Red Hat Data Grid 8.2

Configuring Data Grid

Enable and adjust Data Grid features and capabilities
Enable and adjust Data Grid features and capabilities
Abstract

Data Grid offers flexible configuration options that you can control programmatically or declaratively to handle a wide variety of use cases.
# Table of Contents

## CHAPTER 1. RED HAT DATA GRID

1.1. DATA GRID DOCUMENTATION ........................................ 5
1.2. DATA GRID DOWNLOADS ........................................... 5
1.3. MAKING OPEN SOURCE MORE INCLUSIVE ............................ 5

## CHAPTER 2. DATA GRID CACHES

2.1. CACHE INTERFACE ................................................. 6
2.2. CACHE MANAGERS ................................................ 6
2.3. CACHE CONTAINERS .............................................. 6
2.4. CACHE MODES .................................................. 6
   2.4.1. Cache Mode Comparison ................................... 7

## CHAPTER 3. LOCAL CACHES ............................................. 9

3.1. SIMPLE CACHES .................................................. 9

## CHAPTER 4. CLUSTERED CACHES ...................................... 11

4.1. INVALIDATION MODE ............................................ 11
4.2. REPLICATED CACHES ............................................ 12
4.3. DISTRIBUTED CACHES .......................................... 13
   4.3.1. Read consistency ........................................... 14
   4.3.2. Key Ownership ............................................. 14
   4.3.2.1. Capacity Factors ......................................... 15
   4.3.3. Zero Capacity Node ......................................... 15
   4.3.4. Hashing Configuration ..................................... 15
   4.3.5. Initial cluster size .......................................... 16
   4.3.6. L1 Caching ................................................ 16
   4.3.7. Server Hinting .............................................. 17
      4.3.7.1. Configuration ........................................ 17
   4.3.8. Key affinity service ......................................... 17
      4.3.8.1. API .................................................. 17
      4.3.8.2. Lifecycle ............................................ 18
      4.3.8.3. Topology changes .................................... 18
      4.3.8.4. The Grouping API ................................... 18
      4.3.8.5. How does it work? .................................. 18
      4.3.8.6. How do I use the grouping API? .................. 19
      4.3.8.7. Advanced Interface ................................. 20
   4.4. SCATTERED CACHES ........................................... 21
   4.5. ASYNCHRONOUS COMMUNICATION WITH CLUSTERED CACHES 21
      4.5.1. Asynchronous Communications ......................... 21
      4.5.2. Asynchronous API ....................................... 22
      4.5.3. Return Values in Asynchronous Communication .... 22

## CHAPTER 5. CONFIGURING DATA GRID CACHES ....................... 24

5.1. DECLARATIVE CONFIGURATION .................................... 24
   5.1.1. Cache Templates .......................................... 24
   5.1.2. Cache Configuration Wildcards .......................... 25
   5.1.3. Multiple Configuration Files ............................. 26
5.2. DATA GRID CONFIGURATION API .................................. 26
5.3. CONFIGURING CACHES PROGRAMMATICALLY ..................... 27

## CHAPTER 6. MANAGING MEMORY ...................................... 30

6.1. CONFIGURING EVICTION AND EXPIRATION ....................... 30
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Eviction</td>
<td>30</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Configuring the Total Number of Entries for Data Grid Caches</td>
<td>31</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Configuring the Maximum Amount of Memory for Data Grid Caches</td>
<td>32</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Eviction Examples</td>
<td>33</td>
</tr>
<tr>
<td>6.2</td>
<td>Expiration</td>
<td>35</td>
</tr>
<tr>
<td>6.2.1</td>
<td>How Expiration Works</td>
<td>35</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Expiration Reaper</td>
<td>36</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Maximum Idle and Clustered Caches</td>
<td>36</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Expiration Examples</td>
<td>37</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Using Off-Heap Memory</td>
<td>38</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 7</th>
<th>Chapter 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>8.1</td>
</tr>
<tr>
<td>7.2</td>
<td>8.1.1</td>
</tr>
<tr>
<td>7.3</td>
<td>8.1.2</td>
</tr>
<tr>
<td>7.4</td>
<td>8.1.3.7</td>
</tr>
<tr>
<td>7.5</td>
<td>8.1.4</td>
</tr>
<tr>
<td>7.6</td>
<td>8.1.5</td>
</tr>
<tr>
<td>7.7</td>
<td>8.1.6</td>
</tr>
<tr>
<td>7.8</td>
<td>8.1.7</td>
</tr>
<tr>
<td>7.9</td>
<td>8.1.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 8</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Cache Stores</td>
<td>46</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Configuring Cache Stores</td>
<td>46</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Setting a Global Persistent Location for File-Based Cache Stores</td>
<td>47</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Passivation</td>
<td>48</td>
</tr>
<tr>
<td>8.1.3.1</td>
<td>Passivation and Cache Stores</td>
<td>49</td>
</tr>
<tr>
<td>8.1.4</td>
<td>Cache Loaders and Transactional Caches</td>
<td>50</td>
</tr>
<tr>
<td>8.1.5</td>
<td>Segmented Cache Stores</td>
<td>50</td>
</tr>
<tr>
<td>8.1.6</td>
<td>Filesystem-Based Cache Stores</td>
<td>51</td>
</tr>
<tr>
<td>8.1.7</td>
<td>Write-Through</td>
<td>51</td>
</tr>
<tr>
<td>8.1.8</td>
<td>Write-Behind</td>
<td>52</td>
</tr>
<tr>
<td>8.2</td>
<td>Cache Store Implementations</td>
<td>53</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Cluster Cache Loaders</td>
<td>53</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Single File Cache Stores</td>
<td>54</td>
</tr>
<tr>
<td>8.2.3</td>
<td>JDBC String-Based Cache Stores</td>
<td>55</td>
</tr>
<tr>
<td>8.2.3.1</td>
<td>Connection Factories</td>
<td>55</td>
</tr>
<tr>
<td>8.2.3.2</td>
<td>JDBC String-Based Cache Store Configuration</td>
<td>56</td>
</tr>
<tr>
<td>8.2.4</td>
<td>JPA Cache Stores</td>
<td>58</td>
</tr>
<tr>
<td>8.2.4.1</td>
<td>JPA Cache Store Usage Example</td>
<td>59</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Remote Cache Stores</td>
<td>61</td>
</tr>
<tr>
<td>8.2.6</td>
<td>RocksDB Cache Stores</td>
<td>62</td>
</tr>
<tr>
<td>8.2.7</td>
<td>Soft-Index File Stores</td>
<td>64</td>
</tr>
<tr>
<td>8.2.8</td>
<td>Implementing Custom Cache Stores</td>
<td>65</td>
</tr>
<tr>
<td>8.2.8.1</td>
<td>Data Grid Persistence SPI</td>
<td>65</td>
</tr>
<tr>
<td>8.2.8.2</td>
<td>Creating Cache Stores</td>
<td>66</td>
</tr>
<tr>
<td>8.2.8.3</td>
<td>Configuring Data Grid to Use Custom Stores</td>
<td>66</td>
</tr>
<tr>
<td>8.2.8.4</td>
<td>Deploying Custom Cache Stores</td>
<td>67</td>
</tr>
<tr>
<td>8.3</td>
<td>Migrating Between Cache Stores</td>
<td>67</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Cache Store Migrator</td>
<td>67</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Getting the Store Migrator</td>
<td>67</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Configuring the Store Migrator</td>
<td>68</td>
</tr>
<tr>
<td>8.3.3.1</td>
<td>Store Migrator Properties</td>
<td>69</td>
</tr>
</tbody>
</table>
8.3.4. Migrating Cache Stores

CHAPTER 9. SETTING UP PARTITION HANDLING ....................................................... 74
  9.1. PARTITION HANDLING 74
    9.1.1. Split brain 75
      9.1.1.1. Split Strategies 75
        9.1.1.1.1. ALLOW_READ_WRITES 75
        9.1.1.1.2. DENY_READ_WRITES 75
        9.1.1.1.3. ALLOW_READS 76
      9.1.1.2. Current limitations 76
    9.1.2. Successive nodes stopped 76
    9.1.3. Conflict Manager 77
      9.1.3.1. Detecting Conflicts 77
      9.1.3.2. Merge Policies 77
    9.1.4. Usage 78
    9.1.5. Configuring partition handling 79
      9.1.5.1. Implement a custom merge policy 79
      9.1.5.2. Deploy custom merge policies to a Infinispan server instance 80
    9.1.6. Monitoring and administration 80
CHAPTER 1. RED HAT DATA GRID

Data Grid is a high-performance, distributed in-memory data store.

**Schemaless data structure**
- Flexibility to store different objects as key-value pairs.

**Grid-based data storage**
- Designed to distribute and replicate data across clusters.

**Elastic scaling**
- Dynamically adjust the number of nodes to meet demand without service disruption.

**Data interoperability**
- Store, retrieve, and query data in the grid from different endpoints.

1.1. DATA GRID DOCUMENTATION

Documentation for Data Grid is available on the Red Hat customer portal.

- Data Grid 8.2 Documentation
- Data Grid 8.2 Component Details
- Supported Configurations for Data Grid 8.2
- Data Grid 8 Feature Support
- Data Grid Deprecated Features and Functionality

1.2. DATA GRID DOWNLOADS

Access the [Data Grid Software Downloads](#) on the Red Hat customer portal.

**NOTE**

You must have a Red Hat account to access and download Data Grid software.

1.3. MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see [our CTO Chris Wright’s message](#).
CHAPTER 2. DATA GRID CACHES

Data Grid caches provide flexible, in-memory data stores that you can configure to suit use cases such as:

- boosting application performance with high-speed local caches.
- optimizing databases by decreasing the volume of write operations.
- providing resiliency and durability for consistent data across clusters.

2.1. CACHE INTERFACE

Cache<K,V> is the central interface for Data Grid and extends java.util.concurrent.ConcurrentMap.

Cache entries are highly concurrent data structures in key:value format that support a wide and configurable range of data types, from simple strings to much more complex objects.

2.2. CACHE MANAGERS

Data Grid provides a CacheManager interface that lets you create, modify, and manage local or clustered caches. Cache Managers are the starting point for using Data Grid caches.

There are two CacheManager implementations:

EmbeddedCacheManager

Entry point for caches when running Data Grid inside the same Java Virtual Machine (JVM) as the client application, which is also known as Library Mode.

RemoteCacheManager

Entry point for caches when running Data Grid as a remote server in its own JVM. When it starts running, RemoteCacheManager establishes a persistent TCP connection to a Hot Rod endpoint on a Data Grid server.

NOTE

Both embedded and remote CacheManager implementations share some methods and properties. However, semantic differences do exist between EmbeddedCacheManager and RemoteCacheManager.

2.3. CACHE CONTAINERS

Cache containers declare one or more local or clustered caches that a Cache Manager controls.

Cache container declaration

```xml
<cache-container name="clustered" default-cache="default">
  <!-- Cache Manager configuration goes here. -->
</cache-container>
```

2.4. CACHE MODES
TIP

Data Grid Cache Managers can create and control multiple caches that use different modes. For example, you can use the same Cache Manager for local caches, distributes caches, and caches with invalidation mode.

Local Caches

Data Grid runs as a single node and never replicates read or write operations on cache entries.

Clustered Caches

Data Grid instances running on the same network can automatically discover each other and form clusters to handle cache operations.

Invalidation Mode

Rather than replicating cache entries across the cluster, Data Grid evicts stale data from all nodes whenever operations modify entries in the cache. Data Grid performs local read operations only.

Replicated Caches

Data Grid replicates each cache entry on all nodes and performs local read operations only.

Distributed Caches

Data Grid stores cache entries across a subset of nodes and assigns entries to fixed owner nodes. Data Grid requests read operations from owner nodes to ensure it returns the correct value.

Scattered Caches

Data Grid stores cache entries across a subset of nodes. By default Data Grid assigns a primary owner and a backup owner to each cache entry in scattered caches. Data Grid assigns primary owners in the same way as with distributed caches, while backup owners are always the nodes that initiate the write operations. Data Grid requests read operations from at least one owner node to ensure it returns the correct value.

2.4.1. Cache Mode Comparison

The cache mode that you should choose depends on the qualities and guarantees you need for your data.

The following table summarizes the primary differences between cache modes:

<table>
<thead>
<tr>
<th>Cache mode</th>
<th>Clustered?</th>
<th>Read performance</th>
<th>Write performance</th>
<th>Capacity</th>
<th>Availability</th>
<th>Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>No</td>
<td>High (local)</td>
<td>High (local)</td>
<td>Single node</td>
<td>Single node</td>
<td>Complete</td>
</tr>
<tr>
<td>Simple</td>
<td>No</td>
<td>Highest (local)</td>
<td>Highest (local)</td>
<td>Single node</td>
<td>Single node</td>
<td>Partial: no transactions, persistence, or indexing.</td>
</tr>
<tr>
<td>Invalidation</td>
<td>Yes</td>
<td>High (local)</td>
<td>Low (all nodes, no data)</td>
<td>Single node</td>
<td>Single node</td>
<td>Partial: no indexing.</td>
</tr>
<tr>
<td>Cache mode</td>
<td>Clustered?</td>
<td>Read performance</td>
<td>Write performance</td>
<td>Capacity</td>
<td>Availability</td>
<td>Capabilities</td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
<td>------------------</td>
<td>-------------------</td>
<td>-------------------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Replicated</td>
<td>Yes</td>
<td>High (local)</td>
<td>Lowest (all nodes)</td>
<td>Smallest node</td>
<td>All nodes</td>
<td>Complete</td>
</tr>
<tr>
<td>Distributed</td>
<td>Yes</td>
<td>Medium (owners)</td>
<td>Medium (owner nodes)</td>
<td>Sum of all nodes capacity divided by the number of owners.</td>
<td>Owner nodes</td>
<td>Complete</td>
</tr>
<tr>
<td>Scattered</td>
<td>Yes</td>
<td>Medium (primary)</td>
<td>Higher (single RPC)</td>
<td>Sum of all nodes capacity divided by 2.</td>
<td>Owner nodes</td>
<td>Partial: no transactions.</td>
</tr>
</tbody>
</table>
CHAPTER 3. LOCAL CACHES

While Data Grid is particularly interesting in clustered mode, it also offers a very capable local mode. In this mode, it acts as a simple, in-memory data cache similar to a `ConcurrentHashmap`.

But why would one use a local cache rather than a map? Caches offer a lot of features over and above a simple map, including write-through and write-behind to a persistent store, eviction of entries to prevent running out of memory, and expiration.

Data Grid’s `Cache` interface extends JDK’s `ConcurrentMap` — making migration from a map to Data Grid trivial.

Data Grid caches also support transactions, either integrating with an existing transaction manager or running a separate one. Local caches transactions have two choices:

1. When to lock? **Pessimistic locking** locks keys on a write operation or when the user calls `AdvancedCache.lock(keys)` explicitly. **Optimistic locking** only locks keys during the transaction commit, and instead it throws a `WriteSkewCheckException` at commit time, if another transaction modified the same keys after the current transaction read them.

2. Isolation level. We support **read-committed** and **repeatable read**.

3.1. SIMPLE CACHES

Traditional local caches use the same architecture as clustered caches, i.e. they use the interceptor stack. That way a lot of the implementation can be reused. However, if the advanced features are not needed and performance is more important, the interceptor stack can be stripped away and simple cache can be used.

So, which features are stripped away? From the configuration perspective, simple cache does not support:

- transactions and invocation batching
- persistence (cache stores and loaders)
- custom interceptors (there’s no interceptor stack!)
- indexing
- transcoding
- store as binary (which is hardly useful for local caches)

From the API perspective these features throw an exception:

- adding custom interceptors
- Distributed Executors Framework

So, what’s left?

- basic map-like API
- cache listeners (local ones)
- expiration
- eviction
- security
- JMX access
- statistics (though for max performance it is recommended to switch this off using statistics-available=false)

**Declarative configuration**

```xml
<local-cache name="mySimpleCache" simple-cache="true">
    <!-- Additional cache configuration goes here. -->
</local-cache>
```

**Programmatic configuration**

```java
DefaultCacheManager cm = getCacheManager();
ConfigurationBuilder builder = new ConfigurationBuilder().simpleCache(true);
cm.defineConfiguration("mySimpleCache", builder.build());
Cache cache = cm.getCache("mySimpleCache");
```

Simple cache checks against features it does not support, if you configure it to use e.g. transactions, configuration validation will throw an exception.
CHAPTER 4. CLUSTERED CACHES

Clustered caches store data across multiple Data Grid nodes using JGroups technology as the transport layer to pass data across the network.

4.1. INVALIDATION MODE

You can use Data Grid in invalidation mode to optimize systems that perform high volumes of read operations. A good example is to use invalidation to prevent lots of database writes when state changes occur.

This cache mode only makes sense if you have another, permanent store for your data such as a database and are only using Data Grid as an optimization in a read-heavy system, to prevent hitting the database for every read. If a cache is configured for invalidation, every time data is changed in a cache, other caches in the cluster receive a message informing them that their data is now stale and should be removed from memory and from any local store.

Figure 4.1. Invalidation mode

Sometimes the application reads a value from the external store and wants to write it to the local cache, without removing it from the other nodes. To do this, it must call `Cache.putForExternalRead(key, value)` instead of `Cache.put(key, value)`.

Invalidation mode can be used with a shared cache store. A write operation will both update the shared store, and it would remove the stale values from the other nodes’ memory. The benefit of this is twofold: network traffic is minimized as invalidation messages are very small compared to replicating the entire value, and also other caches in the cluster look up modified data in a lazy manner, only when needed.

NOTE

Never use invalidation mode with a local store. The invalidation message will not remove entries in the local store, and some nodes will keep seeing the stale value.

An invalidation cache can also be configured with a special cache loader, `ClusterLoader`. When `ClusterLoader` is enabled, read operations that do not find the key on the local node will request it from all the other nodes first, and store it in memory locally. In certain situation it will store stale values, so only use it if you have a high tolerance for stale values.

Invalidation mode can be synchronous or asynchronous. When synchronous, a write blocks until all nodes in the cluster have evicted the stale value. When asynchronous, the originator broadcasts invalidation messages but doesn’t wait for responses. That means other nodes still see the stale value for a while after the write completed on the originator.
Transactions can be used to batch the invalidation messages. Transactions acquire the key lock on the primary owner. To find more about how primary owners are assigned, please read the Key Ownership section.

- With pessimistic locking, each write triggers a lock message, which is broadcast to all the nodes. During transaction commit, the originator broadcasts a one-phase prepare message (optionally fire-and-forget) which invalidates all affected keys and releases the locks.

- With optimistic locking, the originator broadcasts a prepare message, a commit message, and an unlock message (optional). Either the one-phase prepare or the unlock message is fire-and-forget, and the last message always releases the locks.

4.2. REPLICATED CACHES

Entries written to a replicated cache on any node will be replicated to all other nodes in the cluster, and can be retrieved locally from any node. Replicated mode provides a quick and easy way to share state across a cluster, however replication practically only performs well in small clusters (under 10 nodes), due to the number of messages needed for a write scaling linearly with the cluster size. Data Grid can be configured to use UDP multicast, which mitigates this problem to some degree.

Each key has a primary owner, which serializes data container updates in order to provide consistency. To find more about how primary owners are assigned, please read the Key Ownership section.

Figure 4.2. Replicated mode

Replicated mode can be synchronous or asynchronous.

- Synchronous replication blocks the caller (e.g. on a cache.put(key, value)) until the modifications have been replicated successfully to all the nodes in the cluster.

- Asynchronous replication performs replication in the background, and write operations return immediately. Asynchronous replication is not recommended, because communication errors, or errors that happen on remote nodes are not reported to the caller.

If transactions are enabled, write operations are not replicated through the primary owner.

- With pessimistic locking, each write triggers a lock message, which is broadcast to all the nodes. During transaction commit, the originator broadcasts a one-phase prepare message and an unlock message (optional). Either the one-phase prepare or the unlock message is fire-and-forget.

- With optimistic locking, the originator broadcasts a prepare message, a commit message, and an unlock message (optional). Again, either the one-phase prepare or the unlock message is fire-and-forget.
4.3. DISTRIBUTED CACHES

Distribution tries to keep a fixed number of copies of any entry in the cache, configured as `numOwners`. This allows the cache to scale linearly, storing more data as nodes are added to the cluster.

As nodes join and leave the cluster, there will be times when a key has more or less than `numOwners` copies. In particular, if `numOwners` nodes leave in quick succession, some entries will be lost, so we say that a distributed cache tolerates `numOwners - 1` node failures.

The number of copies represents a trade-off between performance and durability of data. The more copies you maintain, the lower performance will be, but also the lower the risk of losing data due to server or network failures. Regardless of how many copies are maintained, distribution still scales linearly, and this is key to Data Grid’s scalability.

The owners of a key are split into one primary owner, which coordinates writes to the key, and zero or more backup owners. To find more about how primary and backup owners are assigned, please read the Key Ownership section.

Figure 4.3. Distributed mode

A read operation will request the value from the primary owner, but if it doesn’t respond in a reasonable amount of time, we request the value from the backup owners as well. (The `infinispan.stagger.delay` system property, in milliseconds, controls the delay between requests.) A read operation may require 0 messages if the key is present in the local cache, or up to `2 * numOwners` messages if all the owners are slow.

A write operation will also result in at most `2 * numOwners` messages: one message from the originator to the primary owner, `numOwners - 1` messages from the primary to the backups, and the corresponding ACK messages.

**NOTE**

Cache topology changes may cause retries and additional messages, both for reads and for writes.

Just as replicated mode, distributed mode can also be synchronous or asynchronous. And as in replicated mode, asynchronous replication is not recommended because it can lose updates. In addition to losing updates, asynchronous distributed caches can also see a stale value when a thread writes to a key and then immediately reads the same key.

Transactional distributed caches use the same kinds of messages as transactional replicated caches, except lock/prepare/commit/unlock messages are sent only to the affected nodes (all the nodes that own at least one key affected by the transaction) instead of being broadcast to all the nodes in the
cluster. As an optimization, if the transaction writes to a single key and the originator is the primary owner of the key, lock messages are not replicated.

4.3.1. Read consistency

Even with synchronous replication, distributed caches are not linearizable. (For transactional caches, we say they do not support serialization/snapshot isolation.) We can have one thread doing a single put:

```
cache.get(k) -> v1
cache.put(k, v2)
cache.get(k) -> v2
```

But another thread might see the values in a different order:

```
cache.get(k) -> v2
cache.get(k) -> v1
```

The reason is that read can return the value from any owner, depending on how fast the primary owner replies. The write is not atomic across all the owners — in fact, the primary commits the update only after it receives a confirmation from the backup. While the primary is waiting for the confirmation message from the backup, reads from the backup will see the new value, but reads from the primary will see the old one.

4.3.2. Key Ownership

Distributed caches split entries into a fixed number of segments and assign each segment to a list of owner nodes. Replicated caches do the same, with the exception that every node is an owner.

The first node in the list of owners is the primary owner. The other nodes in the list are backup owners. When the cache topology changes, because a node joins or leaves the cluster, the segment ownership table is broadcast to every node. This allows nodes to locate keys without making multicast requests or maintaining metadata for each key.

The numSegments property configures the number of segments available. However, the number of segments cannot change unless the cluster is restarted.

Likewise the key-to-segment mapping cannot change. Keys must always map to the same segments regardless of cluster topology changes. It is important that the key-to-segment mapping evenly distributes the number of segments allocated to each node while minimizing the number of segments that must move when the cluster topology changes.

You can customize the key-to-segment mapping by configuring a KeyPartitioner or by using the Grouping API.

However, Data Grid provides the following implementations:

**SyncConsistentHashFactory**

Uses an algorithm based on consistent hashing. Selected by default when server hinting is disabled. This implementation always assigns keys to the same nodes in every cache as long as the cluster is symmetric. In other words, all caches run on all nodes. This implementation does have some negative points in that the load distribution is slightly uneven. It also moves more segments than strictly necessary on a join or leave.

**TopologyAwareSyncConsistentHashFactory**
Similar to **SyncConsistentHashFactory**, but adapted for **Server Hinting**. Selected by default when server hinting is enabled.

**DefaultConsistentHashFactory**

Achieves a more even distribution than **SyncConsistentHashFactory**, but with one disadvantage. The order in which nodes join the cluster determines which nodes own which segments. As a result, keys might be assigned to different nodes in different caches.

Was the default from version 5.2 to version 8.1 with server hinting disabled.

**TopologyAwareConsistentHashFactory**

Similar to **DefaultConsistentHashFactory**, but adapted for **Server Hinting**.

Was the default from version 5.2 to version 8.1 with server hinting enabled.

**ReplicatedConsistentHashFactory**

Used internally to implement replicated caches. You should never explicitly select this algorithm in a distributed cache.

### 4.3.2.1. Capacity Factors

Capacity factors allocate segment-to-node mappings based on resources available to nodes.

To configure capacity factors, you specify any non-negative number and the Data Grid hashing algorithm assigns each node a load weighted by its capacity factor (both as a primary owner and as a backup owner).

For example, nodeA has 2x the memory available than nodeB in the same Data Grid cluster. In this case, setting `capacityFactor` to a value of **2** configures Data Grid to allocate 2x the number of segments to nodeA.

Setting a capacity factor of **0** is possible but is recommended only in cases where nodes are not joined to the cluster long enough to be useful data owners.

### 4.3.3. Zero Capacity Node

You might need to configure a whole node where the capacity factor is **0** for every cache, user defined caches and internal caches. When defining a zero capacity node, the node won’t hold any data. This is how you declare a zero capacity node:

```xml
<cache-container zero-capacity-node="true"/>
```

```java
new GlobalConfigurationBuilder().zeroCapacityNode(true);
```

### 4.3.4. Hashing Configuration

This is how you configure hashing declaratively, via XML:

```xml
<distributed-cache name="distributedCache" owners="2" segments="100" capacity-factor="2"/>
```

And this is how you can configure it programatically, in Java:

```java
Configuration c = new ConfigurationBuilder()
    .clustering();
```
4.3.5. Initial cluster size

Data Grid’s very dynamic nature in handling topology changes (i.e. nodes being added / removed at runtime) means that, normally, a node doesn’t wait for the presence of other nodes before starting. While this is very flexible, it might not be suitable for applications which require a specific number of nodes to join the cluster before caches are started. For this reason, you can specify how many nodes should have joined the cluster before proceeding with cache initialization. To do this, use the `initialClusterSize` and `initialClusterTimeout` transport properties. The declarative XML configuration:

```xml
<transport initial-cluster-size="4" initial-cluster-timeout="30000" />
```

The programmatic Java configuration:

```java
GlobalConfiguration global = new GlobalConfigurationBuilder()
    .transport()
    .initialClusterSize(4)
    .initialClusterTimeout(30000, TimeUnit.MILLISECONDS)
    .build();
```

The above configuration will wait for 4 nodes to join the cluster before initialization. If the initial nodes do not appear within the specified timeout, the cache manager will fail to start.

4.3.6. L1 Caching

When L1 is enabled, a node will keep the result of remote reads locally for a short period of time (configurable, 10 minutes by default), and repeated lookups will return the local L1 value instead of asking the owners again.

**Figure 4.4. L1 caching**

L1 caching is not free though. Enabling it comes at a cost, and this cost is that every entry update must broadcast an invalidation message to all the nodes. L1 entries can be evicted just like any other entry when the the cache is configured with a maximum size. Enabling L1 will improve performance for repeated reads of non-local keys, but it will slow down writes and it will increase memory consumption to some degree.
Is L1 caching right for you? The correct approach is to benchmark your application with and without L1 enabled and see what works best for your access pattern.

### 4.3.7. Server Hinting

The following topology hints can be specified:

**Machine**

This is probably the most useful, when multiple JVM instances run on the same node, or even when multiple virtual machines run on the same physical machine.

**Rack**

In larger clusters, nodes located on the same rack are more likely to experience a hardware or network failure at the same time.

**Site**

Some clusters may have nodes in multiple physical locations for extra resilience. Note that Cross site replication is another alternative for clusters that need to span two or more data centres.

All of the above are optional. When provided, the distribution algorithm will try to spread the ownership of each segment across as many sites, racks, and machines (in this order) as possible.

#### 4.3.7.1. Configuration

The hints are configured at transport level:

```xml
<transport
  cluster="MyCluster"
  machine="LinuxServer01"
  rack="Rack01"
  site="US-WestCoast"/>
```

### 4.3.8. Key affinity service

In a distributed cache, a key is allocated to a list of nodes with an opaque algorithm. There is no easy way to reverse the computation and generate a key that maps to a particular node. However, we can generate a sequence of (pseudo-)random keys, see what their primary owner is, and hand them out to the application when it needs a key mapping to a particular node.

#### 4.3.8.1. API

Following code snippet depicts how a reference to this service can be obtained and used.

```java
// 1. Obtain a reference to a cache
Cache cache = ...
Address address = cache.getCacheManager().getAddress();

// 2. Create the affinity service
KeyAffinityService keyAffinityService = KeyAffinityServiceFactory.newLocalKeyAffinityService(
  cache,
  new RndKeyGenerator(),
  Executors.newSingleThreadExecutor(),
  100);
```
The service is started at step 2: after this point it uses the supplied Executor to generate and queue keys. At step 3, we obtain a key from the service, and at step 4 we use it.

4.3.8.2. Lifecycle

**KeyAffinityService** extends **Lifecycle**, which allows stopping and (re)starting it:

```java
public interface Lifecycle {
    void start();
    void stop();
}
```

The service is instantiated through **KeyAffinityServiceFactory**. All the factory methods have an Executor parameter, that is used for asynchronous key generation (so that it won’t happen in the caller’s thread). It is the user’s responsibility to handle the shutdown of this Executor.

The **KeyAffinityService**, once started, needs to be explicitly stopped. This stops the background key generation and releases other held resources.

The only situation in which **KeyAffinityService** stops by itself is when the cache manager with which it was registered is shutdown.

4.3.8.3. Topology changes

When the cache topology changes (i.e. nodes join or leave the cluster), the ownership of the keys generated by the **KeyAffinityService** might change. The key affinity service keep tracks of these topology changes and doesn’t return keys that would currently map to a different node, but it won’t do anything about keys generated earlier.

As such, applications should treat **KeyAffinityService** purely as an optimization, and they should not rely on the location of a generated key for correctness.

In particular, applications should not rely on keys generated by **KeyAffinityService** for the same address to always be located together. Collocation of keys is only provided by the **Grouping API**.

4.3.8.4. The Grouping API

Complementary to **Key affinity service**, the grouping API allows you to co-locate a group of entries on the same nodes, but without being able to select the actual nodes.

4.3.8.5. How does it work?

By default, the segment of a key is computed using the key’s `hashCode()`. If you use the grouping API, Data Grid will compute the segment of the group and use that as the segment of the key. See the **Key Ownership** section for more details on how segments are then mapped to nodes.

When the group API is in use, it is important that every node can still compute the owners of every key without contacting other nodes. For this reason, the group cannot be specified manually. The group can either be intrinsic to the entry (generated by the key class) or extrinsic (generated by an external
4.3.8.6. How do I use the grouping API?

First, you must enable groups. If you are configuring Data Grid programmatically, then call:

```java
Configuration c = new ConfigurationBuilder()
    .clustering().hash().groups().enabled()
    .build();
```

Or, if you are using XML:

```xml
<distributed-cache>
    <groups enabled="true"/>
</distributed-cache>
```

If you have control of the key class (you can alter the class definition, it’s not part of an unmodifiable library), then we recommend using an intrinsic group. The intrinsic group is specified by adding the `@Group` annotation to a method. Let’s take a look at an example:

```java
class User {
    ...
    String office;
    ...

    public int hashCode() {
        // Defines the hash for the key, normally used to determine location
        ...
    }

    // Override the location by specifying a group
    // All keys in the same group end up with the same owners
    @Group
    public String getOffice() {
        return office;
    }
}
```

**NOTE**
The group method must return a `String`.

If you don’t have control over the key class, or the determination of the group is an orthogonal concern to the key class, we recommend using an extrinsic group. An extrinsic group is specified by implementing the `Grouper` interface.

```java
public interface Grouper<T> {
    String computeGroup(T key, String group);
    Class<T> getKeyType();
}
```
If multiple **Grouper** classes are configured for the same key type, all of them will be called, receiving the value computed by the previous one. If the key class also has a [@Group](#) annotation, the first **Grouper** will receive the group computed by the annotated method. This allows you even greater control over the group when using an intrinsic group. Let’s take a look at an example **Grouper** implementation:

```java
public class KXGrouper implements Grouper<String> {
    // The pattern requires a String key, of length 2, where the first character is
    // "k" and the second character is a digit. We take that digit, and perform
    // modular arithmetic on it to assign it to group "0" or group "1".
    private static Pattern kPattern = Pattern.compile("(^k)<a>\d</a>$");

    public String computeGroup(String key, String group) {
        Matcher matcher = kPattern.matcher(key);
        if (matcher.matches()) {
            String g = Integer.parseInt(matcher.group(2)) % 2 + "";
            return g;
        } else {
            return null;
        }
    }

    public Class<String> getKeyType() {
        return String.class;
    }
}
```

**Grouper** implementations must be registered explicitly in the cache configuration. If you are configuring Data Grid programmatically:

```java
Configuration c = new ConfigurationBuilder()
    .clustering().hash().groups().enabled().addGrouper(new KXGrouper())
    .build();
```

Or, if you are using XML:

```xml
<distributed-cache>
    <groups enabled="true">
        <grouper class="com.acme.KXGrouper"/>
    </groups>
</distributed-cache>
```

### 4.3.8.7. Advanced Interface

**AdvancedCache** has two group-specific methods:

- **getGroup(groupName)**
  - Retrieves all keys in the cache that belong to a group.

- **removeGroup(groupName)**
  - Removes all the keys in the cache that belong to a group.

Both methods iterate over the entire data container and store (if present), so they can be slow when a cache contains lots of small groups.
4.4. SCATTERED CACHES

Scattered mode is very similar to Distribution Mode as it allows linear scaling of the cluster. It allows single node failure by maintaining two copies of the data (as Distribution Mode with numOwners=2). Unlike Distributed, the location of data is not fixed; while we use the same Consistent Hash algorithm to locate the primary owner, the backup copy is stored on the node that wrote the data last time. When the write originates on the primary owner, backup copy is stored on any other node (the exact location of this copy is not important).

This has the advantage of single Remote Procedure Call (RPC) for any write (Distribution Mode requires one or two RPCs), but reads have to always target the primary owner. That results in faster writes but possibly slower reads, and therefore this mode is more suitable for write-intensive applications.

Storing multiple backup copies also results in slightly higher memory consumption. In order to remove out-of-date backup copies, invalidation messages are broadcast in the cluster, which generates some overhead. This makes scattered mode less performant in very big clusters (this behaviour might be optimized in the future).

When a node crashes, the primary copy may be lost. Therefore, the cluster has to reconcile the backups and find out the last written backup copy. This process results in more network traffic during state transfer.

Since the writer of data is also a backup, even if we specify machine/rack/site ids on the transport level the cluster cannot be resilient to more than one failure on the same machine/rack/site.

Currently it is not possible to use scattered mode in transactional cache. Asynchronous replication is not supported either; use asynchronous Cache API instead. Functional commands are not implemented neither but these are expected to be added soon.

The cache is configured in a similar way as the other cache modes, here is an example of declarative configuration:

```xml
<scattered-cache name="scatteredCache" />
```

And this is how you can configure it programmatically:

```java
Configuration c = new ConfigurationBuilder()
    .clustering().cacheMode(CacheMode.SCATTERED_SYNC)
    .build();
```

Scattered mode is not exposed in the server configuration as the server is usually accessed through the Hot Rod protocol. The protocol automatically selects primary owner for the writes and therefore the write (in distributed mode with two owner) requires single RPC inside the cluster, too. Therefore, scattered cache would not bring the performance benefit.

4.5. ASYNCHRONOUS COMMUNICATION WITH CLUSTERED CACHES

4.5.1. Asynchronous Communications

All clustered cache modes can be configured to use asynchronous communications with the mode="ASYNC" attribute on the `<replicated-cache/>`, `<distributed-cache>`, or `<invalidation-cache/>` element.
With asynchronous communications, the originator node does not receive any acknowledgement from
the other nodes about the status of the operation, so there is no way to check if it succeeded on other
nodes.

We do not recommend asynchronous communications in general, as they can cause inconsistencies in
the data, and the results are hard to reason about. Nevertheless, sometimes speed is more important
than consistency, and the option is available for those cases.

4.5.2. Asynchronous API

The Asynchronous API allows you to use synchronous communications, but without blocking the user
thread.

There is one caveat: The asynchronous operations do NOT preserve the program order. If a thread calls
cache.putAsync(k, v1); cache.putAsync(k, v2), the final value of k may be either v1 or v2. The
advantage over using asynchronous communications is that the final value can’t be v1 on one node and
v2 on another.

4.5.3. Return Values in Asynchronous Communication

Because the Cache interface extends java.util.Map, write methods like put(key, value) and
remove(key) return the previous value by default.

In some cases, the return value may not be correct:

1. When using AdvancedCache.withFlags() with Flag.IGNORE_RETURN_VALUE,
   Flag.SKIP_REMOTE_LOOKUP, or Flag.SKIP_CACHE_LOAD.

2. When the cache is configured with unreliable-return-values="true".

3. When using asynchronous communications.

4. When there are multiple concurrent writes to the same key, and the cache topology changes.
   The topology change will make Data Grid retry the write operations, and a retried operation’s
   return value is not reliable.

Transactional caches return the correct previous value in cases 3 and 4. However, transactional caches
also have a gotcha: in distributed mode, the read-committed isolation level is implemented as
repeatable-read. That means this example of “double-checked locking” won’t work:

```java
Cache cache = ...
TransactionManager tm = ...

tm.begin();
try {
    Integer v1 = cache.get(k);
    // Increment the value
    Integer v2 = cache.put(k, v1 + 1);
    if (Objects.equals(v1, v2) {
        // success
    } else {
        // retry
    }
} finally {
    tm.commit();
}
```
The correct way to implement this is to use
```
cache.getAdvancedCache().withFlags(Flag.FORCE_WRITE_LOCK).get(k).
```

In caches with optimistic locking, writes can also return stale previous values. Write skew checks can avoid stale previous values.
CHAPTER 5. CONFIGURING DATA GRID CACHES

Data Grid lets you define properties and options for caches both declaratively and programmatically.

Declarative configuration uses XML files that adhere to a Data Grid schema. Programmatic configuration, on the other hand, uses Data Grid APIs.

In most cases, you use declarative configuration as a starting point for cache definitions. At runtime you can then programmatically configure your caches to tune settings or specify additional properties. However, Data Grid provides flexibility so you can choose either declarative, programmatic, or a combination of the two.

5.1. DECLARATIVE CONFIGURATION

Declarative configuration conforms to a schema and is defined in XML or JSON formatted files.

The following example shows the basic structure of a Data Grid configuration:

```xml
<infinispan>
  <!-- Defines properties for all caches within the container and optionally names a default cache. -->
  <cache-container default-cache="local">
    <!-- Configures transport properties for clustered cache modes. -->
    <!-- Specifies the default JGroups UDP stack and names the cluster. -->
    <transport stack="udp" cluster="mycluster"/>
    <!-- Configures a local cache. -->
    <local-cache name="local"/>
    <!-- Configures an invalidation cache. -->
    <invalidation-cache name="invalidation"/>
    <!-- Configures a replicated cache. -->
    <replicated-cache name="replicated"/>
    <!-- Configures a distributed cache. -->
    <distributed-cache name="distributed"/>
  </cache-container>
</infinispan>
```

Reference

- Data Grid 8.2 Configuration Schema
- infinispan-config-8.2.xsd

5.1.1. Cache Templates

Data Grid lets you define templates that you can use to create cache configurations.

For example, the following configuration contains a cache template:

```xml
<infinispan>
  <!-- Specifies the cache named "local" as the default. -->
  <cache-container default-cache="local">
    <!-- Adds a cache template for local caches. -->
    <local-cache-configuration name="local-template">
      <expiration interval="10000" lifespan="10" max-idle="10"/>
    </local-cache-configuration>
  </cache-container>
</infinispan>
```
Inheritance with configuration templates

Configuration templates can also inherit from other templates to extend and override settings.

**NOTE**

Cache template inheritance is hierarchical. For a child configuration template to inherit from a parent, you must include it after the parent template.

The following is an example of template inheritance:

```xml
<infinispan>
  <cache-container>
    <!-- Defines a cache template named "base-template". -->
    <local-cache-configuration name="base-template">
      <expiration interval="10000" lifespan="10" max-idle="10"/>
    </local-cache-configuration>
    <!-- Defines a cache template named "extended-template" that inherits settings from "base-template". -->
    <local-cache-configuration name="extended-template" configuration="base-template">
      <expiration lifespan="20"/>
      <memory max-size="2GB"/>
    </local-cache-configuration>
  </cache-container>
</infinispan>

**IMPORTANT**

Configuration template inheritance is additive for elements that have multiple values, such as `property`. Resulting child configurations merge values from parent configurations.

For example, `<property value_x="foo"/>` in a parent configuration merges with `<property value_y="bar"/>` in a child configuration to result in `<property value_x="foo" value_y="bar"/>`.

5.1.2. Cache Configuration Wildcards

You can use wildcards to match cache definitions to configuration templates.

```xml
<infinispan>
  <cache-container>
    <!-- Uses the "*" wildcard to match any cache names that start with "basecache". -->
    <local-cache-configuration name="basecache*">
      <expiration interval="10500" lifespan="11" max-idle="11"/>
    </local-cache-configuration>
    <!-- Adds local caches that use the "basecache*" configuration. -->
    <local-cache name="basecache-1"/>
  </cache-container>
</infinispan>
```
5.1.3. Multiple Configuration Files

Data Grid supports XML inclusions (XInclude) that allow you to split configuration across multiple files.

If you want to use a schema for your included fragments, use the `infinispan-config-fragment-12.1.xsd` schema:

```xml
<local-cache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="urn:infinispan:config:12.1 https://infinispan.org/schemas/infinispan-config-fragment-12.1.xsd"
             xmlns="urn:infinispan:config:12.1"
             name="mycache"/>
```

NOTE

Data Grid configuration provides only minimal support for the XInclude specification. For example, you cannot use the `xpointer` attribute, the `xi:fallback` element, text processing, or content negotiation.

Reference

XInclude specification

5.2. DATA GRID CONFIGURATION API

Configure Data Grid programmatically.

Global configuration

Use the `GlobalConfiguration` class to apply configuration to all caches under the Cache Manager.

```java
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
    .cacheContainer().statistics(true)  // 1
    .metrics().gauges(true).histograms(true)  // 2
    .jmx().enable()  // 3
    .build();
```
Enables Cache Manager statistics.

Exports statistics through the **metrics** endpoint.

Exports statistics via JMX MBeans.

References:

- org.infinispan.configuration.global.GlobalConfiguration.
- Global Configuration API.

**Cache configuration**

Use the **ConfigurationBuilder** class to configure caches.

```java
ConfigurationBuilder builder = new ConfigurationBuilder();
    builder.clustering()
        .cacheMode(CacheMode.DIST_SYNC)
        .l1().lifespan(25000L)
        .hash().numOwners(3)
        .statistics().enable();
Configuration cfg = builder.build();
```

1. Enables cache clustering.
2. Uses the distributed, synchronous cache mode.
3. Configures maximum lifespan for entries in the L1 cache.
4. Configures three cluster-wide replicas for each cache entry.
5. Enables cache statistics.

References:

- org.infinispan.configuration.cache.ConfigurationBuilder.

### 5.3. CONFIGURING CACHES PROGRAMMATICALLY

Define cache configurations with the Cache Manager.

**NOTE**

The examples in this section use **EmbeddedCacheManager**, which is a Cache Manager that runs in the same JVM as the client.

To configure caches remotely with HotRod clients, you use **RemoteCacheManager**. Refer to the HotRod documentation for more information.

**Configure new cache instances**

The following example configures a new cache instance:
Creates a new Configuration object.

Specifies distributed, synchronous cache mode.

Defines a new cache named "replicatedCache" with the Configuration object.

Create new caches from existing configurations

The following examples create new cache configurations from existing ones:

```java
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-prod.xml");
Configuration c = manager.getConfigurations().get("replicatedCache").getClustering() 1
  .cacheMode(CacheMode.REPL_SYNC) 2
  .build();

String newCacheName = "replicatedCache";
manager.defineConfiguration(newCacheName, c); 3
Cache<String, String> cache = manager.getCache(newCacheName);
```

1 Creates a new Configuration object.
2 Specifies distributed, synchronous cache mode.
3 Defines a new cache named "replicatedCache" with the Configuration object.

```java
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-prod.xml");
Configuration dcc = manager.getDefaultCacheConfiguration(); 1
Configuration c = new ConfigurationBuilder().read(dcc) 2
  .clustering() 3
  .cacheMode(CacheMode.DIST_SYNC) 4
  .l1() 5
  .lifespan(60000L) 6
  .build();

String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c); 5
Cache<String, String> cache = manager.getCache(newCacheName);
```

1 Returns the default cache configuration from the Cache Manager. In this example, *infinispan-prod.xml* defines a replicated cache as the default.
2 Creates a new Configuration object that uses the default cache configuration as a base.
3 Specifies distributed, synchronous cache mode.
4 Adds an L1 lifespan configuration.
5 Defines a new cache named "distributedWithL1" with the Configuration object.

```java
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-prod.xml");
Configuration rc = manager.getCacheConfiguration("replicatedCache"); 1
Configuration c = new ConfigurationBuilder().read(rc) 2
  .clustering() 3
  .cacheMode(CacheMode.DIST_SYNC) 4
  .l1() 5
  .lifespan(60000L) 6
  .build();
```

1 Creates a new Configuration object that uses the default cache configuration as a base.
2 Specifies distributed, synchronous cache mode.
3 Adds an L1 lifespan configuration.
4 Defines a new cache named "replicatedCache" with the Configuration object.
String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);

Uses a cache configuration named "replicatedCache" as a base.

Reference

- CacheManager package summary
- org.infinispan.configuration.cache.ConfigurationBuilder
- org.infinispan.manager.EmbeddedCacheManager
- HotRod Java Client Guide
- org.infinispan.client.hotrod.configuration.ConfigurationBuilder
- org.infinispan.client.hotrod.RemoteCacheManager
CHAPTER 6. MANAGING MEMORY

Configure the data container where Data Grid stores entries. Specify the encoding for cache entries, store data in off-heap memory, and use eviction or expiration policies to keep only active entries in memory.

6.1. CONFIGURING EVICTION AND EXPIRATION

Eviction and expiration are two strategies for cleaning the data container by removing old, unused entries. Although eviction and expiration are similar, they have some important differences.

✓ Eviction lets Data Grid control the size of the data container by removing entries when the container becomes larger than a configured threshold.

✓ Expiration limits the amount of time entries can exist. Data Grid uses a scheduler to periodically remove expired entries. Entries that are expired but not yet removed are immediately removed on access; in this case get() calls for expired entries return "null" values.

✓ Eviction is local to Data Grid nodes.

✓ Expiration takes place across Data Grid clusters.

✓ You can use eviction and expiration together or independently of each other.

✓ You can configure eviction and expiration declaratively in infinispan.xml to apply cache-wide defaults for entries.

✓ You can explicitly define expiration settings for specific entries but you cannot define eviction on a per-entry basis.

✓ You can manually evict entries and manually trigger expiration.

6.1.1. Eviction

Eviction lets you control the size of the data container by removing entries from memory. Eviction drops one entry from the data container at a time and is local to the node on which it occurs.

IMPORTANT

Eviction removes entries from memory but not from persistent cache stores. To ensure that entries remain available after Data Grid evicts them, and to prevent inconsistencies with your data, you should configure a persistent cache store.

Data Grid eviction relies on two configurations:

- Maximum size of the data container.
- Strategy for removing entries.

Data container size

Data Grid lets you store entries either in the Java heap or in native memory (off-heap) and set a maximum size for the data container.

You configure the maximum size of the data container in one of two ways:
- Total number of entries (max-count).
- Maximum amount of memory (max-size).

To perform eviction based on the amount of memory, you define a maximum size in bytes. For this reason, you must encode entries with a binary storage format such as application/x-protobuf.

**Evicting cache entries**

When you configure *memory*, Data Grid approximates the current memory usage of the data container. When entries are added or modified, Data Grid compares the current memory usage of the data container to the maximum size. If the size exceeds the maximum, Data Grid performs eviction.

Eviction happens immediately in the thread that adds an entry that exceeds the maximum size.

Consider the following configuration as an example:

```<memory max-count="50"/>```

In this case, the cache can have a total of 50 entries. After the cache reaches the total number of entries, write operations trigger Data Grid to perform eviction.

**Eviction strategies**

Strategies control how Data Grid performs eviction. You can either perform eviction manually or configure Data Grid to do one of the following:

- Remove old entries to make space for new ones.
- Throw `ContainerFullException` and prevent new entries from being created. The exception eviction strategy works only with transactional caches that use 2 phase commits; not with 1 phase commits or synchronization optimizations.

**NOTE**

Data Grid includes the Caffeine caching library that implements a variation of the Least Frequently Used (LFU) cache replacement algorithm known as TinyLFU. For off-heap storage, Data Grid uses a custom implementation of the Least Recently Used (LRU) algorithm.

**References**

- Caffeine
- Setting Up Persistent Storage

**6.1.1. Configuring the Total Number of Entries for Data Grid Caches**

Limit the size of the data container for cache entries to a total number of entries.

**Procedure**

1. Configure your Data Grid cache encoding with an appropriate storage format.
2. Specify the total number of entries that caches can contain before Data Grid performs eviction.
3. Configure an eviction strategy to control how Data Grid removes entries.
   - Declaratively: Set the **when-full** attribute.
   - Programmatically: Call the **whenFull()** method.

**Declarative example**

```xml
<local-cache name="maximum_count">
    <encoding media-type="application/x-protostream"/>
    <memory max-count="500" when-full="REMOVE"/>
</local-cache>
```

**Programmatic example**

```java
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg
    .encoding()
    .mediaType("application/x-protostream")
    .memory()
    .maxCount(500)
    .whenFull(EvictionStrategy.REMOVE)
    .build();
```

**Reference**

- Data Grid Configuration Schema Reference
- org.infinispan.configuration.cache.MemoryConfigurationBuilder

### 6.1.1.2. Configuring the Maximum Amount of Memory for Data Grid Caches

Limit the size of the data container for cache entries to a maximum amount of memory.

**Procedure**

1. Configure your Data Grid cache to use a storage format that supports binary encoding. You must use a binary storage format to perform eviction based on the maximum amount of memory.

2. Configure the maximum amount of memory, in bytes, that caches can use before Data Grid performs eviction.
   - Declaratively: Set the **max-size** attribute.
   - Programmatically: Use the **maxSize()** method.

3. Optionally specify a byte unit of measurement. The default is B (bytes). Refer to the configuration schema for supported units.
4. Configure an eviction strategy to control how Data Grid removes entries.
   
   - Declaratively: Set the **when-full** attribute.
   - Programatically: Use the **whenFull()** method.

**Declarative example**

```xml
<local-cache name="maximum_size">
  <encoding media-type="application/x-protostream"/>
  <memory max-size="1.5GB" when-full="REMOVE"/>
</local-cache>
```

**Programmatic example**

```java
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg
  .encoding()
  .mediaType("application/x-protostream")
  .memory()
  .maxSize("1.5GB")
  .whenFull(EvictionStrategy.REMOVE)
  .build());
```

**Reference**

- [Data Grid Configuration Schema Reference](org.infinispan.configuration.cache.EncodingConfiguration)
- [org.infinispan.configuration.cache.EncodingConfiguration](org.infinispan.configuration.cache.MemoryConfigurationBuilder)
- [Cache Encoding and Marshalling](org.infinispan.configuration.cache.MemoryConfigurationBuilder)

### 6.1.1.3. Eviction Examples

Configure eviction as part of your cache definition.

**Default memory configuration**

Eviction is not enabled, which is the default configuration. Data Grid stores cache entries as objects in the JVM heap.

```xml
<distributed-cache name="default_memory">
  <memory />
</distributed-cache>
```

**Eviction based on the total number of entries**

Data Grid stores cache entries as objects in the JVM heap. Eviction happens when there are 100 entries in the data container and Data Grid gets a request to create a new entry:
Eviction based maximum size in bytes

Data Grid stores cache entries as byte[] arrays if you encode entries with binary storage formats, for example: application/x-protostream format.

In the following example, Data Grid performs eviction when the size of the data container reaches 500 MB (megabytes) in size and it gets a request to create a new entry:

```xml
<distributed-cache name="binary_storage">
  <!-- Encodes the cache with a binary storage format. -->
  <encoding media-type="application/x-protostream"/>
  <!-- Bounds the data container to a maximum size in MB (megabytes). -->
  <memory max-size="500 MB"/>
</distributed-cache>
```

Off-heap storage

Data Grid stores cache entries as bytes in native memory. Eviction happens when there are 100 entries in the data container and Data Grid gets a request to create a new entry:

```xml
<distributed-cache name="off_heap">
  <memory storage="OFF_HEAP" max-count="100"/>
</distributed-cache>
```

Off-heap storage with the exception strategy

Data Grid stores cache entries as bytes in native memory. When there are 100 entries in the data container, and Data Grid gets a request to create a new entry, it throws an exception and does not allow the new entry:

```xml
<distributed-cache name="eviction_exception">
  <memory storage="OFF_HEAP" max-count="100" when-full="EXCEPTION"/>
</distributed-cache>
```

Manual eviction

Data Grid stores cache entries as objects in the JVM heap. Eviction is not enabled but performed manually using the evict() method.

**TIP**

This configuration prevents a warning message when you enable passivation but do not configure eviction.

```xml
<distributed-cache name="eviction_manual">
  <memory when-full="MANUAL"/>
</distributed-cache>
```

Passivation with eviction
Passivation persists data to cache stores when Data Grid evicts entries. You should always enable eviction if you enable passivation, for example:

```xml
<distributed-cache name="passivation">
  <persistence passivation="true">
    <!-- Persistence configuration goes here. -->
  </persistence>
  <memory max-count="100"/>
</distributed-cache>
```

**References**

- Passivation

### 6.1.2. Expiration

Expiration removes entries from caches when they reach one of the following time limits:

**Lifespan**

Sets the maximum amount of time that entries can exist.

**Maximum idle**

Specifies how long entries can remain idle. If operations do not occur for entries, they become idle.

**IMPORTANT**

Maximum idle expiration does not currently support cache configurations with persistent cache stores.

When using expiration with an exception-based eviction policy, entries that are expired but not yet removed from the cache count towards the size of the data container.

### 6.1.2.1. How Expiration Works

When you configure expiration, Data Grid stores keys with metadata that determines when entries expire.

- Lifespan uses a `creation` timestamp and the value for the `lifespan` configuration property.
- Maximum idle uses a `last used` timestamp and the value for the `max-idle` configuration property.

Data Grid checks if lifespan or maximum idle metadata is set and then compares the values with the current time.

If `(creation + lifespan < currentTime)` or `(lastUsed + maxIdle < currentTime)` then Data Grid detects that the entry is expired.

Expiration occurs whenever entries are accessed or found by the expiration reaper.
For example, k1 reaches the maximum idle time and a Cache.get(k1) request. In this case, Data Grid detects that the entry is expired and removes it from the data container. The Cache.get() returns null.

Data Grid also expires entries from cache stores, but only with lifespan expiration. Maximum idle expiration does not work with cache stores. In the case of cache loaders, Data Grid cannot expire entries because loaders can only read from external storage.

**NOTE**
Data Grid adds expiration metadata as long primitive data types to cache entries. This can increase the size of keys by as much as 32 bytes.

### 6.1.2.2. Expiration Reaper

Data Grid uses a reaper thread that runs periodically to detect and remove expired entries. The expiration reaper ensures that expired entries that are no longer accessed are removed.

The Data Grid ExpirationManager interface handles the expiration reaper and exposes the processExpiration() method.

In some cases, you can disable the expiration reaper and manually expire entries by calling processExpiration(); for instance, if you are using local cache mode with a custom application where a maintenance thread runs periodically.

**IMPORTANT**
If you use clustered cache modes, you should never disable the expiration reaper.

Data Grid always uses the expiration reaper when using cache stores. In this case you cannot disable it.

**Reference**

- org.infinispan.configuration.cache.ExpirationConfigurationBuilder
- org.infinispan.expiration ExpirationManager

### 6.1.2.3. Maximum Idle and Clustered Caches

Because maximum idle expiration relies on the last access time for cache entries, it has some limitations with clustered cache modes.

With lifespan expiration, the creation time for cache entries provides a value that is consistent across clustered caches. For example, the creation time for k1 is always the same on all nodes.

For maximum idle expiration with clustered caches, last access time for entries is not always the same on all nodes. To ensure that entries have the same relative access times across clusters, Data Grid sends touch commands to all owners when keys are accessed.

The touch commands that Data Grid send have the following considerations:

- Cache.get() requests do not return until all touch commands complete. This synchronous behavior increases latency of client requests.
- The touch command also updates the "recently accessed" metadata for cache entries on all owners, which Data Grid uses for eviction.

- With scattered cache mode, Data Grid sends touch commands to all nodes, not just primary and backup owners.

**Additional information**

- Maximum idle expiration does not work with invalidation mode.

- Iteration across a clustered cache can return expired entries that have exceeded the maximum idle time limit. This behavior ensures performance because no remote invocations are performed during the iteration. Also note that iteration does not refresh any expired entries.

### 6.1.2.4. Expiration Examples

When you configure Data Grid to expire entries, you can set lifespan and maximum idle times for:

- All entries in a cache (cache-wide). You can configure cache-wide expiration in `infinispan.xml` or programmatically using the `ConfigurationBuilder`.

- Per entry, which takes priority over cache-wide expiration values. You configure expiration for specific entries when you create them.

**NOTE**

When you explicitly define lifespan and maximum idle time values for cache entries, Data Grid replicates those values across the cluster along with the cache entries. Likewise, Data Grid persists expiration values along with the entries if you configure cache stores.

#### Configuring expiration for all cache entries

Expire all cache entries after 2 seconds:

```
<expiration lifespan="2000"/>
```

Expire all cache entries 1 second after last access time:

```
<expiration max-idle="1000"/>
```

Disable the expiration reaper with the `interval` attribute and manually expire entries 1 second after last access time:

```
<expiration max-idle="1000" interval="-1"/>
```

Expire all cache entries after 5 seconds or 1 second after the last access time, whichever happens first:

```
<expiration lifespan="5000" max-idle="1000"/>
```

#### Configuring expiration when creating cache entries

The following example shows how to configure lifespan and maximum idle values when creating cache entries:
If the Data Grid configuration defines a lifespan value of 1000 for all entries, the preceding `Cache.put()` requests cause the entries to expire:

1. After 1 second.
2. After 2 seconds.
3. 1 second after last access time.
4. After 5 seconds or 1 second after the last access time, whichever happens first.

Reference

- Data Grid Configuration Schema
- `org.infinispan.configuration.cache.ExpirationConfigurationBuilder`
- `org.infinispan.expiration.ExpirationManager`

6.2. OFF-HEAP MEMORY

Data Grid can store cache entries in JVM heap memory or off-heap. Off-heap storage lets your Java objects occupy native memory outside the managed JVM memory space.

Off-heap storage requires less overhead per entry than in JVM heap memory. In other words, off-heap storage uses less memory than heap storage for the same amount of data.

Another benefit of off-heap storage is that it is not affected by Garbage Collector runs, which can improve overall JVM performance. However, there are some trade-offs with off-heap storage; for example, JVM heap dumps do not show entries stored in off-heap memory.

Consider the following simplified illustration of memory space for a JVM process where Data Grid is running:
JVM heap memory

The heap is divided into young and old generations that help keep referenced Java objects and other application data in memory. The Garbage Collector (GC) process reclaims space from unreachable objects, running more frequently on the young generation memory pool.

When Data Grid stores cache entries in JVM heap memory, GC runs can take longer to complete as you start adding data to your caches. Because GC is a fairly intensive process, longer and more frequent runs can degrade application performance.

Off-heap memory

Off-heap memory is native available system memory outside JVM memory management. The preceding diagram shows the Metaspace memory pool that holds class metadata and is allocated from native memory. The diagram also represents a section of native memory that holds Data Grid cache entries.

Storing data off-heap

When you add entries to off-heap caches, Data Grid dynamically allocates native memory to your data.

Data Grid hashes the serialized byte[] for each key into buckets that are similar to a standard Java HashMap. Buckets include address pointers that Data Grid use to locate entries that you store in off-heap memory.

NOTE

Data Grid determines equality of Java objects in off-heap storage using the serialized byte[] representation of each object instead of the object instance.

The following diagram shows a set of keys with names, the hash for each key and bucket array of address pointers, as well as the entries with the name and phone number:
In cases where key hashes collide, Data Grid links entries. As in the preceding diagram, if the William Clay and Luke Cage keys have the same hash, then the first entry added to the cache is the first element in the bucket.

NOTE

Even though Data Grid stores cache entries in native memory, run-time operations require JVM heap representations of those objects. For instance, `cache.get()` operations read objects into heap memory before returning. Likewise, state transfer operations hold subsets of objects in heap memory while they take place.

Memory overhead

Memory overhead is the additional memory that Data Grid uses to store entries. For off-heap storage, Data Grid uses 25 bytes for each entry in the cache.

When you use eviction to create a bounded off-heap data container, memory overhead increases to a total of 61 bytes. This occurs because Data Grid creates an additional linked list to track entries in the cache and perform eviction.

Data consistency

Data Grid uses an array of locks to protect off-heap address spaces. The number of locks is twice the number of cores and then rounded to the nearest power of two. This ensures that there is an even distribution of `ReadWriteLock` instances to prevent write operations from blocking read operations.

6.2.1. Using Off-Heap Memory

Configure Data Grid to store cache entries in native memory outside the JVM heap space.

Procedure

1. Create a configuration for any type of cache.
2. Store cache entries in off-heap memory.
   - Declaratively: Add the `storage="OFF_HEAP"` attribute to the `memory` element.
   - Programmatically: Call the `storage(OFF_HEAP)` method in the `MemoryConfigurationBuilder` class.
3. Configure eviction to limit the amount of off-heap memory that the cache can use. You can use eviction based on total number of entries (`max-count`) or maximum amount of memory (`max-size`).
4. Configure any binary encoding for cache entries. For optimal results, use the `application/x-protobuf` MediaType to store entries in Protobuf format.
5. Create caches that use the configuration.

Declarative cache configuration

```xml
<distributed-cache name="off_heap" mode="SYNC">
  <encoding media-type="application/x-protobuf"/>
  <memory storage="OFF_HEAP" max-size="1.5GB" when-full="REMOVE"/>
</distributed-cache>
```
Programmatic cache configuration

```java
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg
  .encoding()
  .mediaType("application/x-protostream")
  .memory()
  .storage(StorageType.OFF_HEAP)
  .maxCount(500)
  .whenFull(EvictionStrategy.REMOVE)
  .build();
```

- Data Grid Configuration Schema Reference
- org.infinispan.configuration.cache.MemoryConfigurationBuilder
CHAPTER 7. CONFIGURING STATISTICS, METRICS, AND JMX

Enable statistics that Data Grid exports to a metrics endpoint or via JMX MBeans. You can also register JMX MBeans to perform management operations.

7.1. ENABLING DATA GRID STATISTICS

Data Grid lets you enable statistics for Cache Managers and caches. However, enabling statistics for a Cache Manager does not enable statistics for the caches that it controls. You must explicitly enable statistics for your caches.

**NOTE**

Data Grid server enables statistics for Cache Managers by default.

**Procedure**

- Enable statistics declaratively or programmatically.

**Declaratively**

```xml
<!-- Enables statistics for the Cache Manager. -->
<cache-container statistics="true"/>

<!-- Enables statistics for the named cache. -->
<local-cache name="mycache" statistics="true"/>
</cache-container>
```

**Programmatically**

```java
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
    .cacheContainer().statistics(true)
    .build();

Configuration config = new ConfigurationBuilder()
    .statistics().enable()
    .build();
```

7.2. ENABLING DATA GRID METRICS

Configure Data Grid to export gauges and histograms.

**Procedure**

- Configure metrics declaratively or programmatically.

**Declaratively**

```xml
<!-- Computes and collects statistics for the Cache Manager. -->
<cache-container statistics="true"/>

<!-- Exports collected statistics as gauge and histogram metrics. -->
</cache-container>
```
Data Grid metrics are provided at the vendor scope. Metrics related to the JVM are provided in the base scope for Data Grid server.

- Gauges provide values such as the average number of nanoseconds for write operations or JVM uptime. Gauges are enabled by default. If you enable statistics, Data Grid automatically generates gauges.

- Histograms provide details about operation execution times such as read, write, and remove times. Data Grid does not enable histograms by default because they require additional computation.

**Reference**

- Eclipse MicroProfile Metrics

## 7.3. CONFIGURING DATA GRID TO REGISTER JMX MBEANS

Data Grid can register JMX MBeans that you can use to collect statistics and perform administrative operations. However, you must enable statistics separately to JMX otherwise Data Grid provides 0 values for all statistic attributes.

**Procedure**

- Enable JMX declaratively or programmatically to register Data Grid JMX MBeans.

### Declaratively

```
<cache-container>
  <jmx enabled="true" />
</cache-container>
```

### Programatically

```java
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .jmx().enable()
  .build();
```
7.3.1. Naming Multiple Cache Managers

In cases where multiple Data Grid Cache Managers run on the same JVM, you should uniquely identify each Cache Manager to prevent conflicts.

Procedure

- Uniquely identify each cache manager in your environment.

For example, the following examples specify "Hibernate2LC" as the cache manager name, which results in a JMX MBean named `org.infinispan:type=CacheManager,name="Hibernate2LC"`.

Declaratively

```xml
<cache-container name="Hibernate2LC">
   <jmx enabled="true" />
   ...
</cache-container>
```

Programmatically

```java
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
    .cacheManagerName("Hibernate2LC")
    .jmx().enable()
    .build();
```

Reference

- GlobalConfigurationBuilder
- Data Grid Configuration Schema

7.3.2. Registering MBeans In Custom MBean Servers

Data Grid includes an MBeanServerLookup interface that you can use to register MBeans in custom MBeanServer instances.

Procedure

1. Create an implementation of MBeanServerLookup so that the getMBeanServer() method returns the custom MBeanServer instance.

2. Configure Data Grid with the fully qualified name of your class, as in the following example:

Declaratively

```xml
<cache-container>
   <jmx enabled="true" mbean-server-lookup="com.acme.MyMBeanServerLookup" />
</cache-container>
```

Programmatically
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
    .jmx().enable().mBeanServerLookup(new com.acme.MyMBeanServerLookup())
    .build();

Reference

- Data Grid Configuration Schema
- MBeanServerLookup

7.3.3. Data Grid MBeans

Data Grid exposes JMX MBeans that represent manageable resources.

org.infinispan:type=Cache

Attributes and operations available for cache instances.

org.infinispan:type=CacheManager

Attributes and operations available for cache managers, including Data Grid cache and cluster health statistics.

For a complete list of available JMX MBeans along with descriptions and available operations and attributes, see the Data Grid JMX Components documentation.

Reference

Data Grid JMX Components
CHAPTER 8. SETTING UP PERSISTENT STORAGE

Data Grid can persist in-memory data to external storage, giving you additional capabilities to manage your data such as:

**Durability**
Adding cache stores allows you to persist data to non-volatile storage so it survives restarts.

**Write-through caching**
Configuring Data Grid as a caching layer in front of persistent storage simplifies data access for applications because Data Grid handles all interactions with the external storage.

**Data overflow**
Using eviction and passivation techniques ensures that Data Grid keeps only frequently used data in-memory and writes older entries to persistent storage.

8.1. DATA GRID CACHE STORES

Cache stores connect Data Grid to persistent data sources and implement the `NonBlockingStore` interface.

8.1.1. Configuring Cache Stores

Add cache stores to Data Grid caches in a chain either declaratively or programmatically. Cache read operations check each cache store in the configured order until they locate a valid non-null element of data. Write operations affect all cache stores except for those that you configure as read only.

**Procedure**

1. Use the `persistence` parameter to configure the persistence layer for caches.
2. Configure whether cache stores are local to the node or shared across the cluster. Use either the `shared` attribute declaratively or the `shared(false)` method programmatically.
3. Configure other cache stores properties as appropriate. Custom cache stores can also include `property` parameters.

**NOTE**
Configuring cache stores as shared or not shared (local only) determines which parameters you should set. In some cases, using the wrong combination of parameters in your cache store configuration can lead to data loss or performance issues.

For example, if the cache store is local to a node then it makes sense to fetch state and purge on startup. However, if the cache store is shared, then you should not fetch state or purge on startup.

**Local (non-shared) file store**

```xml
<persistence passivation="false">
  <file-store shared="false"
    preload="true"
    fetch-state="true"
```
CHAPTER 8. SETTING UP PERSISTENT STORAGE

Shared custom cache store

```xml
<local-cache name="myCustomStore">
  <persistence passivation="false">
    <!-- Specifies the fully qualified class name of a custom cache store. -->
    <store class="org.acme.CustomStore"
      fetch-state="false"
      preload="true"
      shared="false"
      purge="true"
      read-only="false"
      segmented="true">
      <write-behind modification-queue-size="123" />
    </store>
    <!-- Sets system properties for the custom cache store. -->
    <property name="myProp">${system.property}</property>
  </persistence>
</local-cache>
```

Single file store

```java
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
  .passivation(false)
  .addSingleFileStore()
  .preload(true)
  .shared(false)
  .fetchPersistentState(true)
  .ignoreModifications(false)
  .purgeOnStartup(true)
  .location(System.getProperty("java.io.tmpdir"))
  .async()
  .enabled(true)
```

Reference

- Data Grid Configuration Schema
- Data Grid Cache Store Implementations
- Creating Custom Cache Stores

8.1.2. Setting a Global Persistent Location for File-Based Cache Stores

Data Grid uses a global filesystem location for saving data to persistent storage.
The global persistent location must be unique to each Data Grid instance. To share data between multiple instances, use a shared persistent location.

Data Grid servers use the `$RHDG_HOME/server/data` directory as the global persistent location.

If you are using Data Grid as a library embedded in custom applications and global-state is enabled, the global persistent location defaults to the `user.dir` system property. This system property typically uses the directory where your application starts. You should configure a global persistent location to use a suitable location.

Declarative configuration

```xml
<cache-container default-cache="myCache">
  <global-state>
    <persistent-location path="example" relative-to="my.data"/>
  </global-state>
  ...
</cache-container>

new GlobalConfigurationBuilder().globalState().enable().persistentLocation("example", "my.data");
```

File-Based Cache Stores and Global Persistent Location

When using file-based cache stores, you can optionally specify filesystem directories for storage. Unless absolute paths are declared, directories are always relative to the global persistent location.

For example, you configure your global persistent location as follows:

```xml
<global-state>
  <persistent-location path="/tmp/example" relative-to="my.data"/>
</global-state>
```

You then configure a Single File cache store that uses a path named `myDataStore` as follows:

```xml
<file-store path="myDataStore"/>
```

In this case, the configuration results in a Single File cache store in

```
/tmp/example/myDataStore/myCache.dat
```

If you attempt to set an absolute path that resides outside the global persistent location and global-state is enabled, Data Grid throws the following exception:

```
ISPN000558: "The store location 'foo' is not a child of the global persistent location 'bar'"
```

Reference

- Data Grid configuration schema
- `org.infinispan.configuration.global.GlobalStateConfiguration`

8.1.3. Passivation
Passivation configures Data Grid to write entries to cache stores when it evicts those entries from memory. In this way, passivation ensures that only a single copy of an entry is maintained, either in-memory or in a cache store, which prevents unnecessary and potentially expensive writes to persistent storage.

Activation is the process of restoring entries to memory from the cache store when there is an attempt to access passivated entries. For this reason, when you enable passivation, you must configure cache stores that implement both `CacheWriter` and `CacheLoader` interfaces so they can write and load entries from persistent storage.

When Data Grid evicts an entry from the cache, it notifies cache listeners that the entry is passivated then stores the entry in the cache store. When Data Grid gets an access request for an evicted entry, it lazily loads the entry from the cache store into memory and then notifies cache listeners that the entry is activated.

**NOTE**
- Passivation uses the first cache loader in the Data Grid configuration and ignores all others.
- Passivation is not supported with:
  - Transactional stores. Passivation writes and removes entries from the store outside the scope of the actual Data Grid commit boundaries.
  - Shared stores. Shared cache stores require entries to always exist in the store for other owners. For this reason, passivation is not supported because entries cannot be removed.

If you enable passivation with transactional stores or shared stores, Data Grid throws an exception.

### 8.1.3.1. Passivation and Cache Stores

**Passivation disabled**

 Writes to data in memory result in writes to persistent storage.

If Data Grid evicts data from memory, then data in persistent storage includes entries that are evicted from memory. In this way persistent storage is a superset of the in-memory cache.

If you do not configure eviction, then data in persistent storage provides a copy of data in memory.

**Passivation enabled**

 Data Grid adds data to persistent storage only when it evicts data from memory.

When Data Grid activates entries, it restores data in memory and deletes data from persistent storage. In this way, data in memory and data in persistent storage form separate subsets of the entire data set, with no intersection between the two.
Entries in persistent storage can become stale when using shared cache stores. This occurs because Data Grid does not delete passivated entries from shared cache stores when they are activated.

Values are updated in memory but previously passivated entries remain in persistent storage with out of date values.

The following table shows data in memory and in persistent storage after a series of operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Passivation disabled</th>
<th>Passivation enabled</th>
<th>Passivation enabled with shared cache store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert k1.</td>
<td>Memory: k1</td>
<td>Memory: k1</td>
<td>Memory: k1</td>
</tr>
<tr>
<td></td>
<td>Disk: k1</td>
<td>Disk: -</td>
<td>Disk: -</td>
</tr>
<tr>
<td>Insert k2.</td>
<td>Memory: k1, k2</td>
<td>Memory: k1, k2</td>
<td>Memory: k1, k2</td>
</tr>
<tr>
<td></td>
<td>Disk: k1, k2</td>
<td>Disk: -</td>
<td>Disk: -</td>
</tr>
<tr>
<td>Eviction thread runs and evicts k1.</td>
<td>Memory: k2</td>
<td>Memory: k2</td>
<td>Memory: k2</td>
</tr>
<tr>
<td></td>
<td>Disk: k1, k2</td>
<td>Disk: k1</td>
<td>Disk: k1</td>
</tr>
<tr>
<td>Read k1.</td>
<td>Memory: k1, k2</td>
<td>Memory: k1, k2</td>
<td>Memory: k1, k2</td>
</tr>
<tr>
<td></td>
<td>Disk: k1, k2</td>
<td>Disk: -</td>
<td>Disk: -</td>
</tr>
<tr>
<td>Eviction thread runs and evicts k2.</td>
<td>Memory: k1</td>
<td>Memory: k1</td>
<td>Memory: k1</td>
</tr>
<tr>
<td></td>
<td>Disk: k1, k2</td>
<td>Disk: k2</td>
<td>Disk: k1</td>
</tr>
<tr>
<td>Remove k2.</td>
<td>Memory: k1</td>
<td>Memory: k1</td>
<td>Memory: k1</td>
</tr>
<tr>
<td></td>
<td>Disk: k1</td>
<td>Disk: -</td>
<td>Disk: -</td>
</tr>
</tbody>
</table>

8.1.4. Cache Loaders and Transactional Caches

Only JDBC String-Based cache stores support transactional operations. If you configure caches as transactional, you should set `transactional=true` to keep data in persistent storage synchronized with data in memory.

For all other cache stores, Data Grid does not enlist cache loaders in transactional operations. This can result in data inconsistency if transactions succeed in modifying data in memory but do not completely apply changes to data in the cache store. In this case manual recovery does not work with cache stores.

Reference

- JDBC String-Based Cache Stores

8.1.5. Segmented Cache Stores

Cache stores can organize data into hash space segments to which keys map.

Segmented stores increase read performance for bulk operations; for example, streaming over data (`Cache.size`, `Cache.entrySet.stream`), pre-loading the cache, and doing state transfer operations.
However, segmented stores can also result in loss of performance for write operations. This performance loss applies particularly to batch write operations that can take place with transactions or write-behind stores. For this reason, you should evaluate the overhead for write operations before you enable segmented stores. The performance gain for bulk read operations might not be acceptable if there is a significant performance loss for write operations.

**IMPORTANT**

The number of segments you configure for cache stores must match the number of segments you define in the Data Grid configuration with the `clustering.hash.numSegments` parameter.

If you change the `numSegments` parameter in the configuration after you add a segmented cache store, Data Grid cannot read data from that cache store.

**Reference**

Key Ownership

### 8.1.6. Filesystem-Based Cache Stores

In most cases, filesystem-based cache stores are appropriate for local cache stores for data that overflows from memory because it exceeds size and/or time restrictions.

**WARNING**

You should not use filesystem-based cache stores on shared file systems such as an NFS, Microsoft Windows, or Samba share. Shared file systems do not provide file locking capabilities, which can lead to data corruption.

Likewise, shared file systems are not transactional. If you attempt to use transactional caches with shared file systems, unrecoverable failures can happen when writing to files during the commit phase.

### 8.1.7. Write-Through

Write-Through is a cache writing mode where writes to memory and writes to cache stores are synchronous. When a client application updates a cache entry, in most cases by invoking `Cache.put()`, Data Grid does not return the call until it updates the cache store. This cache writing mode results in updates to the cache store concluding within the boundaries of the client thread.

The primary advantage of Write-Through mode is that the cache and cache store are updated simultaneously, which ensures that the cache store is always consistent with the cache.

However, Write-Through mode can potentially decrease performance because the need to access and update cache stores directly adds latency to cache operations.

Data Grid defaults to Write-Through mode unless you explicitly configure Write-Behind mode on cache stores.
Write-through configuration

```xml
<persistence passivation="false">
  <file-store fetch-state="true"
             read-only="false"
             purge="false" path="${java.io.tmpdir}"/>
</persistence>
```

Reference

Write-Behind

8.1.8. Write-Behind

Write-Behind is a cache writing mode where writes to memory are synchronous and writes to cache stores are asynchronous.

When clients send write requests, Data Grid adds those operations to a modification queue. Data Grid processes operations as they join the queue so that the calling thread is not blocked and the operation completes immediately.

If the number of write operations in the modification queue increases beyond the size of the queue, Data Grid adds those additional operations to the queue. However, those operations do not complete until Data Grid processes operations that are already in the queue.

For example, calling `Cache.putAsync` returns immediately and the Stage also completes immediately if the modification queue is not full. If the modification queue is full, or if Data Grid is currently processing a batch of write operations, then `Cache.putAsync` returns immediately and the Stage completes later.

Write-Behind mode provides a performance advantage over Write-Through mode because cache operations do not need to wait for updates to the underlying cache store to complete. However, data in the cache store remains inconsistent with data in the cache until the modification queue is processed. For this reason, Write-Behind mode is suitable for cache stores with low latency, such as unshared and local filesystem-based cache stores, where the time between the write to the cache and the write to the cache store is as small as possible.

Write-behind configuration

```xml
<persistence passivation="false">
  <file-store fetch-state="true"
             read-only="false"
             purge="false" path="${java.io.tmpdir}"/>
  <write-behind modification-queue-size="123"
                 fail-silently="true"/>
</file-store>
</persistence>
```

The preceding configuration example uses the `fail-silently` parameter to control what happens when either the cache store is unavailable or the modification queue is full.

- If `fail-silently="true"` then Data Grid logs WARN messages and rejects write operations.
- If `fail-silently="false"` then Data Grid throws exceptions if it detects the cache store is unavailable during a write operation. Likewise if the modification queue becomes full, Data Grid throws an exception.
In some cases, data loss can occur if Data Grid restarts and write operations exist in the modification queue. For example the cache store goes offline but, during the time it takes to detect that the cache store is unavailable, write operations are added to the modification queue because it is not full. If Data Grid restarts or otherwise becomes unavailable before the cache store comes back online, then the write operations in the modification queue are lost because they were not persisted.

Reference
Write-Through

8.2. CACHE STORE IMPLEMENTATIONS

Data Grid provides several cache store implementations that you can use. Alternatively you can provide custom cache stores.

8.2.1. Cluster Cache Loaders

ClusterCacheLoader retrieves data from other Data Grid cluster members but does not persist data. In other words, ClusterCacheLoader is not a cache store.

ClusterCacheLoader provides a non-blocking partial alternative to state transfer. ClusterCacheLoader fetches keys from other nodes on demand if those keys are not available on the local node, which is similar to lazily loading cache content.

The following points also apply to ClusterCacheLoader:

- Preloading does not take effect (preload=true).
- Fetching persistent state is not supported (fetch-state=true).
- Segmentation is not supported.

WARNING
The ClusterLoader has been deprecated and will be removed in a future release.

Declarative configuration

```xml
<persistence>
  <cluster-loader remote-timeout="500"/>
</persistence>
```

Programmatic configuration

```java
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()  
  .addClusterLoader()  
  .remoteCallTimeout(500);
```
8.2.2. Single File Cache Stores

Single File cache stores, `SingleFileStore`, persist data to file. Data Grid also maintains an in-memory index of keys while keys and values are stored in the file. By default, Single File cache stores are segmented, which means that Data Grid creates a separate file for each segment.

Because `SingleFileStore` keeps an in-memory index of keys and the location of values, it requires additional memory, depending on the key size and the number of keys. For this reason, `SingleFileStore` is not recommended for use cases where the keys have a large size.

In some cases, `SingleFileStore` can also become fragmented. If the size of values continually increases, available space in the single file is not used but the entry is appended to the end of the file. Available space in the file is used only if an entry can fit within it. Likewise, if you remove all entries from memory, the single file store does not decrease in size or become defragmented.

### Declarative configuration

```
<persistence>
  <file-store max-entries="5000"/>
</persistence>
```

### Programmatic configuration

- For embedded deployments, do the following:

```java
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
  .addSingleFileStore()
  .maxEntries(5000);
```

- For server deployments, do the following:

```java
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.configuration.NearCacheMode;
...

ConfigurationBuilder builder = new ConfigurationBuilder();
builder
  .remoteCache("mycache")
  .configuration("<distributed-cache name="mycache">" +
                 <persistence><file-store max-entries="5000"/>" +
                 </persistence>")
```

### Segmentation

Single File cache stores support segmentation and create a separate instance per segment, which results in multiple directories in the path you configure. Each directory is a number that represents the segment to which the data maps.
8.2.3. JDBC String-Based Cache Stores

JDBC String-Based cache stores, JdbcStringBasedStore, use JDBC drivers to load and store values in the underlying database.

JdbcStringBasedStore stores each entry in its own row in the table to increase throughput for concurrent loads. JdbcStringBasedStore also uses a simple one-to-one mapping that maps each key to a String object using the key-to-string-mapper interface.

Data Grid provides a default implementation, DefaultTwoWayKey2StringMapper, that handles primitive types.

In addition to the data table used to store cache entries, the store also creates a _META table for storing metadata. This table is used to ensure that any existing database content is compatible with the current Data Grid version and configuration.

NOTE

By default Data Grid shares are not stored, which means that all nodes in the cluster write to the underlying store on each update. If you want operations to write to the underlying database once only, you must configure JDBC store as shared.

Segmentation

JdbcStringBasedStore uses segmentation by default and requires a column in the database table to represent the segments to which entries belong.

8.2.3.1. Connection Factories

JdbcStringBasedStore relies on a ConnectionFactory implementation to connection to a database.

Data Grid provides the following ConnectionFactory implementations:

PooledConnectionFactoryConfigurationBuilder

A connection factory based on Agroal that you configure via PooledConnectionFactoryConfiguration.

Alternatively, you can specify configuration properties prefixed with org.infinispan.agroal. as in the following example:

org.infinispan.agroal.metricsEnabled=false
org.infinispan.agroal.minSize=10
org.infinispan.agroal.maxSize=100
org.infinispan.agroal.initialSize=20
org.infinispan.agroal.acquisitionTimeout_s=1
org.infinispan.agroal.validationTimeout_m=1
org.infinispan.agroal.leakTimeout_s=10
You then configure Data Grid to use your properties file via
PooledConnectionFactoryConfiguration.propertyFile.

NOTE
You should use PooledConnectionFactory with standalone deployments, rather than deployments in servlet containers.

ManagedConnectionFactoryConfigurationBuilder
A connection factory that you can use with managed environments such as application servers. This connection factory can explore a configurable location in the JNDI tree and delegate connection management to the DataSource.

SimpleConnectionFactoryConfigurationBuilder
A connection factory that creates database connections on a per invocation basis. You should use this connection factory for test or development environments only.

Reference
- Agroal
- ConnectionFactoryConfigurationBuilder
- PooledConnectionFactoryConfigurationBuilder
- ManagedConnectionFactoryConfigurationBuilder
- SimpleConnectionFactoryConfigurationBuilder

8.2.3.2. JDBC String-Based Cache Store Configuration
You can configure JdbcStringBasedStore programmatically or declaratively.

Declarative configuration
- Using PooledConnectionFactory

```xml
<persistence>
  <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:12.1" shared="true">
    <connection-pool connection-url="jdbc:h2:mem:PooledConnectionFactoryTest;DB_CLOSE_DELAY=-1" username="sa" driver="org.h2.Driver"/>
    <string-keyed-table drop-on-exit="true"/>
  </string-keyed-jdbc-store>
</persistence>
```

org.infinispan.agroal.reapTimeout_m=10
org.infinispan.agroal.metricsEnabled=false
org.infinispan.agroal.autoCommit=true
org.infinispan.agroal.jdbcTransactionIsolation=READ_COMMITTED
org.infinispan.agroal.jdbcUrl=jdbc:h2:mem:PooledConnectionFactoryTest;DB_CLOSE_DELAY=-1
org.infinispan.agroal.driverClassName=org.h2.Driver.class
org.infinispan.agroal.principal=sa
org.infinispan.agroal.credential=sa
Using ManagedConnectionFactory

```xml
<persistence>
  <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:12.1" shared="true">
    <data-source jndi-url="java:/StringStoreWithManagedConnectionTest/DS" />
    <string-keyed-table drop-on-exit="true" create-on-start="true" prefix="ISPN_STRING_TABLE">
      <id-column name="ID_COLUMN" type="VARCHAR(255)" />
      <data-column name="DATA_COLUMN" type="BINARY" />
      <timestamp-column name="TIMESTAMP_COLUMN" type="BIGINT" />
      <segment-column name="SEGMENT_COLUMN" type="INT" />
    </string-keyed-table>
  </string-keyed-jdbc-store>
</persistence>

• Using ManagedConnectionFactory

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
  .fetchPersistentState(false)
  .ignoreModifications(false)
  .purgeOnStartup(false)
  .shared(true)
  .table()
    .dropOnExit(true)
    .createOnStart(true)
    .tableNamePrefix("ISPN_STRING_TABLE")
    .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)"")
    .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
    .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
    .segmentColumnName("SEGMENT_COLUMN").segmentColumnType("INT")
  .connectionPool()
    .connectionUrl("jdbc:h2:mem:infinispan_string_based;DB_CLOSE_DELAY=-1")
    .username("sa")
    .driverClass("org.h2.Driver");
```

Programmatic configuration

• Using PooledConnectionFactory

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
  .fetchPersistentState(false)
  .ignoreModifications(false)

• Using ManagedConnectionFactory

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
  .fetchPersistentState(false)
  .ignoreModifications(false)
8.2.4. JPA Cache Stores

JPA (Java Persistence API) cache stores, JpaStore, use formal schema to persist data. Other applications can then read from persistent storage to load data from Data Grid. However, other applications should not use persistent storage concurrently with Data Grid.

When using JpaStore, you should take the following into consideration:

- Keys should be the ID of the entity. Values should be the entity object.
- Only a single @Id or @EmbeddedId annotation is allowed.
- Auto-generated IDs with the @GeneratedValue annotation are not supported.
- All entries are stored as immortal.
- JpaStore does not support segmentation.

Declarative configuration

```xml
<local-cache name="vehicleCache">
  <persistence passivation="false">
    <jpa-store xmlns="urn:infinispan:config:store:jpa:12.1"
               persistence-unit="org.infinispan.persistence.jpa.configurationTest"
               entity-class="org.infinispan.persistence.jpa.entity.Vehicle">
      .purgeOnStartup(false)
      .shared(true)
      .table()
      .dropOnExit(true)
      .createOnStart(true)
      .tableNamePrefix("ISPN_STRING_TABLE")
      .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)"
      .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
      .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
      .segmentColumnName("SEGMENT_COLUMN").segmentColumnType("INT")
      .dataSource()
      .jndiUrl("java:/StringStoreWithManagedConnectionTest/DS");
    </jpa-store>
  </persistence>
</local-cache>
```

Reference

- JDBC cache store configuration schema
- JdbcStringBasedStore
- JdbcStringBasedStoreConfiguration
Persistence unit

Specifies the JPA persistence unit name in the JPA configuration file, `persistence.xml`, that contains the JPA entity class.

Entity class

Specifies the fully qualified JPA entity class name that is expected to be stored in this cache. Only one class is allowed.

Programmatic configuration

```java
Configuration cacheConfig = new ConfigurationBuilder().persistence()
    .addStore(JpaStoreConfigurationBuilder.class)
    .persistenceUnitName("org.infinispan.loaders.jpa.configurationTest")
    .entityClass(User.class)
    .build();
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>persistence-unit</td>
<td>Specifies the JPA persistence unit name in the JPA configuration file, <code>persistence.xml</code>, that contains the JPA entity class.</td>
</tr>
<tr>
<td>entity-class</td>
<td>Specifies the fully qualified JPA entity class name that is expected to be stored in this cache. Only one class is allowed.</td>
</tr>
</tbody>
</table>

Reference

- JPA cache store configuration schema
- JpaStore
- JpaStoreConfiguration

8.2.4.1. JPA Cache Store Usage Example

This section provides an example for using JPA cache stores.

Prerequisites

- Configure Data Grid to marshall your JPA entities. By default, Data Grid uses ProtoStream for marshalling Java objects. To marshall JPA entities, you must create a `SerializationContextInitializer` implementation that registers a `.proto` schema and marshaller with a `SerializationContext`.

Procedure

1. Define a persistence unit "myPersistenceUnit" in `persistence.xml`. 

2. Create a user entity class.

```java
@Entity
public class User implements Serializable {
    @Id
    private String username;
    private String firstName;
    private String lastName;
    ...
}
```

3. Configure a cache named "usersCache" with a JPA cache store. Then you can configure a cache "usersCache" to use JPA Cache Store, so that when you put data into the cache, the data would be persisted into the database based on JPA configuration.

```java
EmbeddedCacheManager cacheManager = ...;
Configuration cacheConfig = new ConfigurationBuilder().persistence()
    .addStore(JpaStoreConfigurationBuilder.class)
    .entityClass(User.class)
    .build();

cacheManager.defineCache("usersCache", cacheConfig);

Cache<String, User> usersCache = cacheManager.getCache("usersCache");
usersCache.put("raytsang", new User(...));
```

- Caches that use a JPA cache store can store one type of data only, as in the following example:

```java
Cache<String, User> usersCache = cacheManager.getCache("myJPACache");
// Cache is configured for the User entity class
usersCache.put("username", new User());
// Cannot configure caches to use another entity class with JPA cache stores
Cache<Integer, Teacher> teachersCache = cacheManager.getCache("myJPACache");
teachersCache.put(1, new Teacher());
// The put request does not work for the Teacher entity class
```

- The `@Embeddable` annotation allows you to use composite keys, as in the following example:

```java
@Embeddable
public class Vehicle implements Serializable {
    @EmbeddableId
    private VehicleId id;
    private String color; ...
}
```
8.2.5. Remote Cache Stores

Remote cache stores, RemoteStore, use the Hot Rod protocol to store data on Data Grid clusters.

The following is an example RemoteStore configuration that stores data in a cache named "mycache" on two Data Grid Server instances, named "one" and "two":

```
<remote-store xmlns="urn:infinispan:config:store:remote:12.1" cache="mycache" raw-values="true">
  <remote-server host="one" port="12111" />
  <remote-server host="two" />  
  <connection-pool max-active="10" exhausted-action="CREATE_NEW" />
  <write-behind />
</remote-store>
```

Additional resources

- Cache Encoding and Marshalling

**NOTE**

If you configure remote cache stores as shared you cannot preload data. In other words if shared="true" in your configuration then you must set preload="false".

Declarative configuration

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence().addStore(RemoteStoreConfigurationBuilder.class)
  .fetchPersistentState(false)
  .ignoreModifications(false)
  .purgeOnStartup(false)
  .remoteCacheName("mycache")
  .rawValues(true)
  .addServer()
    .host("one").port(12111)
  .addServer()
    .host("two")
  .connectionPool()
    .maxActive(10)
  .exhaustedAction(ExhaustedAction.CREATE_NEW)
  .async().enable();
```

Programmatic configuration
Segmentation

**RemoteStore** supports segmentation and can publish keys and entries by segment, which makes bulk operations more efficient. However, segmentation is available only with Data Grid Hot Rod protocol version 2.3 or later.

![WARNING]

When you enable segmentation for **RemoteStore**, it uses the number of segments that you define in your Data Grid server configuration.

If the source cache is segmented and uses a different number of segments than **RemoteStore**, then incorrect values are returned for bulk operations. In this case, you should disable segmentation for **RemoteStore**.

Reference

- Remote cache store configuration schema
- **RemoteStore**
- **RemoteStoreConfigurationBuilder**

8.2.6. RocksDB Cache Stores

RocksDB provides key-value filesystem-based storage with high performance and reliability for highly concurrent environments.

RocksDB cache stores, **RocksDBStore**, use two databases. One database provides a primary cache store for data in memory; the other database holds entries that Data Grid expires from memory.

Declarative configuration

```xml
<local-cache name="vehicleCache">
  <persistence>
    <rocksdb-store xmlns="urn:infinispan:config:store:rocksdb:12.1" path="rocksdb/data">
      <expiration path="rocksdb/expired"/>
    </rocksdb-store>
  </persistence>
</local-cache>
```

Programmatic configuration

```java
Configuration cacheConfig = new ConfigurationBuilder().persistence()
  .addStore(RocksDBStoreConfigurationBuilder.class)
  .build();
EmbeddedCacheManager cacheManager = new DefaultCacheManager(cacheConfig);

Cache<String, User> usersCache = cacheManager.getCache("usersCache");
usersCache.put("raytsang", new User(...));
```
Properties props = new Properties();
props.put("database.max_background_compactions", "2");
props.put("data.write_buffer_size", "512MB");

Configuration cacheConfig = new ConfigurationBuilder().persistence()
    .addStore(RocksDBStoreConfigurationBuilder.class)
    .location("rocksdb/data")
    .expiredLocation("rocksdb/expired")
    .properties(props)
    .build();

Table 8.1. RocksDBStore configuration parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>Specifies the path to the RocksDB database that provides the primary cache store. If you do not set the location, it is automatically created. Note that the path must be relative to the global persistent location.</td>
</tr>
<tr>
<td>expiredLocation</td>
<td>Specifies the path to the RocksDB database that provides the cache store for expired data. If you do not set the location, it is automatically created. Note that the path must be relative to the global persistent location.</td>
</tr>
<tr>
<td>expiryQueueSize</td>
<td>Sets the size of the in-memory queue for expiring entries. When the queue reaches the size, Data Grid flushes the expired into the RocksDB cache store.</td>
</tr>
<tr>
<td>clearThreshold</td>
<td>Sets the maximum number of entries before deleting and re-initializing (re-init) the RocksDB database. For smaller size cache stores, iterating through all entries and removing each one individually can provide a faster method.</td>
</tr>
</tbody>
</table>

RocksDB tuning parameters

You can also specify the following RocksDB tuning parameters:

- compressionType
- blockSize
- cacheSize

RocksDB configuration properties

Optionally set properties in the configuration as follows:

- Prefix properties with database to adjust and tune RocksDB databases.
- Prefix properties with data to configure the column families in which RocksDB stores your data.
Segmentation

RocksDBStore supports segmentation and creates a separate column family per segment. Segmented RocksDB cache stores improve lookup performance and iteration but slightly lower performance of write operations.

**NOTE**

You should not configure more than a few hundred segments. RocksDB is not designed to have an unlimited number of column families. Too many segments also significantly increases cache store start time.

Reference

- RocksDB cache store configuration schema
- RocksDBStore
- RocksDBStoreConfiguration
- rocksdb.org
- RocksDB Tuning Guide

8.2.7. Soft-Index File Stores

Soft-Index File cache stores, SoftIndexFileStore, provide local file-based storage.

SoftIndexFileStore is a Java implementation that uses a variant of B+ Tree that is cached in-memory using Java soft references. The B+ Tree, called Index is offloaded on the file system to a single file that is purged and rebuilt each time the cache store restarts.

SoftIndexFileStore stores data in a set of files rather than a single file. When usage of any file drops below 50%, the entries in the file are overwriten to another file and the file is then deleted.

SoftIndexFileStore persists data in a set of files that are written in an append-only method. For this reason, if you use SoftIndexFileStore on conventional magnetic disk, it does not need to seek when writing a burst of entries.

Most structures in SoftIndexFileStore are bounded, so out-of-memory exceptions do not pose a risk. You can also configure limits for concurrently open files.

By default the size of a node in the Index is limited to 4096 bytes. This size also limits the key length; more precisely the length of serialized keys. For this reason, you cannot use keys longer than the size of the node, 15 bytes. Additionally, key length is stored as "short", which limits key length to 32767 bytes. SoftIndexFileStore throws an exception if keys are longer after serialization occurs.

SoftIndexFileStore cannot detect expired entries, which can lead to excessive usage of space on the file system.
NOTE

AdvancedStore.purgeExpired() is not implemented in SoftIndexFileStore.

Declarative configuration

```xml
<persistence>
  <soft-index-file-store xmlns="urn:infinispan:config:store:soft-index:12.1">
    <index path="testCache/index" />
    <data path="testCache/data" />
  </soft-index-file-store>
</persistence>
```

Programmatic configuration

```java
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
  .addStore(SoftIndexFileStoreConfigurationBuilder.class)
  .indexLocation("testCache/index");
  .dataLocation("testCache/data")
```

Segmentation

Soft-Index File cache stores support segmentation and create a separate instance per segment, which results in multiple directories in the path you configure. Each directory is a number that represents the segment to which the data maps.

Reference

- Soft-Index File cache store configuration schema
- SoftIndexFileStore
- SoftIndexFileStoreConfiguration

8.2.8. Implementing Custom Cache Stores

You can create custom cache stores through the Data Grid persistent SPI.

8.2.8.1. Data Grid Persistence SPI

The Data Grid Service Provider Interface (SPI) enables read and write operations to external storage through the NonBlockingStore interface and has the following features:

- **Portability across JCache-compliant vendors**
  Data Grid maintains compatibility between the NonBlockingStore interface and the JSR-107 JCache specification by using an adapter that handles blocking code.

- **Simplified transaction integration**
  Data Grid automatically handles locking so your implementations do not need to coordinate concurrent access to persistent stores. Depending on the locking mode you use, concurrent writes to the same key generally do not occur. However, you should expect operations on the persistent storage to originate from multiple threads and create implementations to tolerate this behavior.

- **Parallel iteration**
Data Grid lets you iterate over entries in persistent stores with multiple threads in parallel.

**Reduced serialization resulting in less CPU usage**

Data Grid exposes stored entries in a serialized format that can be transmitted remotely. For this reason, Data Grid does not need to deserialize entries that it retrieves from persistent storage and then serialize again when writing to the wire.

**Reference**

- Persistence SPI
- NonBlockingStore
- JSR-107

**8.2.8.2. Creating Cache Stores**

Create custom cache stores by implementing the NonBlockingStore interface.

1. Implement the appropriate Data Grid persistent SPIs.
2. Annotate your store class with the @ConfiguredBy annotation if it has a custom configuration.
3. Create a custom cache store configuration and builder if desired.
   a. Extend AbstractStoreConfiguration and AbstractStoreConfigurationBuilder.
   b. Optionally add the following annotations to your store Configuration class to ensure that your custom configuration builder parses your cache store configuration from XML:
      - @ConfigurationFor
      - @BuiltBy
      
      If you do not add these annotations, then CustomStoreConfigurationBuilder parses the common store attributes defined in AbstractStoreConfiguration and any additional elements are ignored.

**NOTE**

If a configuration does not declare the @ConfigurationFor annotation, a warning message is logged when Data Grid initializes the cache.

**8.2.8.3. Configuring Data Grid to Use Custom Stores**

After you create your custom cache store implementation, configure Data Grid to use it.

**Declarative configuration**

```
<local-cache name="customStoreExample">
  <persistence>
    <store class="org.infinispan.persistence.dummy.DummyInMemoryStore"/>
  </persistence>
</local-cache>
```

**Programmatic configuration**

-
8.2.8.4. Deploying Custom Cache Stores

You can package custom cache stores into JAR files and deploy them to Data Grid servers as follows:

1. Package your custom cache store implementation in a JAR file.
2. Add your JAR file to the `server/lib` directory of your Data Grid server.

8.3. MIGRATING BETWEEN CACHE STORES

Data Grid provides a utility to migrate data from one cache store to another.

8.3.1. Cache Store Migrator

Data Grid provides the `StoreMigrator.java` utility that recreates data for the latest Data Grid cache store implementations.

`StoreMigrator` takes a cache store from a previous version of Data Grid as source and uses a cache store implementation as target.

When you run `StoreMigrator`, it creates the target cache with the cache store type that you define using the `EmbeddedCacheManager` interface. `StoreMigrator` then loads entries from the source store into memory and then puts them into the target cache.

`StoreMigrator` also lets you migrate data from one type of cache store to another. For example, you can migrate from a JDBC String-Based cache store to a Single File cache store.

**IMPORTANT**

`StoreMigrator` cannot migrate data from segmented cache stores to:

- Non-segmented cache store.
- Segmented cache stores that have a different number of segments.

8.3.2. Getting the Store Migrator

`StoreMigrator` is available as part of the Data Grid tools library, `infinispan-tools`, and is included in the Maven repository.

**Procedure**

- Configure your `pom.xml` for `StoreMigrator` as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
```
8.3.3. Configuring the Store Migrator

Set properties for source and target cache stores in a `migrator.properties` file.

Procedure

1. Create a `migrator.properties` file.

2. Configure the source cache store in `migrator.properties`.
   a. Prepend all configuration properties with `source`. as in the following example:

```
source.type=SOFT_INDEX_FILE_STORE
source.cache_name=myCache
source.location=/path/to/source/sifs
```

3. Configure the target cache store in `migrator.properties`.
Prepend all configuration properties with `target`, as in the following example:

```plaintext
target.type=SINGLE_FILE_STORE
target.cache_name=myCache
target.location=/path/to/target/sfs.dat
```

8.3.3.1. Store Migrator Properties

Configure source and target cache stores in a `StoreMigrator` properties.

Table 8.2. Cache Store Type Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Specifies the type of cache store type for a source or target.</td>
<td>Required</td>
</tr>
<tr>
<td>.type=JDBC_STRING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.type=JDBC_BINARY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.type=JDBC_MIXED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.type=LEVELDB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.type=ROCKSDB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.type=SINGLE_FILE_STORE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.type=SOFT_INDEX_FILE_STORE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.type=JDBC_MIXED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.3. Common Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Example Value</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_name</td>
<td>Names the cache that the store backs.</td>
<td>.cache_name=myCache</td>
<td>Required</td>
</tr>
</tbody>
</table>
The number of segments must match `clustering.hash.num Segments` in the Data Grid configuration.

In other words, the number of segments for a cache store must match the number of segments for the corresponding cache. If the number of segments is not the same, Data Grid cannot read data from the cache store.

### Table 8.4. JDBC Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Example Value</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>segment_count</strong></td>
<td>Specifies the number of segments for target cache stores that can use segmentation.</td>
<td><code>.segment_count=256</code></td>
<td>Optional</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Example Value</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dialect</strong></td>
<td>Specifies the dialect of the underlying database.</td>
<td></td>
<td>Required</td>
</tr>
<tr>
<td><strong>version</strong></td>
<td>Specifies the marshaller version for source cache stores. Set one of the following values:</td>
<td><code>source.version=9</code></td>
<td>Required for source stores only. For example:</td>
</tr>
<tr>
<td></td>
<td>* 8 for Data Grid 7.2.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>* 9 for Data Grid 7.3.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>* 10 Data Grid 8.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>marshaller.class</strong></td>
<td>Specifies a custom marshaller class.</td>
<td></td>
<td>Required if using custom marshellers.</td>
</tr>
<tr>
<td><strong>marshaller.externalizers</strong></td>
<td>Specifies a comma-separated list of custom <code>AdvancedExternalizer</code> implementations to load in this format: <code>[id]:&lt;Externalizer class&gt;</code></td>
<td></td>
<td>Optional</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Required/Optional</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td>connection_pool.connection_url</td>
<td>Specifies the JDBC connection URL.</td>
<td>Required</td>
<td></td>
</tr>
<tr>
<td>connection_pool.driver_class</td>
<td>Specifies the class of the JDBC driver.</td>
<td>Required</td>
<td></td>
</tr>
<tr>
<td>connection_pool.username</td>
<td>Specifies a database username.</td>
<td>Required</td>
<td></td>
</tr>
<tr>
<td>connection_pool.password</td>
<td>Specifies a password for the database username.</td>
<td>Required</td>
<td></td>
</tr>
<tr>
<td>db.major_version</td>
<td>Sets the database major version.</td>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td>db.minor_version</td>
<td>Sets the database minor version.</td>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td>db.disable_upsert</td>
<td>Disables database upsert.</td>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td>db.disable_indexing</td>
<td>Specifies if table indexes are created.</td>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td>table.string.table_name_prefix</td>
<td>Specifies additional prefixes for the table name.</td>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td>table.string.&lt;id</td>
<td>data</td>
<td>timestamp&gt;.name</td>
<td>Specifies the column name.</td>
</tr>
<tr>
<td>table.string.&lt;id</td>
<td>data</td>
<td>timestamp&gt;.type</td>
<td>Specifies the column type.</td>
</tr>
<tr>
<td>key_to_string_mapper</td>
<td>Specifies the TwoWayKey2StringMapper class.</td>
<td>Optional</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE**

To migrate from Binary cache stores in older Data Grid versions, change `table.string.*` to `table.binary.*` in the following properties:

- `source.table.binary.table_name_prefix`
- `source.table.binary.<id\|data\|timestamp>.name`
- `source.table.binary.<id\|data\|timestamp>.type`

# Example configuration for migrating to a JDBC String-Based cache store

```
target.type=STRING
```
target.cache_name=myCache
target.dialect=POSTGRES
target.marshaller.class=org.example.CustomMarshaller
target.marshaller.externalizers=25:Externalizer1,org.example.Externalizer2
target.connection_pool.connection_url=jdbc:postgresql:postgres
target.connection_pool.driver_class=org.postgresql.Driver
target.connection_pool.username=postgres
target.connection_pool.password=redhat
target.db.major_version=9
target.db.minor_version=5
target.db.disable_upsert=false
target.db.disable_indexing=false
target.table.string.table_name_prefix=tablePrefix
target.table.string.id.name=id_column
target.table.string.data.name=datum_column
target.table.string.timestamp.name=timestamp_column
target.table.string.id.type=VARCHAR
target.table.string.data.type=bytea
target.table.string.timestamp.type=BIGINT
target.key_to_string_mapper=org.infinispan.persistence.keymappers.DefaultTwoWayKey2StringMapper

Table 8.5. RocksDB Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>Sets the database directory.</td>
<td>Required</td>
</tr>
<tr>
<td>compression</td>
<td>Specifies the compression type to use.</td>
<td>Optional</td>
</tr>
</tbody>
</table>

# Example configuration for migrating from a RocksDB cache store.
source.type=ROCKSDB
source.cache_name=myCache
source.location=/path/to/rocksdb/database
source.compression=SNAPPY

Table 8.6. SingleFileStore Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>Sets the directory that contains the cache store .dat file.</td>
<td>Required</td>
</tr>
</tbody>
</table>

# Example configuration for migrating to a Single File cache store.
target.type=SINGLE_FILE_STORE
target.cache_name=myCache
target.location=/path/to/sfs.dat

Table 8.7. SoftIndexFileStore Properties
### 8.3.4. Migrating Cache Stores

Run StoreMigrator to migrate data from one cache store to another.

**Prerequisites**

- Get `infinispan-tools.jar`.
- Create a `migrator.properties` file that configures the source and target cache stores.

**Procedure**

- If you build `infinispan-tools.jar` from source, do the following:
  1. Add `infinispan-tools.jar` and dependencies for your source and target databases, such as JDBC drivers, to your classpath.
  2. Specify `migrator.properties` file as an argument for StoreMigrator.
- If you pull `infinispan-tools.jar` from the Maven repository, run the following command: `mvn exec:java`
CHAPTER 9. SETTING UP PARTITION HANDLING

9.1. PARTITION HANDLING

An Data Grid cluster is built out of a number of nodes where data is stored. In order not to lose data in the presence of node failures, Data Grid copies the same data – cache entry in Data Grid parlance – over multiple nodes. This level of data redundancy is configured through the `numOwners` configuration attribute and ensures that as long as fewer than `numOwners` nodes crash simultaneously, Data Grid has a copy of the data available.

However, there might be catastrophic situations in which more than `numOwners` nodes disappear from the cluster:

**Split brain**

Caused e.g. by a router crash, this splits the cluster in two or more partitions, or sub-clusters that operate independently. In these circumstances, multiple clients reading/writing from different partitions see different versions of the same cache entry, which for many application is problematic. Note there are ways to alleviate the possibility for the split brain to happen, such as redundant networks or IP bonding. These only reduce the window of time for the problem to occur, though.

**numOwners nodes crash in sequence**

When at least `numOwners` nodes crash in rapid succession and Data Grid does not have the time to properly rebalance its state between crashes, the result is partial data loss.

The partition handling functionality discussed in this section allows the user to configure what operations can be performed on a cache in the event of a split brain occurring. Data Grid provides multiple partition handling strategies, which in terms of Brewer’s CAP theorem determine whether availability or consistency is sacrificed in the presence of partition(s). Below is a list of the provided strategies:

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
<th>CAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>DENY_READ_WRITES</td>
<td>If the partition does not have all owners for a given segment, both reads and writes are denied for all keys in that segment.</td>
<td>Consistency</td>
</tr>
<tr>
<td>ALLOW_READS</td>
<td>Allows reads for a given key if it exists in this partition, but only allows writes if this partition contains all owners of a segment. This is still a consistent approach because some entries are readable if available in this partition, but from a client application perspective it is not deterministic.</td>
<td>Consistency</td>
</tr>
<tr>
<td>ALLOW_READ_WRITES</td>
<td>Allow entries on each partition to diverge, with conflict resolution attempted upon the partitions merging.</td>
<td>Availability</td>
</tr>
</tbody>
</table>
The requirements of your application should determine which strategy is appropriate. For example, DENY_READ_WRITES is more appropriate for applications that have high consistency requirements; i.e. when the data read from the system must be accurate. Whereas if Data Grid is used as a best-effort cache, partitions maybe perfectly tolerable and the ALLOW_READ_WRITES might be more appropriate as it favours availability over consistency.

The following sections describe how Data Grid handles split brain and successive failures for each of the partition handling strategies. This is followed by a section describing how Data Grid allows for automatic conflict resolution upon partition merges via merge policies. Finally, we provide a section describing how to configure partition handling strategies and merge policies.

9.1. Split brain

In a split brain situation, each network partition will install its own JGroups view, removing the nodes from the other partition(s). We don’t have a direct way of determining whether the has been split into two or more partitions, since the partitions are unaware of each other. Instead, we assume the cluster has split when one or more nodes disappear from the JGroups cluster without sending an explicit leave message.

9.1.1. Split Strategies

In this section, we detail how each partition handling strategy behaves in the event of split brain occurring.

9.1.1.1. ALLOW_READ_WRITES

Each partition continues to function as an independent cluster, with all partitions remaining in AVAILABLE mode. This means that each partition may only see a part of the data, and each partition could write conflicting updates in the cache. During a partition merge these conflicts are automatically resolved by utilising the ConflictManager and the configured EntryMergePolicy.

9.1.1.2. DENY_READ_WRITES

When a split is detected each partition does not start a rebalance immediately, but first it checks whether it should enter DEGRADED mode instead:

- If at least one segment has lost all its owners (meaning at least numOwners nodes left since the last rebalance ended), the partition enters DEGRADED mode.

- If the partition does not contain a simple majority of the nodes (floor(numNodes/2) + 1) in the latest stable topology, the partition also enters DEGRADED mode.

- Otherwise the partition keeps functioning normally, and it starts a rebalance.

The stable topology is updated every time a rebalance operation ends and the coordinator determines that another rebalance is not necessary.

These rules ensures that at most one partition stays in AVAILABLE mode, and the other partitions enter DEGRADED mode.

When a partition is in DEGRADED mode, it only allows access to the keys that are wholly owned:

- Requests (reads and writes) for entries that have all the copies on nodes within this partition are honoured.
Requests for entries that are partially or totally owned by nodes that disappeared are rejected with an `AvailabilityException`.

This guarantees that partitions cannot write different values for the same key (cache is consistent), and also that one partition can not read keys that have been updated in the other partitions (no stale data).

To exemplify, consider the initial cluster $M = \{A, B, C, D\}$, configured in distributed mode with `numOwners = 2`. Further on, consider three keys $k_1, k_2$ and $k_3$ (that might exist in the cache or not) such that $\text{owners}(k_1) = \{A, B\}$, $\text{owners}(k_2) = \{B, C\}$ and $\text{owners}(k_3) = \{C, D\}$. Then the network splits in two partitions, $N_1 = \{A, B\}$ and $N_2 = \{C, D\}$, they enter DEGRADED mode and behave like this:

- on $N_1$, $k_1$ is available for read/write, $k_2$ (partially owned) and $k_3$ (not owned) are not available and accessing them results in an `AvailabilityException`
- on $N_2$, $k_1$ and $k_2$ are not available for read/write, $k_3$ is available

A relevant aspect of the partition handling process is the fact that when a split brain happens, the resulting partitions rely on the original segment mapping (the one that existed before the split brain) in order to calculate key ownership. So it doesn’t matter if $k_1$, $k_2$, or $k_3$ already existed cache or not, their availability is the same.

If at a further point in time the network heals and $N_1$ and $N_2$ partitions merge back together into the initial cluster $M$, then $M$ exits the degraded mode and becomes fully available again. During this merge operation, because $M$ has once again become fully available, the `ConflictManager` and the configured `EntryMergePolicy` are used to check for any conflicts that may have occurred in the interim period between the split brain occurring and being detected.

As another example, the cluster could split in two partitions $O_1 = \{A, B, C\}$ and $O_2 = \{D\}$, partition $O_1$ will stay fully available (rebalancing cache entries on the remaining members). Partition $O_2$, however, will detect a split and enter the degraded mode. Since it doesn’t have any fully owned keys, it will reject any read or write operation with an `AvailabilityException`.

If afterwards partitions $O_1$ and $O_2$ merge back into $M$, then the `ConflictManager` attempts to resolve any conflicts and $D$ once again becomes fully available.

### 9.1.1.3. ALLOW_READS

Partitions are handled in the same manner as `DENY_READ_WRITES`, except that when a partition is in DEGRADED mode read operations on a partially owned key WILL not throw an `AvailabilityException`.

### 9.1.2. Current limitations

Two partitions could start up isolated, and as long as they don’t merge they can read and write inconsistent data. In the future, we will allow custom availability strategies (e.g. check that a certain node is part of the cluster, or check that an external machine is accessible) that could handle that situation as well.

### 9.1.2. Successive nodes stopped

As mentioned in the previous section, Data Grid can’t detect whether a node left the JGroups view because of a process/machine crash, or because of a network failure: whenever a node leaves the JGroups cluster abruptly, it is assumed to be because of a network problem.

If the configured number of copies (`numOwners`) is greater than 1, the cluster can remain available and will try to make new replicas of the data on the crashed node. However, other nodes might crash during the rebalance process. If more than `numOwners` nodes crash in a short interval of time, there is a
chance that some cache entries have disappeared from the cluster altogether. In this case, with the DENY_READ_WRITES or ALLOW_READS strategy enabled, Data Grid assumes (incorrectly) that there is a split brain and enters DEGRADED mode as described in the split–brain section.

The administrator can also shut down more than \texttt{numOwners} nodes in rapid succession, causing the loss of the data stored only on those nodes. When the administrator shuts down a node gracefully, Data Grid knows that the node can’t come back. However, the cluster doesn’t keep track of how each node left, and the cache still enters DEGRADED mode as if those nodes had crashed.

At this stage there is no way for the cluster to recover its state, except stopping it and repopulating it on restart with the data from an external source. Clusters are expected to be configured with an appropriate \texttt{numOwners} in order to avoid \texttt{numOwners} successive node failures, so this situation should be pretty rare. If the application can handle losing some of the data in the cache, the administrator can force the availability mode back to AVAILABLE via JMX.

9.1.3. Conflict Manager

The conflict manager is a tool that allows users to retrieve all stored replica values for a given key. In addition to allowing users to process a stream of cache entries whose stored replicas have conflicting values. Furthermore, by utilising implementations of the \texttt{EntryMergePolicy} interface it is possible for said conflicts to be resolved automatically.

9.1.3.1. Detecting Conflicts

Conflicts are detected by retrieving each of the stored values for a given key. The conflict manager retrieves the value stored from each of the key’s write owners defined by the current consistent hash. The \texttt{.equals} method of the stored values is then used to determine whether all values are equal. If all values are equal then no conflicts exist for the key, otherwise a conflict has occurred. Note that null values are returned if no entry exists on a given node, therefore we deem a conflict to have occurred if both a null and non-null value exists for a given key.

9.1.3.2. Merge Policies

In the event of conflicts arising between one or more replicas of a given CacheEntry, it is necessary for a conflict resolution algorithm to be defined, therefore we provide the \texttt{EntryMergePolicy} interface. This interface consists of a single method, "merge", whose returned CacheEntry is utilised as the "resolved" entry for a given key. When a non-null CacheEntry is returned, this entries value is "put" to all replicas in the cache. However when the merge implementation returns a null value, all replicas associated with the conflicting key are removed from the cache.

The merge method takes two parameters: the "preferredEntry" and "otherEntries". In the context of a partition merge, the preferredEntry is the primary replica of a CacheEntry stored in the partition that contains the most nodes or if partitions are equal the one with the largest topologyId. In the event of overlapping partitions, i.e. a node A is present in the topology of both partitions \{A\}, \{A,B,C\}, we pick \{A\} as the preferred partition as it will have the higher topologId as the other partition’s topology is behind. When a partition merge is not occurring, the "preferredEntry" is simply the primary replica of the CacheEntry. The second parameter, "otherEntries" is simply a list of all other entries associated with the conflicting key are removed from the cache.

\begin{quote}
\textbf{NOTE}

\texttt{EntryMergePolicy::merge} is only called when a conflict has been detected, it is not called if all CacheEntries are the same.
\end{quote}

Currently Data Grid provides the following implementations of EntryMergePolicy:
<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MergePolicy.NONE (default)</td>
<td>No attempt is made to resolve conflicts. Entries hosted on the minority partition are removed and the nodes in this partition do not hold any data until the rebalance starts. Note, this behaviour is equivalent to prior Infinispan versions which did not support conflict resolution. Note, in this case all changes made to entries hosted on the minority partition are lost, but once the rebalance has finished all entries will be consistent.</td>
</tr>
<tr>
<td>MergePolicy.PREFERRED_ALWAYS</td>
<td>Always utilise the &quot;preferredEntry&quot;. MergePolicy.NONE is almost equivalent to PREFERRED_ALWAYS, albeit without the performance impact of performing conflict resolution, therefore MergePolicy.NONE should be chosen unless the following scenario is a concern. When utilising the DENY_READ_WRITES or DENY_READ strategy, it is possible for a write operation to only partially complete when the partitions enter DEGRADED mode, resulting in replicas containing inconsistent values. MergePolicy.PREFERRED_ALWAYS will detect said inconsistency and resolve it, whereas with MergePolicy.NONE the CacheEntry replicas will remain inconsistent after the cluster has rebalanced.</td>
</tr>
<tr>
<td>MergePolicy.PREFERRED_NON_NULL</td>
<td>Utilise the &quot;preferredEntry&quot; if it is non-null, otherwise utilise the first entry from &quot;otherEntries&quot;.</td>
</tr>
<tr>
<td>MergePolicy.REMOVE_ALL</td>
<td>Always remove a key from the cache when a conflict is detected.</td>
</tr>
<tr>
<td>Fully qualified class name</td>
<td>The custom implementation for merge will be used</td>
</tr>
</tbody>
</table>

**9.1.4. Usage**

During a partition merge the ConflictManager automatically attempts to resolve conflicts utilising the configured EntryMergePolicy, however it is also possible to manually search for/resolve conflicts as required by your application.

The code below shows how to retrieve anEmbeddedCacheManager’s ConflictManager, how to retrieve all versions of a given key and how to check for conflicts across a given cache.

```java
EmbeddedCacheManager manager = new DefaultCacheManager("example-config.xml");
Cache<Integer, String> cache = manager.getCache("testCache");
ConflictManager<Integer, String> crm = ConflictManagerFactory.get(cache.getAdvancedCache());

// Get All Versions of Key
Map<Address, InternalCacheValue<String>> versions = crm.getAllVersions(1);
```
NOTE

Although the `ConflictManager::getConflicts` stream is processed per entry, the underlying spliterator is in fact lazily-loading cache entries on a per segment basis.

9.1.5. Configuring partition handling

Unless the cache is distributed or replicated, partition handling configuration is ignored. The default partition handling strategy is `ALLOW_READ_WRITES` and the default EntryMergePolicy is `MergePolicies::PREFERRED_ALWAYS`.

```java
Stream<Map<Address, CacheEntry<Integer, String>>> conflicts = crm.getConflicts();
conflicts.forEach(map -> {
    CacheEntry<Integer, String> entry = map.values().iterator().next();
    Object conflictKey = entry.getKey();
    cache.remove(conflictKey);
});
```

```
// Process conflicts stream and perform some operation on the cache
```

```
// Detect and then resolve conflicts using the configured EntryMergePolicy
```

```
// Detect and then resolve conflicts using the passed EntryMergePolicy instance
```

```
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
    .whenSplit(PartitionHandling.ALLOW_READ_WRITES)
    .mergePolicy(MergePolicy.PREFERRED_ALWAYS);
```

The same can be achieved programmatically:

```java
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
    .whenSplit(PartitionHandling.ALLOW_READ_WRITES)
    .mergePolicy(new CustomMergePolicy());
```

9.1.5.1. Implement a custom merge policy

It’s also possible to provide custom implementations of the EntryMergePolicy

```
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
    .whenSplit(PartitionHandling.ALLOW_READ_WRITES)
    .mergePolicy(new CustomMergePolicy());
```
9.1.5.2. Deploy custom merge policies to an Infinispan server instance

To utilise a custom EntryMergePolicy implementation on the server, it’s necessary for the implementation class(es) to be deployed to the server. This is accomplished by utilising the java service-provider convention and packaging the class files in a jar which has a META-INF/services/org.infinispan.conflict.EntryMergePolicy file containing the fully qualified class name of the EntryMergePolicy implementation.

```java
public class CustomMergePolicy implements EntryMergePolicy<String, String> {
    @Override
    public CacheEntry<String, String> merge(CacheEntry<String, String> preferredEntry,
        List<CacheEntry<String, String>> otherEntries) {
        // decide which entry should be used
        return the_solved_CacheEntry;
    }
}
```

# list all necessary implementations of EntryMergePolicy with the full qualified name

org.example.CustomMergePolicy

In order for a Custom merge policy to be utilised on the server, you should enable object storage, if your policies semantics require access to the stored Key/Value objects. This is because cache entries in the server may be stored in a marshalled format and the Key/Value objects returned to your policy would be instances of WrappedByteArray. However, if the custom policy only depends on the metadata associated with a cache entry, then object storage is not required and should be avoided (unless needed for other reasons) due to the additional performance cost of marshalling data per request. Finally, object storage is never required if one of the provided merge policies is used.

9.1.6. Monitoring and administration

The availability mode of a cache is exposed in JMX as an attribute in the Cache MBean. The attribute is writable, allowing an administrator to forcefully migrate a cache from DEGRADED mode back to AVAILABLE (at the cost of consistency).

The availability mode is also accessible via the AdvancedCache interface:

```java
AdvancedCache ac = cache.getAdvancedCache();

// Read the availability
boolean available = ac.getAvailability() == AvailabilityMode.AVAILABLE;

// Change the availability
if (!available) {
    ac.setAvailability(AvailabilityMode.AVAILABLE);
}
```