required to leverage this feature.

By enabling the One-Way Send Mode options, you allow message producers created by a user-defined connection factory to do one-way message sends, when possible. When active, the associated producers can send messages without internally waiting for a response from the target destination's host JMS server. You can choose to allow queue senders and topic publishers to do one-way sends, or to limit this capability to topic publishers only. You must also specify a One-Way Window Size to determine when a two-way message is required to regulate the producer before it can continue making additional one-way sends.

Configure One-Way Sends On a Connection Factory

You configure one-way message send parameters on a connection factory by using the WebLogic Server Administration Console.

Note:

One-way message sends are disabled if your connection factory is configured with "XA Enabled". This setting disables one-way sends whether or not the sender actually uses transactions.

One-Way Send Support In a Cluster With a Single Destination

To ensure one-way send support in a cluster with a single destination, verify that the connection factory and the JMS server hosting the destination are targeted to the same WebLogic server. The connection factory must not be targeted to any other WebLogic Server instances in the cluster.

One-Way Send Support In a Cluster With Multiple Destinations

To ensure one-way send support in a cluster with multiple destinations that share the same name, special care is required to ensure the WebLogic Server instance that hosts the client connection also hosts the destination. One solution is the following:

1. Configure the cluster wide RMI load balancing algorithm to "Server Affinity".
2. Ensure that no two destinations are hosted on the same WebLogic Server instance.
3. Configure each destination to have the same *local-jndi-name*.

4. Configure a connection factory that is targeted to only those WebLogic Server instances that host the destinations.
5. Ensure sender clients use the JNDI names configured in Steps 3 and 4 to obtain their destination and connection factory from their JNDI context.
6. Ensure sender clients use URLs limited to only those WebLogic Server instances that host the destinations in Step 3.

This solution disables RMI-level load balancing for clustered RMI objects, which includes EJB homes and JMS connection factories. Effectively, the client will obtain a connection and destination based only on the network address used to establish the JNDI context. Load balancing can be achieved by leveraging *network load balancing*, which occurs for URLs that include a comma-separated list of WebLogic Server addresses, or for URLs that specify a DNS name that resolves to a `round-robin` set of IP addresses (as configured by a network administrator).

For more information on Server Affinity for clusters, see "Load Balancing for EJBs and RMI Objects" in *Administering Clusters for Oracle WebLogic Server*.

When One-Way Sends Are Not Supported

This section defines when one-way sends are *not* supported. When one-ways are not supported, the send QOS is automatically upgraded to standard two-ways.

Different Client and Destination Hosts

One-way sends are supported when the client producer's connection host and the JMS server hosting the target destination are the same WebLogic Server instance; otherwise, the one-way mode setting will ignored and standard two-way sends will be used instead.

XA Enabled On Client's Host Connection Factory

One-way message sends are disabled if the client's host connection factory is configured with *XA Enabled*. This setting disables one-way sends whether or not the sender actually uses transactions.

Higher QOS Detected

When the following higher QOS features are detected, then the one-way mode setting will be ignored and standard two-way sends will be used instead:

- XA
- Transacted sessions
- Persistent messaging
- Unit-of-order
- Unit-of-work
- Distributed destinations

Destination Quota Exceeded

When the specified quota is exceeded on the targeted destination, then standard two-way sends will be used until the quota clears.

One-way messages that exceed quota are silently deleted, without immediately throwing exceptions back to the client. The client will eventually get a quota exception if the destination is still over quota at the time the next two-way send occurs. (Even in one-way mode, clients will send a two-way message every *One Way Send Window Size* number of messages configured on the client's connection factory.)

A workaround that helps avoid silently-deleted messages during quota conditions is to increase the value of the Blocking Send Timeout configured on the connection factory, as described in Compressing Messages. The one-way messages will not be deleted immediately, but instead will optimistically wait on the JMS server for the specified time until the quota condition clears (presumably due to messages getting consumed or by messages expiring). The client sender will not block until it sends a two-way message. For each client, no more than One Way Window Size messages will accumulate on the server waiting for quota conditions to clear.

Change In Server Security Policy

A change in the server-side security policy could prevent one-way message sends without notifying the JMS client of the change in security status.

Change In JMS Server or Destination Status

One-way sends can be disabled when a host JMS server or target destination is administratively undeployed, or when message production is paused on either the JMS server or the target destination using the "Production Pause/Resume" feature. See "Production Pause and Production Resume" in *Administering JMS Resources for Oracle WebLogic Server*.

Looking Up Logical Distributed Destination Name

One-way message sends work with distributed destinations provided the client looks up the physical distributed destination members directly rather than using the logical distributed destination's name. See "Using Distributed Destinations" in *Developing JMS Applications for Oracle WebLogic Server*.

One-Way Send QOS Guidelines

Use the following QOS-related guidelines when using the one-way send mode for typical non-persistent messaging.

- When used in conjunction with the Blocking Sends feature, then using one-way sends on a well-running system should achieve similar QOS as when using the two-way send mode.
- One-way send mode for topic publishers falls within the QOS guidelines set by the JMS Specification, but does entail a lower QOS than two-way mode (the WebLogic Server default mode).
- One-way send mode may not improve performance if JMS consumer applications are a system bottleneck, as described in "Asynchronous vs. Synchronous Consumers" in *Developing JMS Applications for Oracle WebLogic Server*.

- Consider enlarging the JVM's heap size on the client and/or server to account for increased batch size (the Window) of sends. The potential memory usage is proportioned to the size of the configured Window and the number of senders.
- The sending application will not receive all quota exceptions. One-way messages that exceed quota are silently deleted, without throwing exceptions back to the sending client. See Destination Quota Exceeded for more information and a possible work around.
- Configuring one-way sends on a connection factory effectively disables any message flow control parameters configured on the connection factory.
- By default, the One-way Window Size is set to "1", which effectively disables one-way sends as every one-way message will be upgraded to a two-way send. (Even in one-way mode, clients will send a two-way message every *One Way Send Window Size* number of messages configured on the client's connection factory.) Therefore, you must set the one-way send window size much higher. It is recommended to try setting the window size to "300" and then adjust it according to your application requirements.

Tuning the Messaging Performance Preference Option

The Messaging Performance Preference tuning option on JMS destinations enables you to control how long a destination should wait (if at all) before creating full batches of available messages for delivery to consumers. At the minimum value, batching is disabled. Tuning above the default value increases the amount of time a destination is willing to wait before batching available messages. The maximum message count of a full batch is controlled by the JMS connection factory's Messages Maximum per Session setting.

Using the WebLogic Server Administration Console, this *advanced* option is available on the General Configuration page for both standalone and uniform distributed destinations, as well as for JMS templates.

Specifically, JMS destinations include internal algorithms that attempt to automatically optimize performance by grouping messages into batches for delivery to consumers. In response to changes in message rate and other factors, these algorithms change batch sizes and delivery times. However, it isn't possible for the algorithms to optimize performance for every messaging environment. The Messaging Performance Preference tuning option enables you to modify how these algorithms react to changes in message rate and other factors so that you can fine-tune the performance of your system.

Messaging Performance Configuration Parameters

The Message Performance Preference option includes the following configuration parameters:

Table 13-4 Message Performance Preference Values

Table 13-4 (Cont.) Message Performance Preference Values

WebLogic Server Administration Console Value	MBean Value	Description
Do Not Batch Messages	0	Effectively disables message batching. Available messages are promptly delivered to consumers. This is equivalent to setting the value of the connection factory's Messages Maximum per Session field to "1".
Batch Messages Without Waiting	25 (default)	Less-than-full batches are immediately delivered with available messages. This is equivalent to the value set on the connection factory's Messages Maximum per Session field.
Low Waiting Threshold for Message Batching	50	Wait briefly before less-than-full batches are delivered with available messages.
Medium Waiting Threshold for Message Batching	75	Possibly wait longer before less-than-full batches are delivered with available messages.
High Waiting Threshold for Message Batching	100	Possibly wait even longer before less-than-full batches are delivered with available messages.

It may take some experimentation to find out which value works best for your system. For example, if you have a queue with many concurrent message consumers, by selecting the WebLogic Server Administration Console's Do Not Batch Messages value, the queue will make every effort to promptly push messages out to its consumers as soon as they are available. Conversely, if you have a queue with only one message consumer that doesn't require fast response times, by selecting the console's High Waiting Threshold for Message Batching value, then the queue will strongly attempt to only push messages to that consumer in batches, which will increase the waiting period but may improve the server's overall throughput by reducing the number of sends.

Compatibility With the Asynchronous Message Pipeline

The Message Performance Preference option is compatible with asynchronous consumers using the Asynchronous Message Pipeline, and is also compatible with synchronous consumers that use the Prefetch Mode for Synchronous Consumers feature, which simulates the Asynchronous Message Pipeline. However, if the value of the Maximum Messages value is set too low, it may negate the impact of the destination's higher-level performance algorithms (e.g., Low, Medium, and High Waiting Threshold for Message Batching). For more information on the Asynchronous Message Pipeline, see "Receiving Messages" in *Developing JMS Applications for Oracle WebLogic Server*.

Client-side Thread Pools

With most java client side applications, the default client thread pool size of 5 threads is sufficient. If, however, the application has a large number of asynchronous consumers, then it is often beneficial to allocate slightly more threads than asynchronous consumers. This allows more asynchronous consumers to run concurrently.

WebLogic client thread pools are configured differently than WebLogic server thread-pools, and are not self tuning. WebLogic clients have a specific thread pool that is used for handling incoming requests from the server, such as JMS MessageListener invocations. This pool can be configured via the command-line property:

```
-Dweblogic.ThreadPoolSize=n
```

where n is the number of threads

You can force a client-side thread dump to verify that this setting is taking effect.

Considerations for Oracle Data Guard Environments

The following sections provide configuration considerations for a WebLogic JMS environment that includes Oracle Data Guard.

- Pause Destinations for Planned Down Time
- Migrate JMS Services for Unexpected Outages

For more information on Oracle Data Guard, see http://www.oracle.com/us/products/database/options/active-data-guard/overview/index.html.

Pause Destinations for Planned Down Time

For planned maintenance windows, pause the impacted JMS destinations before initiating the switch from the production database instance to standby instance. When the standby database has transitioned to production, resume the JMS destinations.

Migrate JMS Services for Unexpected Outages

For unexpected service outages, implement JMS Service migration with the **Restart on Failure** option. Should the amount of time required to switch from the production to standby database exceed the value of the Store `IORetryDelaySeconds` attribute and the JMS Services fails, the JMS service and associated store are restarted in-place. See "In-Place Restarting of Failed Migratable Services" in *Administering Clusters for Oracle WebLogic Server*.

14

Tuning WebLogic JMS Store-and-Forward

WebLogic JMS provides advanced Store-and-Forwarding (SAF) capability for high-performance message forwarding from a local server instance to a remote JMS destination. Get the best performance from SAF applications by following the recommendations described in this chapter.

This chapter includes the following sections:

- Best Practices for JMS SAF
- Tuning Tips for JMS SAF

For more information about SAF, see "Understanding the Store-and-Forward Service" in *Administering the Store-and-Forward Service for Oracle WebLogic Server*.

Best Practices for JMS SAF

- Avoid using SAF if remote destinations are already highly available. JMS clients can send directly to remote destinations. Use SAF in situations where remote destinations are not highly available, such as an unreliable network or different maintenance schedules.
- Use the better performing JMS SAF feature instead of using a Messaging Bridge when forwarding messages to remote destinations. In general, a JMS SAF agent is significantly faster than a Messaging Bridge. One exception is a configuration when sending messages in a non-persistent exactly-once mode.

  Note:

  A Messaging Bridge is still required to store-and-forward messages to foreign destinations and destinations from releases prior to WebLogic 9.0.

- Configure separate SAF Agents for JMS SAF and Web Services Reliable Messaging Agents (WS-RM) to simplify administration and tuning.
- Sharing the same WebLogic Store between subsystems provides increased performance for subsystems requiring persistence. For example, transactions that include SAF and JMS operations, transactions that include multiple SAF destinations, and transactions that include SAF and EJBs. See Tuning the WebLogic Persistent Store.

Tuning Tips for JMS SAF

- Target imported destinations to multiple SAF agents to load balance message sends among available SAF agents.

- Consider using a separate remote SAF context for each SAF destination for better performance. SAF destinations that use the same remote SAF context are typically single threaded.
- Increase the JMS SAF `Window Size` for applications that handle small messages. By default, a JMS SAF agent forwards messages in batches that contain up to 10 messages. For small messages size, it is possible to double or triple performance by increasing the number of messages in each batch to be forwarded. A more appropriate initial value for `Window Size` for small messages is 100. You can then optimize this value for your environment.

  Changing the `Window Size` for applications handling large message sizes is not likely to increase performance and is not recommended. `Window Size` also tunes WS-RM SAF behavior, so it may not be appropriate to tune this parameter for SAF Agents of type `Both`.

  Note:

  For a distributed queue, `WindowSize` is ignored and the batch size is set internally at 1 message.

- Increase the JMS SAF `Window Interval`. By default, a JMS SAF agent has a `Window Interval` value of 0 which forwards messages as soon as they arrive. This can lower performance as it can make the effective `Window size` much smaller than the configured value. A more appropriate initial value for `Window Interval` value is 500 milliseconds. You can then optimize this value for your environment. In this context, small messages are less than a few K, while large messages are on the order of tens of K.

  Changing the `Window Interval` improves performance only in cases where the forwarder is already able to forward messages as fast as they arrive. In this case, instead of immediately forwarding newly arrived messages, the forwarder pauses to accumulate more messages and forward them as a batch. The resulting larger batch size improves forwarding throughput and reduces overall system disk and CPU usage at the expense of increasing latency.

  Note:

  For a distributed queue, `Window Interval` is ignored.

- Set the `Non-Persistent QOS` value to `At-Least-Once` for imported destinations if your application can tolerate duplicate messages.

15 Tuning WebLogic Message Bridge

Improve message bridge performance using the methods described in this chapter.

This chapter includes the following sections:

Best Practices

- Avoid using a Messaging Bridge if remote destinations are already highly available. JMS clients can send directly to remote destinations. Use messaging bridge in situations where remote destinations are not highly available, such as an unreliable network or different maintenance schedules.
- Use the better performing JMS SAF feature instead of using a Messaging Bridge when forwarding messages to remote destinations. In general, a JMS SAF agent is significantly faster than a Messaging Bridge. One exception is a configuration when sending messages in a non-persistent exactly-once mode.

Note:

A Messaging Bridge is still required to store-and-forward messages to foreign destinations and destinations from releases prior to WebLogic 9.0.

Changing the Batch Size

When the `Asynchronous Mode Enabled` attribute is set to false and the quality of service is `Exactly-once`, the `Batch Size` attribute can be used to reduce the number of transaction commits by increasing the number of messages per transaction (batch). The best batch size for a bridge instance depends on the combination of JMS providers used, the hardware, operating system, and other factors in the application environment.

Changing the Batch Interval

When the `Asynchronous Mode Enabled` attribute is set to false and the quality of service is `Exactly-once`, the `BatchInterval` attribute is used to adjust the amount of time the bridge waits for each batch to fill before forwarding batched messages. The best batch interval for a bridge instance depends on the combination of JMS providers used, the hardware, operating system, and other factors in the application environment. For example, if the queue is not very busy, the bridge may frequently stop forwarding in order to wait batches to fill, indicating the need to reduce the value of the `BatchInterval` attribute.

Changing the Quality of Service

An `Exactly-once` quality of service may perform significantly better or worse than `At-most-once` and `Duplicate-okay`.

When the `Exactly-once` quality of service is used, the bridge must undergo a two-phase commit with both JMS servers in order to ensure the transaction semantics and this operation can be very expensive. However, unlike the other qualities of service, the bridge can batch multiple operations together using `Exactly-once` service.

You may need to experiment with this parameter to get the best possible performance. For example, if the queue is not very busy or if non-persistent messages are used, `Exactly-once` batching may be of little benefit.

Using Multiple Bridge Instances

If message ordering is not required, consider deploying multiple bridges.

Multiple instances of the bridge may be deployed using the same destinations. When this is done, each instance of the bridge runs in parallel and message throughput may improve. If multiple bridge instances are used, messages will not be forwarded in the same order they had in the source destination.

Consider the following factors when deciding whether to use multiple bridges:

- Some JMS products do not seem to benefit much from using multiple bridges
- WebLogic JMS messaging performance typically improves significantly, especially when handling persistent messages.
- If the CPU or disk storage is already saturated, increasing the number of bridge instances may decrease throughput.

Changing the Thread Pool Size

A general bridge configuration rule is to provide a thread for each bridge instance targeted to a server instance. Use one of the following options to ensure that an adequate number of threads is available for your environment:

- Use the common thread pool—A server instance changes its thread pool size automatically to maximize throughput, including compensating for the number of bridge instances configured. See "Understanding How WebLogic Server Uses Thread Pools" in *Administering Server Environments for Oracle WebLogic Server*.

- Configure a work manager for the `weblogic.jms.MessagingBridge` class. See "Understanding Work Managers" in *Administering Server Environments for Oracle WebLogic Server*.
- Use the WebLogic Server Administration Console to set the `Thread Pool Size` property in the Messaging Bridge Configuration section on the *Configuration: Services* page for a server instance. To avoid competing with the default execute thread pool in the server, messaging bridges share a separate thread pool. This thread pool is used only in synchronous mode (`Asynchronout Mode Enabled` is not set). In asynchronous mode the bridge runs in a thread created by the JMS provider for the source destination. Deprecated in WebLogic Server 9.0.
- Ensure that the bridge resource adapter pool is twice as large as the number of bridges. See "Resource Adapters" in *Administering the WebLogic Messaging Bridge for Oracle WebLogic Server*.

Avoiding Durable Subscriptions

If the bridge is listening on a topic and it is acceptable that messages are lost when the bridge is not forwarding messages, disable the `Durability Enabled` flag to ensure undeliverable messages do not accumulate in the source server's store. Disabling the flag also makes the messages non-persistent.

Co-locating Bridges with Their Source or Target Destination

If a messaging bridge source or target is a WebLogic destination, deploy the bridge to the same WebLogic server as the destination. Targeting a messaging bridge with one of its destinations eliminates associated network and serialization overhead. Such overhead can be significant in high-throughput applications, particularly if the messages are non-persistent.

Changing the Asynchronous Mode Enabled Attribute

The `Asynchronous Mode Enabled` attribute determines whether the messaging bridge receives messages asynchronously using the `JMS MessageListener` interface at `http://docs.oracle.com/javaee/5/api/javax/jms/MessageListener.html`, or whether the bridge receives messages using the synchronous JMS APIs. In most situations, the `Asynchronous Enabled` attributes value is dependent on the QOS required for the application environment as shown in Table 15-1:

Table 15-1 Asynchronous Mode Enabled Values for QOS Level

QOS	Asynchronous Mode Enabled Attribute value
`Exactly-once`[1]	false
`At-least-once`	true
`At-most-once`	true

[1] If the source destination is a non-WebLogic JMS provider and the QOS is Exactly-once, then the Asynchronous Mode Enabled attribute is disabled and the messages are processed in synchronous mode.

A quality of service of `Exactly-once` has a significant effect on bridge performance. The bridge starts a new transaction for each message and performs a two-phase commit across both JMS servers involved in the transaction. Since the two-phase commit is usually the most expensive part of the bridge transaction, as the number of messages being processed increases, the bridge performance tends to decrease.

Tuning Environments with Many Bridges

This section provides information on how to improve system boot time and general performance of systems that deploy many bridge instances.

- Modify the capacity of the connection factory associated with each resource adaptor by adjusting the `max-capacity` attribute in the `weblogic-ra.xml` descriptor file. The value of the `max-capacity` attribute should be at least two times the number of bridge instances.

  For example, if your environment has up to ten message bridge instances targeted, a `max-capacity` attribute setting of 20 in the default configuration is adequate. But if you increase the number of bridge instances to 15, increase the `max-capacity` attribute to 30. See "Setting the Number of Connection Factories" in *Administering the WebLogic Messaging Bridge for Oracle WebLogic Server*.

- Increase the entire server's thread pool size to something somewhat higher than the number of active bridges. This applies for any XA (transactional) bridge with a batch size higher than 1, or any XA bridge that consumes from a source destination hosted by a non-WebLogic JMS provider.

  For example, pass the following on the command line if you have 90 message bridges:

  ```
  -Dweblogic.threadpool.MinPoolSize=100
  ```

  This ensures there are enough threads available when affected bridges initialize. If there are not enough threads available, there can be a multi-second delay until a new thread is created.

- Provide a thread for each bridge instance targeted to a server instance. See Changing the Thread Pool Size.

16 Tuning Resource Adapters

Tune resource adapters by using the best practices provided in this chapter.

This chapter includes the following sections:

Classloading Optimizations for Resource Adapters

You can package resource adapter classes in one or more JAR files, and then place the JAR files in the RAR file. These are called nested JARs. When you nest JAR files in the RAR file, and classes need to be loaded by the classloader, the JARs within the RAR file must be opened and closed and iterated through for each class that must be loaded.

If there are very few JARs in the RAR file and if the JARs are relatively small in size, there will be no significant performance impact. On the other hand, if there are many JARs and the JARs are large in size, the performance impact can be great.

To avoid such performance issues, you can either:

1. Deploy the resource adapter in an exploded format. This eliminates the nesting of JARs and hence reduces the performance hit involved in looking for classes.
2. If deploying the resource adapter in exploded format is not an option, the JARs can be exploded within the RAR file. This also eliminates the nesting of JARs and thus improves the performance of classloading significantly.

Connection Optimizations

Oracle recommends that resource adapters implement the optional enhancements described in Connection Optimization section of the J2CA 1.6 Specification at`https://jcp.org/aboutJava/communityprocess/final/jsr322/index.html`. Implementing these interfaces allows WebLogic Server to provide several features that will not be available without them.

Lazy Connection Association allows the server to automatically clean up unused connections and prevent applications from hogging resources. Lazy Transaction Enlistment allows applications to start a transaction after a connection is already opened.

Thread Management

Resource adapter implementations should use the `WorkManager` (as described in Chapter 10, "Work Management" in the J2CA 1.6 Specification at `https://jcp.org/aboutJava/communityprocess/final/jsr322/index.html`) to launch operations that need to run in a new thread, rather than creating new threads directly. This allows WebLogic Server to manage and monitor these threads.

InteractionSpec Interface

WebLogic Server supports the Common Client Interface (CCI) for EIS access, as defined in Chapter 17, "Common Client Interface" in the J2CA 1.5 Specification at `http://www.oracle.com/technetwork/java/index.html`. The CCI defines a standard client API for application components that enables application components and EAI frameworks to drive interactions across heterogeneous EISes.

As a best practice, you should not store the `InteractionSpec` class that the CCI resource adapter is required to implement in the RAR file. Instead, you should package it in a separate JAR file outside of the RAR file, so that the client can access it without having to put the `InteractionSpec` interface class in the generic CLASSPATH.

With respect to the `InteractionSpec` interface, it is important to remember that when all application components (EJBs, resource adapters, Web applications) are packaged in an EAR file, all common classes can be placed in the APP-INF/lib directory. This is the easiest possible scenario.

This is not the case for standalone resource adapters (packaged as RAR files). If the interface is serializable (as is the case with `InteractionSpec`), then both the client and the resource adapter need access to the `InteractionSpec` interface as well as the implementation classes. However, if the interface extends `java.io.Remote`, then the client only needs access to the interface class.

17 Tuning Web Applications

Use the best practices for tuning Web applications and managing sessions described in this chapter.

This chapter includes the following sections:

- Best Practices
- Session Management
- Pub-Sub Tuning Guidelines

Best Practices

- Disable Page Checks
- Use Custom JSP Tags
- Precompile JSPs
- Use HTML Template Compression
- Use Service Level Agreements
- Related Reading

Disable Page Checks

You can improve performance by disabling servlet and JDP page checks. Set each of the following parameters to -1:

- `pageCheckSeconds`
- `servlet-reload-check-secs`
- `servlet Reload Check`

These are default values for production mode.

Use Custom JSP Tags

Oracle provides three specialized JSP tags that you can use in your JSP pages: cache, repeat, and process. These tags are packaged in a tag library jar file called `weblogic-tags.jar`. This jar file contains classes for the tags and a tag library descriptor (TLD). To use these tags, you copy this jar file to the Web application that contains your JSPs and reference the tag library in your JSP. See "Using Custom WebLogic JSP Tags (cache, process, repeat)" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

Precompile JSPs

You can configure WebLogic Server to precompile your JSPs when a Web Application is deployed or re-deployed or when WebLogic Server starts up by setting the precompile parameter to true in the `jsp-descriptor` element of the `weblogic.xml` deployment descriptor. To avoid recompiling your JSPs each time the server restarts and when you target additional servers, precompile them using `weblogic.appc` and place them in the WEB-INF/classes folder and archive them in a `.war` file. Keeping your source files in a separate directory from the archived `.war` file eliminates the possibility of errors caused by a JSP having a dependency on one of the class files. For a complete explanation on how to avoid JSP recompilation, see "*Avoiding Unnecessary JSP Compilation"* at `http://www.oracle.com/technetwork/articles/entarch/jsp-reloaded-101726.html`.

Use HTML Template Compression

Using the `compress-html-template` element compresses the HTML in the JSP template blocks which can improve runtime performance. If the JSP's HTML template block contains the `<pre>` HTML tag, do not enable this feature.

See `jsp-descriptor` in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

Use Service Level Agreements

You should assign servlets and JSPs to work managers based on the service level agreements required by your applications. See Thread Management .

Related Reading

- "Servlet Best Practices" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.
- "*Servlet and JSP Performance Tuning*" at `http://www.javaworld.com/javaworld/jw-06-2004/jw-0628-performance_p.html`, by Rahul Chaudhary, JavaWorld, June 2004.

Session Management

As a general rule, you should optimize your application so that it does as little work as possible when handling session persistence and sessions. The following sections provide information on how to design a session management strategy that suits your environment and application:

- Managing Session Persistence
- Minimizing Sessions
- Aggregating Session Data

Managing Session Persistence

WebLogic Server offers many session persistence mechanisms that cater to the differing requirements of your application, including `Async-replicated` and `Async-JDBC` modes. The session persistence mechanisms are configurable at the Web application layer. Which session management strategy you choose for your application depends on real-world factors like HTTP session size, session life cycle, reliability, and session failover requirements. For example, a Web application with no failover requirements could be maintained as a single memory-based session; whereas, a Web application with session fail-over requirements could be maintained as replicated sessions or JDBC-based sessions, based on their life cycle and object size.

In terms of pure performance, replicated session persistence is a better overall choice when compared to JDBC-based persistence for session state. However, replicated-based session persistence requires the use of WebLogic clustering, so it isn't an option in a single-server environment.

On the other hand, an environment using JDBC-based persistence does not require the use of WebLogic clusters and can maintain the session state for longer periods of time in the database. One way to improve JDBC-based session persistence is to optimize your code so that it has as high a granularity for session state persistence as possible. Other factors that can improve the overall performance of JDBC-based session persistence are: the choice of database, proper database server configuration, JDBC driver, and the JDBC connection pool configuration.

For more information on managing session persistence, see:

- "Configuring Session Persistence" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*
- "HTTP Session State Replication" in *Administering Clusters for Oracle WebLogic Server*
- "Using a Database for Persistent Storage (JDBC Persistence)" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*

Minimizing Sessions

Configuring how WebLogic Server manages sessions is a key part of tuning your application for best performance. Consider the following:

- Use of sessions involves a scalability trade-off.
- Use sessions sparingly. In other words, use sessions only for state that cannot realistically be kept on the client or if URL rewriting support is required. For example, keep simple bits of state, such as a user's name, directly in cookies. You can also write a wrapper class to "get" and "set" these cookies, in order to simplify the work of servlet developers working on the same project.
- Keep frequently used values in local variables.

For more information, see "Setting Up Session Management" in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

Aggregating Session Data

This section provides best practices on how to aggregate session data. WebLogic Server tracks and replicates changes in the session by attribute so you should:

- Aggregate session data that changes in tandem into a single session attribute.
- Aggregate session data that changes frequently and read-only session data into separate session attributes

For example: If you use a a single large attribute that contains all the session data and only 10% of that data changes, the entire attribute has to be replicated. This causes unnecessary serialization/deserialization and network overhead. You should move the 10% of the session data that changes into a separate attribute.

Pub-Sub Tuning Guidelines

The following section provides general tuning guidelines for a pub-sub server:

- Increase file descriptors to cater for a large number of long-living connections, especially for applications with thousands of clients.
- Tune logging level for WebLogic Server.
- Tune JVM options. Suggested options: `-Xms1536m -Xmx1536m -Xns512m -XXtlaSize:min=128k,preferred=256k`
- Increase the maximum message. If your application publishes messages under high volumes, consider setting the value to `<max-message-size>10000000</max-message-size>`.

18 Tuning Web Services

Use Oracle best practices for designing, developing, and deploying WebLogic Web Services applications and application resources.

This chapter includes the following sections:

- Web Services Best Practices
- Tuning Web Service Reliable Messaging Agents
- Tuning Heavily Loaded Systems to Improve Web Service Performance

Web Services Best Practices

Design and architectural decisions have a strong impact on runtime performance and scalability of Web Service applications. Here are few key recommendations to achieve best performance.

- Design Web Service applications for course-grained service with moderate size payloads.
- Choose correct service-style & encoding for your Web service application.
- Control serializer overheads and namespaces declarations to achieve better performance.
- Use MTOM/XOP or Fast Infoset to optimizing the format of a SOAP message.
- Carefully design SOAP attachments and security implementations for minimum performance overheads.
- Consider using an asynchronous messaging model for applications with:
  - Slow and unreliable transport.
  - Complex and long-running process.
- For transactional Service Oriented Architectures (SOA) consider using the Last Logging Resource transaction optimization (LLR) to improve performance. See Tuning Transactions.
- Use replication and caching of data and schema definitions to improve performance by minimizing network overhead.
- Consider any XML compression technique only when XML compression/ decompression overheads are less than network overheads involved.
- Applications that are heavy users of XML functionality (parsers) may encounter performance issues or run out of file descriptors. This may occur because XML

parser instances are bootstrapped by doing a lookup in the `jaxp.properties` file (JAXP API). Oracle recommends setting the properties on the command line to avoid unnecessary file operations at runtime and improve performance and resource usage.

- Follow "JWS Programming Best Practices" in *Developing JAX-WS Web Services for Oracle WebLogic Server*.
- Follow best practice and tuning recommendations for all underlying components, such as Tuning WebLogic Server EJBs, Tuning Web Applications, Tuning Data Sources, and Tuning WebLogic JMS.

Tuning Web Service Reliable Messaging Agents

Web Service Reliable Messaging provides advanced store-and-forward capability for high-performance message forwarding from a local server instance to a remote destination. See "Understanding the Store-and-Forward Service" in *Administering the Store-and-Forward Service for Oracle WebLogic Server*. The following section provides information on how to get the best performance from Store-and-Forward (SAF) applications:

- Configure separate SAF Agents for JMS SAF and Web Services Reliable Messaging Agents to simplify administration and tuning.
- Sharing the same WebLogic Store between subsystems provides increased performance for subsystems requiring persistence. For example, transactions that include SAF and JMS operations, transactions that include multiple SAF destinations, and transactions that include SAF and EJBs. See Tuning the WebLogic Persistent Store.
- Consider increasing the `WindowSize` parameter on the remote SAF agent. For small messages of less than 1K, tuning `WindowSize` as high as 300 can improve throughput.

Note:

`WindowSize` also tunes JMS SAF behavior, so it may not be appropriate to tune this parameter for SAF agents of type `both`.

- Ensure that `retry delay` is not set too low. This may cause the system to make unnecessary delivery attempts.

Tuning Heavily Loaded Systems to Improve Web Service Performance

The asynchronous request-response, reliable messaging, and buffering features are all pre-tuned for minimum system resource usage to support a small number of clients (under 10). If you plan on supporting a larger number of clients or high message volumes, you should adjust the tuning parameters to accommodate the additional load, as described in the following sections:

- Setting the Work Manager Thread Pool Minimum Size Constraint
- Setting the Buffering Sessions
- Releasing Asynchronous Resources

Setting the Work Manager Thread Pool Minimum Size Constraint

Define a Work Manager and set the thread pool minimum size constraint (`min-threads-constraint`) to a value that is at least as large as the expected number of concurrent requests or responses into the service.

For example, if a Web service client issues 20 requests in rapid succession, the recommended thread pool minimum size constraint value would be 20 for the application hosting the client. If the configured constraint value is too small, performance can be severely degraded as incoming work waits for a free processing thread.

For more information about the thread pool minimum size constraint, see "Constraints" in *Administering Server Environments for Oracle WebLogic Server*.

Setting the Buffering Sessions

The reliable messaging and buffering features use JMS queue sessions to send messages to the reliability/buffer queues. By default, WebLogic Server allocates 10 sessions for buffering which enables 10 clients to enqueue messages simultaneously onto the reliability/buffer queue.

For asynchronous request-response, the request and response portion of the communication exchange count separately, as two clients. In this case, the default pool of sessions can support five simultaneous asynchronous request-response clients. To accommodate the number of concurrent clients you expect in your application, set the following parameter to twice the number of expected client threads:

```
-Dweblogic.wsee.buffer.QueueSessionPoolSize=size
```

Releasing Asynchronous Resources

When using the asynchronous request-response feature, WebLogic Server persistently stores information about the request until the asynchronous response is returned to the client. These resources remain in the persistent store until they are released by a background thread, called the *store cleaner*.

Often, these resources can be released sooner. Executing the store cleaner more frequently can help to reduce the size of the persistent store and minimize the time required to clean it.

By default, the store cleaner runs every two minutes (120000 ms). Oracle recommends that you set the store cleaner interval to one minute (60000 ms) using the following Java system property:

```
-Dweblogic.wsee.StateCleanInterval=60000
```