Red Hat JBoss A-MQ 6.0

Configuring Broker Persistence

Red Hat JBoss A-MQ's persistence layer can be tailored for speed and robustness

Last Updated: 2017-10-13

Red Hat JBoss A-MQ 6.0 Configuring Broker Persistence

Red Hat JBoss A-MQ's persistence layer can be tailored for speed and robustness

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Abstract

This guide discusses how to configure Red Hat JBoss A-MQ's persistence layer to best suite your application and your environment.

Table of Contents

CHAPTER 1. INTRODUCTION TO RED HAT JBOSS A-MQ PERSISTENCE

01/31/2013

added blurb about lockers

Abstract

Message persistence allows for the recovery of undelivered messages in the event of a system failure. By default, Red Hat JBoss A-MQ'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. Red Hat JBoss A-MQ 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, JBoss A-MQ 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. JBoss A-MQ implements several different kinds of message store, including:

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

MESSAGE CURSORS

JBoss A-MQ 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 Chapter 6, *Message Cursors* 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 JBoss A-MQ 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 JMSDeliveryMode 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 JMSDeliveryMode setting.

Example 1.1, “Turning Off a Broker's Persistence” 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

JBoss A-MQ 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.

CUSTOMIZING THE STORE'S LOCKER

For added flexibility in master/slave deployments JBoss A-MQ's message stores have configurable lockers. All of the message stores have a default locker implementation. The default implementation can be replaced by a custom implementation.

Regardless of the implementation, the locker has two configurable properties:

- if the broker should fail if the store is locked
- how long a broker waits before trying to reacquire a lock

CHAPTER 2. USING THE KAHADB MESSAGE STORE

Abstract

The KahaDB Message Store is the default message store used by Red Hat JBoss A-MQ. 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.

IMPORTANT

If you use antivirus software it can interfere with Red Hat JBoss A-MQ'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.

2.1. UNDERSTANDING THE KAHADB MESSAGE STORE

Overview

The KahaDB message store is the default persistence store used by Red Hat JBoss A-MQ. It is a file-based persistence adapter that is optimized for maximum performance. The main features of KahaDB are:

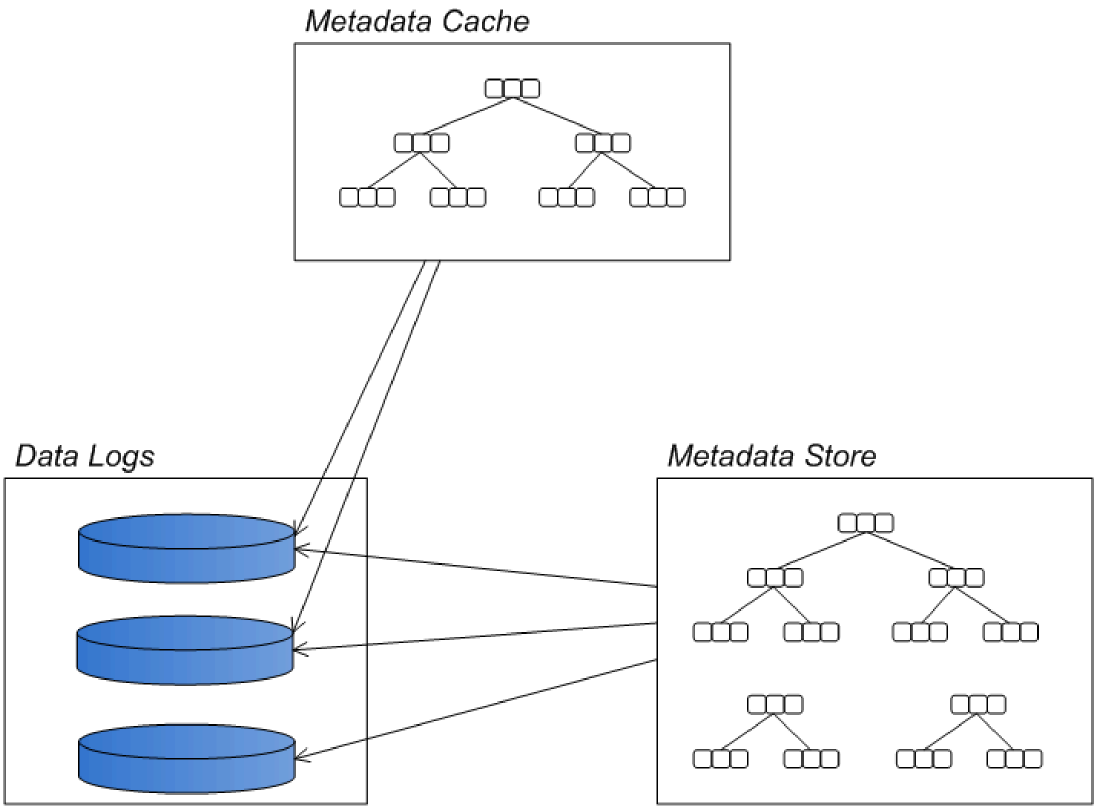
- journal-based storage so that messages can be rapidly written to disk
- allows for the broker to restart quickly
- storing message references in a B-tree index which can be rapidly updated at run time
- full support for JMS transactions
- various strategies to enable recovery after a disorderly shutdown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

Section 2.4, “Optimizing the Metadata Cache”.

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.

2.2. CONFIGURING THE KAHADB MESSAGE STORE

01/30/2013

added locker configuration

Overview

Red Hat JBoss A-MQ'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, “Configuration Properties of the KahaDB Message Store”, 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, “Configuring the KahaDB Message Store” 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, “Configuration Properties of the KahaDB Message Store” 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.

Attribute	Default Value	Description
checkForCorruptJournalFiles	**false**	Specifies whether the message store checks for corrupted journal files on startup and tries to recover them.
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 **true**.
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 section "Shared File System Master/Slave" in "Fault Tolerant Messaging".
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 Section 2.3, “Concurrent Store and Dispatch”.
concurrentStoreAndDispatchTopics	**false**	Specifies if the message store dispatches topic messages to interested clients concurrently with message storage. See Section 2.3, “Concurrent Store and Dispatch”.
concurrentStoreAndDispatchQueues	**true**	Specifies if the message store dispatches queue messages to clients concurrently with message storage. See Section 2.3, “Concurrent Store and Dispatch”.

Attribute	Default Value	Description
archiveCorruptedIndex	**false**	Specifies if corrupted indexes are archived when the broker starts up.
useLock	**true**	Specifies in the adapter uses file locking.

2.3. CONCURRENT STORE AND DISPATCH

Abstract

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.

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

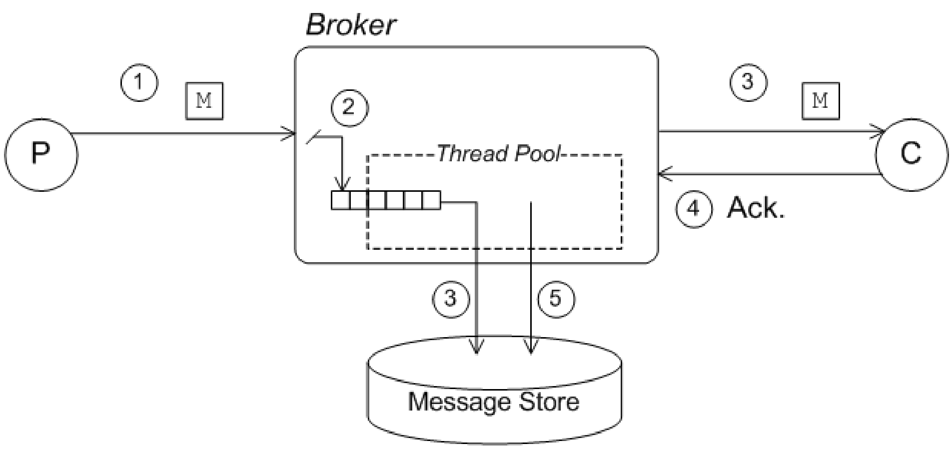
Concurrent store and dispatch is enabled by default for queues.

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, “Concurrent Store and Dispatch—Slow Consumers” 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:

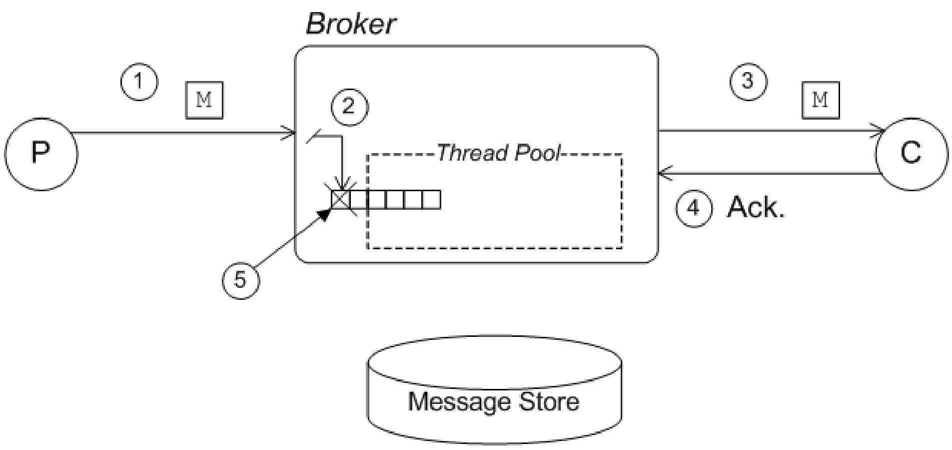
1. The producer sends a message, **M**, to a destination on the broker.
2. The broker sends the message, **M**, to the persistence layer. Because concurrency is enabled, the message is initially held in a task queue, which is serviced by a pool of threads that are responsible for writing to the journal.
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, “Concurrent Store and Dispatch—Fast Consumers” 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, “KahaDB Configured with Serialized Store and Dispatch” explicitly disables the store and dispatch feature for queues and topics.

Example 2.2. KahaDB Configured with Serialized Store and Dispatch

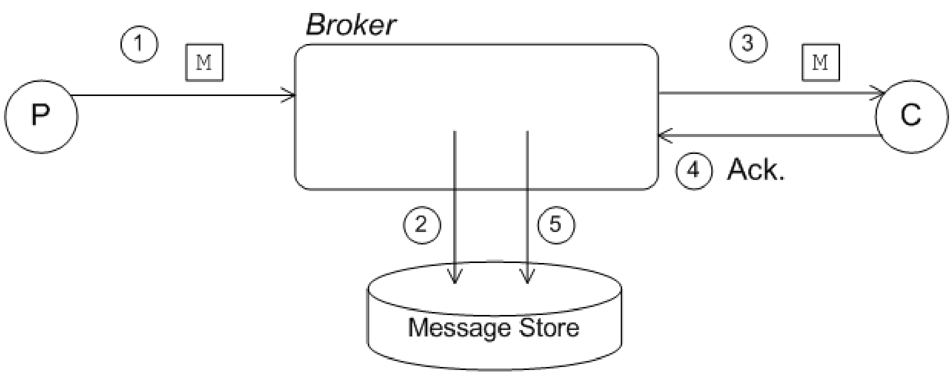
```
<broker brokerName="broker" persistent="true" useShutdownHook="false">
  ...
  <persistenceAdapter>
    <kahaDB directory="activemq-data"
            journalMaxFileLength="32mb"
            concurrentStoreAndDispatchQueues="false"
            concurrentStoreAndDispatchTopics="false"
            />
  </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, “Serialized Store and Dispatch” 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**.

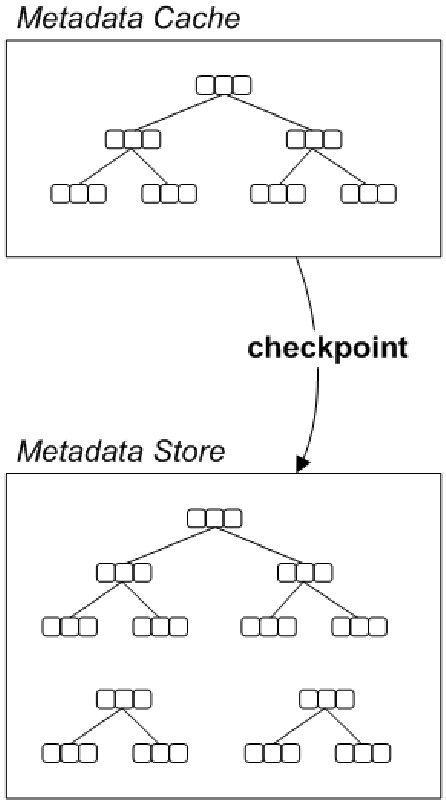
> **NOTE**
>
> **true** is the default value for the **enableJournalDiskSyncs** attribute.

2.4. 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, “Overview of the Metadata Cache and Store” 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 synchonization 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 performace.

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).

2.5. 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 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.

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, “Configuration for Journal Validation” shows hot 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

Abstract

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 the section called “Filters”. 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 Section 2.2, “Configuring the KahaDB Message Store”.

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.

Red Hat JBoss A-MQ 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, "Distributed KahaDB Persistence Adapter Configuration" 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 THE LEVELDB PERSISTENCE ADAPTER

Abstract

The LevelDB persistence adapter uses LevelDB as a high-performance message store. It allows for higher throughput speeds than the default message store at the cost of potentially higher CPU usage and message store size.

IMPORTANT

The LevelDB store is available only on Windows, Linux, and Mac OS platforms.

OVERVIEW

The LevelDB message store is a file based message store implemented using Google's LevelDB library to maintain indexes into log files holding the messages. The main advantages of the LevelDB store include:

- higher persistent throughput
- faster recovery times when a broker restarts
- supports concurrent read access
- no pausing during garbage collection cycles
- uses fewer read IO operations to load stored messages
- exposes status via JMX for monitoring
- supports replication

The LevelDB store does have a few known limitations:

- does not support XA transactions
- does not check for duplicate messages

PLATFORM SUPPORT

LevelDB is implemented in C++ and Red Hat JBoss A-MQ accesses the libraries using a JNI driver. JBoss A-MQ also provides an experimental pure Java driver that can be used as well.

The JNI driver is used by default on the following platforms:

- Linux
- OS X
- Windows Vista
- Windows Server 2008

IMPORTANT

Windows platforms require the MS VC++ 2010 Redistributable package:

- 32 bit JVM— Microsoft Visual C++ 2010 Redistributable Package (x86)
- 64 bit JVM— Microsoft Visual C++ 2010 Redistributable Package (x64)

All other platforms use the experimental Java drivers.

BASIC CONFIGURATION

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

The attributes, listed in Table 4.1, “Configuration Properties of the LevelDB Message Store”, 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 4.1, “Configuring the LevelDB Message Store” shows a basic configuration of the LevelDB message store. The LevelDB files are stored under the **activemq-data** directory.

Example 4.1. Configuring the LevelDB Message Store

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

CONFIGURATION ATTRIBUTES

Table 4.1, “Configuration Properties of the LevelDB Message Store” describes the attributes that can be used to configure the LevelDB message store.

Table 4.1. Configuration Properties of the LevelDB 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.

Attribute	Default Value	Description
flushDelay	**500**	Specifies the amount of time in milliseconds that a store will delay persisting a messaging unit of work in hopes that it will be invalidated shortly thereafter by another unit of work which would negate the operation.
readThreads	**10**	Specifies he number of concurrent IO reads to allow.
sync	**true**	Specifies if the syncs logging operations to disk.
logSize	**104857600**	Specifies the maximum size, in bytes, of each data log file before log file rotation occurs.
logWriteBufferSize	**4194304**	Specifies the maximum amount of log data, in bytes, to build up before writing to the file system.
verifyChecksums	**false**	Specifies if checksum verification is performed on all data that is read from the file system.
paranoidChecks	**false**	Specifies if the store errors out as soon as possible if it detects internal corruption.
indexMaxOpenFiles	**1000**	Specifies the number of open files that can be used by the index.
indexBlockRestartInterval	**16**	Specifies the number of keys between restart points for delta encoding of keys.
indexWriteBufferSize	**4194304**	Specifies the amount of index data to build up in memory before converting to a sorted on-disk file.
indexBlockSize	**4096**	Specifies the size, in bytes, of index data packed per block.
indexCacheSize	**268435456**	Specifies the maximum amount of memory, in bytes, to use to cache index blocks.

Attribute	Default Value	Description
indexCompression	**snappy**	Specifies the type of compression to apply to the index blocks. Can be **snappy** or **none**.
logCompression	**snappy**	Specifies the type of compression to apply to the log records. Can be **snappy** or **none**.
indexFactory	**org.fusesource.leveldbjni.JniDBFactory, org.iq80.leveldb.impl.Iq80DBFactory**	Specifies a comma separated list of leveldb API implementation factory classes that the broker will attempt to load. The broker will use the first one that loads successfully. • **org.fusesource.leveldbjni.JniDBFactory** enables the JNI base implementation • **org.iq80.leveldb.impl.Iq80DBFactory** enables the pure Java implementation
asyncBufferSize	**4096**	Specifies the amount of memory, in bytes, to use for buffering asynchronous writes to disk.
failIfLocked	**false**	Specifies if the broker will fail on start up if the message store's data files are locked. If the broker does not fail it will block until the data files are unlocked.
useLock	**true**	Specifies in the adapter uses file locking.

CHAPTER 5. USING JDBC TO CONNECT TO A DATABASE STORE

Abstract

Red Hat JBoss A-MQ 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.

5.1. 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. When using the JDBC persistence adapter, Red Hat JBoss A-MQ's default database is Apache Derby, but JBoss A-MQ 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, but using the journal considerably improves the speed of the message store.

Supported databases

JBoss A-MQ is known to work with the following databases:

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

In addition, JBoss A-MQ supports a number of generic JDBC providers.

Specifying the type of JDBC store to use

JBoss A-MQ 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 Section 5.2, “Using JDBC with the High Performance Journal”.

- *A non-journaled JDBC store:*

  The non-journaled store is specified using the **jdbcPersistenceAdapter** element inside the **persistenceAdapter** element. For more details see Section 5.3, “Using JDBC without the Journal”.

The journaled JDBC store features better performance than the plain JDBC store.

> **WARNING**
>
> The journaled JDBC store is incompatible with the JDBC master/slave failover pattern—see *Fault Tolerant Messaging*.

Prerequisites

Before you can use one of the JDBC persistence stores you need to ensure that the following are installed in the broker's container:

- The **org.apache.servicemix.bundles.commons-dbcp** bundle
- The JDBC driver for the database being used

> **NOTE**
>
> Depending on the database being used, you may need to wrap the driver in an OSGi bundle by using the wrap: URI prefix when adding it to the container.

Configuring your JDBC driver

JBoss A-MQ 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 the section called “Customizing the SQL statements used by the adapter” for information about modifying the SQL statements. See the section called “Using generic JDBC providers” for information about changing the JDBC methods.

JDBC configuration for Apache Derby

Example 5.1, “Configuration for the Apache Derby Database” shows the configuration for using the default Apache Derby JDBC driver.

Example 5.1. 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"/>
    <property name="createDatabase" value="create"/>
  </bean>

</beans>
```

JDBC configuration for Oracle

Example 5.2, “Configuration for the Oracle JDBC Driver” 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 5.2. 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.

5.2. USING JDBC WITH THE HIGH PERFORMANCE JOURNAL

01/30/2013

added locker configuration

Overview

Using the JDBC persistence adapter with Red Hat JBoss A-MQ'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.

> **WARNING**
>
> The journaled JDBC store is incompatible with the JDBC master/slave failover pattern—see *Fault Tolerant Messaging*.

Prerequisites

Before you can use the journaled JDBC persistence store you need to ensure that the **activeio-core-3.1.4.jar** bundle is installed in the container.

The bundle is available in the archived ActiveMQ installation included in the ***InstallDir*/extras** folder or can be downloaded from Maven at http://mvnrepository.com/artifact/org.apache.activemq/activeio-core/3.1.4.

Example

Example 5.3, “Configuring Red Hat JBoss A-MQ to use the Journaled JDBC Persistence Adapter” shows a configuration fragment that configures the journaled JDBC adapter to use a MySQL database.

Example 5.3. Configuring Red Hat JBoss A-MQ to use the Journaled JDBC Persistence Adapter

```
<beans ... >
 <broker ...>
  ...
  <persistenceFactory>
     <journalPersistenceAdapterFactory journalLogFiles="5" dataDirectory="${data}/kahadb"
    dataSource="#mysql-ds" useDatabaseLock="true" useDedicatedTaskRunner="false />
  </persistenceFactory>
  ...
 <broker>
 ...
 <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 5.3, “Configuring Red Hat JBoss A-MQ to use the Journaled JDBC Persistence Adapter” has three noteworthy elements:

1. The **persistenceFactory** element wraps the configuration for the JDBC persistence adapter.
2. The **journaledJDBC** 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**.
   - The data for the journal will be stored in **${data}/kahadb**.
3. The **bean** element specified the configuration for the MySQL JDBC driver.

Configuration

Table 5.1, “Attributes for Configuring the Journaled JDBC Persistence Adapter” describes the attributes used to configure the journaled JDBC persistence adapter.

Table 5.1. Attributes for Configuring the Journaled JDBC Persistence Adapter

Attribute	Default Value	Description

Attribute	Default Value	Description
adapter		Specifies the strategy to use when accessing a non-supported database. For more information see the section called “Using generic JDBC providers”.
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.
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 the section called “Configuring your JDBC driver”.
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.
useJournal	**true**	Specifies whether or not to use the journal.
useLock	**true**	Specifies in the adapter uses file locking.
lockKeepAlivePeriod	**30000**	Specifies the time period, in milliseconds, at which the current time is saved in the locker table to ensure that the lock does not timeout. **0** specifies unlimited time.

5.3. USING JDBC WITHOUT THE JOURNAL

01/30/2013

added locker configuration

Overview

For those cases where journaling is not appropriate, or you wish to use your own journaling system, you can used the JDBC persistence adapter without the Red Hat JBoss A-MQ high performance journal.

Example

Example 5.4, “Configuring Red Hat JBoss A-MQ to use the Plain JDBC Persistence Adapter” shows a configuration fragment that configures the plain JDBC adapter to use the Apache Derby database.

Example 5.4. Configuring Red Hat JBoss A-MQ to use the Plain JDBC Persistence Adapter

```
<beans ... >
  <broker ...>
  ...
1   <persistenceAdapter>
2       <jdbcPersistenceAdapter dataSource="#derby-ds" />
        </persistenceAdapter>
    ...
  <broker>
  ...
3 <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 5.4, “Configuring Red Hat JBoss A-MQ to use the Plain JDBC Persistence Adapter” has three noteworthy elements:

1. The **persistenceAdapter** element wraps the configuration for the JDBC persistence adapter.
2. 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**.
3. The **bean** element specified the configuration for the Derby JDBC driver.

Configuration

Table 5.2, “Attributes for Configuring the Plain JDBC Persistence Adapter” describes the attributes used to configure the non-journaled JDBC persistence adapter.

Table 5.2. Attributes for Configuring the Plain JDBC Persistence Adapter

Attribute	Default Value	Description

Attribute	Default Value	Description
adapter		Specifies the strategy to use when accessing a non-supported database. For more information see the section called "Using generic JDBC providers".
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.
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 the section called "Configuring your JDBC driver".
transactionIsolation	**Connection.TRANSACTION_READ_UNCOMMITTED**	Specifies the required transaction isolation level. For allowed values, see java.sql.Connection.
useLock	**true**	Specifies in the adapter uses file locking.
lockKeepAlivePeriod	**30000**	Specifies the time period, in milliseconds, at which the current time is saved in the locker table to ensure that the lock does not timeout. **0** specifies unlimited time.

5.4. CUSTOMIZING THE JDBC PERSISTENCE ADAPTER

Overview

Red Hat JBoss A-MQ 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 5.5, “Fine Tuning the Database Schema” shows a configuration snippet that specifies that long strings are going to be stored as VARCHAR(128).

Example 5.5. Fine Tuning the Database Schema

```
<persistenceFactory>
  <journaledJDBC ... >
    <statements>
      <statements stringIdDataType ="VARCHAR(128)"/>
    </statements>
  </journaledJDBC>
</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 5.3, “Statements for Configuring the SQL Statements Used by the JDBC Persistence Adapter” describes the configurable properties.

Table 5.3. 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.
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.

Attribute	Default	Description
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 5.3, “Statements for Configuring the SQL Statements Used by the JDBC Persistence Adapter” 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**
- **lastAckedDurableSubscriberMessageStatement**
- **destinationMessageCountStatement**
- **findNextMessageStatement**
- **createSchemaStatements**
- **dropSchemaStatements**

Using generic JDBC providers

To use a JDBC provider not natively supported by Red Hat JBoss A-MQ, 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 5.6, “Configuring a Generic JDBC Provider” shows a configuration snippet configuring the journaled JDBC persistence adapter to use the blob JDBC adapter.

Example 5.6. 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 6. MESSAGE CURSORS

Abstract

Red Hat JBoss A-MQ 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. JBoss A-MQ also supports two alternative cursor implementations. The type of cursor can be configured on a per-destination basis.

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. Red Hat JBoss A-MQ 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.

6.1. TYPES OF CURSORS

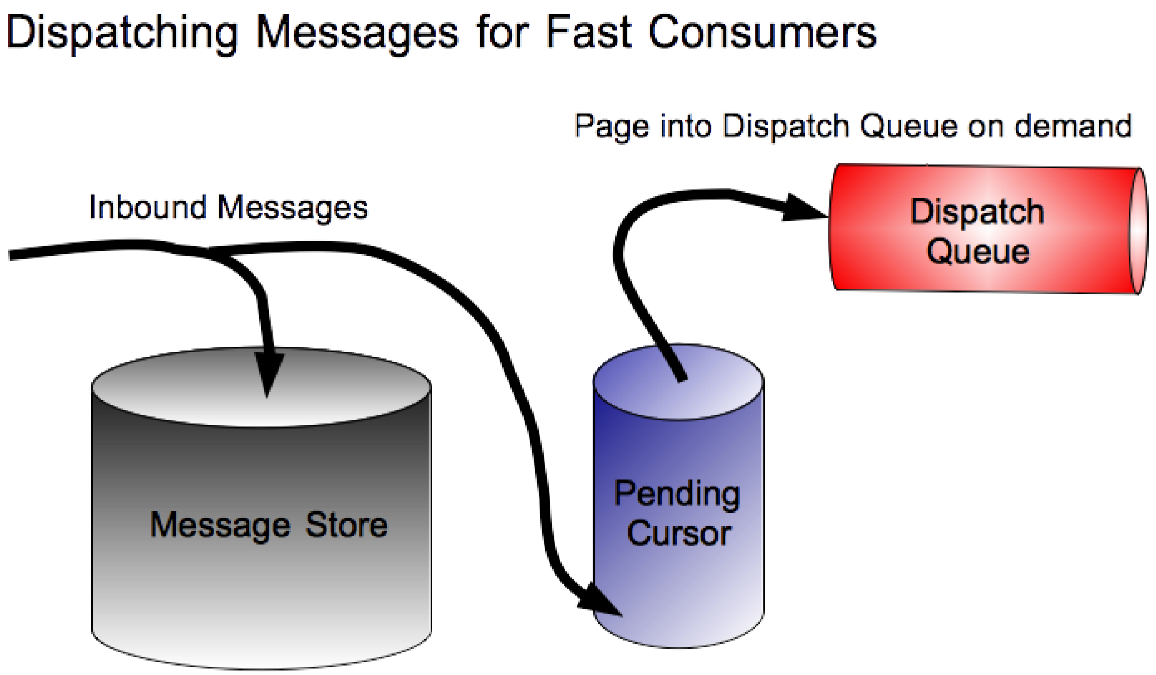
Store-based cursors

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.

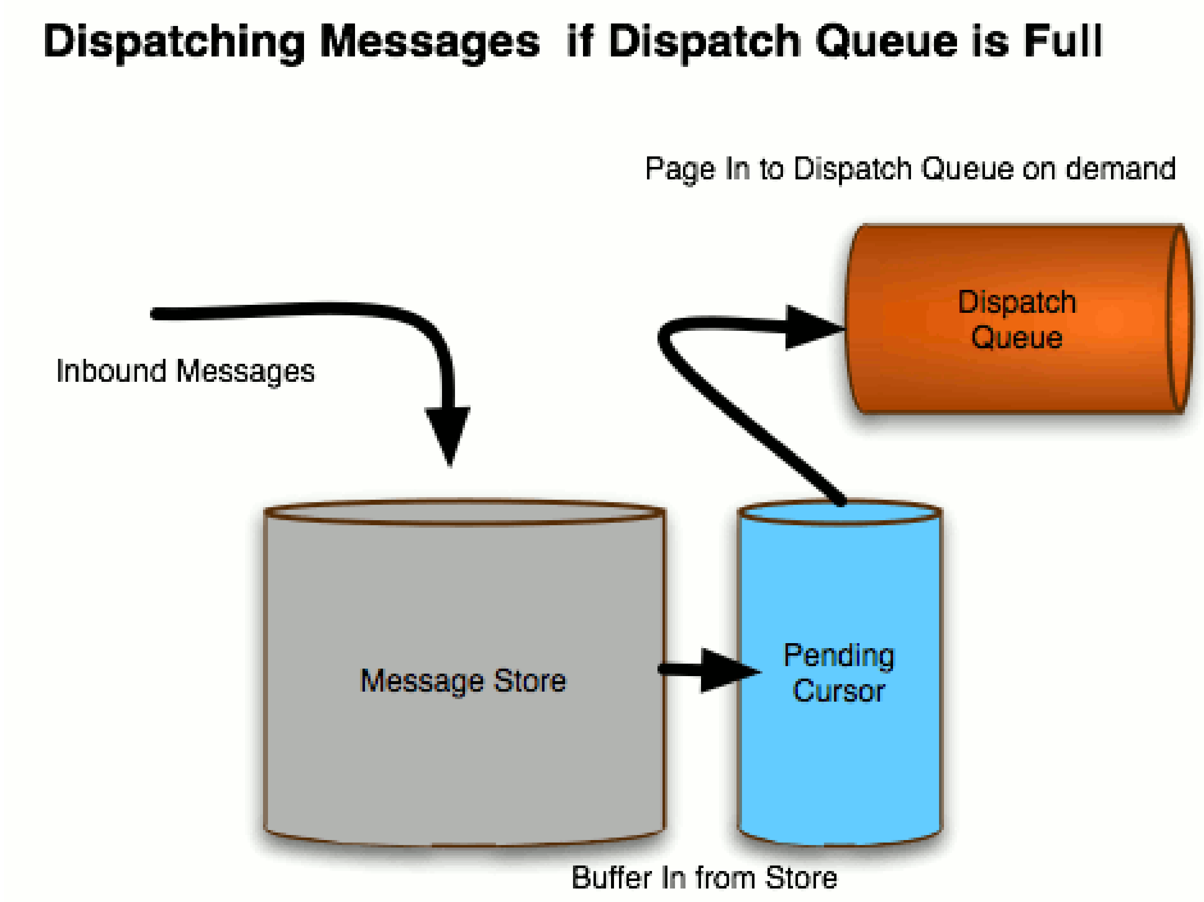
As shown in Figure 6.1, “Store-based Cursors for a Fast Consumer”, the store-based cursor addresses the fast consumer case just like the VM cursor. Messages are written to the persistent store and are also directly stored in the pending cursor, which is held completely in memory. The pending cursor then feeds the messages into the dispatch queue. However, since the store cursor can hold only a limited number of messages in memory, it is a mapping of only a fraction of the persistent message store.

Figure 6.1. Store-based Cursors for a Fast Consumer



When a consumer starts with a back log of messages or falls behind its message producers, JBoss A-MQ changes the strategy used to dispatch messages. As shown in Figure 6.2, “Store-based Cursors for a Slow Consumer”, messages are held in the message store and fed into the consumer's dispatch queue using the pending cursor.

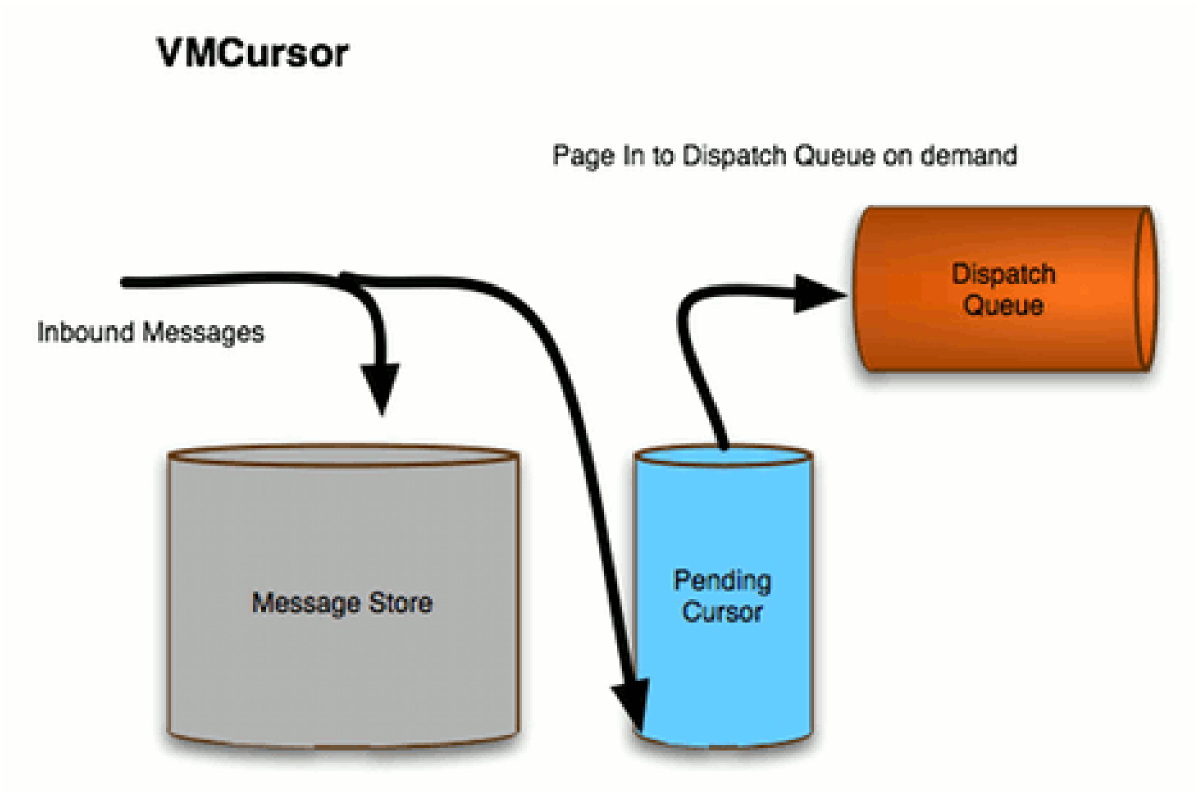
Figure 6.2. Store-based Cursors for a Slow Consumer



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 6.3, "VM Cursors", messages are written to the persistent store and are also stored in the pending cursor, which is held completely in memory. The pending cursor then feeds the messages into the dispatch queue. Since it needs to hold all messages in memory, the VM cursor is a snapshot of the entire persistent message store.

Figure 6.3. VM Cursors

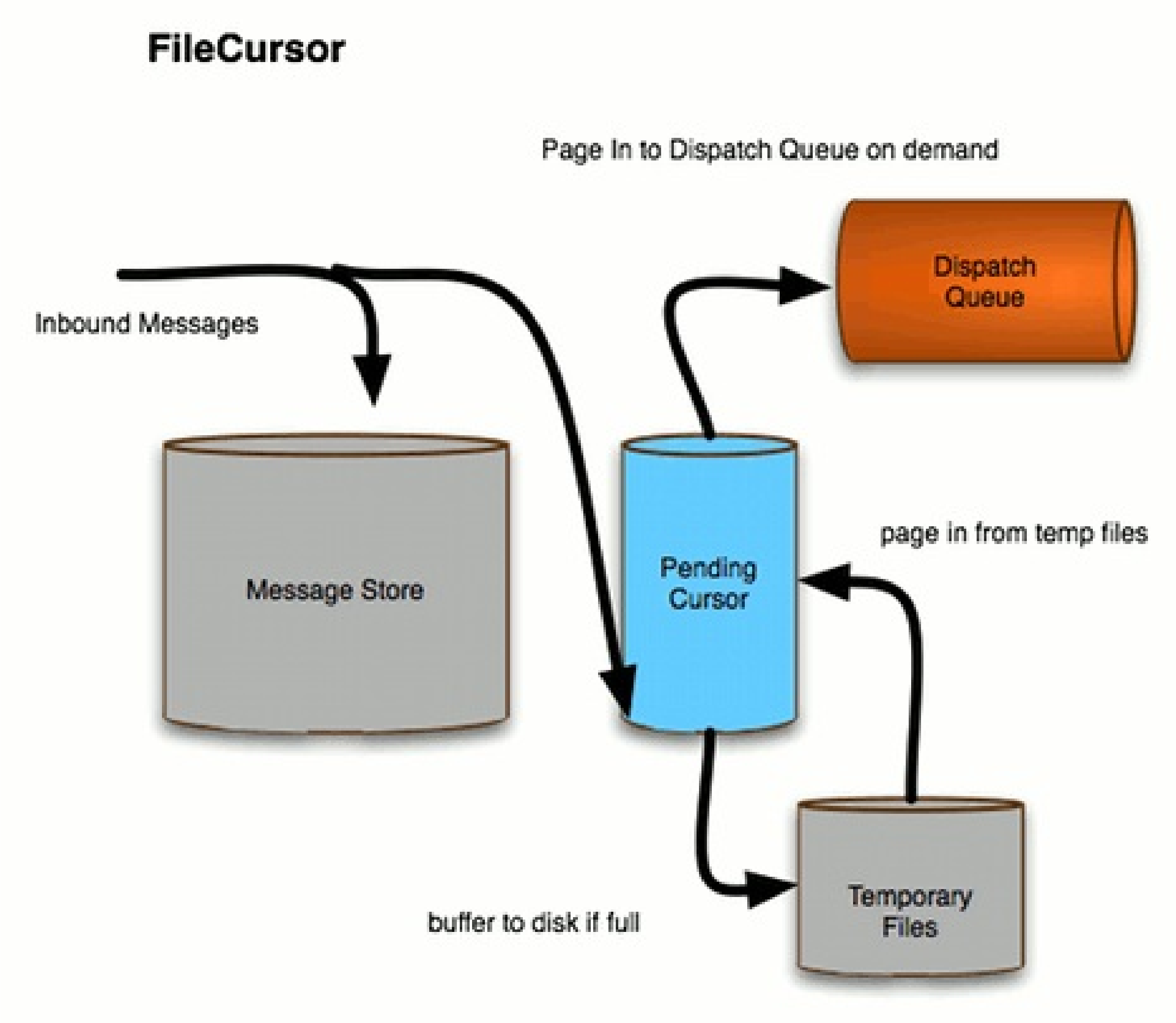


Because the message are dispatched from active memory when using VM cursors, this method is exceptionally fast. However, if the number of unconsumed messages gets large the producers will be throttled to avoid exceeding the available memory.

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 6.4, "File-based Cursors", the broker pages messages out to a temporary file when the broker's memory limit is reached.

Figure 6.4. 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.

6.2. CONFIGURING THE TYPE OF CURSOR USED BY A DESTINATION

Overview

By default, JBoss A-MQ 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 6.1, “Elements for Configuring the Type of Cursor to Use for Durable Subscribers” describes the possible children of **PendingDurableSubscriberMessageStoragePolicy**.

Table 6.1. Elements for Configuring the Type of Cursor to Use for Durable Subscribers

Element	Description
storeDurableSubscriberCursor	Specifies that store-based cursors will be used. See the section called “Store-based cursors” for more information.
vmDurableCursor	Specifies that VM cursors will be used. See the section called “VM cursors” for more information.
fileDurableSubscriberCursor	Specifies that file-based cursors will be used—only suitable for non-persistent messages. See the section called “File-based cursors” for more information.

You configure the cursor implementation used by transient subscribers by adding **pendingSubscriberPolicy** child element to the topic's **policyEntry** element. Table 6.2, “Elements for Configuring the Type of Cursor to Use for Transient Subscribers” describes the possible children of **pendingSubscriberPolicy**.

Table 6.2. Elements for Configuring the Type of Cursor to Use for Transient Subscribers

Element	Description
Unspecified	Default policy is to use store-based cursors. See the section called “Store-based cursors” for more information.
vmCursor	Specifies the VM cursors will be used. See the section called “VM cursors” for more information.
fileCursor	Specifies that file-based cursors will be used. See the section called “File-based cursors” for more information.

Example 6.1, “Configuring a Topic's Cursor Usage” shows a configuration snip-it that configures a topic to use VM cursors for its transient subscribers and file-based cursors for its durable subscribers.

Example 6.1. Configuring a Topic's Cursor Usage

```
<beans ... >
  <broker ... >
    ...
    <destinationPolicy>
      <policyMap>
        <policyEntries>
          <policyEntry topic="com.fusesource.>" >
            ...
            <pendingSubscriberPolicy>
              <vmCursor />
            </pendingSubscriberPolicy>
            <pendingDurableSubscriberPolicy>
              <storeDurableSubscriberCursor />
            </pendingDurableSubscriberPolicy>
            ...
          </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 6.3, "Elements for Configuring the Type of Cursor to Use for a Queue" describes the possible children elements of the **pendingQueuePolicy** element.

Table 6.3. Elements for Configuring the Type of Cursor to Use for a Queue

Element	Description
storeCursor	Specifies that store-based cursors will be used. See the section called "Store-based cursors" for more information.
vmQueueCursor	Specifies the VM cursors will be used. See the section called "VM cursors" for more information.
fileQueueCursor	Specifies that file-based cursors will be used. See the section called "File-based cursors" for more information.

Example 6.2, "Configuring a Queue's Cursor Usage" shows a configuration snippet that configures a queue to use VM cursors.

Example 6.2. Configuring a Queue's Cursor Usage

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

CHAPTER 7. MESSAGE STORE LOCKERS

Abstract

Message store locks are used to elect the master broker in master/slave groups. They are also useful for ensuring that multiple brokers are not attempting to share the same message store. Red Hat JBoss A-MQ's lockers are configurable to allow for tuning.

7.1. LOCKER BASICS

Overview

Red Hat JBoss A-MQ provides two default lockers that are used based on the type of message store being used:

- shared file locker—used by KahaDB and LevelDB stores
- database locker—used by the JDBC store

> **NOTE**
>
> JBoss A-MQ also provides a leased database locker that can be in cases where the brokers may periodically lose their connection to the message store.

These default lockers are configurable to optimize their performance.

For further optimization, you can implement your own locker and plug it into the message store. Doing so involves implementing a simple Java interface and adding some configuration to the persistence adapter.

Message store locks are primarily leveraged by the broker for electing masters in master/slave configurations. For more information on master/slave groups see chapter "Master/Slave" in "Fault Tolerant Messaging".

Configuring a persistence adapter's locker

To configure the locker used by a persistence adapter you add a **locker** element as a child to the adapter's configuration element as shown in Example 7.1, “Configuring a Message Store Locker”.

Example 7.1. Configuring a Message Store Locker

```
<persistenceAdapter>
  <kahaDB directory = "target/activemq-data">
    <locker>
      ...
    </locker>
  </kahaDB>
</persistenceAdapter>
```

Standard locker configuration properties

All locker implementations are required to have the two common configuration properties described in Table 7.1, "Common Locker Properties".

Table 7.1. Common Locker Properties

Property	Default Value	Description
lockAcquireSleepInterval	1000	Specifies the delay interval, in milliseconds, between attempts to acquire a lock.
failIfLocked	false	Specifies in the broker should immediately fail if a lock cannot be obtained.

The properties are specified as attributes to the locker's XML configuration element.

7.2. USING THE PROVIDED LOCKERS

Red Hat JBoss A-MQ includes three standard locker implementations:

- shared file locker—used by file-based message stores like KahaDB and LevelDB
- database locker—used as the default for JDBC message stores
- lease database locker—used as an alternative locker for JDBC message stores in scenarios where brokers have inconsistent connections to the message store

7.2.1. Shared File Locker

Overview

The shared file locker is used by file-based message stores to ensure that only one broker can modify the files used by the message store.

Configuration

As shown in Example 7.2, "Configuring a Shared File Locker", the shared file locker is configured using the **shared-file-locker** element.

Example 7.2. Configuring a Shared File Locker

```
<persistenceAdapter>
  <kahaDB directory = "target/activemq-data">
    <locker>
      <shared-file-locker lockAcquireSleepInterval="5000"/>
    </locker>
  </kahaDB>
</persistenceAdapter>
```

The shared file locker supports the common configuration properties described in Table 7.1, "Common Locker Properties".

7.2.2. Database Locker

Overview

The database locker is the default locker for all JDBC persistence adapters. It locks a database table in a transaction to ensure that only one broker can modify the message store.

The database locker does not perform well in two scenarios:

- intermittent database connectivity
- database failover

Configuration

As shown in Example 7.3, "Configuring a Database Locker", it is configured using the **database-locker** element.

Example 7.3. Configuring a Database Locker

```
<persistenceAdapter>
  <jdbcPersistenceAdapter dataDirectory="${activemq.data}" dataSource="#mysql-ds">
    <locker>
      <database-locker lockAcquireSleepInterval="5000"/>
    </locker>
  </jdbcPersistenceAdapter>
</persistenceAdapter>
```

The database locker supports the common configuration properties described in Table 7.1, "Common Locker Properties".

Intermittent database connectivity

When the master broker loses its connection to the database, or crashes unexpectedly, the information about the lock remains in the database until the database responds to the half-closed socket connection via a TCP timeout. This can prevent the slave from starting for a period of time.

Database failover

When the database used for the message store supports failover issues can arise. When the database connection is dropped in the event of a replica failover, the brokers see this as a database failure and all of the brokers in the master/slave group will begin competing for the lock. This restarts the master election process and can cause the group to failover to a new master. For more information see section "Shared JDBC Master/Slave" in "Fault Tolerant Messaging".

7.2.3. Lease Database Locker

Overview

The lease database locker is designed to overcome the shortcomings of the default database locker by forcing the lock holder to periodically renew the lock. When the lock is first acquired the broker holds it for the period specified in the persistence adapter's **lockKeepAlivePeriod** attribute. After the initial period, the lock is renewed for the period specified by the locker's **lockAcquireSleepInterval** attribute.

When all of broker's system clocks are properly synchronized, the master broker will always renew the lease before any of the slaves in the group can steal it. In the event of a master's failure, the lock will automatically expire within the configured amount of time and one of the slave's in the group will be able to acquire it.

Configuration

As shown in Example 7.4, “Configuring a Lease Database Locker”, it is configured using the **lease-database-locker** element.

Example 7.4. Configuring a Lease Database Locker

```
<persistenceAdapter>
  <jdbcPersistenceAdapter dataDirectory="${activemq.data}" dataSource="#mysql-ds"
lockKeepAlivePeriod="10000">
    <locker>
      <lease-database-locker lockAcquireSleepInterval="5000"/>
    </locker>
  </jdbcPersistenceAdapter>
</persistenceAdapter>
```

The lease database locker supports the common configuration properties described in Table 7.1, “Common Locker Properties”.

Dealing with unsynchronized system clocks

The lease database locker relies on each broker's system clock to enure the proper timing of lease expiration and lock requests. When all of the system clocks are synchronized, the timing works. Once the system clocks start drifting apart, the timing can be thrown off and a slave broker could possibly steal the lock from the group's master.

To avoid this problem the locker can make adjustments based on the database server's current time setting. This feature is controlled by setting the locker's **maxAllowableDiffFromDBTime** to specify the number of milliseconds by which a broker's system clock can differ from the database's before the locker automatically adds an adjustment. The default setting is zero which deactivates the adjustments.

Example 7.5, “Configuring a Lease Database Locker to Adjust for Non-synchronized System Clocks” shows configuration for making adjustments when a broker's clock differs from the database by one second.

Example 7.5. Configuring a Lease Database Locker to Adjust for Non-synchronized System Clocks

```
<persistenceAdapter>
  <jdbcPersistenceAdapter ... >
    <locker>
      <lease-database-locker maxAllowableDiffFromDBTime="1000"/>
```

```
    </locker>
  </jdbcPersistenceAdapter>
</persistenceAdapter>
```

7.3. USING CUSTOM LOCKERS

Overview

If one of the provided lockers are not sufficient for your needs, you can implement a custom locker. All lockers are implementations of the Red Hat JBoss A-MQ **Locker** interface. They are attached to the persistence adapter as a spring bean in the **locker** element.

Interface

All lockers are implementations of the **org.apache.activemq.broker.Locker** interface. Implementing the **Locker** interface involves implementing seven methods:

- **boolean keepAlive()**
  throws IOException;
  Used by the lock's timer to ensure that the lock is still active. If this returns false, the broker is shutdown.

- **void setLockAcquireSleepInterval(long lockAcquireSleepInterval);**
  Sets the delay, in milliseconds, between attempts to acquire the lock. **lockAcquireSleepInterval** is typically supplied through the locker's XML configuration.

- **public void setName(String name);**
  Sets the name of the lock.

- **public void setFailIfLocked(boolean failIfLocked);**
  Sets the property that determines if the broker should fail if it cannot acquire the lock at start-up. **failIfLocked** is typically supplied through the locker's XML configuration.

- **public void configure(PersistenceAdapter persistenceAdapter)**
  throws IOException;
  Allows the locker to access the persistence adapter's configuration. This can be used to obtain the location of the message store.

- **void start();**
  Executed when the locker is initialized by the broker. This is where the bulk of the locker's implementation logic should be placed.

- **void stop();**
  Executed when the broker is shutting down. This method is useful for cleaning up any resources and ensuring that all of the locks are released before the broker is completely shutdown.

Using AbstractLocker

To simplify the implementation of lockers, Red Hat JBoss A-MQ includes a default locker implementation, **org.apache.activemq.broker.AbstractLocker**, that serves as the base for all of the provided lockers. It is recommended that all custom locker implementations also extand the **AbstractLocker** class instead of implementing the plain **Locker** interface.

AbstractLocker provides default implementations for the following methods:

- **keepAlive()**—returns **true**
- **setLockAcquireSleepInterval()**—sets the parameter to the value of the locker beans' ***lockAcquireSleepInterval*** if provided or to **10000** if the parameter is not provided
- **setName()**
- **setFailIfLocked()**—sets the parameter to the value of the locker beans' ***failIfLocked*** if provided or to **false** if the parameter is not provided
- **start()**—starts the locker after calling two additional methods

  IMPORTANT

  This method should not be overridden.

- **stop()**—stops the locker and adds a method that is called before the locker is shutdown and one that is called after the locker is shutdown

  IMPORTANT

  This method should not be overridden.

AbstractLocker adds two methods that must be implemented:

- **void doStart()**
  throws Exception;
  Executed as the locker is started. This is where most of the locking logic is implemented.
- **void doStop(ServiceStopper stopper)**
  throws Exception;
  Executed as the locker is stopped. This is where locks are released and resources are cleaned up.

In addition, **AbstractLocker** adds two methods that can be implemented to provide additional set up and clean up:

- **void preStart()**
  throws Exception;
  Executed before the locker is started. This method can be used to initialize resources needed by the lock. It can also be used to perform any other actions that need to be performed before the locks are created.
- **void doStop(ServiceStopper stopper)**
  throws Exception;
  Executed after the locker is stopped. This method can be used to clean up any resources that are left over after the locker is stopped.

Configuration

Custom lockers are added to a persistence adapter by adding the bean configuration to the persistence adapter's **locker** element as shown in Example 7.6, “Adding a Custom Locker to a Persistence Adapter”.

Example 7.6. Adding a Custom Locker to a Persistence Adapter

```
<persistenceAdapter>
  <kahaDB directory = "target/activemq-data">
    <locker>
      <bean class="my.custom.LockerImpl">
        <property name="lockAcquireSleepInterval" value="5000" />
        ...
      </bean
    </locker>
  </kahaDB>
</persistenceAdapter>
```

INDEX

F

J

K

M

P

S

T

V