



Fuse Message Broker **Configuring Broker Persistence**

Version 5.5
February 2012

Configuring Broker Persistence

Version 5.5

Updated: 27 Mar 2014

Copyright © 2012-2013 Red Hat, Inc. and/or its affiliates.

Trademark Disclaimer

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at <http://creativecommons.org/licenses/by-sa/3.0/>. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

Apache, ServiceMix, Camel, CXF, and ActiveMQ are trademarks of Apache Software Foundation. Any other names contained herein may be trademarks of their respective owners.

Third Party Acknowledgements

One or more products in the Red Hat JBoss Fuse release includes third party components covered by licenses that require that the following documentation notices be provided:

- JLine (<http://jline.sourceforge.net>) jline:jline:jar:1.0

License: BSD (LICENSE.txt) - Copyright (c) 2002-2006, Marc Prud'hommeaux <mwp1@cornell.edu>

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of JLine nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Stax2 API (<http://woodstox.codehaus.org/StAX2>) org.codehaus.woodstox:stax2-api:jar:3.1.1

License: The BSD License (<http://www.opensource.org/licenses/bsd-license.php>)

Copyright (c) <YEAR>, <OWNER> All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- jibx-run - JiBX runtime (<http://www.jibx.org/main-reactor/jibx-run>) org.jibx:jibx-run:bundle:1.2.3

License: BSD (<http://jibx.sourceforge.net/jibx-license.html>) Copyright (c) 2003-2010, Dennis M. Sosnoski.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of JiBX nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON

ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- JavaAssist (<http://www.jboss.org/javassist>) org.jboss.javassist:com.springsource.javassist:jar:3.9.0.GA:compile
License: MPL (<http://www.mozilla.org/MPL/MPL-1.1.html>)
- HAPI-OSGI-Base Module (<http://hl7api.sourceforge.net/hapi-osgi-base/>) ca.uhn.hapi:hapi-osgi-base:bundle:1.2
License: Mozilla Public License 1.1 (<http://www.mozilla.org/MPL/MPL-1.1.txt>)

Table of Contents

1. Introduction to Fuse Message Broker Persistence	11
2. Using the KahaDB Message Store	13
Understanding the KahaDB Message Store	14
Configuring the KahaDB Message Store	17
Concurrent Store and Dispatch	20
Optimizing the Metadata Cache	24
Recovery	26
3. Using a Distributed KahaDB Persistence Adapter	29
4. Using JDBC to Connect to a Database Store	33
Basics of Using the JDBC Persistence Adapter	34
Using JDBC with the High Performance Journal	37
Using JDBC without the Journal	40
Customizing the JDBC Persistence Adapter	43
5. Message Cursors	47
Types of Cursors	48
Configuring the Type of Cursor Used by a Destination	52
Index	55

List of Figures

2.1. Overview of the KahaDB Message Store	15
2.2. Concurrent Store and Dispatch—Slow Consumers	20
2.3. Concurrent Store and Dispatch—Fast Consumers	21
2.4. Serialized Store and Dispatch	23
2.5. Overview of the Metadata Cache and Store	24
5.1. Store-based Cursors for a Fast Consumer	48
5.2. Store-based Cursors for a Slow Consumer	49
5.3. VM Cursors	50
5.4. File-based Cursors	51

List of Tables

1.1. Setting a Broker's Persistence	12
2.1. Configuration Properties of the KahaDB Message Store	18
4.1. Attributes for Configuring the Journaled JDBC Persistence Adapter	38
4.2. Attributes for Configuring the Plain JDBC Persistence Adapter	40
4.3. Statements for Configuring the SQL Statements Used by the JDBC Persistence Adapter	43
5.1. Elements for Configuring the Type of Cursor to Use for Durable Subscribers	52
5.2. Elements for Configuring the Type of Cursor to Use for Transient Subscribers	53
5.3. Elements for Configuring the Type of Cursor to Use for a Queue	54

List of Examples

1.1. Turning Off a Broker's Persistence	12
2.1. Configuring the KahaDB Message Store	17
2.2. KahaDB Configured with Serialized Store and Dispatch	22
2.3. Configuration for Journal Validation	27
3.1. Distributed KahaDB Persistence Adapter Configuration	31
4.1. Configuration for the Apache Derby Database	35
4.2. Configuration for the Oracle JDBC Driver	36
4.3. Configuring Fuse Message Broker to use the Journaled JDBC Persistence Adapter	37
4.4. Configuring Fuse Message Broker to use the Plain JDBC Persistence Adapter	40
4.5. Fine Tuning the Database Schema	43
4.6. Configuring a Generic JDBC Provider	46
5.1. Configuring a Topic's Cursor Usage	53
5.2. Configuring a Queue's Cursor Usage	54

Chapter 1. Introduction to Fuse Message Broker Persistence

Message persistence allows for the recovery of undelivered messages in the event of a system failure. By default, Fuse Message Broker's persistence features are activated. The default set-up is fast and scalable. It is easy to customize the broker configuration to use a JDBC compliant database.

Overview

Loss of messages is not acceptable in mission critical applications. Fuse Message Broker reduces the risk of message loss by using a persistent message store by default. Persistent messages are written to the persistent store when they are sent. The messages persist in the store until their delivery is confirmed. This means that, in the case of a system failure, Fuse Message Broker can recover all of the undelivered messages at the time of the failure.

Persistent message stores

The default message store is embeddable and transactional. It is both very fast and extremely reliable. Fuse Message Broker implements several different kinds of message store, including:

- KahaDB message store
- distributed KahaDB message store
- Journaled JDBC adapter
- Non-journaled JDBC adapter

Message cursors

Fuse Message Broker caches message using *message cursors*. A message cursor represents a batch of messages cached in memory. When necessary, a message cursor can be used to retrieve the batch of persisted messages through the persistence adapter. See ["Message Cursors" on page 47](#) for details.

Activating and deactivating persistence

By default, brokers are configured to use a persistence layer to ensure that persistent messages will survive a broker failure and meet the once-and-only-once requirement of the JMS specification. Having a broker's persistence layer configured comes with a cost in terms of resources used and speed, so for testing purposes or cases where persistence will never be required, it may make sense to disable a broker's persistence layer.

Deactivating a broker's persistence layer means that a broker will treat *all* messages as non-persistent. If a producer sets a message's `JMSDeliveryMode` property to `PERSISTENT` the broker will not respect the setting. The message will be delivered at-most-once instead of once-and-only-once. This means that persistent messages will *not* survive broker shutdown.

Persistence in Fuse Message Broker is controlled by a broker's XML configuration file. To change a broker's persistence behavior you modify the configuration's broker element's `persistent` attribute.

Table 1.1. Setting a Broker's Persistence

Value	Description
true	The broker will use a persistent message store and respect the value of a message's <code>JMSDeliveryMode</code> setting.
false	The broker will not use a persistent message store and will treat all messages as non-persistent regardless of the value of a message's <code>JMSDeliveryMode</code> setting.

[Example 1.1 on page 12](#) shows a configuration snippet for turning off a broker's message persistence.

Example 1.1. Turning Off a Broker's Persistence

```
<broker persistent="false" ... >
  ...
</broker>
```

Configuring persistence adapter behavior

Fuse Message Broker offers a number of different persistence mechanisms besides the default message store. To use one of the alternative message stores, or to modify the behavior of the default message store, you need to configure the persistence adapter. This is done by adding a `persistenceAdapter` element or a `persistenceFactory` element (depending on the kind of adapter you want to use) to the broker's configuration file.

Chapter 2. Using the KahaDB Message Store

The KahaDB Message Store is the default message store used by Fuse Message Broker. It is a light-weight transactional store that is fast and reliable. It uses a hybrid system that couples a transactional journal for message storage and a reference store for quick retrieval.

Understanding the KahaDB Message Store	14
Configuring the KahaDB Message Store	17
Concurrent Store and Dispatch	20
Optimizing the Metadata Cache	24
Recovery	26



Important

If you use antivirus software it can interfere with Fuse Message Broker's ability to access the files in the KahaDB message store. You should configure your antivirus software to skip the KahaDB data folders when doing automatic scans.

Understanding the KahaDB Message Store

Overview

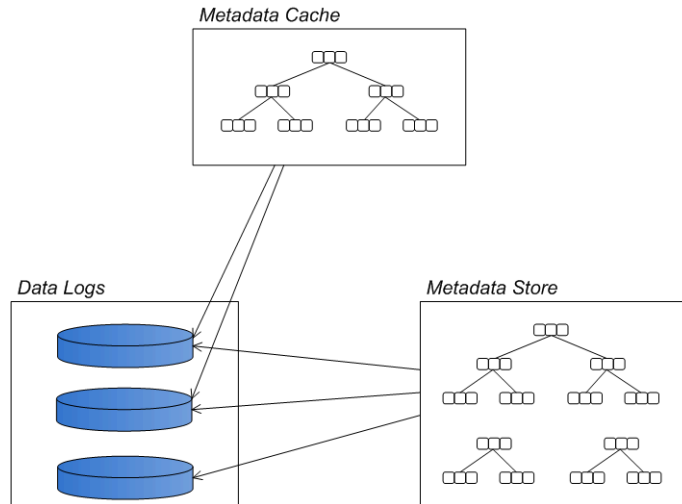
The KahaDB message store is the default persistence store used by Fuse Message Broker. It is a file-based persistence adapter that is optimized for maximum performance. The main features of KahaDB are:

- journal-based storage so that messages can be rapidly written to disk
- allows for the broker to restart quickly
- storing message references in a B-tree index which can be rapidly updated at run time
- full support for JMS transactions
- various strategies to enable recovery after a disorderly shutdown of the broker

Architecture

The KahaDB message store is an embeddable, transactional message store that is fast and reliable. It is an evolution of the AMQ message store used by Fuse Message Broker 5.0 to 5.3. It uses a transactional journal to store message data and a B-tree index to store message locations for quick retrieval.

[Figure 2.1 on page 15](#) shows a high-level view of the KahaDB message store.

Figure 2.1. Overview of the KahaDB Message Store

Messages are stored in file-based data logs. When all of the messages in a data log have been successfully consumed, the data log is marked as deletable. At a predetermined clean-up interval, logs marked as deletable are either removed from the system or moved to an archive.

An index of message locations is cached in memory to facilitate quick retrieval of message data. At configurable checkpoint intervals, the references are inserted into the metadata store.

Data logs

The data logs are used to store data in the form of *journals*, where events of all kinds—messages, acknowledgments, subscriptions, subscription cancellations, transaction boundaries, etc.—are stored in a rolling log. Because new events are always appended to the end of the log, a data log file can be updated extremely rapidly.

Implicitly, the data logs contain all of the message data and all of the information about destinations, subscriptions, transactions, etc.. This data, however, is stored in an arbitrary manner. In order to facilitate rapid access to the content of the logs, the message store constructs metadata to reference the data embedded in the logs.

Metadata cache

The metadata cache is an in-memory cache consisting mainly of destinations and message references. That is, for each JMS destination, the metadata cache holds a tree of message references, giving the location of every message in the data log files. Each message reference maps a message ID to a particular offset in one

of the data log files. The tree of message references is maintained using a B-tree algorithm, which enables rapid searching, insertion, and deletion operations on an ordered list of messages.

The metadata cache is periodically written to the *metadata store* on the file system. This procedure is known as *check pointing* and the length of time between checkpoints is configurable using the `checkpointInterval` configuration attribute. For details on how to configure the metadata cache, see ["Optimizing the Metadata Cache" on page 24](#).

Metadata store

The metadata store contains the complete broker metadata, consisting mainly of a B-tree index giving the message locations in the data logs. The metadata store is written to a file called `db.data`, which is periodically updated from the metadata cache.

The metadata store duplicates data that is already stored in the data logs (in a raw, unordered form). The presence of the metadata store, however, enables the broker instance to restart rapidly. If the metadata store got damaged or was accidentally deleted, the broker could recover by reading the data logs, but the restart would then take a considerable length of time.

Configuring the KahaDB Message Store

Overview

Fuse Message Broker's default configuration includes a persistence adapter that uses a KahaDB message store. The default configuration is suitable for many use cases, but you will likely want to update it for individual broker instances. You do this using the attributes of the kahaDB element.

The basic configuration tells the broker where to write the data files used by the store.

The KahaDB message store also has a number of advanced configuration attributes that customize its behavior.

Basic configuration

The KahaDB message store is configured by placing a kahaDB element in the persistenceAdapter element of your broker's configuration. The kahaDB element's attributes are used to configure the message store.

The attributes, listed in [Table 2.1 on page 18](#), all have reasonable default values, so you are not required to specify values for them. However, you will want to explicitly specify the location of the message store's data files by providing a value for the directory attribute. This will ensure that the broker will not conflict with other brokers.

[Example 2.1 on page 17](#) shows a basic configuration of the KahaDB message store. The KahaDB files are stored under the activemq-data directory.

Example 2.1. Configuring the KahaDB Message Store

```
<broker brokerName="broker" persistent="true" ... >
  ...
  <persistenceAdapter>
    <kahaDB directory="activemq-data" />
  </persistenceAdapter>
  ...
</broker>
```

Configuration attributes

[Table 2.1 on page 18](#) describes the attributes that can be used to configure the KahaDB message store.

Table 2.1. Configuration Properties of the KahaDB Message Store

Attribute	Default Value	Description
directory	activemq-data	Specifies the path to the top-level folder that holds the message store's data files.
indexWriteBatchSize	1000	Specifies the number of B-tree indexes written in a batch. Whenever the number of changed indexes exceeds this value, the metadata cache is written to disk.
indexCacheSize	10000	Specifies the number of B-tree index pages cached in memory.
enableIndexWriteAsync	false	Specifies if kahaDB will asynchronously write indexes.
journalMaxFileLength	32mb	Specifies the maximum size of the data log files.
enableJournalDiskSyncs	true	Specifies whether every non-transactional journal write is followed by a disk sync. If you want to satisfy the JMS durability requirement, you must also disable concurrent store and dispatch.
cleanupInterval	30000	Specifies the time interval, in milliseconds, between cleaning up data logs that are no longer used.
checkpointInterval	5000	Specifies the time interval, in milliseconds, between writing the metadata cache to disk.
ignoreMissingJournalfiles	false	Specifies whether the message store ignores any missing journal files while it starts up. If false, the message store raises an exception when it discovers a missing journal file.
checkForCorruptJournalFiles	false	Specifies whether the message store checks for corrupted journal files on startup and tries to recover them.

Attribute	Default Value	Description
checksumJournalFiles	false	Specifies whether the message store generates a checksum for the journal files. If you want to be able to check for corrupted journals, you must set this to <code>true</code> .
archiveDataLogs	false	Specifies if the message store moves spent data logs to the archive directory.
directoryArchive	null	Specifies the location of the directory to archive data logs.
databaseLockedWaitDelay	10000	Specifies the time delay, in milliseconds, before trying to acquire the database lock in the context of a shared master/slave failover deployment. See "Shared File System Master/Slave" in <i>Fault Tolerant Messaging</i> .
maxAsyncJobs	10000	Specifies the size of the task queue used to buffer the broker commands waiting to be written to the journal. The value should be greater than or equal to the number of concurrent message producers. See "Concurrent Store and Dispatch" on page 20.
concurrentStoreAndDispatchTopics	false	Specifies if the message store dispatches topic messages to interested clients concurrently with message storage. See "Concurrent Store and Dispatch" on page 20.
concurrentStoreAndDispatchQueues	true	Specifies if the message store dispatches queue messages to clients concurrently with message storage. See "Concurrent Store and Dispatch" on page 20.

Concurrent Store and Dispatch

Overview

Concurrent store and dispatch is a strategy that facilitates high rates of message throughput, provided the consumers are able to keep up with the flow of messages from the broker. By allowing the storing of messages to proceed concurrently with the dispatch of those messages to consumers, it can happen that the consumers return acknowledgments before the messages are ever written to disk. In this case, the message writes can be optimized away, because the dispatch has already completed.

Enabling concurrent store and dispatch

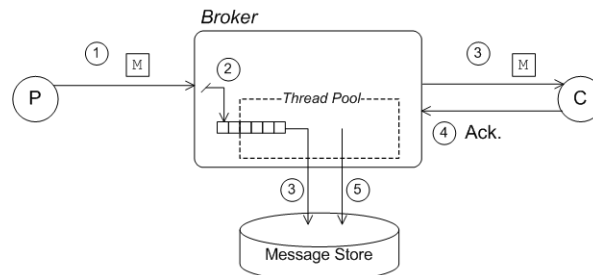
Concurrent store and dispatch is enabled by default for queues.

If you want to enable concurrent store and dispatch for topics, you must set the kahaDB element's `concurrentStoreAndDispatchTopics` attribute to `true`.

Concurrent with slow consumers

Figure 2.2 on page 20 shows an outline what happens in the broker when concurrent store and dispatch is enabled and the attached consumers are relatively *slow* to acknowledge messages.

Figure 2.2. Concurrent Store and Dispatch—Slow Consumers



In the *slow consumer* case, concurrent store and dispatch behaves as follows:

1. The producer sends a message, M, to a destination on the broker.
2. The broker sends the message, M, to the persistence layer. Because concurrency is enabled, the message is initially held in a task queue, which is serviced by a pool of threads that are responsible for writing to the journal.

3. Storing and dispatching are now performed concurrently. The message is dispatched either to one consumer (queue destination) or possibly to multiple destinations (topic consumer). In the meantime, because the attached consumers are slow, we can be sure that the thread pool has already pulled the message off the task queue and written it to the journal.
4. The consumer(s) acknowledge receipt of the message.
5. The broker asks the persistence layer to remove the message from persistent storage, because delivery is now complete.



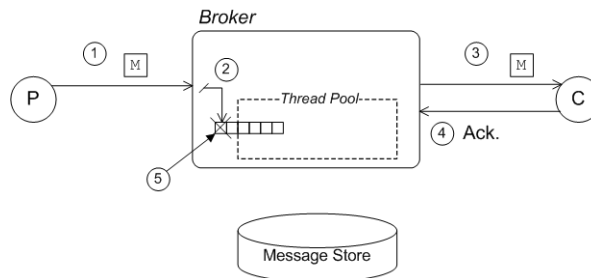
Note

In practice, because the KahaDB persistence layer is *not* able to remove the message from the rolling log files, KahaDB simply logs the fact that delivery of this message is complete. (At some point in the future, when all of the messages in the log file are marked as complete, the entire log file will be deleted.)

Concurrent with fast consumers

Figure 2.3 on page 21 shows an outline what happens in the broker when concurrent store and dispatch is enabled and the attached consumers are relatively *fast* at acknowledging messages.

Figure 2.3. Concurrent Store and Dispatch—Fast Consumers



In the *fast consumer* case, concurrent store and dispatch behaves as follows:

1. The producer sends a message, *M*, to a destination on the broker.
2. The broker sends the message, *M*, to the persistence layer. Because concurrency is enabled, the message is initially held in a queue, which is serviced by a pool of threads.

3. Storing and dispatching are now performed concurrently. The message is dispatched to one or more consumers.

In the meantime, assuming that the broker is fairly heavily loaded, it is probable that the message has not yet been written to the journal.

4. Because the consumers are fast, they rapidly acknowledge receipt of the message.
5. When all of the consumer acknowledgments are received, the broker asks the persistence layer to remove the message from persistent storage. But in the current example, the message is still pending and *has not been written to the journal*. The persistence layer can therefore remove the message just by deleting it from the in-memory task queue.

Disabling concurrent store and dispatch

If you want to configure the KahaDB message store to use serialized store and dispatch, you must explicitly disable concurrent store and dispatch for queues. [Example 2.2 on page 22](#) explicitly disables the store and dispatch feature for queues and topics.

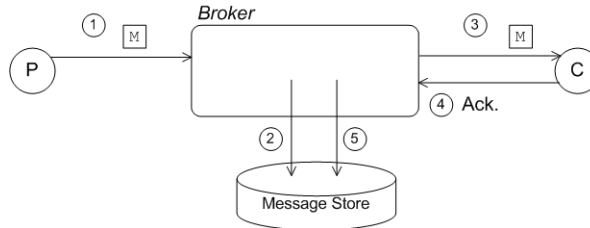
Example 2.2. KahaDB Configured with Serialized Store and Dispatch

```
<broker brokerName="broker" persistent="true" useShutdownHook="false">
  ...
  <persistenceAdapter>
    <kahaDB directory="activemq-data"
      journalMaxFileLength="32mb"
      concurrentStoreAndDispatchQueues="false"
      concurrentStoreAndDispatchTopics="false"
    />
  </persistenceAdapter>
</broker>
```

The serialized configuration results in a slower performance for the broker, but it also eliminates the risk of losing messages in the event of a system failure.

Serialized store and dispatch

[Figure 2.4 on page 23](#) shows an outline what happens in the broker when concurrent store and dispatch is *disabled*, so that the store and dispatch steps are performed in sequence.

Figure 2.4. Serialized Store and Dispatch

In the serialized case, the store and dispatch steps occur as follows:

1. The producer sends a message, *M*, to a destination on the broker.
2. The broker sends the message, *M*, to the persistence layer. Because concurrency is disabled, the message is immediately written to the journal (assuming `enableJournalDiskSyncs` is `true`).
3. The message is dispatched to one or more consumers.
4. The consumers acknowledge receipt of the message.
5. When all of the consumer acknowledgments are received, the broker asks the persistence layer to remove the message from persistent storage (in the case of the KahaDB, this means that the persistence layer records in the journal that delivery of this message is now complete).

JMS durability requirements

In order to avoid losing messages, the JMS specification requires the broker to persist each message received from a producer, *before* sending an acknowledgment back to the producer. In the case of JMS transactions, the requirement is to persist the transaction data (including the messages in the transaction scope), before acknowledging a *commit* directive. Both of these conditions ensure that data is not lost.

Make sure that the message saves are synced to disk right away by setting the kahaDB element's `enableJournalDiskSyncs` attribute to `true`.



Tip

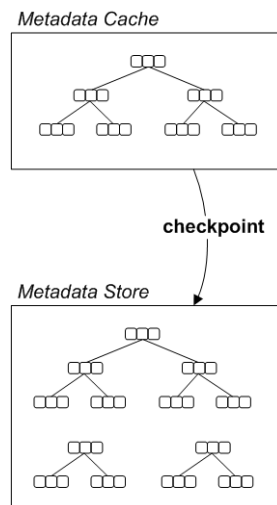
`true` is the default value for the `enableJournalDiskSyncs` attribute.

Optimizing the Metadata Cache

Overview

Proper configuration of the metadata cache is one of the key factors affecting the performance of the KahaDB message store. In a production deployment, therefore, you should always take the time to tune the properties of the metadata cache for maximum performance. [Figure 2.5 on page 24](#) shows an overview of the metadata cache and how it interacts with the metadata store. The most important part of the metadata is the B-tree index, which is shown as a tree of nodes in the figure. The data in the cache is periodically synchronized with the metadata store, when a checkpoint is performed.

Figure 2.5. Overview of the Metadata Cache and Store



Synchronizing with the metadata store

The metadata in the cache is constantly changing, in response to the events occurring in the broker. It is therefore necessary to write the metadata cache to disk, from time to time, in order to restore consistency between the metadata cache and the metadata store. There are two distinct mechanisms that can trigger a synchronization between the cache and the store, as follows:

- **Batch threshold**—as more and more of the B-tree indexes are changed, and thus inconsistent with the metadata store, you can define a threshold for the number of these *dirty indexes*. When the number of dirty indexes exceeds the threshold, KahaDB writes the cache to the store. The threshold value is set using the `indexWriteBatchSize` property.

- *Checkpoint interval*—irrespective of the current number of dirty indexes, the cache is synchronized with the store at regular time intervals, where the time interval is specified in milliseconds using the `checkpointInterval` property.

In addition, during a normal shutdown, the final state of the cache is synchronized with the store.

Setting the cache size

In the ideal case, the cache should be big enough to hold *all* of the KahaDB metadata in memory. Otherwise, the cache is forced to swap pages in and out of the persistent metadata store, which causes a considerable drag on performance.

You can specify the cache size using the `indexCacheSize` property, which specifies the size of the cache in units of pages (where one page is 4 KB by default). Generally, the cache should be as large as possible. You can check the size of your metadata store file, `db.data`, to get some idea of how big the cache needs to be.

Setting the write batch size

The `indexWriteBatchSize` defines the threshold for the number of dirty indexes that are allowed to accumulate, before KahaDB writes the cache to the store. Normally, these batch writes occur between checkpoints.

If you want to maximize performance, however, you could suppress the batch writes by setting `indexWriteBatchSize` to a very large number. In this case, the store would be updated only during checkpoints. The tradeoff here is that there is a risk of losing a relatively large amount of metadata, in the event of a system failure (but the broker should be able to restore the lost metadata when it restarts, by reading the tail of the journal).

Recovery

Overview

KahaDB supports a variety of mechanisms that enable it to recover and restart after a disorderly shutdown (system failure). This includes features to detect missing data files and to restore corrupted metadata. These features on their own, however, are *not* sufficient to guard completely against loss of data in the event of a system failure. If your broker is expected to mediate critical data, it is recommended that you deploy a disaster recovery system, such as a [RAID¹](http://en.wikipedia.org/wiki/RAID) disk array, to protect your data.

Clean shutdown

When the broker shuts down normally, the KahaDB message store flushes its cached data (representing the final state of the broker) to the file system. Specifically, the following information is written to the file system:

- All of the outstanding journal entries.
- All of the cached metadata.

Because this data represents the final state of the broker, the metadata store and the journal's data logs are consistent with each other after shutdown is complete. That is, the stored metadata takes into account *all* the commands recorded in the journal.

Recovery from disorderly shutdown

Normally, the journal tends to run ahead of the metadata store, because the journal is constantly being updated, whereas the metadata store is written only periodically (for example, whenever there is a checkpoint). Consequently, whenever there is a disorderly shutdown (which prevents the final state of the broker from being saved), it is likely that the stored metadata will be inconsistent with the journal, with the journal containing additional events not reflected in the metadata store.

When the broker restarts after a disorderly shutdown, the KahaDB message store recovers by reading the stored metadata into the cache and then reading the additional journal events not yet taken into account in the stored metadata (KahaDB can easily locate the additional journal events, because the metadata store always holds a reference to the last consistent location in the journal). KahaDB replays the additional journal events in order to recreate the original metadata.

¹ <http://en.wikipedia.org/wiki/RAID>



Note

The KahaDB message store also uses a redo log, `db.redo`, to reduce the risk of a system failure occurring while updating the metadata store. Before updating the metadata store, KahaDB always saves the redo log, which summarizes the changes that are about to be made to the store. Because the redo log is a small file, it can be written relatively rapidly and is thus less likely to be affected by a system failure. During recovery, KahaDB checks whether the changes recorded in the redo log need to be applied to the metadata.

Forcing recovery by deleting the metadata store

If the metadata store somehow becomes irretrievably corrupted, you can force recovery as follows (assuming the journal's data logs are clean):

1. While the broker is shut down, delete the metadata store, `db.data`.
2. Start the broker.
3. The broker now recovers by re-reading the *entire* journal and replaying all of the events in the journal in order to recreate the missing metadata.

While this is an effective means of recovering, you should bear in mind that it could take a considerable length of time if the journal is large.

Missing journal files

KahaDB has the ability to detect when journal files are missing. If one or more journal files are detected to be missing, the default behavior is for the broker to raise an exception and shut down. This gives an administrator the opportunity to investigate what happened to the missing journal files and to restore them manually, if necessary.

If you want the broker to ignore any missing journal files and continue processing regardless, you can set the `ignoreMissingJournalfiles` property to `true`.

Checking for corrupted journal files

KahaDB has a feature that checks for corrupted journal files, but this feature must be explicitly enabled. [Example 2.3 on page 27](#) shows how to configure a KahaDB message store to detect corrupted journal files.

Example 2.3. Configuration for Journal Validation

```
<persistenceAdapter>
  <kahaDB directory="activemq-data">
```

```
    journalMaxFileLength="32mb"  
    checksumJournalFiles="true"  
    checkForCorruptJournalFiles="true"  
  />  
</persistenceAdapter>
```

Chapter 3. Using a Distributed KahaDB Persistence Adapter

When you have destinations with different performance profiles or different persistence requirements you can distribute them across multiple KahaDB message stores.

Overview

The stock KahaDB persistence adapter works well when all of the destinations being managed by the broker have similar performance and reliability profiles. When one destination has a radically different performance profile, for example its consumer is exceptionally slow compared to the consumers on other destinations, the message store's disk usage can grow rapidly. When one or more destinations don't require disc synchronization and the others do require it, all of the destinations must take the performance hit.

The *distributed KahaDB persistence adapter* allows you to distribute a broker's destinations across multiple KahaDB message stores. Using multiple message stores allows you to tailor the message store more precisely to the needs of the destinations using it. Destinations and stores are matched using filters that take standard wild card syntax.

Configuration

The distributed KahaDB persistence adapter configuration wraps more than one KahaDB message store configuration.

The distributed KahaDB persistence adapter configuration is specified using the `mKahaDB` element. The `mKahaDB` element has a single attribute, `directory`, that specifies the location where the adapter writes its data stores. This setting is the default value for the `directory` attribute of the embedded KahaDB message store instances. The individual message stores can override this default setting.

The `mKahaDB` element has a single child `filteredPersistenceAdapters`. The `filteredPersistenceAdapters` element contains multiple `filteredKahaDB` elements that configure the KahaDB message stores that are used by the persistence adapter.

Each `filteredKahaDB` element configures one KahaDB message store. The destinations matched to the message store are specified using attributes on the `filteredKahaDB` element:

- `queue`—specifies the name of queues
- `topic`—specifies the name of topics

The destinations can be specified either using explicit destination names or using wildcards. For information on using wildcards see ["Filters" on page 30](#). If no destinations are specified the message store will match any destinations that are not matched by other filters.

The KahaDB message store configured by a `filteredKahaDB` element is configured using the standard KahaDB persistence adapter configuration. It consists of a `kahaDB` element wrapped in a `persistenceAdapter` element. For details on configuring a KahaDB message store see ["Configuring the KahaDB Message Store" on page 17](#).

Filters

You can use wildcards to specify a group of destination names. This is useful for situations where your destinations are set up in federated hierarchies.

For example, imagine you are sending price messages from a stock exchange feed. You might name your destinations as follows:

- `PRICE.STOCK.NASDAQ.ORCL` to publish Oracle Corporation's price on NASDAQ
- `PRICE.STOCK.NYSE.IBM` to publish IBM's price on the New York Stock Exchange

You could use exact destination names to specify which message store will be used to persist message data, or you could use wildcards to define hierarchical pattern matches to the pair the destinations with a message store.

Fuse Message Broker uses the following wild cards:

- `.` separates names in a path
- `*` matches any name in a path
- `>` recursively matches any destination starting from this name

For example using the names above, these filters are possible:

- `PRICE.>`—any price for any product on any exchange
- `PRICE.STOCK.>`—any price for a stock on any exchange
- `PRICE.STOCK.NASDAQ.*`—any stock price on NASDAQ
- `PRICE.STOCK.*.IBM`—any IBM stock price on any exchange

Example

[Example 3.1 on page 31](#) shows a distributed KahaDB persistence adapter that distributes destinations accross two KahaDB message stores. The first message store is used for all queues managed by the broker. The second message store will is used for all other destinations. In this case, it will be used for all topics.

Example 3.1. Distributed KahaDB Persistence Adapter Configuration

```
<persistenceAdapter>
  <mKahaDB directory="${activemq.base}/data/kahadb">
    <filteredPersistenceAdapters>
      <!-- match all queues -->
      <filteredKahaDB queue="">
        <persistenceAdapter>
          <kahaDB journalMaxFileLength="32mb"/>
        </persistenceAdapter>
      </filteredKahaDB>

      <!-- match all destinations -->
      <filteredKahaDB>
        <persistenceAdapter>
          <kahaDB enableJournalDiskSyncs="false"/>
        </persistenceAdapter>
      </filteredKahaDB>
    </filteredPersistenceAdapters>
  </mKahaDB>
</persistenceAdapter>
```

Transactions

Transactions can span multiple journals if the destinations are distributed. This means that two phase completion is required. This does incur the performance penalty of the additional disk sync to record the commit outcome.

If only one journal is involved in the transaction, the additional disk sync is not used. The performance penalty is not incurred in this case.

Chapter 4. Using JDBC to Connect to a Database Store

Fuse Message Broker supports the use of relational databases as a message store through JDBC. You can use the JDBC persistence adapter either coupled with a high performance journal or standalone.

Basics of Using the JDBC Persistence Adapter	34
Using JDBC with the High Performance Journal	37
Using JDBC without the Journal	40
Customizing the JDBC Persistence Adapter	43

Basics of Using the JDBC Persistence Adapter

Overview

For long term persistence you may want to use a relational database as your persistent message store. Fuse Message Broker's default database when using the JDBC persistence adapter is Apache Derby. Fuse Message Broker also supports most major SQL databases. You can enable other databases by properly configuring the JDBC connection in the broker's configuration file.

You can use the JDBC persistence adapter either with or without journaling. Using the journal provides two main benefits. First, it improves the speed of the message store. Second, it provides support for JMS transactions.

Supported databases

Fuse Message Broker is known to work with the following databases:

- Apache Derby
- Axion
- DB2
- HSQL
- Informix
- MaxDB
- MySQL
- Oracle
- PostgreSQL
- SQLServer
- Sybase

In addition, Fuse Message Broker supports a number of generic JDBC providers.

Specifying the type of JDBC store to use

Fuse Message Broker support two types of JDBC store:

- *A journaled JDBC store:*

The journaled JDBC store is specified using the `journaledJDBC` element inside the `persistenceFactory` element. For more details see ["Using JDBC with the High Performance Journal" on page 37](#).

- *A non-journaled JDBC store:*

The non-journaled store is specified using the `jdbcPersistenceAdapter` element inside the `persistenceAdapter` element. For more details see ["Using JDBC without the Journal" on page 40](#).

The journaled JDBC store features better performance than the plain JDBC store; but the journaled JDBC store is incompatible with the JDBC master/slave failover pattern—see ["Shared JDBC Master/Slave" in Fault Tolerant Messaging](#).

Configuring your JDBC driver

Fuse Message Broker autodetects the JDBC driver that is in use at start-up. For the supported databases, the JDBC adapter automatically adjusts the SQL statements and JDBC driver methods to work with the driver. If you wish to customize the names of the database tables or work with an unsupported database, you can modify both the SQL statements and the JDBC driver methods. See ["Customizing the SQL statements used by the adapter" on page 43](#) for information about modifying the SQL statements. See ["Using generic JDBC providers" on page 45](#) for information about changing the JDBC methods.

JDBC configuration for Apache Derby

Fuse Message Broker provides sample configurations of various JDBC databases in `InstallDir/conf/activemq-jdbc.xml`. [Example 4.1 on page 35](#) shows the configuration for using the default JDBC database (Apache Derby).

Example 4.1. Configuration for the Apache Derby Database

```
<beans ...>
  <broker xmlns="http://activemq.apache.org/schema/core"
    brokerName="localhost">
    ...
    <persistenceAdapter>
      <jdbcPersistenceAdapter
        dataDirectory="${activemq.base}/data"
        dataSource="#derby-ds"/>
    </persistenceAdapter>
    ...
  </broker>

  <!-- Embedded Derby DataSource Sample Setup -->
  <bean id="derby-ds" class="org.apache.derby.jdbc.EmbeddedDataSource">
    <property name="databaseName" value="derbydb"/>
  </bean>
</beans>
```

```
<property name="createDatabase" value="create"/>
</bean>

</beans>
```

JDBC configuration for Oracle

[Example 4.2 on page 36](#) shows the configuration for using the Oracle JDBC driver. The persistence adapter configuration refers to the Spring bean element that configures the JDBC driver.

Example 4.2. Configuration for the Oracle JDBC Driver

```
<beans ... >
  <broker xmlns="http://activemq.apache.org/schema/core"
    brokerName="localhost">
    ...
    <persistenceAdapter>
      <jdbcPersistenceAdapter
        dataDirectory="${activemq.base}/data"
        dataSource="#oracle-ds"/>
    </persistenceAdapter>
    ...
  </broker>

  <!-- Oracle DataSource Sample Setup -->
  <bean id="oracle-ds"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:AMQDB"/>
    <property name="username" value="scott"/>
    <property name="password" value="tiger"/>
    <property name="maxActive" value="200"/>
    <property name="poolPreparedStatements" value="true"/>
  </bean>
</beans>
```

The JDBC drivers are configured using a Spring bean element. The `id` attribute specifies the name by which you will refer to the driver when configuring the JDBC persistence adapter. The `class` attribute specifies the class that implements the data source used to interface with the JDBC driver. The `destroy-method` attribute specifies the name of the method to call when the JDBC driver is shutdown.

In addition to the bean element, the JDBC driver configuration includes a number of property elements. Each property element specifies a property required by the JDBC driver. For information about the configurable properties refer to your JDBC driver's documentation.

Using JDBC with the High Performance Journal

Overview

Using the JDBC persistence adapter with Fuse Message Broker's high performance journal boosts the performance of the persistence adapter in two ways:

1. In applications where message consumers keep up with the message producers, the journal makes it possible to lower the number of messages that need to be committed to the data base. For example a message producer could publish 10,000 messages between journal checkpoints. If the message consumer pops 9,900 messages off of the queue during the same interval, only 100 messages will be committed to the database through the JDBC adapter.
2. In applications where the message consumers cannot keep up with the message producers, or in applications where messages must persist for long periods, the journal boosts performance by committing messages in large batches. This means that the JDBC driver can optimize the writes to the external database.

In addition to the performance gains, the high performance journal also makes it possible to ensure the consistency of JMS transactions in the case of a system failure.

Example

[Example 4.3 on page 37](#) shows a configuration fragment that configures the journaled JDBC adapter to use a MySQL database.

Example 4.3. Configuring Fuse Message Broker to use the Journaled JDBC Persistence Adapter

```
<beans ... >
  <broker ...>
    ...
    ❶ <persistenceFactory>
    ❷   <journaledJDBC journalLogFiles="5" dataSource="#mysql-ds" />
    </persistenceFactory>
    ...
  </broker>
  ...
  ❸<bean id="mysql-ds"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost/activemq?relaxAutoCommit=true"/>
    <property name="username" value="activemq"/>
    <property name="password" value="activemq"/>
    <property name="poolPreparedStatements" value="true"/>
  </bean>
```

The configuration in [Example 4.3 on page 37](#) has three noteworthy elements:

- ❶ The `persistenceFactory` element wraps the configuration for the JDBC persistence adapter.



Note

In Fuse Message Broker versions prior to 5.4, the `persistenceAdapter` element was sometimes used to wrap the configuration for the JDBC persistence adapter. This is no longer possible in 5.4, because the broker performs strict schema checking on the configuration file.

- ❷ The `journalledJDBC` element specifies that the broker will use the JDBC persistence adapter with the high performance journal. The element's attributes configure the following properties:
 - The journal will span five log files.
 - The configuration for the JDBC driver is specified in a bean element with the ID, `mysql-ds`.
- ❸ The bean element specified the configuration for the MySQL JDBC driver.

Configuration

[Table 4.1 on page 38](#) describes the attributes used to configure the journaled JDBC persistence adapter.

Table 4.1. Attributes for Configuring the Journaled JDBC Persistence Adapter

Attribute	Default Value	Description
<code>adapter</code>		Specifies the strategy to use when accessing a non-supported database. For more information see "Using generic JDBC providers" on page 45 .
<code>createTablesOnStartup</code>	<code>true</code>	Specifies whether or not new database tables are created when the broker starts. If the database tables already exist, the existing tables are reused.
<code>dataDirectory</code>	<code>activemq-data</code>	Specifies the directory into which the default Derby database writes its files.
<code>dataSource</code>	<code>#derby</code>	Specifies the id of the Spring bean storing the JDBC driver's

Attribute	Default Value	Description
		configuration. For more information see "Configuring your JDBC driver" on page 35 .
journalArchiveDirectory		Specifies the directory used to store archived journal log files.
journalLogFiles	2	Specifies the number of log files to use for storing the journal.
journalLogFileSize	20MB	Specifies the size for a journal's log file.
journalThreadPriority	10	Specifies the thread priority of the thread used for journaling.
useDatabaseLock	true	Specifies whether or not an exclusive database lock should be used to enable JDBC Master/Slave.
useJournal	true	Specifies whether or not to use the journal.

Using JDBC without the Journal

Overview

For those cases where journaling is not appropriate, or you wish to use your own journaling system, you can use the JDBC persistence adapter without the Fuse Message Broker high performance journal.

Example

[Example 4.4 on page 40](#) shows a configuration fragment that configures the plain JDBC adapter to use the Apache Derby database.

Example 4.4. Configuring Fuse Message Broker to use the Plain JDBC Persistence Adapter

```
<beans ... >
  <broker ...>
    ...
    ❶ <persistenceAdapter>
      ❷ <jdbcPersistenceAdapter dataSource="#derby-ds" />
    </persistenceAdapter>
    ...
  </broker>
  ...
  ❸<bean id="derby-ds" class="org.apache.derby.jdbc.EmbeddedDataSource">
    <property name="databaseName" value="derbydb"/>
    <property name="createDatabase" value="create"/>
  </bean>
```

The configuration in [Example 4.4 on page 40](#) has three noteworthy elements:

- ❶ The `persistenceAdapter` element wraps the configuration for the JDBC persistence adapter.
- ❷ The `jdbcPersistenceAdapter` element specifies that the broker will use the plain JDBC persistence adapter and that the JDBC driver's configuration is specified in a bean element with the ID, `derby-ds`.
- ❸ The bean element specified the configuration for the Derby JDBC driver.

Configuration

[Table 4.2 on page 40](#) describes the attributes used to configure the non-journaled JDBC persistence adapter.

Table 4.2. Attributes for Configuring the Plain JDBC Persistence Adapter

Attribute	Default Value	Description
adapter		Specifies the strategy to use when accessing a non-supported

Attribute	Default Value	Description
		database. For more information see "Using generic JDBC providers" on page 45 .
cleanupPeriod	300000	Specifies, in milliseconds, the interval at which acknowledged messages are deleted.
createTablesOnStartup	true	Specifies whether or not new database tables are created when the broker starts. If the database tables already exist, the existing tables are reused.
databaseLocker	DefaultDatabaseLocker instance	The locker instance that prevents more than one broker from accessing the database at a time.
dataDirectory	activemq-data	Specifies the directory into which the default Derby database writes its files.
dataSource	#derby	Specifies the id of the Spring bean storing the JDBC driver's configuration. For more information see "Configuring your JDBC driver" on page 35 .
lockAcquireSleepInterval	1000	Time interval (ms) between lock acquire attempts, when using the default database locker. Does not apply to custom database lockers, specified using databaseLocker.
lockKeepAlivePeriod	30000	Periodically save the current time in the locker table, at time intervals specified by this option (ms). This ensure that the lock does not time out.
useDatabaseLock	true	Specifies whether or not an exclusive database lock should be used to enable JDBC Master/Slave—see "Shared JDBC Master/Slave" in Fault Tolerant Messaging .

Attribute	Default Value	Description
transactionIsolation	Connection.TRANSACTION_READ_UNCOMMITTED	Specifies the required transaction isolation level. For allowed values, see java.sql.Connection ¹ .

¹ http://download-l1nw.oracle.com/docs/cd/E17476_01/javase/1.5.0/docs/api/java/sql/Connection.html

Customizing the JDBC Persistence Adapter

Overview

Fuse Message Broker provides options to customize the interaction between the JDBC persistence adapter and the underlying database. In some cases you might be able to use these customization options to integrate the JDBC persistence adapter with an unsupported database.

Customizing the SQL statements used by the adapter

You can customize the SQL statements that the JDBC persistence adapter uses to access the database. This is done by adding a `statements` element to the JDBC persistence adapter configuration.

[Example 4.5 on page 43](#) shows a configuration snippet that specifies that long strings are going to be stored as `VARCHAR(128)`.

Example 4.5. Fine Tuning the Database Schema

```
<persistenceFactory>
  <journalJDBC ... >
    <statements>
      <statements stringIdDataType ="VARCHAR(128)"/>
    </statements>
  </journalJDBC>
</persistenceFactory>
```

The first `statements` element is a wrapper for one or more nested `statements` elements. Each nested `statements` element specifies a single configuration statement. [Table 4.3 on page 43](#) describes the configurable properties.

Table 4.3. Statements for Configuring the SQL Statements Used by the JDBC Persistence Adapter

Attribute	Default	Description
tablePrefix		Specifies a prefix that is added to every table name. The prefix should be unique per broker if multiple brokers will be sharing the same database.
messageTableName	ACTIVEMQ_MSGS	Specifies the name of the table in which persistent messages are stored.
durableSubAcksTableName	ACTIVEMQ_ACKS	Specifies the name of the database table used to store acknowledgment messages from durable subscribers.

Attribute	Default	Description
lockTableName	ACTIVEMQ_LOCK	Specifies the name of the lock table used to determine the master in a master/slave scenario.
binaryDataType	BLOB	Specifies the data type used to store the messages.
containerNameDataType	VARCHAR(250)	Specifies the data type used to store the destination name.
msgIdDataType	VARCHAR(250)	Specifies the data type used to store a message id.
sequenceDataType	INTEGER	Specifies the datatype used to store the sequence id of a message.
longDataType	BIGINT	Specifies the data type used to store a Java long.
stringIdDataType	VARCHAR(250)	Specifies the data type used to store long strings like client ids, selectors, and broker names.

The properties listed in [Table 4.3 on page 43](#) configure the default SQL statements used by the JDBC adapter and work with all of the supported databases.

Customizing SQL statements for unsupported databases

If you need to override the default statements to work with an unsupported database, there are a number of other properties that can be set. These include:

- `addMessageStatement`
- `updateMessageStatement`
- `removeMessageStatement`
- `findMessageSequenceIdStatement`
- `findMessageStatement`
- `findAllMessagesStatement`
- `findLastSequenceIdInMsgsStatement`
- `findLastSequenceIdInAcksStatement`

- `createDurableSubStatement`
- `findDurableSubStatement`
- `findAllDurableSubsStatement`
- `updateLastAckOfDurableSubStatement`
- `deleteSubscriptionStatement`
- `findAllDurableSubMessagesStatement`
- `findDurableSubMessagesStatement`
- `findAllDestinationsStatement`
- `removeAllMessagesStatement`
- `removeAllSubscriptionsStatement`
- `deleteOldMessagesStatement`
- `lockCreateStatement`
- `lockUpdateStatement`
- `nextDurableSubscriberMessageStatement`
- `durableSubscriberMessageCountStatement`
- `lastAackedDurableSubscriberMessageStatement`
- `destinationMessageCountStatement`
- `findNextMessageStatement`
- `createSchemaStatements`
- `dropSchemaStatements`

Using generic JDBC providers

To use a JDBC provider not natively supported by Fuse Message Broker, you can configure the JDBC persistence adapter, by setting the persistence adapter's `adapter` attribute to reference the bean ID of the relevant adapter. The following adapter types are supported:

- `org.activemq.store.jdbc.adapter.BlobJDBCAdapter`
- `org.activemq.store.jdbc.adapter.BytesJDBCAdapter`
- `org.activemq.store.jdbc.adapter.DefaultJDBCAdapter`
- `org.activemq.store.jdbc.adapter.ImageJDBCAdapter`

Various settings are provided to customize how the JDBC adapter stores and accesses BLOB fields in the database. To determine the proper settings, consult the documentation for your JDBC driver and your database.

[Example 4.6 on page 46](#) shows a configuration snippet configuring the journaled JDBC persistence adapter to use the blob JDBC adapter.

Example 4.6. Configuring a Generic JDBC Provider

```
<broker persistent="true" ... >
  ...
  <persistenceFactory>
    <journaledJDBC adapter="#blobAdapter" ... />
  </persistenceFactory>

  <bean id="blobAdapter"
        class="org.activemq.store.jdbc.adapter.BlobJDBCAdapter"/>
  ...
</broker>
```

Chapter 5. Message Cursors

Fuse Message Broker uses message cursors to improve the scalability of the persistent message store. By default, a hybrid approach that uses an in memory dispatch queue for fast consumers and message cursors for slower consumers is used. Fuse Message Broker also supports two alternative cursor implementations. The type of cursor can be configured on a per-destination basis.

Types of Cursors	48
Configuring the Type of Cursor Used by a Destination	52

Message data is cached in the broker using *message cursors*, where a cursor instance is associated with each destination. A message cursor represents a batch of messages cached in memory. When necessary, a message cursor will retrieve persisted messages through the persistence adapter. But the key point you need to understand about message cursors is that the cursors are essentially *independent* of the persistence layer.

Message cursors provide a means for optimizing a persistent message store. They allow the persistent store to maintain a pointer to the next batch of messages to pull from the persistent message store. Fuse Message Broker has three types of cursors that can be used depending on the needs of your application:

- **Store-based** cursors are used by default to handle persistent messages.
- **VM** cursors are very fast, but cannot handle slow message consumers.
- **File-based** cursors are used by default to handle non-persistent messages. They are useful when the message store is slow and message consumers are relatively fast.

Types of Cursors

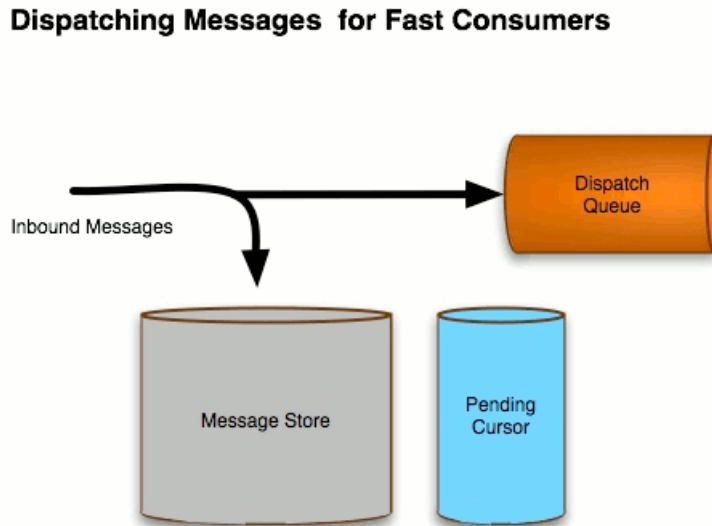
Store-based cursors

Store-based cursors are used, by default, for processing persistent messages. Store-based cursors are a hybrid implementation that offers the robustness of typical cursor implementations and the speed of in-memory message reference implementations.

Typically messaging systems will pull persistent messages from long-term storage in a batch when a client is ready to consume them. A cursor will be used to maintain the position for the next batch of messages. While this approach scales well and provides excellent robustness, it does not perform well when message consumers keep pace with message producers.

As shown in [Figure 5.1 on page 48](#), store-based cursors address the fast consumer case by skipping the message cursor. When a message consumer is keeping pace with the message producers, persistent messages are written to the message store and moved directly into a dispatch queue for the consumer.

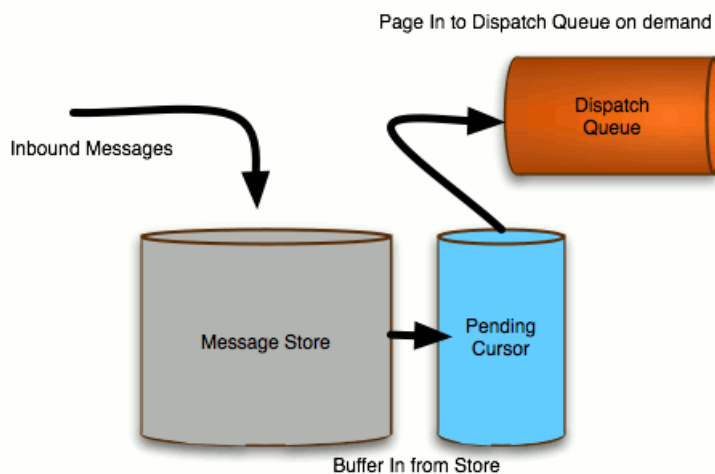
Figure 5.1. Store-based Cursors for a Fast Consumer



When a consumer starts with a back log of messages or falls behind its message producers, Fuse Message Broker changes the strategy used to dispatch messages. As shown in [Figure 5.2 on page 49](#), messages are held in the message store and fed into the consumer's dispatch queue using the pending cursor.

Figure 5.2. Store-based Cursors for a Slow Consumer

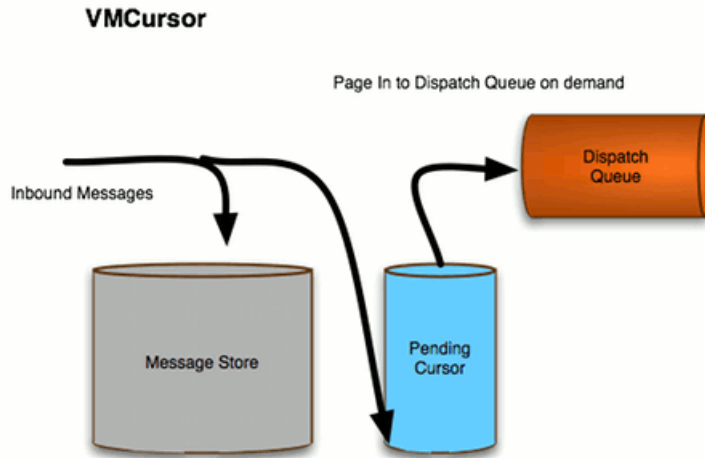
Dispatching Messages if Dispatch Queue is Full



VM cursors

When speed is the top priority and the consumers can definitely keep pace with the message producers, VM cursors could be the best approach. In this approach, shown in [Figure 5.3 on page 50](#), messages are written to the persistent store and then also stored in the pending cursor which is held completely in memory. The messages are fed into the dispatch queue from the pending cursor.

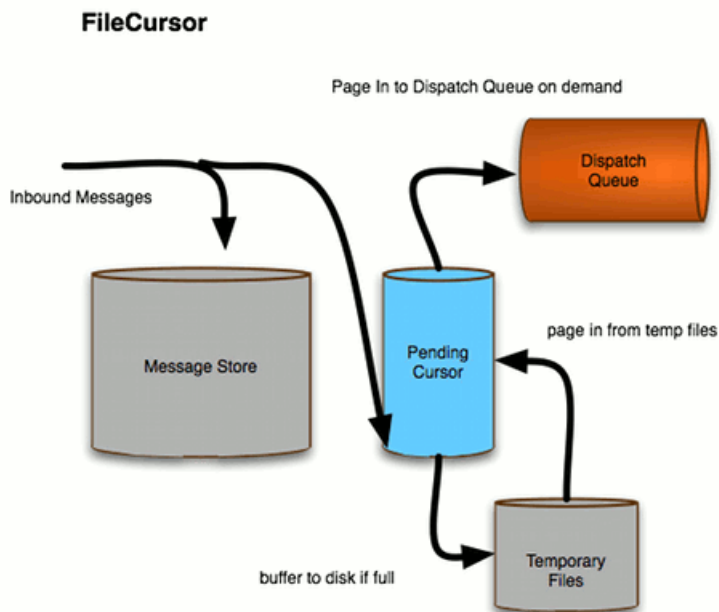
Figure 5.3. VM Cursors



Because the messages are dispatched from active memory when using VM cursors, this method is exceptionally fast. However, if the number of unconsumed messages gets large the producers will be throttled to avoid exceeding the available memory.

File-based cursors

File-based cursors are a variation of VM cursors that provides a buffer against running out of memory when a consumer falls behind. As shown in [Figure 5.4 on page 51](#), the broker pages messages out to a temporary file when the broker's memory limit is reached.

Figure 5.4. File-based Cursors

Using a temporary file cushions the broker against situations where a consumer occasionally falls behind or messages are produced in a burst. The broker uses the temporary file instead of resorting to using slower persistent storage.

File-based cursors do not scale well when consumers are frequently behind by a large margin. It is also not ideal when a fast long term message store is available.

File-based cursors are used, by default, to process non-persistent messages.

Configuring the Type of Cursor Used by a Destination

Overview

By default, Fuse Message Broker uses store-based cursors for persistent messages and file-based cursors for non-persistent messages. You can, however, configure your destinations to use a specified cursor implementation by adding the appropriate policy entries into the destination's policy map.

You configure a destination's policy set using a `destinationPolicy` element. The `destinationPolicy` element is a wrapper for a `policyMap` element. The `policyMap` element is a wrapper for a `policyEntries` element. The `policyEntries` element is a wrapper for one or more `policyEntry` elements.

The cursor policies are entered as children to a `policyEntry` element. The configuration elements used to specify the type of destination you are configuring. Topics use cursors for both durable subscribers and transient subscribers, so it uses two sets of configuration elements. Queues only a single cursor and only require a single set of configuration elements.

Configuring topics

Topics maintain a dispatch queue and a pending cursor for every consumer subscribed to the topic regardless of whether the subscription is durable or transient. You can configure the cursor implementation used by durable subscribers separately from the cursor implementation used by transient subscribers.

You configure the cursor implementation used by durable subscribers by adding `PendingDurableSubscriberMessageStoragePolicy` child element to the topic's `policyEntry` element. [Table 5.1 on page 52](#) describes the possible children of `PendingDurableSubscriberMessageStoragePolicy`.

Table 5.1. Elements for Configuring the Type of Cursor to Use for Durable Subscribers

Element	Description
<code>vmDurableCursor</code>	Specifies the VM cursors will be used. See "VM cursors" on page 49 for more information.
<code>fileDurableSubscriberCursor</code>	Specifies that file-based cursors will be used. See "File-based cursors" on page 50 for more information.

You configure the cursor implementation used by transient subscribers by adding `pendingSubscriberPolicy` child element to the topic's `policyEntry` element. [Table 5.2 on page 53](#) describes the possible children of `pendingSubscriberPolicy`.

Table 5.2. Elements for Configuring the Type of Cursor to Use for Transient Subscribers

Element	Description
vmCursor	Specifies the VM cursors will be used. See "VM cursors" on page 49 for more information.
fileCursor	Specifies that file-based cursors will be used. See "File-based cursors" on page 50 for more information.

[Example 5.1 on page 53](#) shows a configuration snippet that configures a topic to use VM cursors for its transient subscribers and file-based cursors for its durable subscribers.

Example 5.1. Configuring a Topic's Cursor Usage

```
<beans ... >
  <broker ... >
    ...
    <destinationPolicy>
      <policyMap>
        <policyEntries>
          <policyEntry topic="com.fusesource.>">
            ...
            <pendingSubscriberPolicy>
              <vmCursor />
            </pendingSubscriberPolicy>
            <PendingDurableSubscriberMessageStoragePolicy>
              <fileDurableSubscriberPolicy />
            </PendingDurableSubscriberMessageStoragePolicy>
            ...
          </policyEntry>
          ...
        </policyEntries>
      </policyMap>
    </destinationPolicy>
    ...
  </broker>
  ...
</beans>
```

Configuring queues

Queues use a single pending cursor and dispatch queue. You configure the type of cursor to use by adding a `pendingQueuePolicy` element to the queue's `policyEntry` element. [Table 5.3 on page 54](#) describes the possible children elements of the `pendingQueuePolicy` element.

Table 5.3. Elements for Configuring the Type of Cursor to Use for a Queue

Element	Description
vmQueueCursor	Specifies the VM cursors will be used. See "VM cursors" on page 49 for more information.
fileQueueCursor	Specifies that file-based cursors will be used. See "File-based cursors" on page 50 for more information.

[Example 5.2 on page 54](#) shows a configuration snippet that configures a queue to use VM cursors.

Example 5.2. Configuring a Queue's Cursor Usage

```
<beans ... >
  <broker ... >
    ...
    <destinationPolicy>
      <policyMap>
        <policyEntries>
          <policyEntry queue="com.fusesource.">
            ...
            <pendingQueuePolicy>
              <vmQueueCursor />
            </pendingQueuePolicy>
            ...
          </policyEntry>
          ...
        </policyEntries>
      </policyMap>
    </destinationPolicy>
    ...
  </broker>
  ...
</beans>
```

Index

B

- broker element, 11
 - persistent attribute, 11

C

- configuration
 - turning persistence on/off, 11
- cursors
 - file-based, 50
 - store-based, 48
 - VM, 49

D

- destinationPolicy, 52
- distributed Kahadb persistence adapter
 - transactions, 31
- durable subscribers
 - configuring cursors, 52
 - using file-based cursors, 52
 - using VM cursors, 52

F

- fileCursor, 53
- fileDurableSubscriberCursor, 52
- fileQueueCursor, 54
- filteredKahaDB, 29
- filteredPersistenceAdapters, 29

J

- JDBC
 - using generic providers, 45
- jdbcPersistenceAdapter, 40
 - adapter attribute, 40, 45
 - cleanupPeriod attribute, 41
 - createTablesOnStartup attribute, 41
 - dataDirectory attribute, 41
 - dataSource attribute, 41
 - useDatabaseLock attribute, 41

- journalizedJDBC, 38
 - adapter attribute, 38, 45
 - createTablesOnStartup attribute, 38
 - dataDirectory attribute, 38
 - dataSource attribute, 38
 - journalArchiveDirectory attribute, 39
 - journalLogFiles attribute, 39
 - journalLogFileSize attribute, 39
 - journalThreadPriority attribute, 39
 - useDatabaseLock attribute, 39
 - useJournal attribute, 39

K

- kahaDB element, 17
 - archiveDataLogs attribute, 19
 - checkForCorruptJournalFiles attribute, 18
 - checkpointInterval attribute, 18
 - checksumJournalFiles attribute, 19
 - cleanupInterval attribute, 18
 - concurrentStoreAndDispatchQueues attribute, 19
 - concurrentStoreAndDispatchTopics attribute, 19
 - databaseLockedWaitDelay attribute, 19
 - directory attribute, 18
 - directoryArchive attribute, 19
 - enableIndexWriteAsync attribute, 18
 - enableJournalDiskSyncs attribute, 18
 - ignoreMissingJournalfiles attribute, 18
 - indexCacheSize attribute, 18
 - indexWriteBatchSize attribute, 18
 - journalMaxFileLength attribute, 18
 - maxAsyncJobs attribute, 19

- KahaDB message store
 - architecture, 14
 - basic configuration, 17
 - configuration attributes, 17
 - data logs, 15
 - distributed, 29
 - metadata cache, 15
 - metadata store, 16

M

- mKahaDB, 29

P

- PendingDurableSubscriberMessageStoragePolicy, 52
- pendingQueuePolicy, 53
- pendingSubscriberPolicy, 52
- persistenceAdapter, 12
- persistenceFactory, 12
- policyEntries, 52
- policyEntry, 52
- policyMap, 52

S

- SQL data types, 43
- statements, 43
 - binaryDataType attribute, 44
 - containerNameDataType attribute, 44
 - durableSubAcksTableName attribute, 43
 - lockTableName attribute, 44
 - longDataType attribute, 44
 - messageTableName attribute, 43
 - msgIdDataType attribute, 44
 - sequenceDataType attribute, 44
 - stringIdDataType attribute, 44
 - tablePrefix attribute, 43

T

- transactions
 - distributed destination, 31
 - distributed Kahadb persistence adapter, 31
 - multiple journals, 31
- transient subscribers
 - configuring cursors, 52
 - using file-based cursors, 53
 - using VM cursors, 53

V

- vmCursor, 53
- vmDurableCursor, 52
- vmQueueCursor, 54