

Oracle® Fusion Middleware

Tuning Performance of Oracle WebLogic Server



14c (14.1.2.0.0)

F43306-01

December 2024

The Oracle logo, consisting of a solid red square with the word "ORACLE" in white, uppercase, sans-serif font centered within it.

ORACLE®

Oracle Fusion Middleware Tuning Performance of Oracle WebLogic Server, 14c (14.1.2.0.0)

F43306-01

Copyright © 2007, 2024, Oracle and/or its affiliates.

Primary Author: Oracle Corporation

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience	xii
Documentation Accessibility	xii
Diversity and Inclusion	xii
Related Documentation	xii
Conventions	xiii

1 Top Tuning Recommendations for WebLogic Server

Tune Pool Sizes	1-1
Use the Prepared Statement Cache	1-2
Use Logging Last Resource Optimization	1-2
Tune Connection Backlog Buffering	1-2
Use Optimistic or Read-only Concurrency	1-2
Use Local Interfaces	1-2
Use eager-relationship-caching	1-3
Tune HTTP Sessions	1-3
Tune Messaging Applications	1-3

2 Performance Tuning Roadmap and Guidelines

Performance Tuning Roadmap	2-1
Understand Your Performance Objectives	2-1
Measure Your Performance Metrics	2-2
Monitor Disk and CPU Utilization	2-2
Monitor Data Transfers Across the Network	2-3
Locate Bottlenecks in Your System	2-3
Minimize Impact of Bottlenecks	2-3
Tune Your Application	2-3
Tune your DB	2-4
Tune WebLogic Server Performance Parameters	2-4
Tune Your JVM	2-4
Tune the Operating System	2-4
Achieve Performance Objectives	2-4

Tuning Tips	2-4
-------------	-----

3 Tuning Java Virtual Machines (JVMs)

JVM Tuning Considerations	3-1
Changing To a Different JVM	3-2
Garbage Collection	3-2
VM Heap Size and Garbage Collection	3-2
Choosing a Garbage Collection Scheme	3-3
Using Verbose Garbage Collection to Determine Heap Size	3-3
Specifying Heap Size Values	3-4
Tuning Tips for Heap Sizes	3-4
Java HotSpot VM Heap Size Options	3-5
Other Java HotSpot VM Options	3-5
Increasing Java Heap Size for Managed Servers	3-6
Modify the startManagedWebLogic Script to Set Java Heap Size	3-6
Using the Command Line to Set Java Heap Size	3-6
Determining the Memory Values Used by a Managed Server	3-6

4 Tuning WebLogic Diagnostic Framework and Java Flight Recorder Integration

Using Java Flight Recorder	4-1
Using WLDF	4-1
Tuning Considerations	4-2

5 Tuning WebLogic Server

Setting Java Parameters for Starting WebLogic Server	5-2
Development vs. Production Mode Default Tuning Values	5-2
Deployment	5-3
On-demand Deployment of Internal Applications	5-3
Use FastSwap Deployment to Minimize Redeployment Time	5-3
Generic Overrides	5-3
Thread Management	5-3
Tuning a Work Manager	5-4
Self-Tuning Thread Pool Size	5-4
How Many Work Managers are Needed?	5-4
What are the SLA Requirements for Each Work Manager?	5-4
Tuning the Stuck Thread Detection Behavior	5-4
Tuning Network I/O	5-5
Tuning Muxers	5-5

Java Non-Blocking IO (NIO) Muxer	5-5
Native Muxers	5-5
Server Location and Supported Platforms	5-6
Pure-Java Versus Native Socket Reader Implementations	5-7
Configuring Reader Threads for Java Socket Implementation	5-7
Network Channels	5-9
Reducing the Potential for Denial of Service Attacks	5-10
Tuning Message Size	5-10
Tuning Complete Message Timeout	5-10
Tuning Number of File Descriptors	5-11
Tuning Connection Backlog Buffering	5-11
Tuning Cached Connections	5-11
Tuning the Work Manager Maximum Threads Constraint Queue Size	5-11
Optimize Java Expressions	5-12
Using WebLogic Server Clusters to Improve Performance	5-12
Scalability and High Availability	5-13
How to Ensure Scalability for WebLogic Clusters	5-13
Database Bottlenecks	5-14
Session Replication	5-14
Asynchronous HTTP Session Replication	5-14
Invalidation of Entity EJBs	5-15
Invalidation of HTTP sessions	5-16
JNDI Binding, Unbinding and Rebinding	5-16
Running Multiple Server Instances on Multi-Core Machines	5-16
Monitoring a WebLogic Server Domain	5-16
Using the WebLogic Diagnostic Framework	5-16
Using JMX to Monitor WebLogic Server	5-17
Using WLST to Monitor WebLogic Server	5-17
Resources to Monitor WebLogic Server	5-17
Tuning Class and Resource Loading	5-17
Filtering Loader Mechanism	5-17
Class Caching	5-18
SSL Considerations	5-18

6 Tuning the WebLogic Persistent Store

Overview of Persistent Stores	6-1
Using the Default Persistent Store	6-1
Using Custom File Stores and JDBC Stores	6-2
Using a JDBC TLOG Store	6-2
Using JMS Paging Stores	6-2
Using Flash Storage to Page JMS Messages	6-3

Using Diagnostic Stores	6-3
Best Practices When Using Persistent Stores	6-3
Tuning JDBC Stores	6-4
Tuning File Stores	6-4
Basic Tuning Information	6-4
Tuning a File Store Direct-Write-With-Cache Policy	6-6
Using Flash Storage to Increase Performance	6-6
Additional Considerations	6-6
Tuning the File Store Direct-Write Policy	6-7
Tuning the File Store Block Size	6-8
Setting the Block Size for a File Store	6-9
Determining the File Store Block Size	6-9
Determining the File System Block Size	6-9
Converting a Store with Pre-existing Files	6-10
Using a Network File System	6-10
Configuring Synchronous Write Policies	6-10
Test Server Restart Behavior	6-10
Handling NFS Locking Errors	6-11
Solution 1 – Using NFS v4 Instead of NFS v3	6-11
Solution 2 - Copying Data Files to Remove NFS Locks	6-12
Solution 3 - Disabling File Locks in WebLogic Server File Stores	6-13

7 Database Tuning

General Suggestions	7-1
Database-Specific Tuning	7-1
Oracle	7-2
Microsoft SQL Server	7-3

8 Tuning WebLogic Server EJBs

General EJB Tuning Tips	8-1
Tuning EJB Caches	8-2
Tuning the Stateful Session Bean Cache	8-2
Tuning the Entity Bean Cache	8-2
Transaction-Level Caching	8-2
Caching between Transactions	8-3
Ready Bean Caching	8-3
Tuning the Query Cache	8-3
Tuning EJB Pools	8-4
Tuning the Stateless Session Bean Pool	8-4
Tuning the MDB Pool	8-4

Tuning the Entity Bean Pool	8-4
CMP Entity Bean Tuning	8-5
Use Eager Relationship Caching	8-5
Using Inner Joins	8-5
Use JDBC Batch Operations	8-6
Tuned Updates	8-6
Using Field Groups	8-6
include-updates	8-6
call-by-reference	8-7
Bean-level Pessimistic Locking	8-7
Concurrency Strategy	8-7
Tuning In Response to Monitoring Statistics	8-8
Cache Miss Ratio	8-9
Lock Waiter Ratio	8-9
Lock Timeout Ratio	8-9
Pool Miss Ratio	8-10
Destroyed Bean Ratio	8-10
Pool Timeout Ratio	8-11
Transaction Timeout Ratio	8-11

9 Tuning Message-Driven Beans

Use Transaction Batching	9-1
MDB Thread Management	9-2
Determining the Number of Concurrent MDBs	9-2
Selecting a Concurrency Strategy	9-2
Thread Utilization When Using WebLogic Destinations	9-3
Limitations for Multi-threaded Topic MDBs	9-4
Best Practices for Configuring and Deploying MDBs Using Distributed Topics	9-4
Using MDBs with Foreign Destinations	9-4
Concurrency for MDBs that Process Messages from Foreign Destinations	9-5
Thread Utilization for MDBs that Process Messages from Foreign Destinations	9-5
Token-based Message Polling for Transactional MDB Listening on Queues/Topics	9-5
Compatibility for WLS 10.0 and Earlier-style Polling	9-6

10 Tuning Data Sources

Tune the Number of Database Connections	10-1
Waste Not	10-2
Use Test Connections on Reserve with Care	10-3
Cache Prepared and Callable Statements	10-3
Database Listener Timeout under Heavy Server Loads	10-3

Disable Wrapping of Data Type Objects	10-4
Advanced Configurations for Oracle Drivers and Databases	10-4
Use Best Design Practices	10-4

11 Tuning Transactions

Improving Throughput Using XA Transaction Cluster Affinity	11-1
Logging Last Resource Transaction Optimization	11-1
LLR Tuning Guidelines	11-2
Read-only, One-Phase Commit Optimizations	11-2

12 Tuning WebLogic JMS

JMS Performance & Tuning Check List	12-2
Handling Large Message Backlogs	12-3
Improving Message Processing Performance	12-4
Controlling Message Production	12-5
Drawbacks to Controlling Message Production	12-5
Cache and Re-use Client Resources	12-6
Tuning Distributed Queues	12-6
Tuning Topics	12-7
Tuning Non-durable Topic Publishers	12-7
Tuning for Large Messages	12-8
Tuning MessageMaximum	12-8
Tuning MessageMaximum Limitations	12-8
Setting Maximum Message Size for Network Protocols	12-9
Threshold Compression for Remote Producers	12-9
Store Compression	12-10
Selecting a Message Compression Option	12-10
Message Compression for JMS Servers	12-11
Message Compression for Store-and-Forward Sending Agents	12-11
Paging Out Messages To Free Up Memory	12-11
Specifying a Message Paging Directory	12-12
Tuning the Message Buffer Size Option	12-12
Defining Quota	12-13
Quota Resources	12-13
Destination-Level Quota	12-14
JMS Server-Level Quota	12-14
Subscription Message Limits	12-14
Controlling the Flow of Messages on JMS Servers and Destinations	12-15
How Flow Control Works	12-15
Configuring Flow Control	12-16

Flow Control Thresholds	12-17
Tuning Applications Using Unit-of-Order	12-17
Best Practices	12-18
Using UOO and Distributed Destinations	12-18
Using JMS 2.0 Asynchronous Message Sends	12-18
Using One-Way Message Sends	12-20
Configure One-Way Sends On a Connection Factory	12-21
One-Way Send Support In a Cluster With a Single Destination	12-21
One-Way Send Support In a Cluster With Multiple Destinations	12-21
When One-Way Sends Are Not Supported	12-21
Different Client and Destination Hosts	12-22
XA Enabled On Client's Host Connection Factory	12-22
Higher QOS Detected	12-22
Destination Quota Exceeded	12-22
Change In Server Security Policy	12-22
Change In JMS Server or Destination Status	12-23
Looking Up Logical Distributed Destination Name	12-23
Hardware Failure	12-23
One-Way Send QOS Guidelines	12-23
Client-side Thread Pools	12-24
Best Practices for JMS .NET Client Applications	12-24
Considerations for Oracle Data Guard Environments	12-24
Migrate JMS Services for Unexpected Outages	12-25

13 Tuning WebLogic JMS Store-and-Forward

Best Practices for JMS SAF	13-1
Tuning Tips for JMS SAF	13-1

14 Tuning WebLogic Message Bridge

Best Practices	14-1
Changing the Batch Size	14-1
Changing the Batch Interval	14-2
Changing the Quality of Service	14-2
Using Multiple Bridge Instances	14-2
Changing the Thread Pool Size	14-2
Avoiding Durable Subscriptions	14-3
Co-locating Bridges with Their Source or Target Destination	14-3
Changing the Asynchronous Mode Enabled Attribute	14-3
Tuning Environments with Many Bridges	14-4

15 Tuning Resource Adapters

Classloading Optimizations for Resource Adapters	15-1
Connection Optimizations	15-1
Thread Management	15-2
InteractionSpec Interface	15-2

16 Tuning Web Applications

Best Practices	16-1
Disable Page Checks	16-1
Use Custom JSP Tags	16-2
Precompile JSPs	16-2
Use HTML Template Compression	16-2
Use Service Level Agreements	16-2
Related Reading	16-2
Session Management	16-2
Managing Session Persistence	16-3
Minimizing Sessions	16-3
Aggregating Session Data	16-3
Pub-Sub Tuning Guidelines	16-4
Enabling GZIP Compression	16-4

17 Tuning Web Services

Web Services Best Practices	17-1
Tuning Web Service Reliable Messaging Agents	17-2
Tuning Heavily Loaded Systems to Improve Web Service Performance	17-2
Setting the Work Manager Thread Pool Minimum Size Constraint	17-3
Setting the Buffering Sessions	17-3
Releasing Asynchronous Resources	17-3

18 Tuning WebLogic Tuxedo Connector

Configuration Guidelines	18-1
Best Practices	18-1

A Capacity Planning

Capacity Planning Factors	A-1
Programmatic and Web-based Clients	A-2
RMI and Server Traffic	A-3
SSL Connections and Performance	A-3

WebLogic Server Process Load	A-3
Database Server Capacity and User Storage Requirements	A-3
Concurrent Sessions	A-4
Network Load	A-4
Clustered Configurations	A-5
Server Migration	A-5
Application Design	A-5
Assessing Your Application Performance Objectives	A-5
Hardware Tuning	A-5
Benchmarks for Evaluating Performance	A-6
Supported Platforms	A-6
Network Performance	A-6
Determining Network Bandwidth	A-6
Related Information	A-7

Preface

The documentation is intended for administrators who monitor the performance of Oracle WebLogic Server 14.1.2.0.0 and tune the components such as JVMs, EJBs, DBs, Persistent Stores, Data Sources, Messaging Servers, and so on.

- [Audience](#)
- [Documentation Accessibility](#)
- [Diversity and Inclusion](#)
- [Related Documentation](#)
- [Conventions](#)

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.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Access to Oracle Support

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documentation

New and Changed WebLogic Server Features

For a comprehensive listing of the new WebLogic Server features introduced in this release, see *What's New in 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.
<i>italic</i>	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

Top Tuning Recommendations for WebLogic Server

Tuning Oracle WebLogic Server and your WebLogic Server application is a complex and iterative process. To get you started, Oracle recommends various tuning techniques to optimize your application's performance. These tuning techniques are applicable to nearly all WebLogic 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.
- [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.
- [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.
- [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.
- [Use Optimistic or Read-only Concurrency](#)
Use optimistic concurrency with cache-between-transactions or read-only concurrency with query-caching for CMP EJBs to leverage the Entity Bean cache provided by the EJB container.
- [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.
- [Use eager-relationship-caching](#)
Use eager-relationship-caching to allow the EJB container to load related beans using a single SQL statement.
- [Tune HTTP Sessions](#)
Optimize your application so that it does as little work as possible when handling HTTP session persistence and sessions. Also, design a session management strategy that suits your environment and application.
- [Tune Messaging Applications](#)
Oracle provides messaging users a rich set of performance tunables. In general, you should always configure quotas and paging.

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. 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 to 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:

 **Note:**

- In release prior to WebLogic Server 8.1, call-by-reference is turned on by default. For releases of WebLogic Server 8.1 and later, call-by-reference is turned off by default. Older applications migrating to WebLogic Server 8.1 and later 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 to allow 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 its 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 HTTP session persistence and sessions. 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](#)

2

Performance Tuning Roadmap and Guidelines

Use performance tuning roadmap in Oracle WebLogic Server to understand your performance objectives and tune your application environment to optimize performance.

- [Performance Tuning Roadmap](#)
The performance tuning roadmap includes the methods you use to quantify your performance objectives, such as measuring your performance metrics, locating bottlenecks in system, and minimizing the impact of bottlenecks.
- [Tuning Tips](#)
Follow the tuning tips and guidelines when tuning overall system performance.

Performance Tuning Roadmap

The performance tuning roadmap includes the methods you use to quantify your performance objectives, such as measuring your performance metrics, locating bottlenecks in system, and minimizing the impact of bottlenecks.

- [Understand Your Performance Objectives](#)
- [Measure Your Performance Metrics](#)
- [Monitor Disk and CPU Utilization](#)
- [Monitor Data Transfers Across the Network](#)
- [Locate Bottlenecks in Your System](#)
- [Minimize Impact of Bottlenecks](#)
- [Tune Your Application](#)
- [Tune your DB](#)
- [Tune WebLogic Server Performance Parameters](#)
- [Tune Your JVM](#)
- [Tune the Operating System](#)
- [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 is recommended. A good target for non-latency sensitive applications is about 90%.

Performance objectives are limited by constraints, such as

- The configuration of hardware and software such as CPU type, disk size vs. disk speed, sufficient memory.

There is no single formula for determining your hardware requirements. The process of determining what type of hardware and software configuration is required to meet application needs adequately is called capacity planning. Capacity planning requires assessment of your system performance goals and an understanding of your application. Capacity planning for server hardware should focus on maximum performance requirements. See [Capacity Planning](#).

- 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. The following sections provide information on measuring basic performance metrics:

- [Monitor Disk and CPU Utilization](#)
- [Monitor Data Transfers Across the Network](#)

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 bottlenecked. 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 than 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 or RAID, disabling synchronous JMS writes, using JTA direct writes for tlogs, or increasing the HTTP log buffer.

Monitor Data Transfers Across the Network

Check the amount of data transferred between the application and the application server, and between the application server and any remote endpoint. This amount should not exceed your network bandwidth; otherwise, your network becomes the bottleneck.

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](#)
- [Tune the Operating System](#)
- [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.

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](#)
- [Tuning WebLogic Tuxedo Connector](#)

Tune your DB

Your database can be a major enterprise-level bottleneck. Database optimization can be complex and vendor dependent. 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 \(JVMs\)](#).

Tune the Operating System

Tune your operating system according to your operating system documentation based on your application environment.

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

Follow the tuning 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 Jakarta EE performance patterns.
 - Optimize your Java code.

3

Tuning Java Virtual Machines (JVMs)

The Java virtual machine (JVM) in Oracle WebLogic Server 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. Configure the JVM tuning options for WebLogic Server.

- [JVM Tuning Considerations](#)
Examine some general JVM tuning considerations for WebLogic Server.
- [Changing To a Different JVM](#)
When you create a domain, you choose the JVM that you want to run your domain and the Configuration Wizard configures the Oracle start scripts based on your choice. Modify the values for the `JAVA_HOME` and `JAVA_VENDOR` variables in the Configuration Wizard to change the JVM.
- [Garbage Collection](#)
Garbage collection is the VM's process of freeing up unused Java objects in the Java heap.
- [Increasing Java Heap Size for Managed Servers](#)
For better performance, increase the heap size for each Managed Server in your environment.

JVM Tuning Considerations

Examine some general JVM tuning considerations for WebLogic Server.

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

Table 3-1 General JVM Tuning Considerations

Tuning Factor	Information Reference
JVM vendor and version	Use only production JVMs on which WebLogic Server has been certified. See Supported Configurations in <i>What's New in Oracle WebLogic Server</i> for links to the latest certification information on various platforms.
Tuning heap size and garbage collection	For WebLogic Server heap size tuning details, see Garbage Collection .
Choosing a GC (garbage collection) scheme	Depending on your application, there are a number of GC schemes available for managing your system memory, as described in Choosing a Garbage Collection Scheme .
Mixed client/server JVMs	Deployments using different JVM versions for the client and server are supported in WebLogic Server.

Table 3-1 (Cont.) General JVM Tuning Considerations

Tuning Factor	Information Reference
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 <i>Performance Documentation For the Java Hotspot Virtual Machine: Threading</i> at http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html .

Changing To a Different JVM

When you create a domain, you choose the JVM that you want to run your domain and the Configuration Wizard configures the Oracle start scripts based on your choice. Modify the values for the `JAVA_HOME` and `JAVA_VENDOR` variables in the Configuration Wizard to change the JVM.

After you create a domain, if you want to use a different JVM, see Changing the JVM That Runs Servers in *Administering Server Startup and Shutdown for Oracle WebLogic Server*.

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](#)
- [Choosing a Garbage Collection Scheme](#)
- [Using Verbose Garbage Collection to Determine Heap Size](#)
- [Specifying Heap Size Values](#)
- [Tuning Tips for Heap Sizes](#)
- [Java HotSpot VM Heap Size Options](#)

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 in the server, see [Specifying Heap Size Values](#).

Choosing a Garbage Collection Scheme

Depending on which JVM you are using, you can choose from several garbage collection schemes to manage your system memory. For example, some garbage collection schemes are more appropriate for a given type of application. Once you have an understanding of the workload of the application and the different garbage collection algorithms utilized by the JVM, you can optimize the configuration of the garbage collection.

For an overview of the garbage collection schemes available with Sun's HotSpot VM, see [JDK 17 Documentation - HotSpot Virtual Machine Garbage Collection Tuning Guide](#) and [JDK 21 Documentation - HotSpot Virtual Machine Garbage Collection Tuning Guide](#).

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. Review the New generation heap sizes, see [Java HotSpot VM Heap Size Options](#).
 5. 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.
 6. 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.
 7. 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](#)
- [Java HotSpot VM Heap Size Options](#)

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.

Java HotSpot VM Heap Size Options

You achieve best performance by individually tuning each application. However, configuring the Java HotSpot VM heap size options listed in the following table when starting WebLogic Server increases performance for most applications.

These options may differ depending on your architecture and operating system. See your vendor's documentation for platform-specific JVM tuning options.

Table 3-2 Java Heap Size Options

Task	Option	Comments
Setting the New generation heap size	<code>-XX:NewSize</code>	As a general rule, set <code>-XX:NewSize</code> to be one-fourth the size of the heap size. Increase the value of this option for larger numbers of short-lived objects. Be sure to increase the New generation as you increase the number of processors. Memory allocation can be parallel, but garbage collection is not parallel.
Setting the maximum New generation heap size	<code>-XX:MaxNewSize</code>	Set the maximum size of the New Generation heap size.
Setting New heap size ratios	<code>-</code> <code>XX:SurvivorRati</code> <code>o</code>	The New generation area is divided into three sub-areas: Eden, and two survivor spaces that are equal in size. Configure the ratio of the Eden/survivor space size. Try setting this value to 8, and then monitor your garbage collection.
Setting initial heap size	<code>-Xms</code>	As a general rule, set initial heap size (<code>-Xms</code>) equal to the maximum heap size (<code>-Xmx</code>) to minimize garbage collections.
Setting maximum heap size	<code>-Xmx</code>	Set the maximum size of the heap.

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

```
$ java -XX:NewSize=128m -XX:MaxNewSize=128m -XX:SurvivorRatio=8 -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 128 megabytes of memory to the New generation and maximum New generation heap sizes, and 512 megabytes of memory to the minimum and maximum heap sizes for the WebLogic Server instance running in the JVM.

- [Other Java HotSpot VM Options](#)

Other Java HotSpot VM Options

Oracle provides other standard and non-standard command-line options to improve the performance of your VM. How you use these options depends on how your application is coded.

Test both your client and server JVMs to see which options perform better for your particular application. For additional examples of the HotSpot VM options, see:

- *Standard Options for Windows (Win32) VMs* at <http://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html>.
- *Standard Options for Solaris VMs and Linux VMs* at <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html>.

The *Java Virtual Machine* document provides a detailed discussion of the Client and Server implementations of the Java virtual machine for Java SE at <http://docs.oracle.com/javase/8/docs/technotes/guides/vm/>.

Increasing Java Heap Size for Managed Servers

For better performance, 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.

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

- [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](#)

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 Remote 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.

4

Tuning WebLogic Diagnostic Framework and Java Flight Recorder Integration

Follow the recommended tips and guidelines to tune WebLogic Diagnostic Framework (WLDF) and Java Flight Recorder of Oracle WebLogic Server.

- [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.
- [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.
- [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.

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 Remote 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.

5

Tuning WebLogic Server

Learn how to tune Oracle WebLogic Server to match your application.

- [Setting Java Parameters for Starting WebLogic Server](#)
Java parameters must be specified whenever you start WebLogic Server.
- [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.
- [Deployment](#)
Learn techniques to improve deployment performance.
- [Thread Management](#)
WebLogic Server provides the following mechanisms to manage threads to perform work.
- [Tuning Network I/O](#)
Learn about network communication between clients and servers (including T3 and IIOP protocols, and their secure versions).
- [Tuning the Work Manager Maximum Threads Constraint Queue Size](#)
By default, the queue size for the Work Manager's maximum threads constraint is 8,192 (8K). During times of high load (when the machine CPU runs at 100% utilization), Work Manager instances may be unable to process messages in the queue quickly enough using this default setting.
- [Optimize Java Expressions](#)
Set the `optimize-java-expression` element to optimize Java expressions to improve runtime performance.
- [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.
- [Monitoring a WebLogic Server Domain](#)
Learn several different ways to monitor a WebLogic Server domain.
- [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.
- [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.

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.

Table 5-1 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.
Secured Production	Your application is running in its final form and you want rigid policies and configuration to ensure a highly secure environment for your production domain.

For information about how the security and performance-related configuration parameters differ when switching from one domain mode to another, see *How Domain Mode Affects the Default Security Configuration* in *Securing a Production Environment for Oracle WebLogic Server* .

Deployment

Learn techniques 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](#)
- [Self-Tuning Thread Pool Size](#)
- [How Many Work Managers are Needed?](#)
- [What are the SLA Requirements for Each Work Manager?](#)
- [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 Manager 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*.

Self-Tuning Thread Pool Size

The thread pool allocates threads to process the requests of service servers and client servers. The default value of the `selfTuningThreadPoolSizeMax` MBean attribute is 400. Depending on the provider and consumer requests, you can increase the pool size to a maximum of 65534.

We recommend that you increase the pool size if:

- The service provider and the service consumer share the same WebLogic server.
- The number of concurrent requests from the service consumer is greater than the self-tuning thread pool size.
- Service consumer requests occupy all the threads from the thread pool, and no thread is available for the service provider to respond to the requests.

See Self-Tuning Thread Pool 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*.

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. See *Configuring WebLogic Server to Avoid Overload Conditions in Administering Server Environments for Oracle WebLogic Server*.

Tuning Network I/O

Learn about 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

Two factors determine the efficiency of socket communications in WebLogic Server:

- Whether the server instance host system uses a native or a pure-Java socket reader implementation.
- For systems that use pure-Java socket readers, whether the server instance is configured to use enough socket reader threads.

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:

- [Java Non-Blocking IO \(NIO\) Muxer](#)
- [Native Muxers](#)
- [Server Location and Supported Platforms](#)
- [Pure-Java Versus Native Socket Reader Implementations](#)
- [Configuring Reader Threads for Java Socket Implementation](#)

Java Non-Blocking IO (NIO) 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`.

Native Muxers

Native Muxers 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`

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 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 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. Prior to WebLogic Server 12.1.2, Oracle recommended to use native muxers and referred to as performance packs.

For WebLogic Server 12.1.2 and subsequent releases, the Non-Blocking IO (NIO) muxer is recommended by default. However, Oracle still provides native muxer as an option for users upgrading WebLogic Server versions prior to 12.1.2 to maximize consistency of the runtime environment after upgrading.

With native muxers, you may be able to improve throughput for some cpu-bound applications 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.

Server Location and Supported Platforms

You can refer to the below example for WebLogic Server installation and supported platforms.

Install Location

Libraries for native muxers can be found under `ORACLE_HOME/wlserver/server/native` directory. For example, for linux x86_64 platform, `libmuxer.so` is located at: `ORACLE_HOME/wlserver/server/native/linux/x86_64/libmuxer.so`.

Supported Platforms

The native library supports the following platforms:

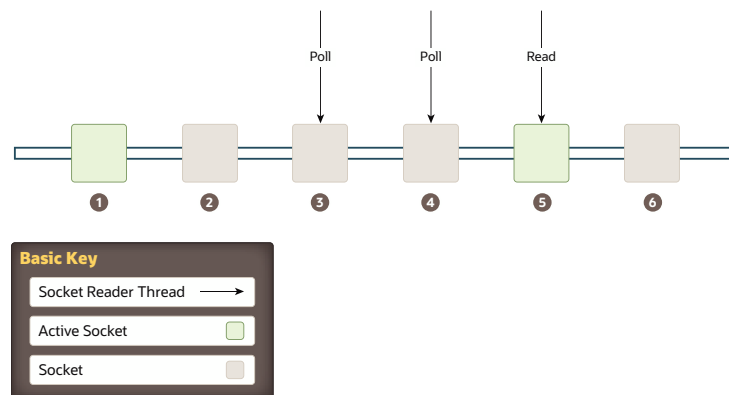
```
oracle.wls.core.app.server.nativelib/template.xml:  
dest="server/native/solaris/sparc64/libmuxer.so" source="wlserver/server/  
native/solaris/sparc64/libmuxer.so"  
oracle.wls.core.app.server.nativelib/template.xml:  
dest="server/native/aix/ppc64/libmuxer.so" source="wlserver/server/native/aix/  
ppc64/libmuxer.so"  
oracle.wls.core.app.server.tier1nativelib/template.xml:  
dest="server/native/linux/aarch64/libmuxer.so" source="wlserver/server/native/  
linux/aarch64/libmuxer.so"  
oracle.wls.core.app.server.tier1nativelib/template.xml:  
dest="server/native/linux/x86_64/libmuxer.so" source="wlserver/server/native/  
linux/x86_64/libmuxer.so"
```

Pure-Java Versus Native Socket Reader Implementations

Although the pure-Java implementation of socket reader threads is a reliable and portable method of peer-to-peer communication, it does not provide the optimal performance for heavy-duty socket usage in a WebLogic Server cluster. With pure-Java socket readers, threads must actively poll all opened sockets to determine if they contain data to read. In other words, socket reader threads are always "busy" polling sockets, even if the sockets have no data to read. This unnecessary overhead can reduce performance.

The performance issue is magnified when a server instance has more open sockets than the socket reader threads. Each reader thread must poll more than one open socket. When the socket reader encounters an inactive socket, it waits for a timeout before servicing another. During this timeout period, an active socket may go unread while the socket reader polls inactive sockets, as shown in [Figure 5-1](#).

Figure 5-1 Pure-Java Socket Reader Threads Poll Inactive Sockets



For optimal socket performance, configure the WebLogic Server host machine to use the native socket reader implementation for your operating system, rather than the pure-Java implementation. Native socket readers use far more efficient techniques to determine if there is data to read on a socket. With a native socket reader implementation, reader threads do not need to poll inactive sockets. They service only active sockets, and they are immediately notified (via an interrupt) when a given socket becomes active.



Note:

Applets cannot use native socket reader implementations, and therefore have limited efficiency in socket communication.

For instructions on how to configure the WebLogic Server host machine to use the native socket reader implementation for your operating system, see [Configure Native IP Sockets Readers on Machines that Host Server Instances](#).

Configuring Reader Threads for Java Socket Implementation

If you do use the pure-Java socket reader implementation, you can still improve the performance of socket communication by configuring the proper number of socket reader threads for each server instance. For optimal performance, the number of socket reader

threads in WebLogic Server should equal the potential maximum number of opened sockets. This configuration avoids the situation in which a reader thread must service multiple sockets, and ensures that socket data is read immediately.

To determine the proper number of reader threads for server instances in your cluster, see the following section, [Determining Potential Socket Usage](#).

For instructions on how to configure socket reader threads, see [Set the Number of Reader Threads on Machines that Host Server Instances](#).

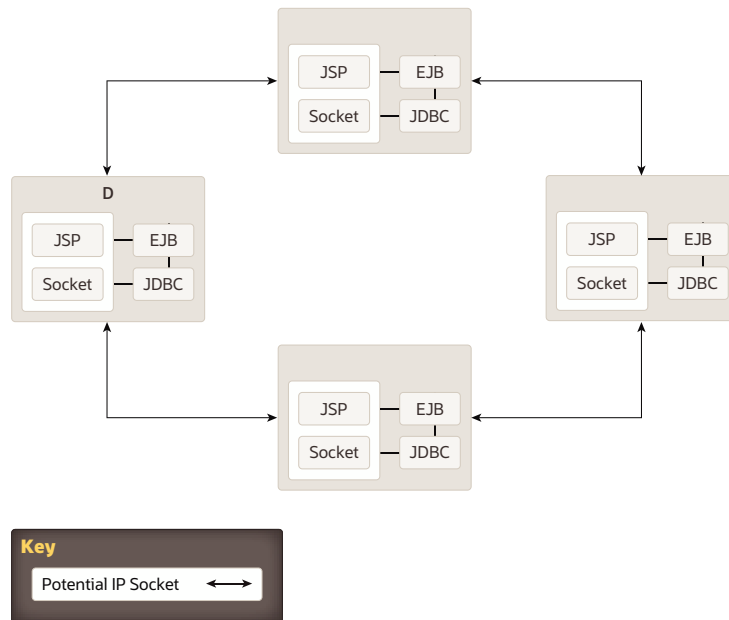
- [Determining Potential Socket Usage](#)

Determining Potential Socket Usage

Each WebLogic Server instance can potentially open a socket for every other server instance in the cluster. However, the actual maximum number of sockets used at a given time depends on the configuration of your cluster. In practice, clustered systems generally do not open a socket for every other server instance, because objects are deployed homogeneously—to each server instance in the cluster.

If your cluster uses in-memory HTTP session state replication, and you deploy objects homogeneously, each server instance potentially opens a maximum of only two sockets, as shown in [Figure 5-2](#).

Figure 5-2 Homogeneous Deployment Minimizes Socket Requirements

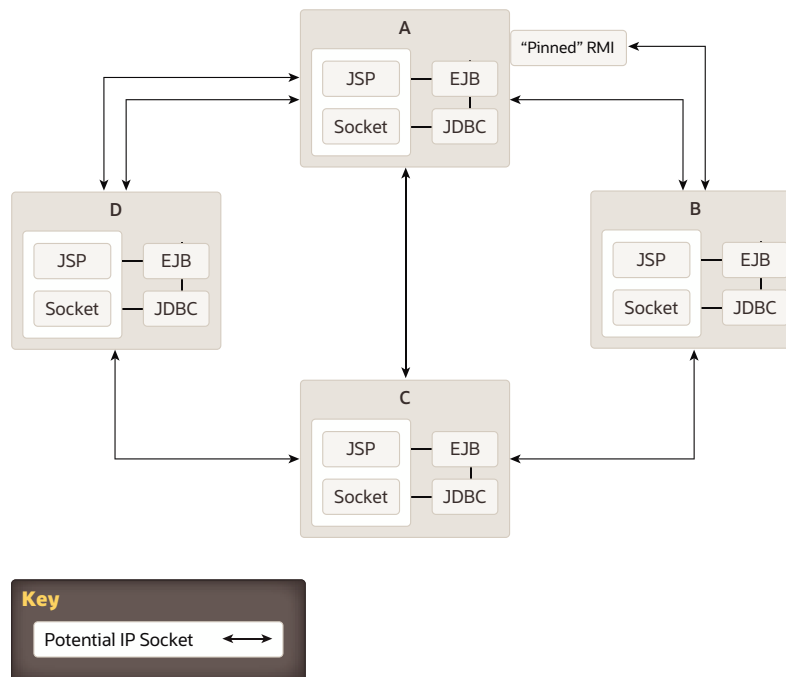


The two sockets in this example are used to replicate HTTP session states between primary and secondary server instances. Sockets are not required for accessing clustered objects, due to the collocation optimizations that WebLogic Server uses to access those objects. (These optimizations are described in [Optimization for Collocated Objects](#).) In this configuration, the default socket reader thread configuration is sufficient.

Deployment of "pinned" services (services that are active on only one server instance at a time) can increase socket usage, because server instances may need to open additional sockets to access the pinned object. (This potential can only be released if a remote server

instance actually accesses the pinned object.) [Figure 5-3](#) shows the potential effect of deploying a non-clustered RMI object to Server A.

Figure 5-3 Non-Clustered Objects Increase Potential Socket Requirements



In this example, each server instance can potentially open a maximum of three sockets at a given time, to accommodate HTTP session state replication and to access the pinned RMI object on Server A.

 **Note:**

Additional sockets may also be required for servlet clusters in a multitier cluster architecture, as described in Configuration Notes for Multitier Architecture.

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. See *Configure Custom Network Channels* in *Oracle WebLogic Remote Console Online Help*.
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](#)
- [Tuning Complete Message Timeout](#)
- [Tuning Number of File Descriptors](#)

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.

For information about displaying the WebLogic Remote Console page from which the complete message timeout parameter can be set, see *Configure General Protocol Settings* in the *Oracle WebLogic Remote Console Online Help*.

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.

Note:

The `Accept Backlog` parameter will be used by the Operating System when listening to the TCP sockets. The TCP backlog semantics are platform implementation specific.

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 checks the health condition before returning the cached connection to the client for use. If there is no need to check the connection's health condition before returning it to the client, set the `http.keepAliveCache.socketHealthCheckTimeout` value to 0.

Tuning the Work Manager Maximum Threads Constraint Queue Size

By default, the queue size for the Work Manager's maximum threads constraint is 8,192 (8K). During times of high load (when the machine CPU runs at 100% utilization), Work Manager instances may be unable to process messages in the queue quickly enough using this default setting.

You may need to increase the queue size in response to the following runtime exception:


```
java.lang.RuntimeException: [WorkManager:002943]Maximum Threads Constraint
"ClusterMessaging" queue for work manager "ClusterMessaging" reached maximum
capacity of 8,192 elements. Consider setting a larger queue size for the maximum
threads constraint.
```

In the following example, the target is specified by the server name (Server-0), and the queue size is increased to 65,536 (64K).

```
<max-threads-constraint>
  <name>ClusterMessaging-max</name>
  <target>Server-0</target>
  <count>1</count>
  <queue-size>65536</queue-size>
</max-threads-constraint>

<work-manager>
  <name>ClusterMessaging</name>
  <target>Server-0</target>
  <max-threads-constraint>ClusterMessaging-max</max-threads-constraint>
</work-manager>
```

You can specify the target using either the server name or the cluster name. You can use either server or cluster as target names.

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*.

For information about improving cluster throughput of global transactions, see [Improving Throughput Using XA Transaction Cluster Affinity](#).

- [Scalability and High Availability](#)
- [How to Ensure Scalability for WebLogic Clusters](#)
- [Running Multiple Server Instances on Multi-Core Machines](#)

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 Jakarta 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 Jakarta EE application: stateful session EJBs or HTTP sessions. By themselves, they rarely impact cluster scalability. However, when coupled with a session replication mechanism required to provide high-availability, bottlenecks are introduced. If a Jakarta 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 5-2 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.

Table 5-2 (Cont.) Asynchronous Replication Behavior by Cluster Topology

Topology	Behavior
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.

Running Multiple Server Instances on Multi-Core Machines

With multi-core machines, additional consideration must be given to the ratio of the number of available cores to clustered WebLogic Server instances. Because WebLogic Server has no built-in limit to the number of server instances that reside in a cluster, large, multi-core servers, can potentially host very large clusters or multiple clusters.

Consider the following when determining the optimal ratio of cores to WebLogic Server instances:

- The memory requirements of the application. Choose the heap sizes of an individual instance and the total number of instances to ensure that you're providing sufficient memory for the application and achieving good GC performance. For some applications, allocating very large heaps to a single instance may lead to longer GC pause times. In this case the performance may benefit from increasing the number of instances and giving each instance a smaller heap.
- Maximizing CPU utilization. While WebLogic Server is capable of utilizing multiple cores per instance, for some applications, increasing the number of instances on a given machine (reducing the number of cores per instance) can improve CPU utilization and overall performance.

Monitoring a WebLogic Server Domain

Learn several different ways to monitor a WebLogic Server domain.

- [Using the WebLogic Diagnostic Framework](#)
- [Using JMX to Monitor WebLogic Server](#)
- [Using WLST to Monitor WebLogic Server](#)
- [Resources to Monitor WebLogic Server](#)

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](#)
- [Class Caching](#)

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.

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*.

6

Tuning the WebLogic Persistent Store

The persistent store provides a built-in, high-performance storage solution for Oracle WebLogic Server subsystems and services that require persistence. Tune the persistent store by tuning JDBC stores, file stores, and following the best practices when using persistent stores.

- [Overview of Persistent Stores](#)
Each server instance, including the Administration Server, has a default persistent store that requires no configuration. In addition to using the default file store, you can also configure a file-based store or JDBC-accessible store, JDBC TLOG store, and a file-based paging store.
- [Best Practices When Using Persistent Stores](#)
Learn the best practices for using WebLogic persistent stores.
- [Tuning JDBC Stores](#)
Review information on tuning JDBC stores.
- [Tuning File Stores](#)
Learn about tuning file stores.
- [Using a Network File System](#)
Learn about using a WebLogic persistent store with a Network File System (NFS).

Overview of Persistent Stores

Each server instance, including the Administration Server, has a default persistent store that requires no configuration. In addition to using the default file store, you can also configure a file-based store or JDBC-accessible store, JDBC TLOG store, and a file-based paging store.

- [Using the Default Persistent Store](#)
- [Using Custom File Stores and JDBC Stores](#)
- [Using a JDBC TLOG Store](#)
- [Using JMS Paging Stores](#)
- [Using Diagnostic 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. For high availability, it is a best practice to configure custom file or JDBC stores instead of a default store. See:

- Using the WebLogic Persistent Store in *Administering the WebLogic Persistent Store*.
- Create a File Store in *Oracle WebLogic Remote Console Online Help*.

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 the WebLogic Persistent Store*](#).
- [Comparing File Stores and JDBC Stores in *Administering the WebLogic Persistent Store*](#).
- [Creating a Custom \(User-Defined\) File Store in *Administering the WebLogic Persistent Store*](#).
- [Creating a JDBC Store in *Administering the WebLogic Persistent Store*](#).

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 the WebLogic Persistent Store*](#).

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 Flash Storage to Page JMS Messages](#)

Using Flash Storage to Page JMS Messages

You can often improve paging performance for JMS messages (persistent or non-persistent) by configuring JMS server paging directories to reference a directory on a locally mounted enterprise-class flash storage device. This can be significantly faster than other technologies

Note:

Most Flash storage devices are a single point of failure and are typically only accessible as a local device. They are suitable for JMS server paging stores which do not recover data after a failure and automatically reconstruct themselves as needed.

In most cases, Flash storage devices are not suitable for custom or default stores which typically contains data that must be safely recoverable. A configured `Directory` attribute of a default or custom store should not normally reference a directory that is on a single point of failure device.

Use the following steps to use a Flash storage device to page JMS messages:

1. Set the JMS server `Message Paging Directory` attribute to the path of your flash storage device, see [Specifying a Message Paging Directory](#).
2. Tune the `Message Buffer Size` attribute (it controls when paging becomes active). You may be able to use lower threshold values as faster I/O operations provide improved load absorption. See [Tuning the Message Buffer Size Option](#).
3. Tune JMS Server quotas to safely account for any Flash storage space limitations. This ensures that your JMS server(s) will not attempt to page more messages than the device can store, potentially yielding runtime errors and/or automatic shutdowns. As a conservative rule of thumb, assume page file usage will be at least $1.5 * ((\text{Maximum Number of Active Messages}) * (512 + \text{average message body size}))$ rounded up to the nearest 16MB. See [Defining Quota](#).

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

Learn the best practices for using WebLogic 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

Review information on tuning JDBC stores.

- By default, a WebLogic JDBC store instance obtains two JDBC connections from its data source and caches these connections for its entire lifetime. The JDBC store can be tuned to retry more often on a connection failure, and the data source should be tuned to test connections. See *Using a JDBC Store in Administering the WebLogic Persistent Store*.
- 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 the WebLogic Persistent Store*.
- When using Oracle BLOBS, you may be able to improve performance by tuning the [ThreeStepThreshold](#) value. See *Enabling Oracle BLOB Record Columns in Administering the WebLogic Persistent Store*.
- 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 *Create a JDBC Store in Oracle WebLogic Remote Console Online Help* and *Using the WebLogic Persistent Store in Administering the WebLogic Persistent Store*.

Tuning File Stores

Learn about tuning file stores.

- [Basic Tuning Information](#)
- [Tuning a File Store Direct-Write-With-Cache Policy](#)
- [Tuning the File Store Direct-Write Policy](#)
- [Tuning the File Store Block Size](#)

Basic Tuning Information

The following section provides general information on tuning File Stores:

- Take care when configuring file store directory locations.
 - Paging stores should reference a location on a local disk for best performance (paging stores are not reloaded after a failure and do not need to be on a highly available storage).
 - Custom or default file stores that may migrate to a different machine or JVM must be configured to reference a directory that is in a centrally accessible shared location.
 - See High Availability Best Practices in *Administering JMS Resources for Oracle WebLogic Server*.

- See File Locations in *Administering the WebLogic Persistent Store*.
- For basic (non-RAID) disk hardware, consider dedicating one disk per file store. A store can operate up to four to five times faster if it does not have to compete with any other store on the disk. Remember to consider the existence of the default file store in addition to each configured store and a JMS paging store for each JMS server.
- 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.

- 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`.
- Depending on the synchronous write policy, custom and default stores have a variety of additional tunable attributes that may improve performance. These include `CacheDirectory`, `MaxWindowBufferSize`, `IOBufferSize`, `BlockSize`, `InitialSize`, and `MaxFileSize`.
See [FileStoreMBean](#) in the *MBean Reference for Oracle WebLogic Server*.
- 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.
- Tune the `BlockSize` attribute. See [Tuning the File Store Block Size](#).

For more information on individual tuning parameters, see the [FileStoreMBean](#) in the *MBean Reference for Oracle WebLogic Server*.

- [Using Flash Storage to Increase Performance](#)
- [Additional Considerations](#)

Using Flash Storage to Increase Performance

You can gain additional I/O performance by using enterprise-class flash drives, which can be significantly faster than spinning disks for accessing data in real-time applications and allows you to free up memory for other processing tasks.

Simply update the `CacheDirectory` attribute with the path to your flash storage device and ensure that the device contains sufficient free storage to accommodate a full copy of the store's primary files. See the `CacheDirectory` attribute in the *MBean Reference for Oracle WebLogic Server*.

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 the [FileStoreMBean](#) in the *MBean Reference for Oracle WebLogic Server*.

- 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.

Tuning the File Store Direct-Write Policy

Note:

The `AvoidDirectIO` properties described in this section are still supported in this release, but have been deprecated as of 11gR1PS2. Use the configurable `Direct-Write-With-Cache` synchronous write policy as an alternative to the `Direct-Write` policy.

For file stores with the synchronous write policy of `Direct-Write`, you may be directed by Oracle Support or a release note to set `weblogic.Server` options on the command line or start script of the JVM that runs the store:

- Globally changes all stores running in the JVM:

```
-Dweblogic.store.AvoidDirectIO=true
```
- For a single store, where `store-name` is the name of the store:

```
-Dweblogic.store.store-name.AvoidDirectIO=true
```
- For the default store, where `server-name` is the name of the server hosting the store:

```
-Dweblogic.store._WLS_server-name.AvoidDirectIO=true
```

Setting `AvoidDirectIO` on an individual store overrides the setting of the global `-Dweblogic.store.AvoidDirectIO` option. For example: If you have two stores, A and B, and set the following options:

```
-Dweblogic.store.AvoidDirectIO=true  
-Dweblogic.store.A.AvoidDirectIO=false
```

then only store B has the setting `AvoidDirectIO=true`.

**Note:**

Setting the `AvoidDirectIO` option may have performance implications which often can be mitigated using the block size setting described in [Tuning the File Store Block Size](#).

Tuning the File Store Block Size

You may want to tune the file store block size for file stores that are configured with a synchronous write policy of `Direct-Write` (default), `Direct-Write-With-Cache`, or `Cache-Flush`, especially when using `Direct-Write` with `AvoidDirectIO=true` as described in [Tuning the File Store Direct-Write Policy](#) or for systems with a hard-drive-based write-back cache where you see that performance is limited by physical storage latency.

Consider the following example:

- A single WebLogic JMS producer sends persistent messages one by one.
- The network overhead is known to be negligible.
- The file store's disk drive has a 10,000 RPM rotational rate.
- The disk drive has a battery-backed write-back cache.

and the messaging rate is measured at 166 messages per second.

In this example, the low messaging rate matches the disk drive's latency (10,000 RPM / 60 seconds = 166 RPS) even though a much higher rate is expected due to the battery-backed write-back cache. Tuning the store's block size to match the file systems' block size could result in a significant improvement.

In some other cases, tuning the block size may result in marginal or no improvement:

- The caches are observed to yield low latency (so the I/O subsystem is not a significant bottleneck).
- Write-back caching is not used and performance is limited by larger disk drive latencies.

There may be a trade off between performance and file space when using higher block sizes. Multiple application records are packed into a single block only when they are written concurrently. Consequently, a large block size may cause a significant increase in store file sizes for applications that have little concurrent server activity and produce small records. In this case, one small record is stored per block and the remaining space in each block is unused. As an example, consider a Web Service Reliable Messaging (WS-RM) application with a single producer that sends small 100 byte length messages, where the application is the only active user of the store.

Oracle recommends tuning the store block size to match the block size of the file system that hosts the file store (typically 4096 for most file systems) when this yields a performance improvement. Alternately, tuning the block size to other values (such as paging and cache units) may yield performance gains. If tuning the block size does not yield a performance improvement, Oracle recommends leaving the block size at the default as this helps to minimize use of file system resources.

- [Setting the Block Size for a File Store](#)
- [Determining the File Store Block Size](#)
- [Determining the File System Block Size](#)

- [Converting a Store with Pre-existing Files](#)

Setting the Block Size for a File Store

Note:

The `BlockSize` command line properties that are described in this section are still supported in 11gR1PS2, but are deprecated. Oracle recommends using the `BlockSize` configurable on custom and default file stores instead.

To set the block size of a store, use one of the following properties on the command line or start script of the JVM that runs the store:

- Globally sets the block size of all file stores that don't have pre-existing files.

```
-Dweblogic.store.BlockSize=block-size
```

- Sets the block size for a specific file store that doesn't have pre-existing files.

```
-Dweblogic.store.store-name.BlockSize=block-size
```

- Sets the block size for the default file store, if the store doesn't have pre-existing files:

```
-Dweblogic.store._WLS_server-name. BlockSize=block-size
```

The value used to set the block size is an integer between 512 and 8192 which is automatically rounded down to the nearest power of 2.

Setting `BlockSize` on an individual store overrides the setting of the global `-Dweblogic.store.BlockSize` option. For example: If you have two stores, A and B, and set the following options:

```
-Dweblogic.store.BlockSize=8192  
-Dweblogic.store.A.BlockSize=512
```

then store B has a block size of 8192 and store A has a block size of 512.

Note:

Setting the block size using command line properties only takes effect for file stores that have no pre-existing files. If a store has pre-existing files, the store continues to use the block size that was set when the store was first created.

Determining the File Store Block Size

You can verify a file store's current block size and synchronous write policy by viewing the server log of the server that hosts the store. Search for a "280009" store opened message.

Determining the File System Block Size

To determine your file system's actual block size, consult your operating system documentation. For example:

- Linux ext2 and ext3 file systems: run `/sbin/dumpe2fs /dev/device-name` and look for "Block size"
- Windows NTFS: run `fsutil fsinfo ntfsinfo device letter:` and look for "Bytes Per Cluster"

Converting a Store with Pre-existing Files

If the data in a store's pre-existing files do not need to be preserved, then simply shutdown the host WebLogic Server instance and delete the files to allow the block size to change when the store is restarted. If you need to preserve the data, convert a store with pre-existing files to a different block size by creating a version of the file store with the new block size using the compact command of the command line store administration utility:

1. `java -Dweblogic.store.BlockSize=block-size weblogic.store.Admin`
2. Type help for available commands.
3. `Storeadmin->compact -dir file-store-directory`

See Store Administration Using a Java Command-line in *Administering the WebLogic Persistent Store*.

Using a Network File System

Learn about using a WebLogic persistent store with a Network File System (NFS).

- [Configuring Synchronous Write Policies](#)
- [Test Server Restart Behavior](#)
- [Handling NFS Locking Errors](#)

Configuring Synchronous Write Policies

NFS storage may not fully protect transactional data, as it may be configured to silently buffer synchronous write requests in volatile memory. If a file store Directory is located on an NFS mount, and the file store's Synchronous Write Policy is anything other than Disabled, check your NFS implementation and configuration to make sure that it is configured to support synchronous writes. A Disabled synchronous write policy does not perform synchronous writes, but, as a consequence, is generally not transactionally safe. You may detect undesirable buffering of synchronous write requests by observing high persistent message or transaction throughput that exceeds the physical capabilities of your storage device. On the NFS server, check the synchronous write setting of the exported NFS directory hosting your File Store. A SAN based file store, or a JDBC store, may provide an easier solution for safe centralized storage.

Test Server Restart Behavior

Oracle strongly recommends verifying the behavior of a server restart after abrupt machine failures when the JMS messages and transaction logs are stored on an NFS mounted directory. Depending on the NFS implementation, different issues can arise after a failover or restart. The behavior can be verified by abruptly shutting down the node hosting the Web Logic servers while these are running. If the server is configured for server migration, it should be started automatically in the failover node after the corresponding failover period. If not, a manual restart of the WebLogic Server on the same host (after the node has completely rebooted) can be performed.

Handling NFS Locking Errors

WebLogic Server maintains locks on files used for storing JMS data and transaction logs. These locks provide protection against potential data corruption caused when two instances of the same WebLogic Server or File Store are accidentally started. When two WebLogic file stores having the same name and directory are mistakenly started or when Oracle WebLogic Server does not restart after a catastrophic failure, the following errors may appear in the server log files:

Example 6-1 Store Restart Failure Error Message

```
<Error> <Store> <BEA-280061> <The persistent store "_WLS_server_soal" could not be
deployed:
weblogic.store.PersistentStoreException: java.io.IOException:
[Store:280021]There was an error while opening the file store file
"_WLS_SERVER_SOA1000000.DAT"
at weblogic.store.io.file.Heap.open(Heap.java:168)
at weblogic.store.io.file.FileStoreIO.open(FileStoreIO.java:88)
...
java.io.IOException: Error from fcntl() for file locking, Resource temporarily
unavailable, errno=11
```

File locking errors are caused when two file stores using same files are started simultaneously. The following information can be used to prevent or correct the errors:

- If two file stores from two different domains having the same name are configured to use the same directory, shut down WebLogic Server and change the configuration of the conflicting stores to use different directories. This is prevented from occurring with two different file stores within the same domain, as WebLogic Server prevents the two file stores from being configured with the same name within the same domain.
- WebLogic does not support starting multiple deployments of the same domain from different sites when they are using shared persistent storage locations.
- WebLogic does not support starting multiple WebLogic Servers of the same name from different sites when they are using shared persistent storage locations.

File locking errors are also caused due to an "abandoned lock" which occurs when the owning file store is no longer running after a machine failure, operating system crash, and virtual machine destruction. The NFS storage device does not become aware of the problem immediately. As a result, any subsequent attempt by the WebLogic Server to acquire locks on the previously locked files may fail. You can perform the tasks described in the following solutions to unlock the logs and data files:

For more information about locking of files stored in NFS mounted directories on the storage device, see your storage vendor's documentation.

- [Solution 1 – Using NFS v4 Instead of NFS v3](#)
- [Solution 2 - Copying Data Files to Remove NFS Locks](#)
- [Solution 3 - Disabling File Locks in WebLogic Server File Stores](#)

Solution 1 – Using NFS v4 Instead of NFS v3

Most abandoned locks are due to using NFS v3 based Network Attached Storage (NAS) which can leave files locked in the event of catastrophic failures such as a hardware failure, operating system crash, deletion a virtual machine without allowing it to shutdown normally.

To mitigate this problem, Oracle strongly recommends using a NFS v4 NAS server. It should be tuned to automatically release abandoned locks within the approximate time required to complete a server restart or migration and is the most recommended solution.

For more information about locking files stored in NFS-mounted directories on the storage device, see your storage vendor's documentation.

Solution 2 - Copying Data Files to Remove NFS Locks

Note:

You should be very cautious when using this option. It is critical to ensure that a file store is shut down before copying the file store's files. Otherwise, the files may get corrupted. Additional procedural precautions must be implemented to avoid any human error and to ensure that no instance of file store is manually started at any given point in time during the copy. Similarly, extra precautions must be taken to ensure that no two domains have a store with the same name that references the same directory.

Manually unlock the default store, paging store, and JMS store data files and start the servers by first ensuring that there are no running file stores that are using the files, then creating a copy of the locked persistence store file, and using the copy for subsequent operations.

To create a copy of the locked persistence store file, rename the file, and then copy it back to its original name. The following sample steps assume that transaction logs are stored in the `/shared/tlogs` directory and JMS data is stored in the `/shared/jms` directory.

Example 6-2 Sample Steps to Remove NFS Locks

```
cd /shared/tlogs

mv _WLS_SOA_SERVER1000000.DAT _WLS_SOA_SERVER1000000.DAT.old

cp _WLS_SOA_SERVER1000000.DAT.old _WLS_SOA_SERVER1000000.DAT

cd /shared/jms

mv SOAJMSFILESTORE_AUTO_1000000.DAT SOAJMSFILESTORE_AUTO_1000000.DAT.old

cp SOAJMSFILESTORE_AUTO_1000000.DAT.old SOAJMSFILESTORE_AUTO_1000000.DAT

mv UMSJMSFILESTORE_AUTO_1000000.DAT UMSJMSFILESTORE_AUTO_1000000.DAT.old

cp UMSJMSFILESTORE_AUTO_1000000.DAT.old UMSJMSFILESTORE_AUTO_1000000.DAT
```

In this solution, the WebLogic file locking mechanism continues to provide protection from any accidental data corruption caused when multiple instances of the same servers are started accidentally. However, the servers must be restarted manually after abrupt machine failures. In addition, file stores create multiple `numbered.DAT` files consecutively when they are used to store large amounts of data. All files may need to be copied and renamed when this occurs.

Solution 3 - Disabling File Locks in WebLogic Server File Stores

Note:

With this solution, because the WebLogic Server locking is disabled, automated server restarts and failovers are expected to succeed. However, you should be very cautious when using this option. The WebLogic file locking feature is designed to help prevent severe file corruptions that can occur in concurrency scenarios.

The following prerequisites mitigate the risk of disabling file locks:

- If the server using the file store is configured for server migration, configure the database-based leasing option. This enforces additional locking mechanisms using database tables and prevents automated restart of more than one instance of the same WebLogic Server.
- Avoid disabling file locks on a file store that is using Automatic Service Migration (ASM).
 - ASM requires file store locking to work safely and is activated in the following scenarios:
 1. A custom file store target is set to a Migratable Target and the Migratable Target is configured with a Migration Policy other than 'manual' (the default).
 2. A custom file store target is set to a WebLogic cluster when the store is configured with a Migration Policy other than 'Off' (the default).
 3. A WebLogic Server is configured with a JTA Migratable Target with a Migration Policy value other than 'manual' (the default), as this in turn can lead to default file store migrations.
 - If both ASM and disabling file locks are required, store your critical data in database stores instead of file stores to avoid the risk of file corruptions. For example, use a custom JDBC store instead of a file store and configure JTA to use a JDBC TLOG store instead of each WebLogic Server's default file store.
- Additional procedural precautions must be implemented to avoid any human error and to ensure that only one instance of a server is manually started at any given point in time. Similarly, precautions must be taken to ensure that no two domains have a store with the same name that references the same directory.

You can use a system property or WebLogic Server configuration to disable WebLogic file locking mechanisms for the default file store, custom file store, a JMS paging file store, and a Diagnostics file store, as described in the following sections:

- [Disabling File Locking for all Stores Using a System Property](#)

Disabling File Locking for all Stores Using a System Property

Starting from WebLogic Server 14.1.2 release, you can specify a Java system property on the `weblogic.Server` command line or start script of the JVM to disable locking on all of its file stores including default, paging, diagnostic, and custom file stores as shown below:

```
"-Dweblogic.store.file.LockEnabled=false"
```

7

Database Tuning

Follow the Oracle WebLogic Server tuning guidelines to prevent your database from becoming a major enterprise-level bottleneck by configuring it for optimal performance.

- [General Suggestions](#)
Review general database tuning suggestions.
- [Database-Specific Tuning](#)
Consider the basic tuning suggestions for Oracle, SQL Server, and Sybase databases.

General Suggestions

Review 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).

Database-Specific Tuning

Consider the basic tuning suggestions for Oracle, SQL Server, and Sybase databases.

- [Oracle](#)

- [Microsoft SQL Server](#)

Oracle

This section describes performance tuning for Oracle.

For proper initial configuration of the database to gain significant performance benefits, see [Configuring a Database for Performance](#).

- **Number of processes** — On most operating systems, each connection to the Oracle server spawns a shadow process to service the connection. Thus, the maximum number of processes allowed for the Oracle server must account for the number of simultaneous users, as well as the number of background processes used by the Oracle server. The default number is usually not big enough for a system that needs to support a large number of concurrent operations. For platform-specific issues, see your Oracle administrator's guide. The current setting of this parameter can be obtained with the following query:

```
SELECT name, value FROM v$parameter WHERE name = 'processes';
```

- **Buffer pool size** — The buffer pool usually is the largest part of the Oracle server system global area (SGA). This is the location where the Oracle server caches data that it has read from disk. For read-mostly applications, the single most important statistic that affects data base performance is the buffer cache hit ratio. The buffer pool should be large enough to provide upwards of a 95% cache hit ratio. Set the buffer pool size by changing the value, in data base blocks, of the `db_cache_size` parameter in the `init.ora` file.
- **Shared pool size** — The share pool in an important part of the Oracle server system global area (SGA). The SGA is a group of shared memory structures that contain data and control information for one Oracle database instance. If multiple users are concurrently connected to the same instance, the data in the instance's SGA is shared among the users. The shared pool portion of the SGA caches data for two major areas: the library cache and the dictionary cache. The library cache stores SQL-related information and control structures (for example, parsed SQL statement, locks). The dictionary cache stores operational metadata for SQL processing.

For most applications, the shared pool size is critical to Oracle performance. If the shared pool is too small, the server must dedicate resources to managing the limited amount of available space. This consumes CPU resources and causes contention because Oracle imposes restrictions on the parallel management of the various caches. The more you use triggers and stored procedures, the larger the shared pool must be. The `SHARED_POOL_SIZE` initialization parameter specifies the size of the shared pool in bytes.

The following query monitors the amount of free memory in the share pool:

```
SELECT * FROM v$sgastat  
WHERE name = 'free memory' AND pool = 'shared pool';
```

- **Maximum opened cursor** — To prevent any single connection taking all the resources in the Oracle server, the `OPEN_CURSORS` initialization parameter allows administrators to limit the maximum number of opened cursors for each connection. Unfortunately, the default value for this parameter is too small for systems such as WebLogic Server. Cursor information can be monitored using the following query:

```
SELECT name, value FROM v$sysstat  
WHERE name LIKE 'opened cursor%';
```

- **Database block size** — A block is Oracle's basic unit for storing data and the smallest unit of I/O. One data block corresponds to a specific number of bytes of physical database space on disk. This concept of a block is specific to Oracle RDBMS and should not be confused with the block size of the underlying operating system. Since the block size

affects physical storage, this value can be set only during the creation of the database; it cannot be changed once the database has been created. The current setting of this parameter can be obtained with the following query:

```
SELECT name, value FROM v$parameter WHERE name = 'db_block_size';
```

- **Sort area size** — Increasing the sort area increases the performance of large sorts because it allows the sort to be performed in memory during query processing. This can be important, as there is only one sort area for each connection at any point in time. The default value of this `init.ora` parameter is usually the size of 6–8 data blocks. This value is usually sufficient for OLTP operations but should be increased for decision support operation, large bulk operations, or large index-related operations (for example, recreating an index). When performing these types of operations, you should tune the following `init.ora` parameters (which are currently set for 8K data blocks):

```
sort_area_size = 65536  
sort_area_retained_size = 65536
```

Microsoft SQL Server

The following guidelines pertain to performance tuning parameters for Microsoft SQL Server databases. For more information about these parameters, see your Microsoft SQL Server documentation.

- Store `tempdb` on a fast I/O device.
- Increase the recovery interval if `perfmon` shows an increase in I/O.
- Use an I/O block size larger than 2 KB.

8

Tuning WebLogic Server EJBs

Tune the Oracle WebLogic Server EJBs for your application environment by following the general EJB tuning tips, and tuning EJB caches and pools.

- [General EJB Tuning Tips](#)
Use the general EJB tuning tips to optimize the application's performance.
- [Tuning EJB Caches](#)
Learn how to tune EJB caches.
- [Tuning EJB Pools](#)
Learn how to tune EJB pools.
- [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. Learn about the WebLogic Server EJB container features that you can use to minimize database interaction safely.
- [Tuning In Response to Monitoring Statistics](#)
The WebLogic Remote Console reports a wide variety of EJB runtime monitoring statistics, many of which are useful for tuning the performance of your EJBs. Learn how some of these statistics can help you tune the performance of EJBs.

General EJB Tuning Tips

Use the general EJB tuning tips to optimize the application's performance.

- 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 database connection is 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 releases prior to WebLogic Server 8.1, call-by-reference is turned on by default. For releases of WebLogic Server 8.1 and later, call-by-reference is turned off by default. Older applications migrating to WebLogic Server 8.1 and later 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.

The calls across different applications can be between:

- * applications on different JVMs
- * applications on the same JVM

For example, when you have a JVM that contains `EJBApp1.ear` and `EJBApp2.ear` on the same server, and you deploy one

EJB on `EJBApp1.ear` and another EJB on `EJBApp2.ear`, the calls between the applications on `EJBApp1.ear` and `EJBApp2.ear` are considered as calls across different applications even though they are on the same JVM.

- 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

Learn 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

Learn 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 `setEntityContext()` 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. Learn about the WebLogic Server EJB container features 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](#)

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 may 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 its 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 Database 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 Remote Console reports a wide variety of EJB runtime monitoring statistics, many of which are useful for tuning the performance of your EJBs. Learn how some of these statistics can help you tune the performance of EJBs.

If you prefer to write a custom monitoring application, you can access the monitoring statistics using JMX or WLST by accessing the relevant runtime MBeans. See [Runtime MBeans in MBean Reference for Oracle WebLogic Server](#).

- [Cache Miss Ratio](#)
- [Lock Waiter Ratio](#)
- [Lock Timeout Ratio](#)
- [Pool Miss Ratio](#)
- [Destroyed Bean Ratio](#)
- [Pool Timeout Ratio](#)
- [Transaction Timeout Ratio](#)

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):

$$\text{Cache Miss Ratio} = (\text{Cache Total Miss Count} / \text{Cache Total Access Count}) * 100$$

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:

$$\text{Lock Waiter Ratio} = (\text{Current Waiter Count} / \text{Current Lock Entry Count}) * 100$$

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:

$$\text{Lock Timeout Ratio} = (\text{Lock Manager Timeout Total Count} / \text{Lock Manager Total Access Count}) * 100$$

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 may 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 Remote 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 a 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:

$$\text{Pool Timeout Ratio} = (\text{Pool Total Timeout Count} / \text{Pool Total Access Count}) * 100$$

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 Timeout Ratio

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

$$\text{Transaction Timeout Ratio} = (\text{Transaction Total Timeout Count} / \text{Transaction Total Count}) * 100$$

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.

9

Tuning Message-Driven Beans

Use the tuning and best practice information of Oracle WebLogic Server for Message-Driven Beans (MDBs).

- [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.
- [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. Review information about MDB concurrency.
- [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. Follow the best practices for configuring and deploying MDBs.
- [Using MDBs with Foreign Destinations](#)
Review information on the behavior of WebLogic Server when using MDBs that consume messages from foreign destinations
- [Token-based Message Polling for Transactional MDB Listening on Queues/Topics](#)
Token-based polling mechanism approach provides better control of the concurrent poller thread count under changing message loads. Transactional WebLogic MDB uses a synchronous polling mechanism to retrieve messages from JMS destinations. 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.
- [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.

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. Review information about 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 9-1](#) provides information on how to determine the number of concurrently running MDB instances for a server instance.

Table 9-1 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 $\text{Min}(\text{max-beans-in-free-pool}, 16)$
Default work manager with self-tuning disabled	$\text{Min}(\text{default-thread-pool-size}/2+1, \text{max-beans-in-free-pool})$ This is also the default thread pool concurrency algorithm for WebLogic Server 8.1
Custom execute queue	$\text{Min}(\text{execute-queue-size}, \text{max-beans-in-free-pool})$
Custom work manager with constraint	varies due to self-tuning, between $\text{min-thread-constraint}$ and $\text{Min}(\text{max-threads-constraint}, \text{max-beans-in-free-pool})$

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 9-1](#).

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. Follow the best practices for configuring and deploying MDBs.

See *Configuring and Deploying MDBs Using Distributed Topics* in *Developing Message-Driven Beans for Oracle WebLogic Server*.

Using MDBs with Foreign Destinations

Review information on the behavior of WebLogic Server when using MDBs that consume messages from 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.

- [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 MDB Listening on Queues/Topics

Token-based polling mechanism approach provides better control of the concurrent poller thread count under changing message loads. Transactional WebLogic MDB uses a synchronous polling mechanism to retrieve messages from JMS destinations. 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.

Transactional WebLogic MDB uses 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

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.

WebLogic 12.2.1.2.0 introduces two properties of type `activation-config-property`:

`mdbDestinationPollIntervalMillis` and `minimizeAQSessions`. The

`mdbDestinationPollIntervalMillis` property controls the message polling interval that is used by the synchronous polling mechanism for MDB. When the JMS provider is AQJMS, the `minimizeAQSessions` property reduces the use of database resources by minimizing the number of AQ JMS sessions.

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.

10

Tuning Data Sources

To get the best performance from your Oracle WebLogic Server data sources, use the recommended tips to tune the data sources.

- [Tune the Number of Database Connections](#)
Creating a database connection is a relatively expensive process in any environment. 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.
- [Waste Not](#)
Another simple way to boost performance is to avoid wasting resources. Read about situations in which you can avoid wasting JDBC related resources.
- [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 reduces the risk of passing invalid connections to clients.
- [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.
- [Database Listener Timeout under Heavy Server Loads](#)
In some situations where WebLogic Server is under heavy loads, the database listener may timeout and throw an exception while creating a new connection. To workaroud this issue, increase the listener timeout on the database server.
- [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 of data type objects.
- [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.
- [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.

Tune the Number of Database Connections

Creating a database connection is a relatively expensive process in any environment. 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.

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. See *Control JDBC Data Sources* in the *Oracle WebLogic Remote Console Online Help*. You will still need to determine the optimal value for the `Maximum Capacity` as part of your pre-production performance testing.

Note that if you configure the value of `Initial Capacity` to be zero, WebLogic Server does not get a connection during startup. This provides a big startup performance gain, especially if several data sources are available. But more importantly, it allows the data source to be deployed on startup, even if the database is not available or has problems at startup (or it could be a standby data source that is not even available when the primary service is running).

There are two situations in which a connection is reserved, even if `Initial Capacity` is zero:

1. For a multi data source configured for LLR, a connection is reserved on each member data source to determine if the underlying database is an Oracle Real Application Clusters (Oracle RAC) database. If it is Oracle RAC, only one of the member data sources must be available.
2. For an Active GridLink (AGL) data source configured with *auto-ONS* (that is, with no ONS host and port pairs provided), a connection is created to get the ONS configuration information from the database.

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. Read about situations in which 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 reduces 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. 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. See *Usage Restrictions for the Statement Cache in Administering JDBC Data Sources for Oracle WebLogic Server*.

Database Listener Timeout under Heavy Server Loads

In some situations where WebLogic Server is under heavy loads, 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 of data type objects.

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*.

Tuning Transactions

Learn tuning guidelines of Oracle WebLogic Server to optimize transaction performance.

- [Improving 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.
- [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).
- [Read-only, One-Phase Commit Optimizations](#)
When resource managers, such as the Oracle Database (including Oracle AQ and Oracle RAC), 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.

Improving 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 *Configure Clusters* in *Oracle WebLogic Remote Console Online Help* and *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](#)

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 (including Oracle AQ and Oracle RAC), 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*.

Tuning WebLogic JMS

Get the most out of your applications by implementing the administrative performance tuning features available with Oracle WebLogic Server JMS.

- [JMS Performance & Tuning Check List](#)
Review a checklist of items to consider when tuning WebLogic JMS.
- [Handling Large Message Backlogs](#)
When message senders inject messages faster than consumers, messages accumulate into a message *backlog*.
- [Cache and Re-use Client Resources](#)
JMS client resources are relatively expensive to create in comparison with 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.
- [Tuning Distributed Queues](#)
Each distributed queue member is individually advertised in JNDI as `jms-server-name@distributed-destination-jndi-name`. 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) or set *Producer Load Balancing Policy* to `Per-JVM`.
- [Tuning Topics](#)
Review information on how to tune WebLogic Topics.
- [Tuning for Large Messages](#)
Learn how to improve JMS performance when handling large messages.
- [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.
- [Subscription Message Limits](#)
In Oracle WebLogic JMS 12.2.1.3.0 and later, you can help prevent overloaded subscriptions from using all the available resources by configuring a message limit for a topic or a template. To configure a message limit, set the `MessagesLimitOverride` attribute on a destination template, a standalone topic, or a uniform distributed topic.
- [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.
- [Tuning Applications Using Unit-of-Order](#)
Message Unit-of-Order is a WebLogic Server value-added feature that enables a standalone 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.
- [Using JMS 2.0 Asynchronous Message Sends](#)
WebLogic Server 12.2.1.0 introduced a standard way to do asynchronous sends, that is flexible, powerful, and supported by the standard JMS 2.0 *asynchronous send* method.

- [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). 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.
- [Client-side Thread Pools](#)
WebLogic client thread pools are configured differently than WebLogic server thread-pools, and are not self tuning. Use the `-Dweblogic.ThreadPoolSize=n` command-line property to configure the thread pools.
- [Best Practices for JMS .NET Client Applications](#)
Review a short list of performance related best practices to use when creating a JMS .NET client application.
- [Considerations for Oracle Data Guard Environments](#)
Review the configuration considerations for a WebLogic JMS environment that includes Oracle Data Guard.

JMS Performance & Tuning Check List

Review 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 Standalone 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](#).
- For asynchronous message sends, see [Using JMS 2.0 Asynchronous Message Sends](#) (preferred), or if JMS 2.0 is not an option, and 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.

There are two areas to investigate in order to avoid or reduce message backlogs:

- [Improving Message Processing Performance](#)
- [Controlling Message Production](#)

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 Remote 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 replicated distributed topic behavior when not all members are active. In replicated 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.
- In certain applications it may be fine to automatically delete old unprocessed 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](#). For topics, additionally consider tuning a subscription limit. See [Subscription Message Limits](#).
- Use fewer producer threads.
- Tune a sender blocking timeout that occurs during a quota condition. 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](#)

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 with 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*.

- 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

Each distributed queue member is individually advertised in JNDI as `jms-server-name@distributed-destination-jndi-name`. 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) or set *Producer Load Balancing Policy* to `Per-JVM`.

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 and consider setting the *Producer Load Balancing Policy* to `Per-JVM`.
- 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.

- 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

Review information on how to tune WebLogic Topics.

- You may want to convert singleton topics to distributed topics.
- 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 the application can tolerate the deletion of old messages without having them be processed by a consumer, consider using message expirations or subscription limits. See [Subscription Message Limits](#).
- If produced messages are failing to load balance evenly across the members of a Distributed Topic, you may need to change the configuration of your producer connection factories to disable *server affinity* (enabled by default) or set *Producer Load Balancing Policy* to `Per-JVM`.
- 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](#)

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

Learn how to improve JMS performance when handling large messages.

- [Tuning MessageMaximum](#)
- [Setting Maximum Message Size for Network Protocols](#)
- [Threshold Compression for Remote Producers](#)
- [Store Compression](#)
- [Paging Out Messages To Free Up Memory](#)

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 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.

Threshold Compression for Remote Producers

A message compression threshold can be set programmatically using a JMS API extension to the [WLMessageProducer](#) interface, or 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.

Store Compression

WebLogic Server provides the ability to configure message compression for JMS Store I/O operations.

By selecting an appropriate message body compression option, JMS store I/O performance may improve for:

- Persistent messages that are read from or written to disk.
- Persistent and non-persistent messages are paged in or paged out when JMS paging is enabled.

For general tuning information on JMS message compression, see [Threshold Compression for Remote Producers](#).

The following sections provide information on how to configure message compression.

- [Selecting a Message Compression Option](#)
This section provides information on the types of message compression available for use when message body compression is enabled.
- [Message Compression for JMS Servers](#)
- [Message Compression for Store-and-Forward Sending Agents](#)
To configure message body compression for SAF Sending Agents:

Selecting a Message Compression Option

This section provides information on the types of message compression available for use when message body compression is enabled.

Note:

The performance of each compression option is dependent on the operating environment, data type, and data size. Oracle recommends users test their environments to determine the most appropriate compression option.

Table 12-1 Message Body Compression Options

Compression Type	Description
GZIP_DEFAULT_COMPRESSION	Use GZIP_DEFAULT_COMPRESSION to enable message compression using the JDK GZIP API with DEFAULT_COMPRESSION level. See java.util.zip package .
GZIP_BEST_COMPRESSION	Use GZIP_BEST_COMPRESSION to enable message compression using the JDK GZIP API with BEST_COMPRESSION level. See java.util.zip package .
GZIP_BEST_SPEED	Use GZIP_BEST_SPEED to enable message compression using the JDK GZIP API with BEST_SPEED level. See java.util.zip package .

Table 12-1 (Cont.) Message Body Compression Options

Compression Type	Description
LZF	Use LZF to enable message compression using Open Source LZF. See https://github.com/ning/compress .

Message Compression for JMS Servers

To configure message body compression for JMS servers:

1. If you have not done so, create a JMS Server, see *Create a JMS Server* in the *Oracle WebLogic Remote Console Online Help*.
2. Use the instructions to *Configure Resources for JMS System Modules* in the *Oracle WebLogic Remote Console Online Help*. Update the following **Advanced JMS server** attributes for your environment:
 - a. Optionally, select **Store Message Compression Enabled** to enable the JMS store to perform message body compression. See [StoreMessageCompressionEnabled](#) in *MBean Reference for Oracle WebLogic Server*.
 - b. Optionally, select **Paging Message Compression Enabled** to enable the JMS paging store to perform message body compression on persistent and non-persistent messages. See [PagingMessageCompressionEnabled](#) in *MBean Reference for Oracle WebLogic Server*.
 - c. In **Message Compression Options**, specify the type of message compression used. See [MessageCompressionOptions](#) in *MBean Reference for Oracle WebLogic Server*.

Message Compression for Store-and-Forward Sending Agents

To configure message body compression for SAF Sending Agents:

1. If you have not done so, create a SAF Sending Agent, see *Create a Store-and-Forward Agent* in the *Oracle WebLogic Remote Console Online Help*.
2. . Update the following **Advanced Sending Agent** attributes for your environment:
 - a. Optionally, select **Store Message Compression Enabled** to enable the JMS store to perform message body compression. See [StoreMessageCompressionEnabled](#) in *MBean Reference for Oracle WebLogic Server*.
 - b. Optionally, select **Paging Message Compression Enabled** to enable the JMS paging store to perform message body compression on persistent and non-persistent messages. See [PagingMessageCompressionEnabled](#) in *MBean Reference for Oracle WebLogic Server*.
 - c. In **Message Compression Options**, specify the type of message compression used. See [MessageCompressionOptions](#) in *MBean Reference for Oracle WebLogic Server*.

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](#)
- [Tuning the Message Buffer Size Option](#)

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.
```

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.

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 $256 + 256 \times 3 = 1024$ 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](#)
- [Destination-Level Quota](#)
- [JMS Server-Level Quota](#)

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 12-2 Quota Parameters

Attribute	Description
Bytes Maximum and Messages Maximum	The <code>Messages Maximum/Bytes Maximum</code> 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 <code>Shared</code> parameter for a quota resource defines whether multiple destinations referring to the same quota resource compete for resources with each other.

Table 12-2 (Cont.) Quota Parameters

Attribute	Description
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 .

For more information about quota configuration parameters, see [QuotaBean](#) in the *MBean Reference for Oracle WebLogic Server*.

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. It behaves just like the Policy setting of any other quota.


Subscription Message Limits

In Oracle WebLogic JMS 12.2.1.3.0 and later, you can help prevent overloaded subscriptions from using all the available resources by configuring a message limit for a topic or a template. To configure a message limit, set the `MessagesLimitOverride` attribute on a destination template, a standalone topic, or a uniform distributed topic.

When a subscription reaches its specified limit and receives a new message, the head message of the subscription is deleted to provide space for the new message. For a default FIFO subscription, the head message is the oldest. Messages are deleted only from

subscriptions that have reached their limit. If a message exists on multiple subscriptions and is deleted on one subscription, then the message can still be received by the other subscriptions.

A subscription limit differs from a quota in multiple ways. A topic that has reached its quota disallows new messages until existing messages have been processed or expired; on the other hand, a subscription that has reached its subscription limit allows the new message and makes room for it by deleting current messages. Also, a topic that has reached its quota affects all subscriptions on the topic, as this disallows new messages from being added to any subscription. By contrast, a subscription limit only affects subscriptions that have reached their limits.

 **Note:**

- Subscription limits are not substitutes for quotas. Oracle always recommends configuring quotas, even when a subscription limit is also configured.
- Regardless of subscription limits, subscription messages are not deleted if they are participating in a pending transaction, are part of a Unit-of-Work that is still waiting to accumulate all of its messages, or have already been passed to a consumer and are awaiting acknowledgement.
- If a topic has not reached its quota, and all messages are immune from deletion, then a new message is accepted regardless of whether this causes a subscription to exceed its limit.

To configure a subscription limit, set the [MessagesLimitOverride](#) attribute on a destination template, stand-alone topic, or uniform distributed topic. You can see whether a topic's runtime MBean has a subscription limit configured via its [SubscriptionMessagesLimit](#) attribute ("-1" indicates that no limit has been configured). You can monitor the number of messages that have been deleted due to a subscription limit on a durable subscription by checking its [SubscriptionLimitDeletedCount](#) attribute.

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](#).

- [How Flow Control Works](#)
- [Configuring Flow Control](#)
- [Flow Control Thresholds](#)

How Flow Control Works

Specifically, when either a JMS server or its 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 12-3 Flow Control Parameters

Attribute	Description
Flow Control Enabled	Determines whether a producer can be flow controlled by the JMS server.
Flow Maximum	<p>The maximum number of messages per second for a producer that is experiencing a threshold condition.</p> <p>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.</p> <p>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.</p>
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.

Table 12-3 (Cont.) Flow Control Parameters

Attribute	Description
Flow Steps	<p>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).</p> <p>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:</p> <p>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).</p> <p>The movement upward is linear. The difference is simply divided by the number of Flow Steps.</p>

Flow Control Thresholds

The attributes used for configuring bytes/messages thresholds are defined as part of the JMS server and/or its destination. [Table 12-4](#) defines how the upper and lower thresholds start and stop flow control on a JMS server and/or JMS destination.

Table 12-4 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.
Bytes/Messages Threshold Low	<p>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.</p> <p>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.</p>

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*.

- [Best Practices](#)
- [Using UOO and Distributed Destinations](#)

Best Practices

The following sections provide best practice information when using UOO:

- Ideal for applications that have strict message ordering requirements. UOO simplifies administration and application design, and in most applications improves performance.
- Use MDB batching to:
 - Speed-up processing of the messages within a single sub-ordering.
 - Consume multiple messages at a time under the same transaction.See [Tuning Message-Driven Beans](#).
- You can configure a default UOO for the destination. Only one consumer on the destination processes messages for the default UOO at a time.

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.

Using JMS 2.0 Asynchronous Message Sends

WebLogic Server 12.2.1.0 introduced a standard way to do asynchronous sends, that is flexible, powerful, and supported by the standard JMS 2.0 *asynchronous send* method.

The JMS 2.0 *asynchronous send* feature allows messages to be sent asynchronously without waiting for a JMS Server to accept them. This feature may yield a substantial performance gain, even a 'multi-x' gain, for applications that are bottlenecked on message send latency, especially for batches of small non-persistent messages.

Asynchronous send calls each get an asynchronous reply from the server indicating the message has been successfully sent with the same degree of confidence as if a synchronous send had been performed. The JMS provider notifies the application by invoking the callback method `onCompletion`, on an application-specified `CompletionListener` object. For a given message producer, callbacks to the `CompletionListener` will be performed, single threaded per session, in the same order as the corresponding calls to the asynchronous send method.

 **Note:**

Oracle recommends using JMS 2.0 asynchronous sends instead of the proprietary WebLogic one-way message sends as described in [Using One-Way Message Sends](#).

The JMS 2.0 asynchronous send has a performance similar to that of the One-Way Sends.

The JMS 2.0 asynchronous send:

- Can handle both non-persistent and persistent messages.
- Can handle Unit of Order messages.
- Does not get degraded performance when a client's connection host is connected to a different server in the cluster, than the producer's target destination.
- Provides best effort flow control (block) internally, without a need for special tuning when the amount of outstanding, asynchronously sent data without a completion-event gets too high.

See [JMS 2.0 javadoc](#) for `send()` calls with `CompletionListeners`.

See [What's New in JMS 2.0, Part Two—New Messaging Features](#) for example usage.

 **Note:**

- To get asynchronous send performance gains, it is important to cache or pool message producers between asynchronous send calls. The following calls will block until all outstanding asynchronous send call `CompletionListener` objects have been processed.
 - `Connection.close()`
 - `Session.close()`
 - `MessageProducer.close()`
 - `Session.commit()`
 - `Session.rollback()`
- An implementation of the `CompletionListener` interface must not make calls on their owning session unless no other threads are using the session. This is because the behavior of multi-threaded JMS session access is undefined and unpredictable (as per the JMS specification).
- As required by the JMS specification, asynchronous send calls fail within standard Jakarta EE server applications. If it is necessary to bypass this check, then a non-standard (proprietary to WebLogic) application can still access asynchronous sends by accessing JMS connection factories or contexts directly, instead of via context injection, or via a resource reference to a connection factory. Bypassing JMS in this way is for advanced users only; this disables Jakarta EE restriction checks and the automatic pooling of JMS client objects that are built into the server-side WebLogic applications.
- Asynchronous send calls are not compatible with JTA (XA) transactions, and will fail if a JTA transaction is active when called and the sender's connection was created with a connection factory configured with XA Enabled.

Using One-Way Message Sends

One-way message sends can greatly improve the performance of applications that are bottlenecked by senders, but do so at the risk of introducing a lower QOS (quality-of-service). 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.

 **Note:**

Oracle recommends using the JMS 2.0 asynchronous send feature instead of the proprietary WebLogic one-way send feature. The asynchronous send feature was introduced in 12.2.1.0 and has less activation restrictions. For example, the JMS 2.0 asynchronous send feature works well in a cluster without requiring additional configuration changes.

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.

When the One-Way Send Mode is 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](#)
- [One-Way Send Support In a Cluster With a Single Destination](#)
- [One-Way Send Support In a Cluster With Multiple Destinations](#)
- [When One-Way Sends Are Not Supported](#)
- [Different Client and Destination Hosts](#)
- [XA Enabled On Client's Host Connection Factory](#)
- [Higher QOS Detected](#)
- [Destination Quota Exceeded](#)
- [Change In Server Security Policy](#)
- [Change In JMS Server or Destination Status](#)
- [Looking Up Logical Distributed Destination Name](#)
- [Hardware Failure](#)
- [One-Way Send QOS Guidelines](#)

Configure One-Way Sends On a Connection Factory

You configure one-way message send parameters on a connection factory by using the WebLogic Scripting Tool (WLST) or JMX via the `FlowControlParamsBean` MBean.

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 be 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*.

Hardware Failure

A hardware or network failure will disable one-way sends. In such cases, the JMS producer is notified by an `OnException` or by the next two-way message 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.) The producer will be closed. The worst-case scenario is that all messages can be lost up to the last two-way message before the failure occurred.

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.

- The client application will not immediately receive network or server failure exceptions, some messages may be sent but silently deleted until the failure is detected by WebLogic Server and the producer is automatically closed. See [Hardware Failure](#) for more information.

Client-side Thread Pools

WebLogic client thread pools are configured differently than WebLogic server thread-pools, and are not self tuning. Use the `-Dweblogic.ThreadPoolSize=n` command-line property to configure the 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 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.

Best Practices for JMS .NET Client Applications

Review a short list of performance related best practices to use when creating a JMS .NET client application.

- Always register a connection exception listener using an `IConnection` if the application needs to take action when an idle connection fails.
- Have multiple .NET client threads share a single context to ensure that they use a single socket.
- Cache and reuse frequently accessed JMS resources, such as contexts, connections, sessions, producers, destinations, and connection factories. Creating and closing these resources consumes significant CPU and network bandwidth.
- Use DNS aliases or comma separated addresses for load balancing JMS .NET clients across multiple JMS .NET client host servers in a cluster.

For more information on best practices and other programming considerations for JMS .NET client applications, see *Programming Considerations in Developing JMS .NET Client Applications for Oracle WebLogic Server*.

Considerations for Oracle Data Guard Environments

Review the configuration considerations for a WebLogic JMS environment that includes Oracle Data Guard.

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

- [Migrate JMS Services for Unexpected Outages](#)

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*.

Tuning WebLogic JMS Store-and-Forward

Oracle WebLogic Server 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 recommended practices and tips.

See Understanding the Store-and-Forward Service in *Administering the Store-and-Forward Service for Oracle WebLogic Server*.

- [Best Practices for JMS SAF](#)
Learn the best practices for JMS SAF.
- [Tuning Tips for JMS SAF](#)
For better performance of JMS SAF, use the recommended tuning tips.

Best Practices for JMS SAF

Learn the 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](#).
- Tune message load balancing to match your preference. See SAF Load Balancing in *Administering the Store-and-Forward Service*.

Tuning Tips for JMS SAF

For better performance of JMS SAF, use the recommended tuning tips.

- 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.

Tuning WebLogic Message Bridge

Learn how to improve message bridge performance using the best practices available in Oracle WebLogic Server.

- [Best Practices](#)
- [Changing the Batch Size](#)
- [Changing the Batch Interval](#)
- [Changing the Quality of Service](#)
- [Using Multiple Bridge Instances](#)
- [Changing the Thread Pool Size](#)
- [Avoiding Durable Subscriptions](#)
- [Co-locating Bridges with Their Source or Target Destination](#)
- [Changing the Asynchronous Mode Enabled Attribute](#)
- [Tuning Environments with Many Bridges](#)

Best Practices

Learn the best practices for tuning WebLogic message bridge.

- Avoid using a message bridge if remote destinations are already highly available. JMS clients can send directly to remote destinations. Use a 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 store-and-forward feature instead of using a message bridge when forwarding messages to remote destinations. In general, a JMS SAF agent is significantly faster than a message bridge. One exception is a configuration when sending messages in a non-persistent exactly-once mode.

 **Note:**

A message 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. See *Create a Messaging Bridge Instance in Oracle WebLogic Remote Console Online Help*.

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. See *Create a Messaging Bridge Instance in Oracle WebLogic Remote Console Online Help*.

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. You can change the thread pool size 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*.
- 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.

See *Create a Messaging Bridge Instance in Oracle WebLogic Remote Console Online Help*.

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 <https://javaee.github.io/javaee-spec/javadocs/javax/jms/MessageListener.html>, or whether the bridge receives messages using the synchronous JMS APIs. In most situations, the `Asynchronous Enabled` attribute's value is dependent on the QOS required for the application environment as shown in [Table 14-1](#):

Table 14-1 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.

See *Create a Messaging Bridge Instance in Oracle WebLogic Remote Console Online Help*.

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

Learn 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](#).

Tuning Resource Adapters

Learn the best practices available in Oracle WebLogic Server to tune resource adapters.

- [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.
- [Connection Optimizations](#)
Oracle recommends that resource adapters implement the optional enhancements described in Connection Optimization section of the J2CA 1.5 Specification.
- [Thread Management](#)
Resource adapter implementations use the `WorkManager` to launch operations that need to run in a new thread, rather than creating new threads directly. WebLogic Server manages and monitors these threads.
- [InteractionSpec Interface](#)
An `InteractionSpec` holds properties for driving an Interaction with an EIS instance. The CCI specification defines a set of standard properties for an `InteractionSpec`. The `InteractionSpec` implementation class must provide getter and setter methods for each of its supported properties.

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 1.5 Specification.

See <http://www.oracle.com/technetwork/java/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 use the `WorkManager` to launch operations that need to run in a new thread, rather than creating new threads directly. WebLogic Server manages and monitors these threads.

See Chapter 10, "Work Management" in the J2CA 1.5 Specification at <http://www.oracle.com/technetwork/java/index.html>.

InteractionSpec Interface

An `InteractionSpec` holds properties for driving an Interaction with an EIS instance. The CCI specification defines a set of standard properties for an `InteractionSpec`. The `InteractionSpec` implementation class must provide getter and setter methods for each of its supported properties.

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.

16

Tuning Web Applications

Learn the best practices available in Oracle WebLogic Server for tuning Web applications and managing sessions.

- [Best Practices](#)
Learn the best practices for tuning Web applications.
- [Session Management](#)
Optimize your application so that it does as little work as possible when handling session persistence and sessions. Learn to design a session management strategy that suits your environment and application.
- [Pub-Sub Tuning Guidelines](#)
Follow the general tuning guidelines for a pub-sub server such as increasing the file descriptors, tuning the JVM options, and so on.
- [Enabling GZIP Compression](#)
The WebLogic Server Web container supports HTTP content-encoding GZIP compression, which is part of HTTP/1.1. With GZIP compression, you can reduce the size of the data that a Web browser has to download, improving network bandwidth. You can tune Web applications by enabling and configuring GZIP compression at either the domain level or Web application level.

Best Practices

Learn the best practices for tuning Web applications.

- [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 JSP 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 Recompilation](#).

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 <https://www.infoworld.com/article/2072812/servlet-and-jsp-performance-tuning.html>, by Rahul Chaudhary, JavaWorld, June 2004.

Session Management

Optimize your application so that it does as little work as possible when handling session persistence and sessions. Learn 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.

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

Follow the general tuning guidelines for a pub-sub server such as increasing the file descriptors, tuning the JVM options, and so on.

- 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>`.

Enabling GZIP Compression

The WebLogic Server Web container supports HTTP content-encoding GZIP compression, which is part of HTTP/1.1. With GZIP compression, you can reduce the size of the data that a Web browser has to download, improving network bandwidth. You can tune Web applications by enabling and configuring GZIP compression at either the domain level or Web application level.

See Enabling GZIP Compression for Web Applications in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

Tuning Web Services

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

- [Web Services Best Practices](#)
Design and architectural decisions have a strong impact on runtime performance and scalability of Web Service applications. Follow the key recommendations to achieve best performance.
- [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.
- [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, adjust the tuning parameters to accommodate the additional load.

Web Services Best Practices

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

- Design Web Service applications for coarse-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, adjust the tuning parameters to accommodate the additional load.

- [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
```


Tuning WebLogic Tuxedo Connector

The WebLogic Tuxedo Connector (WTC) provides interoperability between Oracle WebLogic Server applications and Tuxedo services. WTC allows WebLogic Server clients to invoke Tuxedo services and Tuxedo clients to invoke WebLogic Server Enterprise Java Beans (EJBs) in response to a service request.

Get the best performance from WebLogic Tuxedo Connector (WTC) applications using the tips provided.

See Introduction to Oracle WebLogic Tuxedo Connector Programming in *Developing Oracle WebLogic Tuxedo Connector Applications for Oracle WebLogic Server*.

- [Configuration Guidelines](#)
Refer the recommended guidelines when configuring WebLogic Tuxedo Connector.
- [Best Practices](#)
Learn the best practices for using WebLogic Tuxedo Connector.

Configuration Guidelines

Refer the recommended guidelines when configuring WebLogic Tuxedo Connector.

- You may have more than one WTC Service in your configuration.
- You can only target one WTC Service to a server instance.
- WTC does not support connection pooling. WTC multiplexes requests though a single physical connection.
- Configuration changes implemented as follows:
 - Changing the session/connection configuration (local APs, remote APs, Passwords, and Resources) before a connection/session is established. The changes are accepted and are implemented in the new session/connection.
 - Changing the Imported and Exported services configuration. The changes are accepted and are implemented in the next inbound or outbound request. Oracle does not recommend this practice as it can leave in-flight requests in an unknown state.
 - Changing the tBridge configuration. Any change in a deployed WTC service causes an exception. You must untarget the WTC service before making any tBridge configuration changes. After untargeting and making configuration changes, you must target the WTC service to implement the changes.

Best Practices

Learn the best practices for using WebLogic Tuxedo Connector.

- When configuring the connection policy, use `ON_STARTUP` and `INCOMING_ONLY`. `ON_STARTUP` and `INCOMING_ONLY` always paired. For example: If a WTC remote access point is configured with `ON_STARTUP`, the `DM_TDOMAIN` section of the Tuxedo domain configuration must be configured with the remote access point as `INCOMING_ONLY`. In this case, WTC always acts as the session initiator. See *Configuring the Connections Between Access Points* in the *Administering WebLogic Tuxedo Connector for Oracle WebLogic Server*.

- Avoid using connection policy `ON_DEMAND`. The preferred connection policy is `ON_STARTUP` and `INCOMING_ONLY`. This reduces the chance of service request failure due to the routing semantics of `ON_DEMAND`. See *Configuring the Connections Between Access Points in the Administering WebLogic Tuxedo Connector for Oracle WebLogic Server*.
- Consider using the following WTC features: Link Level Failover, Service Level failover and load balancing when designing your application. See *Configuring Failover and Failback in the Administering WebLogic Tuxedo Connector for Oracle WebLogic Server*.
- Consider using WebLogic Server clusters to provide additional load balancing and failover. To use WTC in a WebLogic Server cluster:

- Configure a WTC instance on all the nodes of the WebLogic Server cluster.
- Each WTC instance in each cluster node must have the same configuration.

See *How to Manage WebLogic Tuxedo Connector in a Clustered Environment in the Administering WebLogic Tuxedo Connector for Oracle WebLogic Server*.

- If your WTC to Tuxedo connection uses the internet, use the following security settings:
 - Set the value of `Security` to `DM_PW`. See *Authentication of Remote Access Points in the Administering WebLogic Tuxedo Connector for Oracle WebLogic Server*.
 - Enable Link-level encryption and set the `min-encrypt-bits` parameter to 40 and the `max-encrypt-bits` to 128. See *Link-Level Encryption in the Administering WebLogic Tuxedo Connector for Oracle WebLogic Server*.
- Your application logic should provide mechanisms to manage and interpret error conditions in your applications.
 - See *Application Error Management in the Developing Oracle WebLogic Tuxedo Connector Applications for Oracle WebLogic Server*.
 - See *System Level Debug Settings in the Administering WebLogic Tuxedo Connector for Oracle WebLogic Server*.
- Avoid using embedded `TypedFML32` buffers inside `TypedFML32` buffers. See *Using FML with WebLogic Tuxedo Connector in the Developing Oracle WebLogic Tuxedo Connector Applications for Oracle WebLogic Server*.
- If your application handles heavy loads, consider configuring more remote Tuxedo access points and let WTC load balance the work load among the access points. See *Configuring Failover and Failback in the Administering WebLogic Tuxedo Connector for Oracle WebLogic Server*.
- When using transactional applications, try to make the remote services involved in the same transaction available from the same remote access point. See *WebLogic Tuxedo Connector JATMI Transactions in the Developing Oracle WebLogic Tuxedo Connector Applications for Oracle WebLogic Server*.
- The number of client threads available when dispatching services from the gateway may limit the number of concurrent services running. There is no WebLogic Tuxedo Connector attribute to increase the number of available threads. Use a reasonable thread model when invoking service. See [Thread Management](#) and *Using Work Managers to Optimize Scheduled Work in Administering Server Environments for Oracle WebLogic Server*.
- WebLogic Server Releases 9.2 and higher provide improved routing algorithms which enhance transaction performance. Specifically, performance is improved when there are more than one Tuxedo service requests involved in a 2 phase commit (2PC) transaction. If your application does only single service request to the Tuxedo domain, you can disable this feature by setting the following WebLogic Server command line parameter:

```
-Dweblogic.wtc.xaAffinity=false
```

- Call the constructor `TypedFML32` using the maximum number of objects in the buffer. Even if the maximum number is difficult to predict, providing a reasonable number improves performance. You approximate the maximum number by multiplying the number of fields by 1.33.

 **Note:**

This performance tip does not apply to `TypedFML` buffer type.

For example:

If there are 50 fields in a `TypedFML32` buffer type then the maximum number is 63. Calling the constructor `TypedFML32(63, 50)` performs better than `TypedFML32()`.

If there are 50 fields in a `TypedFML32` buffer type and each can have maximum 10 occurrences, then call the constructor `TypedFML32(625, 50)` will give better performance than `TypedFML32()`.

- When configuring Tuxedo applications that act as servers interoperating with WTC clients, take into account of parallelism that may be achieved by carefully configuring different servers on different Tuxedo machines.
- Be aware of the possibility of database access deadlock in Tuxedo applications. You can avoid deadlock through careful Tuxedo application configuration.
- If your are using WTC load balancing or service level failover, Oracle recommends that you do not disable WTC transaction affinity.
- For load balancing outbound requests, configure the imported service with multiple entries using a different key. The imported service uses composite key to determine each record's uniqueness. The composite key is compose of "the service name + the local access point + the primary route in the remote access point list".

The following is an example of how to correctly configure load balancing requests for `service1` between `TDomainSession(WDOM1,TUXDOM1)` and `TDomainSession(WDOM1,TUXDOM2)`:

Table 18-1 Example of Correctly Configured Load Balancing

ResourceName	LocalAccessPoint	RemoteAccessPointList	RemoteName
service1	WDOM1	TUXDOM1	TOLOWER
service1	WDOM1	TUXDOM2	TOLOWER2

The following is an example an incorrectly configured load balancing requests. The following configuration results in the same composite key for `service1`:

Table 18-2 Example of Incorrectly Configured Load Balancing

ResourceName	LocalAccessPoint	RemoteAccessPointList	RemoteName
service1	WDOM1	TUXDOM1	TOLOWER
service1	WDOM1	TUXDOM1	TOLOWER

A

Capacity Planning

Capacity planning in Oracle WebLogic Server is the process of determining what type of hardware and software configuration is required to meet application needs adequately. Capacity planning is not an exact science. Every application is different and every user behavior is different.

- [Capacity Planning Factors](#)
A number of factors influence how much capacity a given hardware configuration will need in order to support a WebLogic Server instance and a given application. The hardware capacity required to support your application depends on the specifics of the application and configuration.
- [Assessing Your Application Performance Objectives](#)
Capacity planning for server hardware focuses on the maximum performance requirements and sets measurable objectives for capacity. Assess your application performance by gathering information about the level of activity expected on your server, the anticipated number of users, the number of requests, acceptable response time, and preferred hardware configuration.
- [Hardware Tuning](#)
The hardware capacity required to support your application depends on the specifics of the application and configuration. Consider how each factor applies to your configuration and application.
- [Network Performance](#)
Network performance is affected when the supply of resources is unable to keep up with the demand for resources. It is important to continually monitor your network performance to troubleshoot potential performance bottlenecks.
- [Related Information](#)
Information on topics related to capacity planning is available from numerous third-party software sources. The Oracle Technology Network provides detailed documentation for WebLogic Server.

Capacity Planning Factors

A number of factors influence how much capacity a given hardware configuration will need in order to support a WebLogic Server instance and a given application. The hardware capacity required to support your application depends on the specifics of the application and configuration.

You should consider how each of these factors applies to your configuration and application.

The following sections discuss several of these factors. Understanding these factors and considering the requirements of your application will aid you in generating server hardware requirements for your configuration. Consider the capacity planning questions in [Table A-1](#).

Table A-1 Capacity Planning Factors and Information Reference

Capacity Planning Questions	For Information, See:
Is WebLogic Server well-tuned?	Assessing Your Application Performance Objectives
How well-designed is the user application?	Database Server Capacity and User Storage Requirements
Is there enough bandwidth?	Network Load
How many transactions need to run simultaneously?	Concurrent Sessions
Is the database a limiting factor? Are there additional user storage requirements?	Database Server Capacity and User Storage Requirements
What is running on the machine in addition to WebLogic Server?	Network Load
Do clients use SSL to connect to WebLogic Server?	SSL Connections and Performance
What types of traffic do the clients generate?	RMI and Server Traffic
What types of clients connect to the WebLogic Server application?	Programmatic and Web-based Clients
Is your deployment configured for a cluster?	Clustered Configurations
Are your servers configured for migration?	Server Migration

- [Programmatic and Web-based Clients](#)
- [RMI and Server Traffic](#)
- [SSL Connections and Performance](#)
- [WebLogic Server Process Load](#)
- [Database Server Capacity and User Storage Requirements](#)
- [Concurrent Sessions](#)
- [Network Load](#)
- [Clustered Configurations](#)
- [Server Migration](#)
- [Application Design](#)

Programmatic and Web-based Clients

Primarily, two types of clients can connect to WebLogic Server:

- Web-based clients, such as Web browsers and HTTP proxies, use the HTTP or HTTPS (secure) protocol to obtain HTML or servlet output.
- Programmatic clients, such as Java applications and applets, can connect through the T3 protocol and use RMI to connect to the server.

The stateless nature of HTTP requires that the server handle more overhead than is the case with programmatic clients. However, the benefits of HTTP clients are numerous, such as the availability of browsers and firewall compatibility, and are usually worth the performance costs.

Programmatic clients are generally more efficient than HTTP clients because T3 does more of the presentation work on the client side. Programmatic clients typically call directly into EJBs while Web clients usually go through servlets. This eliminates the work the server must do for presentation. The T3 protocol operates using sockets and has a long-standing connection to the server.

A WebLogic Server installation that relies only on programmatic clients should be able to handle more concurrent clients than an HTTP proxy that is serving installations. If you are tunneling T3 over HTTP, you should not expect this performance benefit. In fact, performance of T3 over HTTP is generally 15 percent worse than typical HTTP and similarly reduces the optimum capacity of your WebLogic Server installation.

RMI and Server Traffic

What types of server traffic do the clients generate? If you are using T3 clients, most interaction with the server involves Remote Method Invocation (RMI.) Clients using RMI do not generate heavy traffic to the server because there is only one sender and one listener.

RMI can use HTTP tunneling to allow RMI calls to traverse a firewall. RMI tunneled through HTTP often does not deliver the higher degree of performance provided by non-tunneled RMI.

SSL Connections and Performance

Secure sockets layer (SSL) is a standard for secure Internet communications. WebLogic Server security services support X.509 digital certificates and access control lists (ACLs) to authenticate participants and manage access to network services. For example, SSL can protect JSP pages listing employee salaries, blocking access to confidential information.

SSL involves intensive computing operations. When supporting the cryptography operations in the SSL protocol, WebLogic Server can not handle as many simultaneous connections.

The number of SSL connections required out of the total number of clients required. Typically, for every SSL connection that the server can handle, it can handle three non-SSL connections. SSL substantially reduces the capacity of the server depending upon the strength of encryption used in the SSL connections. Also, the amount of overhead SSL imposes is related to how many client interactions have SSL enabled. WebLogic Server includes native performance packs for SSL operations.

WebLogic Server Process Load

What is running on the machine in addition to a WebLogic Server? The machine may be processing much more than presentation and business logic. For example, it could be running a Web server or maintaining a remote information feed, such as a stock information feed from a quote service.

Consider how much of your WebLogic Server machine's processing power is consumed by processes unrelated to WebLogic Server. In the case in which WebLogic Server (or the machine on which it resides) is doing substantial additional work, you need to determine how much processing power will be drained by other processes. When a Web server proxy is running on the same machine as WebLogic Server, expect anywhere from 25 to 50 percent of the computing capacity.

Database Server Capacity and User Storage Requirements

Is the database a bottleneck? Are there additional user storage requirements? Often the database server runs out of capacity much sooner than WebLogic Server does. Plan for a

database that is sufficiently robust to handle the application. Typically, a good application's database requires hardware three to four times more powerful than the application server hardware. It is good practice to use a separate machine for your database server.

Generally, you can tell if your database is the bottleneck if you are unable to maintain WebLogic Server CPU usage in the 85 to 95 percent range. This indicates that WebLogic Server is often idle and waiting for the database to return results. With load balancing in a cluster, the CPU utilization across the nodes should be about even.

Some database vendors are beginning to provide capacity planning information for application servers. Frequently this is a response to the three-tier model for applications.

An application might require user storage for operations that do not interact with a database. For example, in a secure system disk and memory are required to store security information for each user. You should calculate the size required to store one user's information, and multiply by the maximum number of expected users.

Concurrent Sessions

How many transactions must run concurrently? Determine the maximum number of concurrent sessions WebLogic Server will be called upon to handle. For each session, you will need to add more RAM for efficiency. Oracle recommends that you install a *minimum* of 256 MB of memory for each WebLogic Server installation that will be handling more than minimal capacity.

Next, research the maximum number of clients that will make requests at the same time, and how frequently each client will be making a request. The number of user interactions per second with WebLogic Server represents the total number of interactions that should be handled per second by a given WebLogic Server deployment. Typically for Web deployments, user interactions access JSP pages or servlets. User interactions in application deployments typically access EJBs.

Consider also the maximum number of transactions in a given period to handle spikes in demand. For example, in a stock report application, plan for a surge after the stock market opens and before it closes. If your company is broadcasting a Web site as part of an advertisement during the World Series or World Cup Soccer playoffs, you should expect spikes in demand.

Network Load

Is the bandwidth sufficient? WebLogic Server requires enough bandwidth to handle all connections from clients. In the case of programmatic clients, each client JVM will have a single socket to the server. Each socket requires bandwidth. A WebLogic Server handling programmatic clients should have 125 to 150 percent the bandwidth that a server with Web-based clients would handle. If you are interested in the bandwidth required to run a web server, you can assume that each 56kbps (kilobits per second) of bandwidth can handle between seven and ten simultaneous requests depending upon the size of the content that you are delivering. If you are handling only HTTP clients, expect a similar bandwidth requirement as a Web server serving static pages.

The primary factor affecting the requirements for a LAN infrastructure is the use of replicated sessions for servlets and stateful session EJBs. In a cluster, replicated sessions are the biggest consumer of LAN bandwidth. Consider whether your application will require the replication of session information for servlets and EJBs.

To determine whether you have enough bandwidth in a given deployment, look at the network tools provided by your network operating system vendor. In most cases, including Windows

and Solaris platforms, you can inspect the load on the network system. If the load is very high, bandwidth may be a bottleneck for your system.

Clustered Configurations

Clusters greatly improve efficiency and failover. Customers using clustering should not see any noticeable performance degradation. A number of WebLogic Server deployments in production involve placing a cluster of WebLogic Server instances on a single multiprocessor server.

Large clusters performing replicated sessions for Enterprise JavaBeans (EJBs) or servlets require more bandwidth than smaller clusters. Consider the size of session data and the size of the cluster.

Server Migration

Are your servers configured for migration? Migration in WebLogic Server is the process of moving a clustered WebLogic Server instance or a component running on a clustered instance elsewhere in the event of failure. In the case of whole server migration, the server instance is migrated to a different physical machine upon failure, either manually or automatically.

For capacity planning in a production environment, keep in mind that server startup during migration taxes CPU utilization. You cannot assume that because a machine can handle x number of servers running concurrently that it also can handle that same number of servers starting up on the same machine at the same time.

Application Design

How well-designed is the application? WebLogic Server is a platform for user applications. Badly designed or unoptimized user applications can drastically slow down the performance of a given configuration from 10 to 50 percent. The prudent course is to assume that every application that is developed for WebLogic Server will not be optimal and will not perform as well as benchmark applications. Increase the maximum capacity that you calculate or expect. See [Tune Your Application](#).

Assessing Your Application Performance Objectives

Capacity planning for server hardware focuses on the maximum performance requirements and sets measurable objectives for capacity. Assess your application performance by gathering information about the level of activity expected on your server, the anticipated number of users, the number of requests, acceptable response time, and preferred hardware configuration.

The numbers that you calculate from using one of our sample applications are of course just a rough approximation of what you may see with your application. There is no substitute for benchmarking with the actual production application using production hardware. In particular, your application may reveal subtle contention or other issues not captured by our test applications.

Hardware Tuning

The hardware capacity required to support your application depends on the specifics of the application and configuration. Consider how each factor applies to your configuration and application.

When you examine performance, a number of factors influence how much capacity a given hardware configuration will need in order to support WebLogic Server and a given application.

- [Benchmarks for Evaluating Performance](#)
- [Supported Platforms](#)

Benchmarks for Evaluating Performance

The Standard Performance Evaluation Corporation, at <http://www.spec.org>, provides a set of standardized benchmarks and metrics for evaluating computer system performance.

Supported Platforms

See Supported Configurations in *What's New in Oracle WebLogic Server* for links to the latest certification information on the hardware/operating system platforms that are supported for each release of WebLogic Server.

Network Performance

Network performance is affected when the supply of resources is unable to keep up with the demand for resources. It is important to continually monitor your network performance to troubleshoot potential performance bottlenecks.

Today's enterprise-level networks are very fast and are now rarely the direct cause of performance in well-designed applications. However, if you find that you have a problem with one or more network components (hardware or software), work with your network administrator to isolate and eliminate the problem. You should also verify that you have an appropriate amount of network bandwidth available for WebLogic Server and the connections it makes to other tiers in your architecture, such as client and database connections.

- [Determining Network Bandwidth](#)

Determining Network Bandwidth

A common definition of bandwidth is "the rate of the data communications transmission, usually measured in bits-per-second, which is the capacity of the link to send and receive communications." A machine running WebLogic Server requires enough network bandwidth to handle all WebLogic Server client connections. In the case of programmatic clients, each client JVM has a single socket to the server, and each socket requires dedicated bandwidth. A WebLogic Server instance handling programmatic clients should have 125–150 percent of the bandwidth that a similar Web server would handle. If you are handling only HTTP clients, expect a bandwidth requirement similar to a Web server serving static pages.

To determine whether you have enough bandwidth in a given deployment, you can use the network monitoring tools provided by your network operating system vendor to see what the load is on the network system. You can also use common operating system tools, such as the `netstat` command for Solaris or the System Monitor (`perfmon`) for Windows, to monitor your network utilization. If the load is very high, bandwidth may be a bottleneck for your system.

Also monitor the amount of data being transferred across your network by checking the data transferred between the application and the application server, and between the application server and the database server. This amount should not exceed your network bandwidth; otherwise, your network becomes the bottleneck. To verify this, monitor the network statistics for retransmission and duplicate packets, as follows:

```
netstat -s -P tcp
```

Related Information

Information on topics related to capacity planning is available from numerous third-party software sources. The Oracle Technology Network provides detailed documentation for WebLogic Server.

See <https://www.oracle.com/middleware/technologies/weblogic.html>.