BigMemory Max Configuration Guide

Version 4.3.3

October 2016

TERRACOTTA

This document applies to BigMemory Max Version 4.3.3 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: BMM-CG-433-20161018

Table of Contents

1 Configuring BigMemory Max

About BigMemory Max Configuration

BigMemory Maxsupports declarative configuration via an XML configuration file, as well as programmatic configuration via class-constructor APIs. Choosing one approach over the other can be a matter of preference or a requirement, such as when an application requires a certain run-time context to determine appropriate configuration settings.

If your project permits the separation of configuration from run time use, there are advantages to the declarative approach:

- Cache configuration can be changed more easily at deployment time.
- Configuration can be centrally organized for greater visibility.
- Configuration lifecycle can be separated from application-code lifecycle.
- Configuration errors are checked at startup rather than causing an unexpected runtime error.
- If the configuration file is not provided, a default configuration is always loaded at runtime.

This guide focuses on XML declarative configuration. Programmatic configuration is explored in certain examples and is documented in the Javadoc at http://www.ehcache.org/apidocs/2.10.1/.

XML Configuration

BigMemory Max uses Ehcache as its user-facing interface and is configured using the Ehcache configuration system. By default, Ehcache looks for an ASCII or UTF8 encoded XML configuration file called ehcache.xml at the top level of the Java classpath. You may specify alternate paths and filenames for the XML configuration file by using the various CacheManager constructors as described in the CacheManager Javadoc at http://www.ehcache.org/apidocs/2.10.1/.

To avoid resource conflicts, one XML configuration is required for each CacheManager that is created. For example, directory paths and listener ports require unique values. BigMemory Max will attempt to resolve conflicts, and, if one is found, it will emit a warning reminding the user to use separate configurations for multiple CacheManagers.

A sample ehcache.xml file is included in the BigMemory Max distribution. It contains full commentary on how to configure each element. This file can also be downloaded from http://ehcache.org/ehcache.xml.

ehcache.xsd

Ehcache configuration files must comply with the Ehcache XML schema, ehcache.xsd, which can be downloaded from http://ehcache.org/ehcache.xsd.

The BigMemory Max distribution also contains a copy of ehcache.xsd.

ehcache-failsafe.xml

If the CacheManager default constructor or factory method is called, Ehcache looks for a file called ehcache.xml in the top level of the classpath. Failing that, it looks for ehcache-failsafe.xml in the classpath. The ehcache-failsafe.xml file is packaged in the Ehcache JAR and should always be found.

ehcache-failsafe.xml provides a simple default configuration to enable users to get started before they create their own ehcache.xml.

When ehcache-failsafe.xml is used, Ehcache will emit a warning, reminding the user to set up a proper configuration. The meaning of the elements and attributes are explained in the section on ehcache.xml.

```
<ehcache>
  <diskStore path="java.io.tmpdir"/>
  <defaultCache
     maxEntriesLocalHeap="10000"
     eternal="false"
     timeToIdleSeconds="120"
     timeToLiveSeconds="120"
     maxEntriesLocalDisk="10000000"
     diskExpiryThreadIntervalSeconds="120"
     memoryStoreEvictionPolicy="LRU">
     <persistence strategy="localTempSwap"/>
  </defaultCache>
</ehcache>
```

About Default Cache

The *defaultCache* configuration is applied to any cache that is not explicitly configured. The defaultCache appears in the ehcache-failsafe.xml file by default, and can also be added to any BigMemory Max configuration file.

While the defaultCache configuration is not required, an error is generated if caches are created by name (programmatically) with no defaultCache loaded.

Dynamically Changing Cache Configuration

While most of the BigMemory Max configuration is not changeable after startup, certain cache configuration parameters can be modified dynamically at runtime. These include the following:

- Expiration settings
  - **timeToLive** – The maximum number of seconds an element can exist in the cache regardless of access. The element expires at this limit and will no longer be returned from the cache. The default value is 0, which means no TTL eviction takes place (infinite lifetime).
  - **timeToIdle** – The maximum number of seconds an element can exist in the cache without being accessed. The element expires at this limit and will no longer be

returned from the cache. The default value is 0, which means no TTI eviction takes place (infinite lifetime).

Note that the `eternal` attribute, when set to "true", overrides `timeToLive` and `timeToIdle` so that no expiration can take place.

- Local sizing attributes
  - `maxEntriesLocalHeap`
  - `maxBytesLocalHeap`
  - `maxEntriesLocalDisk`
  - `maxBytesLocalDisk`.
- memory-store eviction policy.
- CacheEventListeners can be added and removed dynamically

This example shows how to dynamically modify the cache configuration of a running cache:

```
Cache cache = manager.getCache("sampleCache");
CacheConfiguration config = cache.getCacheConfiguration();
config.setTimeToIdleSeconds(60);
config.setTimeToLiveSeconds(120);
config.setmaxEntriesLocalHeap(10000);
config.setmaxEntriesLocalDisk(1000000);
```

Dynamic cache configurations can also be disabled to prevent future changes:

```
Cache cache = manager.getCache("sampleCache");
cache.disableDynamicFeatures();
```

In ehcache.xml, you can disable dynamic configuration by setting the `<ehcache>` element's `dynamicConfig` attribute to "false".

Dynamically Changing a Distributed Configuration

Just as for standalone BigMemory, mutating the configuration of distributed BigMemory requires access to the set methods of cache.getCacheConfiguration().

The following table provides information for dynamically changing common configuration options in a Terracotta cluster. The Scope column, which specifies where the configuration is in effect, can have one of the following values:

- Client – The Terracotta client where the CacheManager runs.
- TSA – The Terracotta Server Array for the cluster.
- BOTH – Both the client and the TSA.

Note that configuration options whose scope covers "BOTH" are distributed and therefore affect a cache on all clients.

Configuration Option	Dynamic	Scope	Notes
Cache name	NO	TSA	
Nonstop	NO	Client	Enable High Availability
Timeout	YES	Client	For nonstop.
Timeout Behavior	YES	Client	For nonstop.
Immediate Timeout When Disconnected	YES	Client	For nonstop.
Time to Idle	YES	BOTH	
Time to Live	YES	BOTH	
Maximum Entries or Bytes in Local Stores	YES	Client	This and certain other sizing attributes may be pooled by the CacheManager, creating limitations on how they can be changed.
Maximum Entries in Cache	YES	TSA	
Persistence Strategy	N/A	N/A	
Disk Expiry Thread Interval	N/A	N/A	
Disk Spool Buffer Size	N/A	N/A	
Maximum Off-heap	N/A	N/A	Maximum off-heap memory allotted to the TSA.
Eternal	YES	BOTH	

Configuration Option	Dynamic	Scope	Notes
Pinning	NO	BOTH	For details, see "Managing Data Life" on page 39.
Clear on Flush	NO	Client	
Copy on Read	NO	Client	
Copy on Write	NO	Client	
Statistics	YES	Client	Cache statistics. Change dynamically with `cache.setStatistics(boolean)` method.
Logging	NO	Client	Ehcache and Terracotta logging is specified in configuration. However, cluster events can be set dynamically. For details, see "Cluster Events" in the *Developer Guide* for BigMemory Max.
Consistency	NO	Client	It is possible to switch to and from bulk mode. For details, see "Bulk Loading" in the *Developer Guide* for BigMemory Max..
Synchronous Writes	NO	Client	

To apply non-dynamic L1 changes, remove the existing cache and then add (to the same CacheManager) a new cache with the same name as the removed cache, and which has the new configuration. Restarting the CacheManager with an updated configuration, where all cache names are the same as in the previous configuration, will also apply non-dynamic L1 changes.

Passing Copies Instead of References

By default, a get() operation on a store returns a reference to the requested data, and any changes to that data are immediately reflected in the memory store. In cases where an application requires a *copy* of data rather than a reference to it, you can configure the store to return a copy. This allows you to change a copy of the data without affecting the original data in the memory store.

This is configured using the `copyOnRead` and `copyOnWrite` attributes of the <cache> and <defaultCache> elements in your configuration, or programmatically as follows:

```
CacheConfiguration config = new CacheConfiguration("copyCache", 1000)
                                    .copyOnRead(true).copyOnWrite(true);
Cache copyCache = new Cache(config);
```

The default configuration is "false" for both options.

To copy elements on put()-like and/or get()-like operations, a copy strategy is used. The default implementation uses serialization to copy elements. You can provide your own implementation of net.sf.ehcache.store.compound.CopyStrategy using the <copyStrategy> element:

```
<cache name="copyCache"
    maxEntriesLocalHeap="10"
    eternal="false"
    timeToIdleSeconds="5"
    timeToLiveSeconds="10"
    copyOnRead="true"
    copyOnWrite="true">
  <copyStrategy class="com.company.ehcache.MyCopyStrategy"/>
</cache>
```

A single instance of your CopyStrategy is used per cache. Therefore, in your implementation of CopyStrategy.copy(T), T must be thread-safe.

A copy strategy can be added programmatically in the following way:

```
CacheConfiguration cacheConfiguration = new CacheConfiguration("copyCache", 10);
CopyStrategyConfiguration copyStrategyConfiguration = new CopyStrategyConfiguration();
copyStrategyConfiguration.setClass("com.company.ehcache.MyCopyStrategy");
cacheConfiguration.addCopyStrategy(copyStrategyConfiguration);
```

2 Configuring Storage Tiers

About Storage Tiers

BigMemory Max has three storage tiers, summarized here:

- **Memory store** – Heap memory that holds a copy of the hottest subset of data from the off-heap store. Subject to Java GC.
- **Off-heap store** – Limited in size only by available RAM. Not subject to Java GC. Can store serialized data only. Provides overflow capacity to the memory store.
- **Disk store** – Backs up in-memory data and provides overflow capacity to the other tiers. Can store serialized data only.

> **Note:** The disk tier is available for local (standalone) instances of BigMemory Max. For distributed BigMemory Max, a Terracotta server or server array supplies the third tier.

This document defines the standalone storage tiers and their suitable element types and then details the configuration for each storage tier.

Before running in production, it is strongly recommended that you test the tiers with the actual amount of data you expect to use in production. For information about sizing the tiers, refer to "Sizing Storage Tiers" on page 25.

Configuring Memory Store

The memory store is always enabled and exists in heap memory. For the best performance, allot as much heap memory as possible without triggering garbage collection (GC) pauses, and use the off-heap store to hold the data that cannot fit in heap (without causing GC pauses).

The memory store has the following characteristics:

- Accepts all data, whether serializable or not
- Fastest storage option
- Thread safe for use by multiple concurrent threads

The memory store is the top tier and is automatically used by BigMemory Max to store the data hotset because it is the fastest store. It requires no special configuration to enable, and its overall size is taken from the Java heap size. Since it exists in the heap, it is limited by Java GC constraints.

Memory Use, Spooling, and Expiry Strategy in the Memory Store

All caches specify their maximum in-memory size, in terms of the number of elements, at configuration time.

When an element is added to a cache and it goes beyond its maximum memory size, an existing element is either deleted, if overflow is not enabled, or evaluated for spooling to another tier, if overflow is enabled. The overflow options are `overflowToOffHeap` and `<persistence>` (disk store).

If overflow is enabled, a check for expiry is carried out. If it is expired it is deleted; if not it is spooled. The eviction of an item from the memory store is based on the optional `MemoryStoreEvictionPolicy` attribute specified in the configuration file. Legal values are LRU (default), LFU and FIFO:

- **Least Recently Used (LRU)**—LRU is the default setting. The last-used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.
- **Least Frequently Used (LFU)** —For each get call on the element the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element with least number of hits, the Less Frequently Used element, is evicted.
- **First In First Out (FIFO)** — Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

For all the eviction policies there are also putQuiet() and getQuiet() methods which do not update the last used timestamp.

When there is a get() or a getQuiet() on an element, it is checked for expiry. If expired, it is removed and null is returned. Note that at any point in time there will usually be some expired elements in the cache. Memory sizing of an application must always take into account the maximum size of each cache.

Tip: calculateInMemorySize() is a convenient method that can provide an estimate of the size (in bytes) of the memory store. It returns the serialized size of the cache, providing a rough estimate. Do not use this method in production as it is has a negative effect on performance.

An alternative is to have an expiry thread. This is a trade-off between lower memory use and short locking periods and CPU utilization. The design is in favor of the latter. For those concerned with memory use, simply reduce the tier size. For more information, refer to "Sizing Storage Tiers" on page 25.

Configuring Off-Heap Store

The off-heap store extends the in-memory store to memory outside the of the object heap. This store, which is not subject to Java garbage collection (GC), is limited only by the amount of RAM available.

Because off-heap data is stored in bytes, only data that is Serializable is suitable for the off-heap store. Any non serializable data overflowing to the OffHeapMemoryStore is simply removed, and a WARNING level log message emitted.

Since serialization and deserialization take place on putting and getting from the off-heap store, it is theoretically slower than the memory store. This difference, however, is mitigated when GC involved with larger heaps is taken into account.

Allocating Direct Memory in the JVM

The off-heap store uses the direct-memory portion of the JVM. You must allocate sufficient direct memory for the off-heap store by using the JVM property `MaxDirectMemorySize`.

For example, to allocate 2GB of direct memory in the JVM:

```
java -XX:MaxDirectMemorySize=2G ...
```

Since direct memory may be shared with other processes, allocate at least 256MB (or preferably 1GB) more to direct memory than will be allocated to the off-heap store.

Note the following about allocating direct memory:

- If you configure off-heap memory but do not allocate direct memory with `-XX:MaxDirectMemorySize`, the default value for direct memory depends on your version of your JVM. Oracle HotSpot has a default equal to maximum heap size (-Xmx value), although some early versions may default to a particular value.
- `MaxDirectMemorySize` must be added to the local node's startup environment.
- Direct memory, which is part of the Java process heap, is separate from the object heap allocated by -Xmx. The value allocated by `MaxDirectMemorySize`must not exceed physical RAM, and is likely to be less than total available RAM due to other memory requirements.
- The amount of direct memory allocated must be within the constraints of available system memory and configured off-heap memory.
- The maximum amount of direct memory space you can use depends on the process data model (32-bit or 64-bit) and the associated operating system limitations, the amount of virtual memory available on the system, and the amount of physical memory available on the system.

Using Off-Heap Store with 32-bit JVMs

The amount of heap-offload you can achieve is limited by addressable memory. 64-bit systems can allow as much memory as the hardware and operating system can handle, while 32-bit systems have strict limitations on the amount of memory that can be effectively managed.

For a 32-bit process model, the maximum virtual address size of the process is typically 4GB, though most 32-bit operating systems have a 2GB limit. The maximum heap size available to Java is lower still due to particular operating-system (OS) limitations, other operations that may run on the machine (such as mmap operations used by certain

APIs), and various JVM requirements for loading shared libraries and other code. A useful rule to observe is to allocate no more to off-heap memory than what is left over after -Xmx is set. For example, if you set -Xmx3G, then off-heap should be no more than 1GB. Breaking this rule may not cause an OutOfMemoryError on startup, but one is likely to occur at some point during the JVM's life.

If Java GC issues are afflicting a 32-bit JVM, then off-heap store can help. However, note the following:

- Everything has to fit in 4GB of addressable space. If 2GB of heap is allocated (with -Xmx2g) then at most are 2GB left for off-heap data.
- The JVM process requires some of the 4GB of addressable space for its code and shared libraries plus any extra Operating System overhead.
- Allocating a 3GB heap with -Xmx, as well as 2047MB of off-heap memory, will not cause an error at startup, but when it's time to grow the heap an OutOfMemoryError is likely.
- If both -Xms3G and -Xmx3G are used with 2047MB of off-heap memory, the virtual machine will start but then complain as soon as the off-heap store tries to allocate the off-heap buffers.
- Some APIs, such as java.util.zip.ZipFile on a 1.5 JVM, may <mmap> files in memory. This will also use up process space and may trigger an OutOfMemoryError.

Configuring the Off-Heap Store

If an off-heap store is configured, the corresponding memory store overflows to that off-heap store. Configuring an off-heap store can be done either through XML or programmatically. With either method, the off-heap store is configured on a per-cache basis.

Declarative Configuration

The following XML configuration creates an off-heap cache with an in-heap store (`maxEntriesLocalHeap`) of 10,000 elements which overflow to a 1-gigabyte off-heap area.

```
<ehcache updateCheck="false" monitoring="off" dynamicConfig="false">
  <defaultCache maxEntriesLocalHeap="10000"
              eternal="true"
              memoryStoreEvictionPolicy="LRU"
              statistics="false" />
  <cache name="sample-offheap-cache"
       maxEntriesLocalHeap="10000"
       eternal="true"
       memoryStoreEvictionPolicy="LRU"
       overflowToOffHeap="true"
       maxBytesLocalOffHeap="1G"/>
</ehcache>
```

The configuration options are:

overflowToOffHeap

Values may be true or false. When set to true, enables the cache to utilize off-heap memory storage to improve performance. Off-heap memory is not subject to Java GC cycles and has a size limit set by the Java property `MaxDirectMemorySize`. The default value is false.

maxBytesLocalOffHeap

Sets the amount of off-heap memory available to the cache and is in effect only if `overflowToOffHeap` is true. The minimum amount that can be allocated is 1MB. There is no maximum.

For more information on sizing data stores, see "Sizing Storage Tiers" on page 25.

Note: You should set `maxEntriesLocalHeap` to at least 100 elements when using an off-heap store to avoid performance degradation. Lower values for `maxEntriesLocalHeap` trigger a warning to be logged.

Programmatic Configuration

The equivalent cache can be created using the following programmatic configuration:

```
public Cache createOffHeapCache()
{
  CacheConfiguration config = new CacheConfiguration("sample-offheap-cache", 10000)
  .overflowToOffHeap(true).maxBytesLocalOffHeap("1G");
  Cache cache = new Cache(config);
  manager.addCache(cache);
  return cache;
}
```

Off-Heap Configuration Examples

These examples show how to allocate 8GB of machine memory to different stores. It assumes a data set of 7GB - say for a cache of 7M items (each 1kb in size).

Those who want minimal application response-time variance (or minimizing GC pause times), will likely want all the cache to be off-heap. Assuming that 1GB of heap is needed for the rest of the application, they will set their Java config as follows:

```
java -Xms1G -Xmx1G -XX:maxDirectMemorySize=7G
```

And their Ehcache config as:

```
<cache
 maxEntriesLocalHeap=100
 overflowToOffHeap="true"
 maxBytesLocalOffHeap="6G"
... />
```

Note: To accommodate server communications layer requirements, the value of `maxDirectMemorySize` must be greater than the value of `maxBytesLocalOffHeap`. The exact amount greater depends upon the size

of `maxBytesLocalOffHeap`. The minimum is 256MB, but if you allocate 1GB more to the `maxDirectMemorySize`, it will certainly be sufficient. The server will only use what it needs and the rest will remain available.

Those who want best possible performance for a hot set of data, while still reducing overall application response time variance, will likely want a combination of on-heap and off-heap. The heap will be used for the hotset, the off-heap for the rest. So, for example if the hot set is 1M items (or 1GB) of the 7GB data. They will set their Java config as follows

```
java -Xms2G -Xmx2G -XX:maxDirectMemorySize=6G
```

And their Ehcache config as:

```
<cache
  maxEntriesLocalHeap=1M
  overflowToOffHeap="true"
  maxBytesLocalOffHeap="5G"
  ...>
```

This configuration will compare *very* favorably against the alternative of keeping the less-hot set in a database (100x slower) or caching on local disk (20x slower).

Where the data set is small and pauses are not a problem, the whole data set can be kept on heap:

```
<cache
  maxEntriesLocalHeap=1M
  overflowToOffHeap="false"
  ...>
```

Where latency isn't an issue, the disk can be used:

```
<cache
  maxEntriesLocalHeap=1M
  <persistence strategy="localTempSwap"/>
  ...>
```

Tuning Off-Heap Store Performance

Memory-related or performance issues that arise during operations can be related to improper allocation of memory to the off-heap store. If performance or functional issues arise that can be traced back to the off-heap store, see the suggested tuning tips in this section.

General Memory allocation

Committing too much of a system's physical memory is likely to result in paging of virtual memory to disk, quite likely during garbage-collection operations, leading to significant performance issues. On systems with multiple Java processes, or multiple processes in general, the sum of the Java heaps and off-heap stores for those processes should also not exceed the size of the physical RAM in the system. Besides memory allocated to the heap, Java processes require memory for other items, such as code (classes), stacks, and PermGen.

Note that MaxDirectMemorySize sets an upper limit for the JVM to enforce, but does not actually allocate the specified memory. Overallocation of direct memory (or buffer) space is therefore possible, and could lead to paging or even memory-related errors. The limit on direct buffer space set by MaxDirectMemorySize should take into account the total physical memory available, the amount of memory that is allotted to the JVM object heap, and the portion of direct buffer space that other Java processes may consume.

In addition, be sure to allocate at least 15 percent more off-heap memory than the size of your data set. To maximize performance, a portion of off-heap memory is reserved for meta-data and other purposes.

Note also that there could be other users of direct buffers (such as NIO and certain frameworks and containers). Consider allocating additional direct buffer memory to account for that additional usage.

Compressed References

For 64-bit JVMs running Java 6 Update 14 or higher, consider enabling compressed references to improve overall performance. For heaps up to 32GB, this feature causes references to be stored at half the size, as if the JVM is running in 32-bit mode, freeing substantial amounts of heap for memory-intensive applications. The JVM, however, remains in 64-bit mode, retaining the advantages of that mode.

For the Oracle HotSpot, compressed references are enabled using the option `-XX:+UseCompressedOops`. For IBM JVMs, use `-Xcompressedrefs`.

Slow Off-Heap Allocation

Based configuration, usage, and memory requirements, BigMemory could allocate off-heap memory multiple times. If off-heap memory comes under pressure due to over-allocation, the host OS may begin paging to disk, thus slowing down allocation operations. As the situation worsens, an off-heap buffer too large to fit in memory can quickly deplete critical system resources such as RAM and swap space and crash the host OS.

To stop this situation from degrading, off-heap allocation time is measured to avoid allocating buffers too large to fit in memory. If it takes more than 1.5 seconds to allocate a buffer, a warning is issued. If it takes more than 15 seconds, the JVM is halted with System.exit() (or a different method if the Security Manager prevents this).

To prevent a JVM shutdown after a 15-second delay has occurred, set the `net.sf.ehcache.offheap.DoNotHaltOnCriticalAllocationDelay` system property to true. In this case, an error is logged instead.

Swappiness and Huge Pages

An OS could swap data from memory to disk even if memory is not running low. For the purpose of optimization, data that appears to be unused may be a target for swapping. Because BigMemory can store substantial amounts of data in RAM, its data may be swapped by the OS. But swapping can degrade overall cluster performance by

introducing thrashing, the condition where data is frequently moved forth and back between memory and disk.

To make heap memory use more efficient, Linux, Microsoft Windows, and Oracle Solaris users should review their configuration and usage of swappiness as well as the size of the swapped memory pages. In general, BigMemory benefits from lowered swappiness and the use of *huge pages* (also known as *big pages*, *large pages*, and *superpages*).

Settings for these behaviors vary by OS and JVM. For Oracle HotSpot, `-XX:+UseLargePages` and `-XX:LargePageSizeInBytes=<size>` (where <size> is a value allowed by the OS for specific CPUs) can be used to control page size. However, note that this setting does *not* affect how off-heap memory is allocated. Over-allocating huge pages while also configuring substantial off-heap memory *can starve off-heap allocation and lead to memory and performance problems*.

Maximum Serialized Size of an Element

This section applies when using BigMemory through the Ehcache API.

Unlike the memory and the disk stores, by default the off-heap store has a 4MB limit for classes with high quality hashcodes, and 256KB limit for those with pathologically bad hashcodes. The built-in classes such as String and the java.lang.Number subclasses Long and Integer have high quality hashcodes. This can issues when objects are expected to be larger than the default limits.

To override the default size limits, set the system property `net.sf.ehcache.offheap.cache_name.config.idealMaxSegmentSize` to the size you require.

For example,

```
net.sf.ehcache.offheap.com.company.domain.State.config.idealMaxSegmentSize=30M
```

Reducing Faulting

While the memory store holds a hotset (a subset) of the entire data set, the off-heap store should be large enough to hold the entire data set. The frequency of misses (get operations that fail to find the data in memory) begins to rise when the data is too large to fit into off-heap memory, forcing gets to fetch data from the disk store (called *faulting*). More misses in turn raise latency and lower performance.

For example, tests with a 4GB data set and a 5GB off-heap store recorded no misses. With the off-heap store reduced to 4GB, 1.7 percent of cache operations resulted in misses. With the off-heap store at 3GB, misses reached 15 percent.

Configuring Disk Store

The disk store provides a thread-safe disk-spooling facility that can be used for either additional storage or persisting data through system restarts.

Note: The disk tier is available for local (standalone) instances of BigMemory Max. For distributed BigMemory Max, a Terracotta server or server array is used instead of a disk tier.

This section describes local disk usage. You can find additional information about configuring the disk store in "Configuring Fast Restart (FRS)" on page 47.

Serialization

Only data that is Serializable can be placed in the disk store. Writes to and from the disk use ObjectInputStream and the Java serialization mechanism. Any non-serializable data overflowing to the disk store is removed and a NotSerializableException is thrown.

Serialization speed is affected by the size of the objects being serialized and their type. It has been found that:

- The serialization time for a Java object consisting of a large Map of String arrays was 126ms, where the serialized size was 349,225 bytes.
- The serialization time for a byte[] was 7ms, where the serialized size was 310,232 bytes.

Byte arrays are 20 times faster to serialize, making them a better choice for increasing disk-store performance.

Configuring the Disk Store

Disk stores are configured on a per CacheManager basis. If one or more caches requires a disk store but none is configured, a default directory is used and a warning message is logged to encourage explicit configuration of the disk store path.

Configuring a disk store is optional. If all caches use only memory and off-heap stores, then there is no need to configure a disk store. This simplifies configuration, and uses fewer threads. This also makes it unnecessary to configure multiple disk store paths when multiple CacheManagers are being used.

Two disk store options are available:

- Temporary store (`localTempSwap`)
- Persistent store (`localRestartable`)

localTempSwap

The `localTempSwap` persistence strategy allows local disk usage during BigMemory operation, providing an extra tier for storage. This disk storage is temporary and is cleared after a restart.

If the disk store path is not specified, a default path is used, and the default will be auto-resolved in the case of a conflict with another CacheManager.

The localTempSwap disk store creates a data file for each cache on startup called "<*cache_name*>.data".

localRestartable

This option implements a restartable store for all in-memory data. After any restart, the data set is automatically reloaded from disk to the in-memory stores.

The path to the directory where any required disk files will be created is configured with the `<diskStore>` sub-element of the Ehcache configuration. In order to use the restartable store, a unique and explicitly specified path is required.

The diskStore Configuration Element

Files are created in the directory specified by the `<diskStore>` configuration element. The `<diskStore>` element has one attribute called `path`.

```
<diskStore path="/path/to/store/data"/>
```

Legal values for the path attribute are legal file system paths. For example, for Unix:

```
/home/application/cache
```

The following system properties are also legal, in which case they are translated:

- `user.home` - User's home directory
- `user.dir`- User's current working directory
- `java.io.tmpdir` - Default temp file path
- `ehcache.disk.store.dir`- A system property you would normally specify on the command line—for example, `java -Dehcache.disk.store.dir=/u01/myapp/diskdir`.

Subdirectories can be specified below the system property, for example:

```
user.dir/one
```

To programmatically set a disk store path:

```
DiskStoreConfiguration diskStoreConfiguration = new DiskStoreConfiguration();
diskStoreConfiguration.setPath("/my/path/dir");
// Already created a configuration object ...
configuration.addDiskStore(diskStoreConfiguration);
CacheManager mgr = new CacheManager(configuration);
```

Note: A CacheManager's disk store path cannot be changed once it is set in configuration. If the disk store path is changed, the CacheManager must be recycled for the new path to take effect.

Disk Store Expiry and Eviction

Expired elements are eventually evicted to free up disk space. The element is also removed from the in-memory index of elements.

One thread per cache is used to remove expired elements. The optional attribute `diskExpiryThreadIntervalSeconds` sets the interval between runs of the expiry thread.

Important: Setting `diskExpiryThreadIntervalSeconds` to a low value can cause excessive disk-store locking and high CPU utilization. The default value is 120 seconds.

If a cache's disk store has a limited size, Elements will be evicted from the disk store when it exceeds this limit. The LFU algorithm is used for these evictions. It is not configurable or changeable.

Note: With the `localTempSwap` strategy, you can use `maxEntriesLocalDisk` or `maxBytesLocalDisk` at either the Cache or CacheManager level to control the size of the disk tier.

Turning off Disk Stores

To turn off disk store path creation, comment out the `diskStore` element in ehcache.xml.

The default Ehcache configuration, ehcache-failsafe.xml, uses a disk store. To avoid use of a disk store, specify a custom ehcache.xml with the `diskStore` element commented out.

3 Sizing Storage Tiers

The Sizing Attributes

Tuning BigMemory Max often involves sizing the data storage tiers appropriately. You can size the different data tiers in a number of ways using simple sizing attributes. These sizing attributes affect memory and disk resources.

The following table summarizes the sizing attributes you can use.

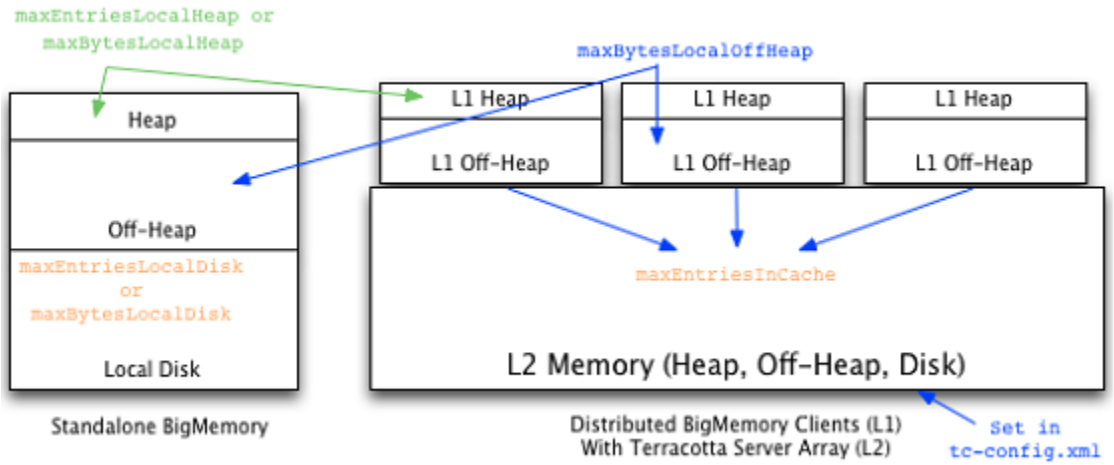
Tier	Attribute	Description
Memory Store (Heap)	`maxEntriesLocalHeap` `maxBytesLocalHeap`	The maximum number of entries or bytes a data set can use in local heap memory, or when set at the CacheManager level (maxBytesLocalHeap only), as a pool available to all data sets under that CacheManager. This setting is required for every cache or at the CacheManager level. Pooling is available at the CacheManager level using `maxBytesLocalHeap` only.
Off-heap Store	`maxBytesLocalOffHeap`	The maximum number of bytes a data set can use in off-heap memory, or when set at the CacheManager level, as a pool available to all data sets under that CacheManager. Pooling is available at the CacheManager level.
Disk Store	`maxEntriesLocalDisk` `maxBytesLocalDisk`	The maximum number of entries or bytes a data set can use on the local disk, or when set at the CacheManager level (maxBytesLocalDisk only), as a pool available to all data sets under that CacheManager. Note that these settings apply to temporary disk usage (`localTempSwap`); these settings do not apply to disk persistence. Pooling is available at the CacheManager level using `maxBytesLocalDisk` only.

The following table summarizes sizing attributes for distributed BigMemory Max (using a Terracotta Server or a Server Array).

Tier	Attribute	Description
Heap	`maxEntriesLocalHeap` `maxBytesLocalHeap`	The maximum number of cache entries or bytes a cache can use in local heap memory, or, when set at the CacheManager level (`maxBytesLocalHeap` only), a local pool available to all caches under that CacheManager. This setting is required for every cache or at the CacheManager level. Pooling is available at the CacheManager level using `maxBytesLocalHeap` only.
Off-heap	`maxBytesLocalOffHeap`	The maximum number of bytes a cache can use in off-heap memory, or, when set at the CacheManager level, as a pool available to all caches under that CacheManager. This setting requires BigMemory. Pooling is available at the CacheManager level.
Local disk	N/A	Distributed caches cannot use the local disk. Pooling at the CacheManager level is not available.
Terracotta Server Array	`maxEntriesInCache`	The number of cache elements that the Terracotta Server Array will store for a distributed cache. This value must be greater than or equal to the number of entries in all of the clients of the server array. Set on individual distributed caches only, this setting is unlimited by default. Pooling at the CacheManager level is not available.

Attributes that set a number of entries or elements take an integer. Attributes that set a memory size (bytes) use the Java -Xmx syntax (for example: "500k", "200m", "2g") or percentage (for example: "20%"). Percentages, however, can be used only in the case where a CacheManager-level pool has been configured.

The following diagram illustrates the tiers and their effective sizing attributes.



Pooling Resources Versus Sizing Individual Data Sets

You can constrain the size of any data set on a specific tier in that data set's configuration. You can also constrain the size of all of a CacheManager's data sets in a specific tier by configuring an overall size at the CacheManager level.

If there is no CacheManager-level pool specified for a tier, an individual data set claims the amount of that tier specified in its configuration. If there is a CacheManager-level pool specified for a tier, an individual data set claims that amount *from the pool*. In this case, data sets with no size configuration for that tier receive an equal share of the remainder of the pool (after data sets with explicit sizing configuration have claimed their portion).

For example, if a CacheManager with eight data sets pools one gigabyte of heap, and two data sets each explicitly specify 200MB of heap while the remaining data sets do not specify a size, the remaining data sets will share 600MB of heap equally. Note that data sets must use bytes-based attributes to claim a portion of a pool; entries-based attributes such as `maxEntriesLocal` cannot be used with a pool.

On startup, the sizes specified by data sets are checked to ensure that any CacheManager-level pools are not over-allocated. If over-allocation occurs for any pool, an InvalidConfigurationException is thrown. Note that percentages should not add up to more than 100% of a single pool.

If the sizes specified by data sets for any tier take exactly the entire CacheManager-level pool specified for that tier, a warning is logged. In this case, data sets that do not specify a size for that tier cannot use the tier, as nothing is left over.

Memory Store (Heap)

A size must be provided for the heap, either in the CacheManager (`maxBytesLocalHeap` only) or in each individual cache (`maxBytesLocalHeap` or `maxEntriesLocalHeap`). Not doing so causes an InvalidConfigurationException.

If a pool is configured, it can be combined with a heap setting in an individual cache. This allows the cache to claim a specified portion of the heap setting configured in the pool. However, in this case the cache setting must use `maxBytesLocalHeap` (same as the CacheManager).

In any case, every cache *must* have a heap setting, either configured explicitly or taken from the pool configured in the CacheManager.

Off-Heap Store

Off-heap sizing can be configured in bytes only, never by entries.

If a CacheManager has a pool configured for off-heap, your application cannot add caches dynamically that have off-heap configuration — doing so generates an error. In addition, if any caches that used the pool are removed programmatically or through the Terracotta Management Console (TMC), other caches in the pool cannot claim the unused portion. To allot the entire off-heap pool to the remaining caches, remove the unwanted cache from the Ehcache configuration and then reload the configuration.

To use off-heap as a data tier, a cache must have `overflowToOffHeap` set to "true". If a CacheManager has a pool configured for using off-heap, the `overflowToOffHeap` attribute is automatically set to "true" for all caches. In this case, you can prevent a specific cache from overflowing to off-heap by explicitly setting its `overflowToOffHeap` attribute to "false".

Note that an exception is thrown if any cache using an off-heap store attempts to put an element that will cause the off-heap store to exceed its allotted size. The exception will contain a message similar to the following:

```
The element '[ key = 25274, value=[B@3ebb2a91, version=1, hitCount=0,
CreationTime = 1367966069431, LastAccessTime = 1367966069431 ]'
is too large to be stored in this offheap store.
```

Local Disk Store

The local disk can be used as a data tier, either for temporary storage or for disk persistence, but not both at once.

To use the disk as a temporary tier during BigMemory operation, set the `persistenceStrategy` to "localTempSwap", and use the `maxBytesLocalDisk` setting to configure the size of this tier. For more information about using the disk as a temporary tier, see "Configuring Disk Store" on page 21.

For information about using the disk store for data persistence, see "Data Persistence Implementation" on page 48.

Note that BigMemory Max distributed across a Terracotta Server Array cannot use the local disk.

Sizing Examples

The following examples illustrate both pooled and individual cache-sizing configurations.

Note: Some of the following examples include allocations for off-heap storage. Off-heap data storage (i.e., the off-heap tier) is only available with the Terracotta BigMemory products.

Pooled Resources

The following configuration sets pools for all of this CacheManager's caches:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="100M"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="50G">
...
<cache name="Cache1" ... </cache>
<cache name="Cache2" ... </cache>
<cache name="Cache3" ... </cache>
</ehcache>
```

CacheManager CM1 automatically allocates these pools equally among its three caches. Each cache gets one third of the allocated heap, off-heap, and local disk. Note that at the CacheManager level, resources can be allocated in bytes only.

Explicitly Sizing Caches

You can explicitly allocate resources to specific caches:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="100M"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="60G">
...
<cache name="Cache1" ...
        maxBytesLocalHeap="50M"
        ...
  </cache>
<cache name="Cache2" ...
       maxBytesLocalOffHeap="5G"
       ...
  </cache>
<cache name="Cache3" ... </cache>
</ehcache>
```

In the example above, Cache1 reserves 50Mb of the 100Mb local-heap pool; the other caches divide the remaining portion of the pool equally. Cache2 takes half of the local off-heap pool; the other caches divide the remaining portion of the pool equally. Cache3 receives 25Mb of local heap, 2.5Gb of off-heap, and 20Gb of the local disk.

Caches that reserve a portion of a pool are not required to use that portion. Cache1, for example, has a fixed portion of the local heap but may have any amount of data in heap up to the configured value of 50Mb.

Note that caches must use the same sizing attributes used to create the pool. Cache1, for example, cannot use `maxEntriesLocalHeap` to reserve a portion of the pool.

Mixed Sizing Configurations

If a CacheManager does not pool a particular resource, that resource can still be allocated in cache configuration, as shown in the following example.

```
<ehcache xmlns...
        Name="CM2"
        maxBytesLocalHeap="100M">
...
<cache name="Cache4" ...
        maxBytesLocalHeap="50M"
        maxEntriesLocalDisk="100000"
        ...
  </cache>
<cache name="Cache5" ...
       maxBytesLocalOffHeap="10G"
       ...
  </cache>
<cache name="Cache6" ... </cache>
</ehcache>
```

CacheManager CM2 creates one pool (local heap). Its caches all use the local heap and are constrained by the pool setting, as expected. However, cache configuration can allocate other resources as desired. In this example, Cache4 allocates disk space for its data, and Cache5 allocates off-heap space for its data. Cache6 gets 25Mb of local heap only.

Using Percents

The following configuration sets pools for each tier:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="1G"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="50G">
...
<!-- Cache1 gets 400Mb of heap, 2.5Gb of off-heap, and 5Gb of disk. -->
<cache name="Cache1" ...
maxBytesLocalHeap="40%">
</cache>
<!-- Cache2 gets 300Mb of heap, 5Gb of off-heap, and 5Gb of disk. -->
<cache name="Cache2" ...
maxBytesLocalOffHeap="50%">
</cache>
<!-- Cache2 gets 300Mb of heap, 2.5Gb of off-heap, and 40Gb of disk. -->
<cache name="Cache3" ...
maxBytesLocalDisk="80%">
</cache>
</ehcache>
```

Note: You can use a percentage of the total JVM heap for the CacheManager maxBytesLocalHeap. The CacheManager percentage, then, is a portion of the total JVM heap, and in turn, the Cache percentage is the portion of the CacheManager pool for that tier.

Sizing Without a Pool

The CacheManager in this example does not pool any resources.

```
<ehcache xmlns...
        Name="CM3"
       ... >
...
<cache name="Cache7" ...
        maxBytesLocalHeap="50M"
        maxEntriesLocalDisk="100000"
        ...
  </cache>
<cache name="Cache8" ...
       maxEntriesLocalHeap="1000"
       maxBytesLocalOffHeap="10G"
       ...
  </cache>
<cache name="Cache9" ...
       maxBytesLocalHeap="50M"
...
</cache>
</ehcache>
```

Caches can be configured to use resources as necessary. Note that every cache in this example must declare a value for local heap. This is because no pool exists for the local heap; implicit (CacheManager configuration) or explicit (cache configuration) local-heap allocation is required.

Sizing Distributed Caches

Terracotta distributed caches can be sized as other caches except that they do not use the local disk and therefore cannot be configured with *LocalDisk sizing attributes. Distributed caches use the storage resources available on the Terracotta Server Array, BigMemory Max and the Fast Restart store. For more information about fast restartability for distributed caches, see "Terracotta Cluster with Reliability" in the *Terracotta Server Array Administrator Guide*.

Cache-configuration sizing attributes behave as local configuration, which means that every node can load its own sizing attributes for the same caches. That is, while some elements and attributes are fixed by the first Ehcache configuration loaded in the cluster, cache-configuration sizing attributes can vary across nodes for the same cache.

For example, a cache may have the following configuration on one node:

```
<cache name="myCache"
     maxEntriesLocalHeap="10000"
     maxBytesLocalOffHeap="8g"
     eternal="false"
     timeToIdleSeconds="3600"
```

```
     timeToLiveSeconds="1800">
    <persistence strategy="distributed"/>
    <terracotta/>
</cache>
```

The same cache may have the following size configuration on another node:

```
<cache name="myCache"
     maxEntriesLocalHeap="10000"
     maxBytesLocalOffHeap="10g"
     eternal="false"
     timeToIdleSeconds="3600"
     timeToLiveSeconds="1800">
    <persistence strategy="distributed"/>
    <terracotta/>
</cache>
```

If the cache exceeds its size constraints on a node, then with this configuration the Terracotta Server Array provides myCache with an unlimited amount of space for spillover and backup. To impose a limit, you must set `maxEntriesInCache` to a positive non-zero value:

```
<cache name="myCache"
     maxEntriesLocalHeap="10000"
     maxBytesLocalOffHeap="10g"
     eternal="false"
     timeToIdleSeconds="3600"
     timeToLiveSeconds="1800"
     maxEntriesInCache="1000000">
    <persistence strategy="distributed"/>
    <terracotta/>
</cache>
```

The Terracotta Server Array will now evict myCache entries to stay within the limit set by `maxEntriesInCache`. However, for any particular cache, eviction on the Terracotta Server Array is based on the largest size configured for that cache. In addition, the Terracotta Server Array will *not* evict any cache entries that exist on at least one client node, regardless of the limit imposed by `maxEntriesInCache`.

Note: If `maxEntriesInCache` is not set, the default value 0 is used, which means that the cache is unbounded and will not undergo capacity eviction (but periodic and resource evictions are still allowed). Although bounds are not enforced by capacity eviction, an "unbounded" cache in this respect has a practical limitation equal to Integer.MAX_VALUE (app. 2 billion entries). For more information about eviction, see "Understanding Automatic Resource Management" in the *Terracotta Server Array Administrator Guide*.

Sizing the Terracotta Server Array

Since `maxEntriesInCache` is based on entries, you must size the Terracotta Server Array based on the expected average size of an entry. One way to discover this value is by using the Terracotta Management Console (TMC). Set up a test cluster with the expected data set, and connect it to the TMC. Then navigate to **Application Data** > **Sizing Panel**, and review the **Relative Cache Sizes by Tier** section.

Note that the average cache-entry size reported in the TMC is an estimate. For more information, see the *Terracotta Management Console User Guide*.

Pinning and Size Limitations

Pinned caches can override the limits set by cache-configuration sizing attributes, potentially causing OutOfMemory errors. This is because pinning prevents flushing of cache entries to lower tiers. For more information on pinning, see "Pinning Data" on page 42.

Built-In Sizing Computation and Enforcement

Internal BigMemory Max mechanisms track data-element sizes and enforce the limits set by CacheManager sizing pools.

Sizing of Elements

Elements put in a memory-limited cache will have their memory sizes measured. The entire Element instance added to the cache is measured, including key and value, as well as the memory footprint of adding that instance to internal data structures. Key and value are measured as object graphs – each reference is followed and the object reference also measured. This goes on recursively.

Shared references will be measured by each class that references it. This will result in an overstatement. Shared references should therefore be ignored.

Ignoring for Size Calculations

For the purposes of measurement, references can be ignored using the `@IgnoreSizeOf` annotation. The annotation may be declared at the class level, on a field, or on a package. You can also specify a file containing the fully qualified names of classes, fields, and packages to be ignored.

This annotation is not inherited, and must be added to any subclasses that should also be excluded from sizing.

The following example shows how to ignore the `Dog` class.

```
@IgnoreSizeOf
public class Dog {
  private Gender gender;
  private String name;
}
```

The following example shows how to ignore the `sharedInstance` field.

```
public class MyCacheEntry {
  @IgnoreSizeOf
  private final SharedClass sharedInstance;
    ...
}
```

Packages may be also ignored if you add the @IgnoreSizeOf annotation to appropriate package-info.java of the desired package. Here is a sample package-info.java for and in the com.pany.ignore package:

```
@IgnoreSizeOf
package com.pany.ignore;
import net.sf.ehcache.pool.sizeof.filter.IgnoreSizeOf;
```

Alternatively, you may declare ignored classes and fields in a file and specify a `net.sf.ehcache.sizeof.filter` system property to point to that file.

```
# That field references a common graph between all cached entries
com.pany.domain.cache.MyCacheEntry.sharedInstance
# This will ignore all instances of that type
com.pany.domain.SharedState
# This ignores a package
com.pany.example
```

Note that these measurements and configurations apply only to on-heap storage. Once Elements are moved to off-heap memory ordisk, they are serialized as byte arrays. The serialized size is then used as the basis for measurement.

Configuration for Limiting the Traversed Object Graph

As noted above, sizing caches involves traversing object graphs, a process that can be limited with annotations. This process can also be controlled at both the CacheManager and cache levels.

Size-Of Limitation at the CacheManager Level

Control how deep the size-of engine can go when sizing on-heap elements by adding the following element at the CacheManager level:

```
<sizeOfPolicy maxDepth="100" maxDepthExceededBehavior="abort"/>
```

This element has the following attributes

- `maxDepth` – Controls how many linked objects can be visited before the size-of engine takes any action. This attribute is required.
- `maxDepthExceededBehavior` – Specifies what happens when the max depth is exceeded while sizing an object graph:
  - "continue" – (DEFAULT) Forces the size-of engine to log a warning and continue the sizing operation. If this attribute is not specified, "continue" is the behavior used.
  - "abort" – Forces the SizeOf engine to abort the sizing, log a warning, and mark the cache as not correctly tracking memory usage. With this setting, `Ehcache.hasAbortedSizeOf()` returns true.

The SizeOf policy can be configured at the CacheManager level (directly under `<ehcache>`) and at the cache level (under `<cache>` or `<defaultCache>`). The cache policy always overrides the CacheManager if both are set.

Size-Of Limitation at the Cache level

Use the `<sizeOfPolicy>` as a sub-element in any `<cache>` block to control how deep the size-of engine can go when sizing on-heap elements belonging to the target cache. This cache-level setting overrides the CacheManager size-of setting.

Debugging of Size-Of Related Errors

If warnings or errors appear that seem related to size-of measurement (usually caused by the size-of engine walking the graph), generate more log information on sizing activities:

- Set the `net.sf.ehcache.sizeof.verboseDebugLogging` system property to true.
- Enable debug logs on `net.sf.ehcache.pool.sizeof` in your chosen implementation of SLF4J.

Eviction When Using CacheManager-Level Storage

When a CacheManager-level storage pool is exhausted, a cache is selected on which to perform eviction to recover pool space. The eviction from the selected cache is performed using the cache's configured eviction algorithm (LRU, LFU, etc...). The cache from which eviction is performed is selected using the "minimal eviction cost" algorithm described below:

```
eviction-cost = mean-entry-size * drop-in-hit-rate
```

Eviction cost is defined as the increase in bytes requested from the underlying SOR (System of Record, e.g., database) per unit time used by evicting the requested number of bytes from the cache.

If we model the hit distribution as a simple power-law then:

```
P(hit n'th element) ~ 1/n^{alpha}
```

In the continuous limit, this means the total hit rate is proportional to the integral of this distribution function over the elements in the cache. The change in hit rate due to an eviction is then the integral of this distribution function between the initial size and the final size. Assuming that the eviction size is small compared to the overall cache size, we can model this as:

```
drop ~ access * 1/x^{alpha} * Delta(x)
```

where "access" is the overall access rate (hits + misses), and x is a unit-less measure of the "fill level" of the cache. Approximating the fill level as the ratio of hit rate to access rate, and substituting in to the eviction-cost expression, we get:

```
eviction-cost = mean-entry-size * access * 1/(hits/access)^{alpha}
                    * (eviction / (byteSize / (hits/access)))
```

Simplifying:

```
eviction-cost = (byteSize / countSize) * access * 1/(h/A)^{alpha}
                    * (eviction * hits)/(access * byteSize)
eviction-cost = (eviction * hits) / (countSize * (hits/access)^{alpha})
```

Removing the common factor of "eviction", which is the same in all caches, lead us to evicting from the cache with the minimum value of:

```
eviction-cost = (hits / countSize) / (hits/access)^{alpha}
```

When a cache has a zero hit-rate (it is in a pure loading phase), we deviate from this algorithm and allow the cache to occupy 1/nth of the pool space, where "n" is the number of caches using the pool. Once the cache starts to be accessed, we re-adjust to match the actual usage pattern of that cache.

4 Managing Data Life

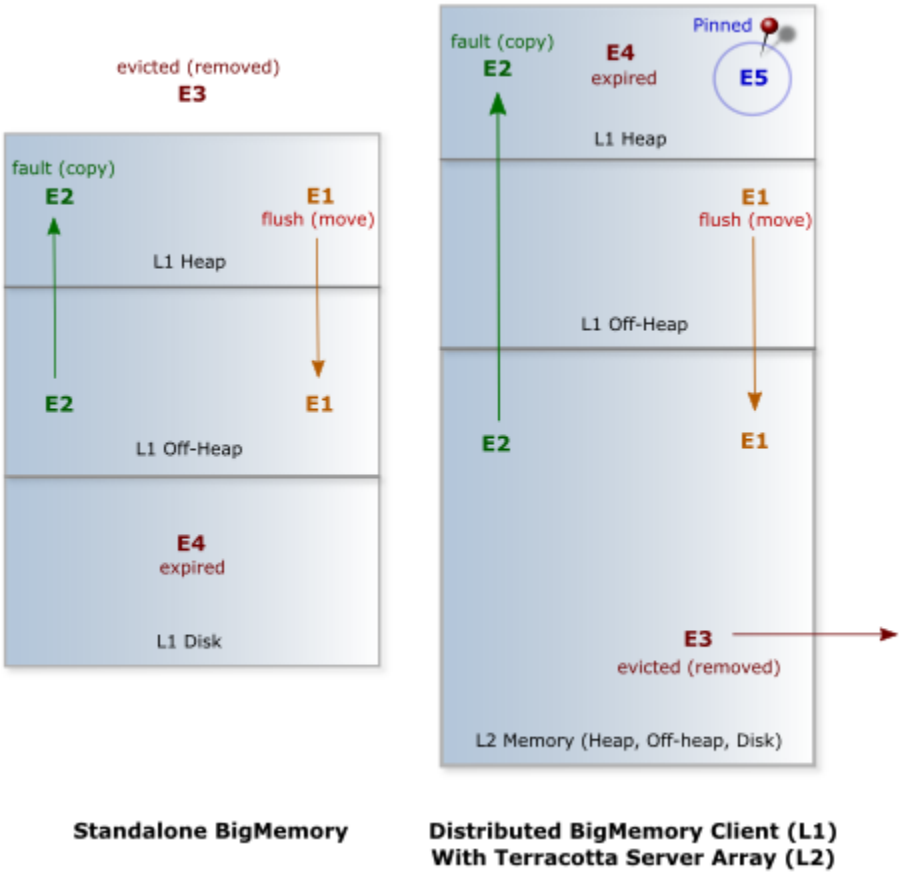
Configuration Options that Affect Data Life

This topic covers managing the life of the data in each of the data-storage tiers, including the pinning features of Automatic Resource Control (ARC).

You use the options to manage data life within the data-storage tiers:

- **Flush** – To move an entry to a lower tier. Flushing is used to free up resources while still keeping data in BigMemory Max.
- **Fault** – To copy an entry from a lower tier to a higher tier. Faulting occurs when data is required at a higher tier but is not resident there. The entry is not deleted from the lower tiers after being faulted.
- **Eviction** – To remove an entry from BigMemory Max. The entry is deleted; it can only be reloaded from an outside source. Entries are evicted to free up resources.
- **Expiration** – A status based on Time-To-Live and Time-To-Idle settings. To maintain performance, expired entries may not be immediately flushed or evicted.
- **Pinning** – To keep data in memory indefinitely.

The following figure depicts the movement of data across the storage tiers.



Setting Expiration

Data entries expire based on parameters with configurable values. When eviction occurs, expired elements are the first to be removed. Having an effective expiration configuration is critical to optimizing the use of resources such as heap and maintaining overall performance.

To add expiration, specify values for the following `<cache>` attributes, and tune these values based on results of performance tests:

- `timeToIdleSeconds` – The maximum number of seconds an element can exist in the BigMemory data store without being accessed. The element expires at this limit and will no longer be returned from BigMemory Max. The default value is 0, which means no TTI eviction takes place (infinite lifetime).
- `timeToLiveSeconds` – The maximum number of seconds an element can exist in the BigMemory data store regardless of use. The element expires at this limit and will no longer be returned from BigMemory Max. The default value is 0, which means no TTL eviction takes place (infinite lifetime).

- `maxEntriesInCache` – The maximum sum total number of elements (cache entries) allowed for a distributed cache in all Terracotta clients. If this target is exceeded, eviction occurs to bring the count within the allowed target. The default value is 0, which means that the cache will not undergo capacity eviction (but periodic and resource evictions are still allowed). Note that `maxEntriesInCache` reflects storage allocated on the Terracotta Server Array.
- `eternal` – If the cache's `eternal` flag is set, it overrides any finite TTI/TTL values that have been set. Individual cache elements may also be set to eternal. Eternal elements and caches do not expire, however they may be evicted.

For information about how configuration can impact eviction, see "How Configuration Affects Element Flushing and Eviction" on page 44.

Pinning Data

Without pinning, expired cache entries can be flushed and eventually evicted, and even most non-expired elements can also be flushed and evicted as well, if resource limitations are reached. Pinning gives per-cache control over whether data can be evicted from BigMemory Max.

Data that should remain in memory can be pinned. You cannot pin individual entries, only an entire cache. As described in the following topics, there are two types of pinning, depending upon whether the pinning configuration should take precedence over resource constraints or the other way around.

Configuring Pinning

Entire caches can be pinned using the `pinning` element in the Ehcache configuration. This element has a required attribute (`store`) to specify how the pinning will be accomplished.

The `store` attribute can have either of the following values:

- inCache – Data is pinned in the cache, in any tier in which cache data is stored. The tier is chosen based on performance-enhancing efficiency algorithms. Unexpired entries can never be evicted.
- localMemory – Data is pinned to the memory store or the off-heap store. Entries are evicted only in the event that the store's configured size is exceeded. Updated copies of invalidated elements are faulted in from the server, a behavior you can override by setting the Terracotta property `l1.servermapmanager.faultInvalidatedPinnedEntries` to false.

For example, the following cache is configured to pin its entries:

```
<cache name="Cache1" ... >
     <pinning store="inCache" />
</cache>
```

The following cache is configured to pin its entries to heap or off-heap only:

```
<cache name="Cache2" ... >
```

```
    <pinning store="localMemory" />
</cache>
```

Pinning and Cache Sizing

The interaction of the pinning configuration with the cache sizing configuration depends upon which pinning option is used.

For `inCache` pinning, the pinning setting takes priority over the configured cache size. Elements resident in a cache with this pinning option cannot be evicted if they have not expired. This type of pinned cache is not eligible for eviction at all, and `maxEntriesInCache` should not be configured for this cache.

Important: Potentially, pinned caches could grow to an unlimited size. Caches should never be pinned unless they are designed to hold a limited amount of data (such as reference data) or their usage and expiration characteristics are understood well enough to conclude that they cannot cause errors.

For `localMemory` pinning, the configured cache size takes priority over the pinning setting. `localMemory` pinning should be used for optimization, to keep data in heap or off-heap memory, unless or until the tier becomes too full. If the number of entries surpasses the configured size, entries will be evicted. For example, in the following cache the `maxEntriesLocalHeap` and `maxBytesLocalOffHeap` settings override the pinning configuration. (Off-heap storage is only available in the Terracotta BigMemory products.)

```
<cache name="myCache"
    maxEntriesLocalHeap="10000"
    maxBytesLocalOffHeap="8g"
    ... >
   <pinning store="localMemory" />
</cache>
```

Scope of Pinning

Pinning as a setting exists in the local Ehcache client (L1) memory. It is never distributed in the cluster. In case the pinned cache is bulk loaded, the pinned data is removed from the L1 and will be faulted back in from the Terracotta Server Array.

Pinning achieved programmatically will not be persisted — after a restart the pinned entries are no longer pinned. Pinning is also lost when an L1 rejoins a cluster. Cache pinning in configuration is reinstated with the configuration file.

Explicitly Removing Data from a Pinned Cache

To unpin all of a cache's pinned entries, clear the cache. Specific entries can be removed from a cache using Cache.remove(). To empty the cache, Cache.removeAll(). If the cache itself is removed (Cache.dispose() or CacheManager.removeCache()), then any data still remaining in the cache is also removed locally. However, that remaining data is *not* removed from the Terracotta Server Array or disk (if localRestartable).

How Configuration Affects Element Flushing and Eviction

The following example shows a cache with certain expiration settings:

```
<cache name="myCache"
      eternal="false" timeToIdleSeconds="3600"
      timeToLiveSeconds="0" memoryStoreEvictionPolicy="LFU">
</cache>
```

Note the following about the myCache configuration:

- If a client accesses an entry in myCache that has been idle for more than an hour (`timeToIdleSeconds`), that element is evicted.
- If an entry expires but is not accessed, and no resource constraints force eviction, then the expired entry remains in place until a periodic evictor removes it.
- Entries in myCache can live forever if accessed at least once per 60 minutes (`timeToLiveSeconds`). However, unexpired entries may still be flushed based on other limitations. For details, see "Sizing Storage Tiers" on page 25.

How Configuration Affects Eviction in Distributed Cache

Element eviction is a crucial part of keeping cluster resources operating efficiently. Element eviction and expiration are related, but an expired element is not necessarily evicted immediately and an evicted element is not necessarily an expired element. Cache elements might be evicted due to resource and configuration constraints, while expired elements are evicted from the Terracotta client when a get() or put() operation occurs on that element (sometimes called *inline* eviction).

The Terracotta server array contains the full key set (as well as all values), while clients contain a subset of keys and values based on elements they have faulted in from the server array.

Typically, an expired cache element is evicted, or more accurately flushed, from a client tier to a lower tier when a get() or put() operation occurs on that element. However, a client may also flush expired, and then unexpired elements, whenever a cache's sizing limit for a specific tier is reached or it is under memory pressure. This type of eviction is intended to meet configured and real memory constraints.

To learn about eviction and controlling the size of the cache, see "Managing Data Life" on page 39 and "Sizing Storage Tiers" on page 25.

Flushing from clients does not mean eviction from the server array. Servers will evict expired elements and elements can become candidates for eviction from the server array when servers run low on allocated BigMemory. Unexpired elements can also be evicted if they meet the following criteria:

- They are in a cache with infinite TTI/TTL (Time To Idle and Time To Live), or no explicit settings for TTI/TTL. Enabling a cache's `eternal` flag overrides any finite TTI/TTL values that have been set.
- They are not resident on any Terracotta client. These elements can be said to have been "orphaned." Once evicted, they will have to be faulted back in from a system of record if requested by a client.

For more information about Terracotta Server Array data eviction, see "Automatic Resource Management" in the *BigMemory Max Administrator Guide*.

Data Freshness and Expiration

Databases and other systems of record (SORs) that were not built to accommodate caching outside of the database do not normally come with any default mechanism for notifying external processes when data has been updated or modified.

When using BigMemory Max as a caching system, the following strategies can help to keep the data in the cache in sync:

- **Data Expiration** Use the eviction algorithms included with Ehcache, along with the `timeToIdleSeconds` and `timetoLiveSeconds` settings, to enforce a maximum time for elements to live in the cache (forcing a re-load from the database or SOR).
- **Message Bus**: Use an application to make all updates to the database. When updates are made, post a message onto a message queue with a key to the item that was updated. All application instances can subscribe to the message bus and receive messages about data that is updated, and can synchronize their local copy of the data accordingly (for example by invalidating the cache entry for updated data)
- **Triggers**: Using a database trigger can accomplish a similar task as the message bus approach. Use the database trigger to execute code that can publish a message to a message bus. The advantage to this approach is that updates to the database do not have to be made only through a special application. The downside is that not all database triggers support full execution environments and it is often inadvisable to execute heavy-weight processing such as publishing messages on a queue during a database trigger.

The Data Expiration method is the simplest and most straightforward. It gives you the most control over the data synchronization, and doesn't require cooperation from any external systems. You simply set a data expiration policy and let Ehcache expire data from the cache, thus allowing fresh reads to re-populate and re-synchronize the cache.

The most important consideration when using the expiration method is balancing data freshness with database load. The shorter you make the expiration settings - meaning the more "fresh" you try to make the data - the more load you will place on the database.

Try out some numbers and see what kind of load your application generates. Even modestly short values such as five or ten minutes will produce significant load reductions.

5 Configuring Fast Restart (FRS)

About Fast Restart (FRS)

BigMemory's Fast Restart (FRS) feature provides enterprise-ready crash resilience by keeping a fully consistent, real-time record of your in-memory data. After any kind of shutdown — planned or unplanned — the next time your application starts up, all of the data that was in BigMemory is still available and very quickly accessible.

The advantages of the Fast Restart feature include:

- In-memory data survives crashes and enables fast restarts. Because your in-memory data does not need to be reloaded from a remote data source, applications can resume at full speed after a restart.
- A real-time record of your in-memory data provides true fault tolerance. Fast Restart provides the equivalent of a local "hot mirror," which guarantees full data consistency.
- A consistent record of your in-memory data opens many possibilities for business innovation, such as arranging data sets according to time-based needs or moving data sets around to different locations. The uses of the Fast Restart store can range from a simple key-value persistence mechanism with fast read performance, to an operational store with in-memory speeds during operation for both reads and writes.

Data Persistence Implementation

The BigMemory Fast Restart feature works by creating a real-time record of the in-memory data, which it persists in a Fast Restart store on the local disk. After any restart, the data that was last in memory (both heap and off-heap stores) automatically loads from the Fast Restart store back into memory.

Data persistence is configured by adding the `<persistence>` sub-element to a cache configuration. The `<persistence>` sub-element includes two attributes: `strategy` and `synchronousWrites`.

```
<cache>
   <persistence strategy="localRestartable|localTempSwap|none|distributed"
                synchronousWrites="false|true"/>
</cache>
```

Strategy Options

The options for the `strategy` attribute are:

- `"localRestartable"` — Enables the Fast Restart feature which automatically logs all BigMemory data. This option provides fast restartability with fault-tolerant data persistence.

- `"localTempSwap"` — Enables temporary local disk usage. This option provides an extra tier for data storage during operation, but this store is not persisted. After a restart, the disk is cleared of any BigMemory data.
- `"none"` — Does not offload data to disk. With this option, all of the working data is kept in memory only. This is the default mode.
-
- `"distributed"` — Defers to the `<terracotta>` configuration for persistence settings. For more information, refer to "Terracotta Clustering Configuration Elements" on page 55.

Synchronous Writes Options

If the `strategy` attribute is set to "localRestartable", then the `synchronousWrites` attribute can be configured. The options for `synchronousWrites` are:

- **synchronousWrites="false"** — This option specifies that an eventually consistent record of the data is kept on disk at all times. Writes to disk happen when efficient, and cache operations proceed without waiting for acknowledgement of writing to disk. After a restart, the data is recovered as it was when last synced. This option is faster than `synchronousWrites="true"`, but after a crash, the last 2-3 seconds of written data may be lost.

  If not specified, the default for `synchronousWrites` is "false".
- **synchronousWrites="true"** — This option specifies that a fully consistent record of the data is kept on disk at all times. As changes are made to the data set, they are synchronously recorded on disk. The write to disk happens before a return to the caller. After a restart, the data is recovered exactly as it was before shutdown. This option is slower than `synchronousWrites="false"`, but after a crash, it provides full data consistency.

For transaction caching with `synchronousWrites`, soft locks are used to protect access. If there is a crash in the middle of a transaction, then upon recovery the soft locks are cleared on next access.

Note: The `synchronousWrites` attribute is also available in the `<terracotta>` sub-element. If configured in both places, it must have the same value.

DiskStore Path

The path to the directory where any required disk files will be created is configured with the `<diskStore>` sub-element of the Ehcache configuration.

- For "localRestartable", a unique and explicitly specified path is required for each node.
- For "localTempSwap", if the disk store path is not specified, a default path is used for the disk tier, and the default path will be auto-resolved in the case of a conflict with another CacheManager.

Note: The Fast Restart feature does not use the disk tier in the same way that conventional disk persistence does. Therefore, when configured for "localRestartable," disk store size measures such as Cache.getDiskStoreSize() or Cache.calculateOnDiskSize() are not applicable and will return zero. On the other hand, when configured for "localTempSwap", these measures will return size values.

Configuration Examples

This section presents possible disk usage configurations when using a standalone deployment.

Options for Crash Resilience

The following configuration provides fast restartability with fully consistent data persistence:

```
<ehcache>
  <diskStore path="/path/to/store/data"/>
  <cache>
    <persistence strategy="localRestartable" synchronousWrites="true"/>
  </cache>
</ehcache>
```

The following configuration provides fast restartability with eventually consistent data persistence:

```
<ehcache>
  <diskStore path="/path/to/store/data"/>
  <cache>
    <persistence strategy="localRestartable" synchronousWrites="false"/>
  </cache>
</ehcache>
```

Clustered Caches

For distributing BigMemory Max across a Terracotta Server Array (TSA), the persistence strategy in the ehcache.xml should be set to "distributed", and the `<terracotta>` sub-element must be present in the configuration.

```
<cache>
  maxEntriesInCache="100000">
  <persistence strategy="distributed"/>
  <terracotta clustered="true" consistency="eventual" synchronousWrites="false"/>
</cache>
```

The attribute `maxEntriesInCache` configures the maximum number of entries in a distributed cache. (`maxEntriesInCache` is not required, but if it is not set, the default is unlimited.)

Note: Restartability must be enabled in the TSA in order for clients to be restartable.

Temporary Disk Storage

The "localTempSwap" persistence strategy create a local disk tier for in-memory data during BigMemory operation. The disk storage is temporary and is cleared after a restart.

```
<ehcache>
  <diskStore path="/auto/default/path"/>
  <cache>
    <persistence strategy="localTempSwap"/>
  </cache>
</ehcache>
```

Note: With the "localTempSwap" strategy, you can use `maxEntriesLocalDisk` or `maxBytesLocalDisk` at either the Cache or CacheManager level to control the size of the disk tier.

In-memory Only Cache

When the persistence strategy is "none", all cache stays in memory (with no overflow to disk nor persistence on disk).

```
<cache>
  <persistence strategy="none"/>
</cache>
```

Programmatic Configuration Example

The following is an example of how to programmatically configure cache persistence on disk:

```
Configuration cacheManagerConfig = new Configuration()
    .diskStore(new DiskStoreConfiguration()
    .path("/path/to/store/data"));
CacheConfiguration cacheConfig = new CacheConfiguration()
    .name("my-cache")
    .maxBytesLocalHeap(16, MemoryUnit.MEGABYTES)
    .maxBytesLocalOffHeap(256, MemoryUnit.MEGABYTES)
    .persistence(new PersistenceConfiguration().strategy(Strategy.LOCALRESTARTABLE));
cacheManagerConfig.addCache(cacheConfig);
CacheManager cacheManager = new CacheManager(cacheManagerConfig);
Ehcache myCache = cacheManager.getEhcache("my-cache");
```

Fast Restart Performance

When configured for fast restartability ("localRestartable" persistence strategy), BigMemory becomes active on restart after all of the in-memory data is loaded. The amount of time until BigMemory is restarted is proportionate to the amount of in-memory data and the speed of the underlying infrastructure. Generally, recovery can occur as fast as the disk speed. With an SSD, for example, if you have a read throughput of 1 GB per second, you will see a similar loading speed during recovery.

Fast Restart Limitations

The following recommendations should be observed when configuring BigMemory for fast restartability:

- The size of on-heap or off-heap stores should not be changed during a shutdown. If the amount of memory allocated is reduced, elements will be evicted upon restart.
- Restartable caches should not be removed from the CacheManager during a shutdown.
- If a restartable cache is disposed, the reference to the cache is deleted, but the cache contents remain in memory and on disk. After a restart, the cache contents are once again recovered into memory and on disk. The way to safely dispose of an unused restartable cache, so that it does not take any space in disk or memory, is to call clear on the cache and then dispose.

6 Defining a Distributed Configuration

About Distributed Configurations

The BigMemory Max configuration file (ehcache.xml by default) contains the configuration for one instance of a CacheManager (the Ehcache class managing a set of defined caches). This configuration file must be in your application's classpath to be found. When using a WAR file, ehcache.xml should be copied to WEB-INF/classes.

Note the following about ehcache.xml in a Terracotta cluster:

- The copy on disk is loaded into memory from the first Terracotta client (also called application server or node) to join the cluster.
- Once loaded, the configuration is persisted in memory by the Terracotta servers in the cluster and survives client restarts.
- In-memory configuration can be edited in the Terracotta Management Console (TMC). Changes take effect immediately but are *not* written to the original on-disk copy of ehcache.xml.
- The in-memory cache configuration is removed with server restarts if the servers are not in persistent mode (`<restartable enabled="false">`), which is the default. The original (on-disk) ehcache.xml is loaded.
- The in-memory cache configuration survives server restarts if the servers are in persistent mode (`<restartable enabled="true">`). If you are using the Terracotta servers with persistence of shared data, and you want the cluster to load the original (on-disk) ehcache.xml, the servers' database must be wiped by removing the data files from the servers' server-data directory. This directory is specified in the Terracotta configuration file in effect (tc-config.xml by default). Wiping the database causes *all persisted shared data to be lost*.

A minimal distributed-cache configuration should have the following configured:

- A CacheManager. See "Configuring a CacheManager" on page 54.
- A Clustering element in individual cache configurations. See "terracotta" on page 55.
- A source for Terracotta client configuration. See "terracottaConfig" on page 57.

Configuring a CacheManager

CacheManager configuration elements and attributes are fully described in the ehcache.xml reference file available in the kit.

Via ehcache.xml

The attributes of `<ehcache>` are:

- **name** – an optional name for the CacheManager. The name is optional and primarily used for documentation or to distinguish Terracotta clustered cache state. With Terracotta clustered caches, a combination of CacheManager name and cache name uniquely identify a particular cache store in the Terracotta clustered memory. The name will show up in the TMC.

Note: If you employ multiple Ehcache configuration files, use the `name` attribute in the <ehcache> element to identify specific CacheManagers in the cluster. The TMC provides a menu listing these names, allowing you to choose the CacheManager you want to view

- **updateCheck** – an optional boolean flag specifying whether this CacheManager should check for new versions of Ehcache over the Internet. If not specified, updateCheck="true".
- **monitoring** – an optional setting that determines whether the CacheManager should automatically register the SampledCacheMBean with the system MBean server. Currently, this monitoring is only useful when using Terracotta clustering. The "autodetect" value detects the presence of Terracotta clustering and registers the MBean. Other allowed values are "on" and "off". The default is "autodetect".

```
<Ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ehcache.xsd"
updateCheck="true" monitoring="autodetect">
```

Programmatic Configuration

CacheManagers can be configured programmatically with a fluent API. The example below creates a CacheManager with a Terracotta configuration specified in an URL, and creates a defaultCache and a cache named "example".

```
Configuration configuration = new Configuration()
.terracotta(new TerracottaClientConfiguration().url("localhost:9510"))
.defaultCache(new CacheConfiguration("defaultCache", 100))
.cache(new CacheConfiguration("example", 100)
.timeToIdleSeconds(5)
.timeToLiveSeconds(120)
.terracotta(new TerracottaConfiguration()));
CacheManager manager = new CacheManager(configuration);
```

Terracotta Clustering Configuration Elements

The `terracotta` and `terracottaConfig` elements in the Ehcache configuration control the clustering of caches with Terracotta.

terracotta

The `<terracotta>` element is an optional sub-element of `<cache>`. It can be set differently for each `<cache>` defined in ehcache.xml.

`<terracotta>` has one sub-element, `<nonstop>`. For more information about the `<nonstop>` element, see "Configuring Nonstop Cache" on page 72.

The following `<terracotta>` attribute allows you to control the type of data consistency for the distributed cache:

- **consistency** – Uses no cache-level locks for better performance ("eventual" DEFAULT) or uses cache-level locks for immediate cache consistency ("strong"). When set to "eventual", allows reads without locks, which means the cache may temporarily return stale data in exchange for substantially improved performance. When set to "strong", guarantees that after any update is completed no local read can return a stale value, but at a potentially high cost to performance, because a large number of locks may need to be stored in client and server heaps.

  Once set, this consistency mode cannot be changed except by reconfiguring the cache using a configuration file and reloading the file. *This setting cannot be changed programmatically.* For more information, see "Consistency Modes" on page 61.

Except for special cases, the following `<terracotta>` attributes should operate with their default values:

- **clustered** – Enables ("true" DEFAULT) or disables ("false") Terracotta clustering of a specific cache. Clustering is enabled if this attribute is not specified. Disabling clustering also disables the effects of all of the other attributes.
- **localCacheEnabled** – Enables ("true" DEFAULT) or disables ("false") local caching of distributed cache data, forcing all of that cached data to reside on the Terracotta Server Array. Disabling local caching may improve performance for distributed caches in write-heavy use cases.
- **synchronousWrites** – Enables ("true") or disables ("false" DEFAULT) synchronous writes from Terracotta clients (application servers) to Terracotta servers. Asynchronous writes (synchronousWrites="false") maximize performance by allowing clients to proceed without waiting for a "transaction received" acknowledgment from the server. This acknowledgment is unnecessary in most use cases. Synchronous writes (synchronousWrites="true") provide extreme data safety but at a performance cost by requiring that a client receive server acknowledgment of a transaction before that client can proceed. If the cache's consistency mode is eventual, or while it is set to bulk load using the bulk-load API, only asynchronous writes can occur (synchronousWrites="true" is ignored).
- **concurrency** – Sets the number of segments for the map backing the underlying server store managed by the Terracotta Server Array. If concurrency is not explicitly set (or set to "0"), the system selects an optimized value.

The following attributes are used with Hibernate:

- **localKeyCache** – Enables ("true") or disables ("false" DEFAULT) a local key cache. BigMemory Max can cache a "hotset" of keys on clients to add locality-of-reference, a feature suitable for read-only cases. Note that the set of keys must be small enough for available memory.

- **localKeyCacheSize** – Defines the size of the local key cache in number (positive integer) of elements. In effect if localKeyCache is enabled. The default value, 300000, should be tuned to meet application requirements and environmental limitations.
- **orphanEviction** – Enables ("true" DEFAULT) or disables ("false") orphan eviction. *Orphans* are cache elements that are not resident in any Hibernate second-level cache but still present on the cluster's Terracotta server instances.
- **orphanEvictionPeriod** – The number of local eviction cycles (that occur on Hibernate) that must be completed before an orphan eviction can take place. The default number of cycles is 4. Raise this value for less aggressive orphan eviction that can reduce faulting on the Terracotta server, or lower it if garbage on the Terracotta server is a concern.

Default Behavior

By default, adding `<terracotta/>` to a `<cache>` block is equivalent to adding the following:

```
<cache name="sampleTerracottaCache"
    maxEntriesLocalHeap="1000"
    eternal="false"
    timeToIdleSeconds="3600"
    timeToLiveSeconds="1800"">
  <persistence strategy="distributed"/>
  <terracotta clustered="true" consistency="eventual" />
</cache>
```

terracottaConfig

The `<terracottaConfig>` element enables the distributed-cache client to identify a source of Terracotta configuration. It also allows a client to rejoin a cluster after disconnecting from that cluster and being timed out by a Terracotta server. For more information on the rejoin feature, see the *BigMemory Max High-Availability Guide*. In addition, the `<terracottaConfig>` element allows you to enable caches for WAN replication.

The client must load the configuration from a file or a Terracotta server. The value of the `url` attribute should contain a path to the file, a system property, or the address and TSA port (9510 by default) of a server.

Note: In a Terracotta cluster, the application server is also known as the client.

For more information on client configuration, see the "Terracotta Configuration Parameters" in the *BigMemory Max Administrator Guide*.

Adding an URL Attribute

Add the `url` attribute to the `<terracottaConfig>` element as follows:

```
<terracottaConfig url="<source>" />
```

where `<source>` must be one of the following:

- A path (for example, `url="/path/to/tc-config.xml"`)
- An URL (for example, `url="http://www.mydomain.com/path/to/tc-config.xml`)
- A system property (for example, `url="${terracotta.config.location}"`), where the system property is defined like this:

```
System.setProperty("terracotta.config.location","10.x.x.x:9510"");
```

- A Terracotta host address in the form `<host>:<tsa-port>` (for example, `url="host1:9510"`). Note the following about using server addresses in the form `<host>:<tsa-port>`:
  - The default TSA port is 9510.
  - In a multi-server cluster, you can specify a comma-delimited list (for example, `url="host1:9510,host2:9510,host3:9510"`).
  - If the Terracotta configuration source changes at a later time, it must be updated in configuration.

Adding the WAN Attribute

For each cache to be replicated, its `ehcache.xml` configuration file must include the `wanEnabledTSA` attribute set to "true" within the `<terracottaConfig>` element. Add the `wanEnabledTSA` attribute to the `<terracottaConfig>` element as follows:

```
<terracottaConfig wanEnabledTSA="true"/>
```

For more information, see the *WAN Replication User Guide*.

Embedding Terracotta Configuration

You can embed the contents of a Terracotta configuration file in ehcache.xml as follows:

```
<terracottaConfig>
   <tc-config>
       <servers>
           <server host="server1" name="s1"/>
           <server host="server2" name="s2"/>
       </servers>
       <clients>
           <logs>app/logs-%i</logs>
       </clients>
   </tc-config>
</terracottaConfig>
```

Embedding Terracotta Configuration

You can embed the contents of a Terracotta configuration file in ehcache.xml as follows:

```
<terracottaConfig>
   <tc-config>
       <servers>
           <server host="server1" name="s1"/>
           <server host="server2" name="s2"/>
       </servers>
       <clients>
           <logs>app/logs-%i</logs>
```

```
        </clients>
    </tc-config>
</terracottaConfig>
```

Incompatible Configuration Attributes

For any clustered cache, you must delete, disable, or edit configuration elements in ehcache.xml that are incompatible when clustering with Terracotta. Clustered caches have a `<terracotta/>` or `<terracotta clustered="true"/>` element.

The following Ehcache configuration attributes or elements should be deleted or disabled:

- Replication-related attributes such as `replicateAsynchronously` and `replicatePuts`.
- The attribute `MemoryStoreEvictionPolicy` is ignored (a clock eviction policy is used instead), however, if allowed to remain in a clustered cache configuration, the `MemoryStoreEvictionPolicy` may cause an exception.

How Server Settings Can Override Client Settings

The tc.property, `ehcache.clustered.config.override.mode`, allows you to determine if and how the cache settings on the server override the cache settings on the client. You can set this tc.property in the tc-config.xml file or define it as a system property with the `com.tc.` prefix.

The possible values are:

- `NONE`- Default behavior: any discrepancy between the client and server versions of a cache configuration option causes the client to throw an exception.
- `GLOBAL`- Allows any cache-wide settings from the server to override those from the client. For example, if a client were to join with a `TTI=10` while the server has a `TTI=15`, then
  - The server TTI value overrides the client TTI value.
  - Local settings, such as `maxEntriesLocalHeap`, are not overridden.
- `ALL` - Causes the client to accept all values from the server's configuration. This includes the cache-wide settings of GLOBAL as well as the local settings, such as `maxBytesLocalHeap`.

Controlling Cache Size

Certain Ehcache cache configuration attributes affect caches clustered with Terracotta.

See "How Configuration Affects Eviction in Distributed Cache" for more information on how configuration affects eviction.

To learn about eviction and controlling the size of the cache, see "Managing Data Life" and "Sizing Storage Tiers."

Cache Events Configuration

The `<cache>` sub-element `<cacheEventListenerFactory>`, which registers listeners for cache events such as puts and updates, has a notification scope controlled by the attribute `listenFor`. This attribute can have one of the following values:

- **local**– Listen for events on the local node. No remote events are detected.
- **remote** – Listen for events on other nodes. No local events are detected.
- **local** – (DEFAULT) Listen for events on both the local node and on remote nodes.

In order for cache events to be detected by remote nodes in a Terracotta cluster, event listeners must have a scope that includes remote events. For example, the following configuration allows listeners of type MyCacheListener to detect both local and remote events:

```
<cache name="myCache" ... >
 <!-- Not defining the listenFor attribute for <cacheEventListenerFactory> is by default equivale
 <cacheEventListenerFactory
    class="net.sf.ehcache.event.TerracottaCacheEventReplicationFactory" />
 <terracotta />
</cache>
```

You must use `net.sf.ehcache.event.TerracottaCacheEventReplicationFactory` as the factory class to enable cluster-wide cache-event broadcasts in a Terracotta cluster.

For more information about cache events in a Terracotta cluster, see "Cache Event Listeners" in the *Developers Guide* for BigMemory Max.

Copy On Read

The `copyOnRead` setting is most easily explained by first examining what it does when not enabled and exploring the potential problems that can arise. For a cache in which `copyOnRead` is not enabled, the following reference comparison will always be true:

```
Object obj1 = c.get("key").getValue();
// Assume no other thread changes the cache mapping between these get() operations ....
Object obj2 = c.get("key").getValue();
if (obj1 == obj2) {
 System.err.println("Same value objects!");
}
```

The fact that the same object reference is returned across multiple get() operations implies that the cache is storing a direct reference to cache value. This default behavior (copyOnRead=false) is usually desired, although there are at least two scenarios in which it is problematic:

1. Caches shared between classloaders

and

2. Mutable value objects

Imagine two web applications that both have access to the same Cache instance (this implies that the core Ehcache classes are in a common classloader). Imagine further that the classes for value types in the cache are duplicated in the web application (so they are not present in the common loader). In this scenario you would get ClassCastExceptions when one web application accessed a value previously read by the other application.

One obvious solution to this problem is to move the value types to the common loader, but another is to enable `copyOnRead`. When copyOnRead is in effect, the object references are unique with every get(). Having unique object references means that the thread context loader of the caller will be used to materialize the cache values on each get(). This feature has utility in OSGi environments as well where a common cache service might be shared between bundles.

Another subtle issue concerns mutable value objects in a distributed cache. Consider this simple code with a Cache containing a mutable value type (Foo):

```
class Foo {
 int field;
}
Foo foo = (Foo) c.get("key").getValue();
foo.field++;
// foo instance is never re-put() to the cache
// ...
```

If the Foo instance is never put back into the Cache your local cache is no longer consistent with the cluster (it is locally modified only). Enabling `copyOnRead` eliminates this possibility since the only way to affect cache values is to call mutator methods on the Cache.

It is worth noting that there is a performance penalty to copyOnRead since values are deserialized on every get().

Consistency Modes

Cache consistency modes are configuration settings and API methods that control the behavior of clustered caches with respect to balancing data consistency and application performance. A cache can be in one of the following consistency modes:

- **Strong** – This mode ensures that data in the cache remains consistent across the cluster at all times. It guarantees that a read gets an updated value only after all write operations to that value are completed, and that each put operation is in a separate transaction. The use of locking and transaction acknowledgments maximizes consistency at *a potentially substantial cost in performance*. This mode is set using the Ehcache configuration file and cannot be changed programmatically (see the `consistency` attribute in `<terracotta>` element).
- **Eventual** – This mode guarantees that data in the cache will eventually be consistent. Read/write performance is substantially boosted at the cost of potentially having an inconsistent cache for brief periods of time. This mode is set using the Ehcache

configuration file and cannot be changed programmatically (see the `consistency` attribute in `<terracotta>` element).

To optimize consistency and performance, consider using eventually consistent caches while selectively using appropriate locking in your application where cluster-wide consistency is critical at all times. Eventual cache operations that are explicitly locked become strongly consistent with respect to each other (but not with respect to non locked operations). For example, a reservation system could be designed with an eventual cache that is only explicitly locked when a customer starts to make a reservation; with this design, eventually consistent data would be available during a customer's search, but during the reservation process, reads and writes would be strongly consistent.

- **Bulk Load** – This mode is optimized for bulk-loading data into the cache without the slowness introduced by locks or regular eviction. It is similar to the eventual mode, but has batching, higher write speeds, and weaker consistency guarantees. This mode is set using the bulk-load API only. For information, see "Bulk Loading" in the *Developer Guide* for BigMemory Max. When turned off, allows the configured consistency mode (either strong or eventual) to take effect again.

Use configuration to set the permanent consistency mode for a cache as required for your application, and the bulk-load mode only during the time when populating (warming) or refreshing the cache.

APIs Related to Consistency

The following APIs and settings also affect consistency:

- **Explicit Locking** – This API provides methods for cluster-wide (application-level) locking on specific elements in a cache. There is guaranteed consistency across the cluster at all times for operations on elements covered by a lock. When used with the strong consistency mode in a cache, *each cache operation* is committed in a single transaction. When used with the eventual consistency mode in a cache, *all cache operations covered by an explicit lock* are committed in a single transaction. While explicit locking of elements provides fine-grained locking, there is still the potential for contention, blocked threads, and increased performance overhead from managing clustered locks. For information, see "Explicit Locking" in the *Developer Guide* for BigMemory Max.
- **Bulk-loading methods** – Bulk-loading Cache methods putAll(), getAll(), and removeAll() provide high-performance and eventual consistency. These can also be used with strong consistency. If you can use them, it's unnecessary to use bulk-load mode.
- **Atomic methods** – To guarantee write consistency at all times and avoid potential race conditions for put operations, use atomic methods. The following Compare and Swap (CAS) operations are available:
  - `cache.replace(Element old, Element new)` — Conditionally replaces the specified key's old value with the new value, if the currently existing value matches the old value.

- `cache.replace(Element)` — Maps the specified key to the specified value, if the key is currently mapped to some value.
- `cache.putIfAbsent(Element)` — Puts the specified key/value pair into the cache only if the key has no currently assigned value. Unlike `put`, which can replace an existing key/value pair, `putIfAbsent` creates the key/value pair only if it is not present in the cache.
- `cache.removeElement(Element)` — Conditionally removes the specified key, if it is mapped to the specified value.

Normally, these methods are used with strong consistency. Unless the property `org.terracotta.clusteredStore.eventual.cas.enabled` is set to "true", these methods throw an UnsupportedOperationException if used with eventual consistency since a race condition cannot be prevented. Other ways to guarantee the return value in eventual consistency are to use the cache decorator StronglyConsistentCacheAccessor, or to use explicit locking. The StronglyConsistentCacheAccessor will use locks with its special substituted versions of the atomic methods. Note that using locks may impact performance. For information about cache decorators and explicit locking, see "Cache Decorators" and "Explicit Locking" in the *Developer Guide for BigMemory Max*.

Configuring Robust Distributed In-Memory Data Sets

Making the BigMemory Max in-memory data system robust is typically a combination of Ehcache configuration and Terracotta configuration and architecture. For more information, see the following documentation:

- **Nonstop caches** – Configure caches to take a specified action after an Ehcache node appears to be disconnected from the cluster. For information, see "Configuring Nonstop Cache" on page 72.
- **Rejoin the cluster** – Allow Ehcache nodes to rejoin the cluster as new clients after being disconnected from the cluster. For information, see the *BigMemory Max High-Availability Guide*.
- **High Availability in a Terracotta cluster** – Configure nodes to ride out network interruptions and long Java garbage collection cycles, connect to a backup Terracotta server, and more. For information, see the *BigMemory Max High-Availability Guide*.
- **Architecture** – Design a cluster that provides failover. For information, see the *BigMemory Max High-Availability Guide*.
- **BigMemory Hybrid** – Extend BigMemory on Terracotta servers with hybrid storage that can include SSD and Flash technologies as well as DRAM. For information, see the *BigMemory Max Administrator Guide*.

Exporting Configuration from the Terracotta Management Console

To create or edit a cache configuration in a live cluster, see the *Terracotta Management Console User Guide*.

To persist custom cache configuration values, create a cache configuration file by exporting customized configuration from the Terracotta Management Console or create a file that conforms to the required format. This file must take the place of any configuration file used when the cluster was last started.

7 Default Settings for a Distributed Configuration

Terracotta Server Array Properties

A Terracotta cluster is composed of clients and servers. Terracotta properties often use a shorthand notation where a client is referred to as "l1" and a server as "l2".

These properties are set at the top of tc-config.xml using a configuration block similar to the following:

```
<tc-properties>
   <property name="l2.nha.tcgroupcomm.reconnect.enabled" value="true" />
<!-- More properties here. -->
</tc-properties>
```

See the *BigMemory Max Administrator Guide* for more information on the Terracotta Server Array.

Reconnection and Logging Properties

The following reconnection properties are shown with default values. These properties can be set to custom values using Terracotta configuration properties (`<tc-properties>/<property>` elements in `tc-config.xml`).

Property	Default Value	Notes
`l2.nha.tcgroupcomm.reconnect.enabled`	true	Enables server-to-server reconnections.
`l2.nha.tcgroupcomm.reconnect.timeout`	5000ms	l2-l2 reconnection timeout.
`l2.l1reconnect.enabled`	true	Enables an l1 to reconnect to servers.
`l2.l1reconnect.timeout.millis`	5000ms	The reconnection time out, after which an l1 disconnects.
`tc.config.getFromSource.timeout`	30000ms	Timeout for getting configuration from a source. For example, this controls how long a client can try to access configuration from a server. If the client fails to do so, it will fail to connect to the cluster.
`logging.maxBackups`	20	Upper limit for number of backups of Terracotta log files.

Property	Default Value	Notes
logging.maxLogFileSize	512MB	Maximum size of Terracotta log files before rolling logging starts.

HealthChecker Tolerances

The following properties control disconnection tolerances between Terracotta servers (l2 <-> l2), Terracotta servers and Terracotta clients (l2 -> l1), and Terracotta clients and Terracotta servers (l1 -> l2).

l2<->l2 GC tolerance : 40 secs, cable pull/network down tolerance : 10secs

```
l2.healthcheck.l2.ping.enabled = true
l2.healthcheck.l2.ping.idletime = 5000
l2.healthcheck.l2.ping.interval = 1000
l2.healthcheck.l2.ping.probes = 3
l2.healthcheck.l2.socketConnect = true
l2.healthcheck.l2.socketConnectTimeout = 5
l2.healthcheck.l2.socketConnectCount = 10
```

l2->l1 GC tolerance : 40 secs, cable pull/network down tolerance : 10secs

```
l2.healthcheck.l1.ping.enabled = true
l2.healthcheck.l1.ping.idletime = 5000
l2.healthcheck.l1.ping.interval = 1000
l2.healthcheck.l1.ping.probes = 3
l2.healthcheck.l1.socketConnect = true
l2.healthcheck.l1.socketConnectTimeout = 5
l2.healthcheck.l1.socketConnectCount = 10
```

l1->l1 GC tolerance : 50 secs, cable pull/network down tolerance : 10secs

```
l1.healthcheck.l2.ping.enabled = true
l1.healthcheck.l2.ping.idletime = 5000
l1.healthcheck.l2.ping.interval = 1000
l1.healthcheck.l2.ping.probes = 3
l1.healthcheck.l2.socketConnect = true
l1.healthcheck.l2.socketConnectTimeout = 5
l1.healthcheck.l2.socketConnectCount = 13
```

Terracotta Client Properties

Client configuration properties typically address the behavior, size, and functionality of in-memory data stores. Others affect certain types of cache-related bulk operations.

See also "How Server Settings Can Override Client Settings" on page 59.

Properties are set in ehcache.xml except as noted.

General Settings

The following default settings affect in-memory data.

Property	Default Value	Notes
value mode	SERIALIZATION	
consistency	EVENTUAL	
XA	false	
orphan eviction	true	
local key cache	false	
synchronous writes	false	
ttl	0	0 means never expire.
tti	0	0 means never expire.
transactional mode	off	
persistence strategy	none	
maxEntriesInCache	0	0 means that the cache will not undergo capacity eviction (but periodic and resource evictions are still allowed)
maxBytesLocalHeap	0	
maxBytesLocalOffHeap	0	
maxEntriesLocalHeap	0	0 means infinite.

NonStop Cache

The following default settings affect the behavior of the cache when while the client is disconnected from the cluster. For more information on these settings, see "Configuring Nonstop Cache" on page 72.

Property	Default Value	Notes
`enable`	false	
`timeout behavior`	exception	
`timeout`	30000ms	
`net.sf.ehcache.nonstop bulkOpsTimeoutMultiply Factor`	10	This value is a timeout multiplication factor affecting bulk operations such as `removeAll()` and `getAll()`. Since the default nonstop timeout is 30 seconds, it sets a timeout of 300 seconds for those operations. The default can be changed programmatically: `cache.getTerracottaConfiguration() .getNonstopConfiguration() .setBulkOpsTimeoutMultiplyFactor(10)`

Bulk Operations

The following properties are shown with default values. These properties can be set to custom values using "Terracotta Server Array Properties" on page 66.

Increasing batch sizes may improve throughput, but could raise latency due to the load on resources from processing larger blocks of data.

Property	Default Value	Notes
`ehcache.bulkOps.maxKBSize`	1MB	Batch size for bulk operations such as putAll and removeAll.
`ehcache.getAll.batchSize`	1000	The number of elements per batch in a getAll operation.
`ehcache.incoherent.putsBatch ByteSize`	5MB	For bulk-loading mode. The minimum size of a batch in a bulk-load operation. Increasing batch sizes may improve throughput, but could raise latency due to the load on

Property	Default Value	Notes
		resources from processing larger blocks of data.
`ehcache.incoherent.putsBatchTimeInMillis`	600 ms	For bulk-loading mode. The maximum time the bulk-load operation takes to batch puts before flushing to the Terracotta Server Array.

8 Configuring Nonstop Operation

About Nonstop Operation

The nonstop feature allows certain operations to proceed on Terracotta clients that have become disconnected from the cluster, or if an operation cannot complete by the nonstop timeout value. This is useful in meeting service-level agreement (SLA) requirements, responding to node failures, and building a more robust High Availability cluster.

One way BigMemory Max can go into nonstop mode is when a client receives a "cluster offline" event. Note that a nonstop instance can go into nonstop mode even if the client is not disconnected, such as when an operation is unable to complete within the timeout allotted by the nonstop configuration. In addition, nonstop instances running in a client that is unable to locate the TSA at startup will initiate nonstop behavior as if the client had disconnected.

Nonstop can be used in conjunction with rejoin. For information about the rejoin feature, see "Configuring Reconnection and Rejoin Properties" in the book *Configuring a Terracotta Cluster for High Availability*.

Use cases include:

- Setting timeouts on operations.

  For example, say you use BigMemory Max rather than a mainframe. The SLA calls for 3 seconds. There is a temporary network interruption that delays the response to a cache request. With the timeout you can return after 3 seconds. The lookup is then done against the mainframe. This could also be useful for write-through, writes to disk, or synchronous writes.
- Automatically responding to cluster topology events to take a pre-configured action.
- Allowing Availability over Consistency within the CAP theorem when a network partition occurs.
- Providing graceful degradation to user applications when distributed BigMemory Max becomes unavailable.

Configuring Nonstop Cache

Nonstop behavior is configured by specifying a `<nonstop>` element in the `<terracotta>` element within a `<cache>` block. In the following example, myCache is configured for nonstop operation:

```
<cache name="myCache" maxEntriesLocalHeap="10000" eternal="false">
 <terracotta>
   <nonstop immediateTimeout="false" timeoutMillis="30000">
     <timeoutBehavior type="noop" />
   </nonstop>
 </terracotta>
</cache>
```

Nonstop is enabled if the <nonstop> element is present. However, if <nonstop> is specified with the attribute enabled="false", nonstop is disabled. The default setting of this optional attribute is enabled="true".

The minimal syntax to enable nonstop is <nonstop />.

If the <nonstop> element is not supplied, then nonstop is disabled.

Nonstop Timeouts and Behaviors

Nonstop caches can be configured with the following attributes:

- enabled – Enables ("true" DEFAULT) or disables ("false") the ability of a cache to execute certain actions after a Terracotta client disconnects. This attribute is optional for enabling nonstop.
- immediateTimeout – Enables ("true") or disables ("false" DEFAULT) an immediate timeout response if the Terracotta client detects a network interruption (the node is disconnected from the cluster). If enabled, this parameter overrides timeoutMillis, so that the option set in timeoutBehavior is in effect immediately.
- timeoutMillis – Specifies the number of milliseconds an application waits for any cache operation to return before timing out. The default value is 30000 (thirty seconds). The behavior after the timeout occurs is determined by timeoutBehavior.
- searchTimeoutMillis – Specifies the number of milliseconds an application waits for search operations to return before timing out. The default value is 30000 (thirty seconds). The behavior after the timeout occurs is determined by timeoutBehavior.

<nonstop> has one self-closing sub-element, <timeoutBehavior>. This sub-element determines the response after a timeout occurs (timeoutMillis expires or an immediate timeout occurs). The response can be set by the <timeoutBehavior>attribute type. This attribute can have one of the values listed in the following table:

Value	Behavior
exception	(DEFAULT) Throw NonStopCacheException. For more information about this exception, see "When is NonStopCacheException Thrown?" on page 75.
noop	Return null for gets. Ignore all other cache operations. Hibernate users may want to use this option to allow their application to continue with an alternative data source.

Value	Behavior
`localReads`	For caches with Terracotta clustering, allow inconsistent reads of cache data. Ignore all other cache operations. For caches without Terracotta clustering, throw an exception.
`localReadsAndExceptionOnWrite`	For caches with Terracotta clustering, allow inconsistent reads of cache data, and throw a NonStopCacheException for write operations. For caches without Terracotta clustering, throw an exception.

Tuning for Nonstop Timeouts and Behaviors

You can tune the default timeout values and behaviors of nonstop caches to fit your environment.

Tuning for Network Interruptions

For example, in an environment with regular network interruptions, consider disabling `immediateTimeout` and increasing `timeoutMillis` to prevent timeouts for most of the interruptions.

For a cluster that experiences regular but short network interruptions, and in which caches clustered with Terracotta carry read-mostly data or there is tolerance of potentially stale data, you may want to set `timeoutBehavior` to `localReads`.

Tuning for Slow Cache Operations

In an environment where cache operations can be slow to return and data is required to always be in sync, increase `timeoutMillis` to prevent frequent timeouts. Set `timeoutBehavior` to `noop` to force the application to get data from another source or `exception` if the application should stop.

For example, a cache.acquireWriteLockOnKey(key) operation may exceed the nonstop timeout while waiting for a lock. This would trigger nonstop mode only because the lock couldn't be acquired in time. Using cache.tryWriteLockOnKey(key, timeout), with the method's timeout set to less than the nonstop timeout, avoids this problem.

Bulk Loading

If a nonstop cache is bulk-loaded using the Bulk-Load API, a multiplier is applied to the configured nonstop timeout whenever the method net.sf.ehcache.Ehcache.setNodeBulkLoadEnabled(boolean) is used. The

default value of the multiplier is 10. You can tune the multiplier using the `bulkOpsTimeoutMultiplyFactor` system property:

```
-Dnet.sf.ehcache.nonstop.bulkOpsTimeoutMultiplyFactor=10
```

Note that when nonstop is enabled, the cache size displayed in the Terracotta Management Console is subject to the `bulkOpsTimeoutMultiplyFactor`. Increasing this multiplier on the clients can facilitate more accurate size reporting.

This multiplier also affects the methods:

net.sf.ehcache.Ehcache.getAll()

net.sf.ehcache.Ehcache.removeAll()

net.sf.ehcache.Ehcache.removeAll(boolean)

Printing Stack Traces on Exceptions

You can turn on additional logging (at the INFO level) for nonstop timeouts using:

```
-Dnet.sf.ehcache.nonstop.printStackTraceOnException=true
```

This is a dynamic property that can also be managed programmatically:

```
System.setProperty(PRINT_STACK_TRACE_ON_EXCEPTION_PROPERTY, "true")
```

This property logs the stack trace for the nonstop thread. Note that, with this property set, log files can become very large in environments in which a large number of timeouts occur.

Nonstop Exceptions

Typically, application code may access the cache frequently and at various points. Therefore, with a nonstop cache, where your application could encounter NonStopCacheExceptions is difficult to predict. The following sections provide guidance on when to expect NonStopCacheExceptions and how to handle them.

When is NonStopCacheException Thrown?

NonStopCacheException is usually thrown when it is the configured behavior for a nonstop cache in a client that disconnects from the cluster. In the following example, the exception would be thrown 30 seconds after the disconnection (or the "cluster offline" event is received):

```
<nonstop immediateTimeout="false" timeoutMillis="30000">
<timeoutBehavior type="exception" />
</nonstop>
```

However, under certain circumstances the NonStopCache exception can be thrown even if a nonstop cache's timeout behavior is *not* set to throw the exception. This can happen when the cache goes into nonstop mode during an attempt to acquire or release a lock. These lock operations are associated with certain lock APIs and special cache types such as Explicit Locking, BlockingCache, SelfPopulatingCache, and UpdatingSelfPopulatingCache.

A NonStopCacheException can also be thrown if the cache must fault in an element to satisfy a get() operation. If the Terracotta Server Array cannot respond within the configured nonstop timeout, the exception is thrown.

A related exception, InvalidLockAfterRejoinException, can be thrown during or after client rejoin. This exception occurs when an unlock operation takes place on a lock obtained *before* the rejoin attempt completed. For more information about rejoin, see "Using Rejoin to Automatically Reconnect Terracotta Clients" in the *BigMemory Max High-Availability Guide*.

Using try-finally Blocks to Release Locks

To ensure that locks are released properly, application code using Ehcache lock APIs should encapsulate lock-unlock operations with try-finally blocks:

```
    myLock.acquireLock();
    try {
      // Do some work.
    } finally {
      myLock.unlock();
    }
```

Handling Nonstop Exceptions

Your application can handle nonstop exceptions in the same way it handles other exceptions. For nonstop caches, an unhandled-exceptions handler could, for example, refer to a separate thread any cleanup needed to manage the problem effectively.

Another way to handle nonstop exceptions is by using a dedicated Ehcache decorator that manages the exception outside of the application framework. The following is an example of how the decorator might operate:

```
try { cache.put(element); }
catch(NonStopCacheException e) {
  handler.handleException(cache, element, e);
}
```

9 Working with Transactions

About Transactional Caches

Transactional caches add a level of safety to cached data and ensure that the cached data and external data stores are in sync. Distributed caches can support Java Transaction API (JTA) transactions as an XA resource. This is useful in JTA applications requiring caching, or where cached data is critical and must be persisted and remain consistent with System of Record data.

However, transactional caches are slower than non-transactional caches due to the overhead from having to write transactionally. Transactional caches also have the following restrictions:

- Data can be accessed only transactionally, even for read-only purposes. You must encapsulate data access with begin() and commit() statements. This may not be necessary under certain circumstances (see, for example, the discussion on Spring in "Transaction Support" in the *Developer Guide* for BigMemory Max.
- `copyOnRead` and `copyOnWrite` must be enabled. These `<cache>` attributes are "false" by default and must set to "true".
- Caches must be strongly consistent. A transactional cache's `consistency` attribute must be set to "strong".
- Nonstop caches cannot be made transactional except in strict mode (xa_strict). Transactional caches in other modes must *not* contain the `<nonstop>` sub-element.
- Objects stored in a transactional cache must override `equals()` and `hashCode()`. If overriding `equals()` and `hashCode()` is not possible, see "Implementing an Element Comparator" on page 83.
- Caches can be dynamically changed to bulk-load mode, but any attempt to perform a transaction when this is the case will throw a `CacheException`.

For more information about transactional caches, see "Transaction Support" in the *Developer Guide* for BigMemory Max.

You can choose one of three different modes for transactional caches:

- **Strict XA** – Has full support for XA transactions. May not be compatible with transaction managers that do not fully support JTA.
- **XA**XA – Has support for the most common JTA components, so likely to be compatible with most transaction managers. But unlike strict XA, may fall out of sync with a database after a failure (has no recovery). Integrity of cache data, however, is preserved.
- **Local** – Local transactions written to a local store and likely to be faster than the other transaction modes. This mode does not require a transaction manager and does not synchronize with remote data sources. Integrity of cache data is preserved in case of failure.

Note: Both the XA and local mode write to the underlying store synchronously and using pessimistic locking. Under certain circumstances, this can result in a deadlock, which generates a DeadLockException after a transaction times out and a commit fails. Your application should catch DeadLockException (or TransactionException) and call `rollback()`. Deadlocks can have a severe impact on performance. A high number of deadlocks indicates a need to refactor application code to prevent races between concurrent threads attempting to update the same data.

Strict XA (Support for All JTA Components)

Note that Ehcache as an XA resource:

- Has an isolation level of ReadCommitted.
- Updates the underlying store asynchronously, potentially creating update conflicts. With this optimistic locking approach, Ehcache might force the transaction manager to roll back the entire transaction if a `commit()` generates a RollbackException (indicating a conflict).
- Can work alongside other resources such as JDBC or JMS resources.
- Guarantees that its data is always synchronized with other XA resources.
- Can be configured on a per-cache basis (transactional and non-transactional caches can exist in the same configuration).
- Automatically performs enlistment.
- Can be used standalone or integrated with frameworks such as Hibernate.
- Is tested with the most common transaction managers by Atomikos, Bitronix, JBoss, WebLogic, and others.

Configuration

To configure a cache as an XA resource able to participate in JTA transactions, the following `<cache>` attributes must be set as shown:

- transactionalMode="xa_strict"
- copyOnRead="true"
- copyOnWrite="true"

In addition, the `<cache>` sub-element `<terracotta>` must not have clustering disabled.

For example, the following cache is configured for JTA transactions with strict XA:

```
<cache name="com.my.package.Foo"
     maxEntriesLocalHeap="500"
     eternal="false"
     copyOnRead="true"
     copyOnWrite="true"
     consistency="strong"
```

```
     transactionalMode="xa_strict">
   <persistence strategy="distributed"/>
   <terracotta />
</cache>
```

Any other XA resource that could be involved in the transaction, such as a database, must also be configured to be XA compliant.

Usage

Your application can directly use a transactional cache in transactions. This usage must occur after the transaction manager has been set to start a new transaction and before it has ended the transaction.

For example:

```
...
myTransactionMan.begin();
Cache fooCache = cacheManager.getCache("Foo");
fooCache.put("1", "Bar");
myTransactionMan.commit();
...
```

If more than one transaction writes to a cache, it is possible for an XA transaction to fail. See "Avoiding XA Commit Failures With Atomic Methods" on page 82.

Setting Up Transactional Caches in Hibernate {#45557}

If your application is using JTA, you can set up transactional caches in a second-level cache with Ehcache for Hibernate. To do so, ensure the following:

Ehcache

- You are using Ehcache 2.1.0 or higher.
- The attribute `transactionalMode` is set to "xa" or "xa-strict".
- The cache is clustered (the <cache> element has the sub-element <terracotta clustered="true">). For example, the following cache is configured to be transactional:

  ```
  <cache name="com.my.package.Foo"
       ...
       transactionalMode="xa"> <terracotta /> </cache>
  ```

- The cache UpdateTimestampsCache is not configured to be transactional. Hibernate updates to `org.hibernate.cache.UpdateTimestampsCache` prevent it from being able to participate in XA transactions.

Hibernate

- You are using Hibernate 3.3.
- The factory class used for the second-level cache is `net.sf.ehcache.hibernate.EhCacheRegionFactory`.
- Query cache is turned off.

- The value of `current_session_context_class` is `jta`.
- The value of `transaction.manager_lookup_class` is the name of a TransactionManagerLookup class (see your Transaction Manager).
- The value of `transaction.factory_class` is the name of a TransactionFactory class to use with the Hibernate Transaction API.
- The cache concurrency strategy is set to TRANSACTIONAL. For example, to set the cache concurrency strategy for `com.my.package.Foo` in `hibernate.cfg.xml`:

  ```
  <class-cache class="com.my.package.Foo" usage="transactional"/>
  ```

  Or in a Hibernate mapping file (hbm file):

  ```
  <cache usage="transactional"/>
  ```

  Or using annotations:

  ```
  @Cache(usage=CacheConcurrencyStrategy.TRANSACTIONAL)
  public class Foo {...}
  ```

Important: WARNING: Use the TRANSACTIONAL concurrency strategy with transactional caches only. Using with other types of caches will cause errors.

XA (Basic JTA Support)

Transactional caches set to "xa" provide support for basic JTA operations. Configuring and using XA does not differ from using local transactions, except that "xa" mode requires a transaction manager and allows the cache to participate in JTA transactions.

Note: When using XA with an Atomikos transaction Manager, be sure to set `com.atomikos.icatch.threaded_2pc=false` in the Atomikos configuration. This helps prevent unintended rollbacks due to a bug in the way Atomikos behaves under certain conditions

For example, the following cache is configured for JTA transactions with XA:

```
<cache name="com.my.package.Foo"
     maxEntriesLocalHeap="500"
     eternal="false"
     copyOnRead="true"
     copyOnWrite="true"
     consistency="strong"
     transactionalMode="xa">
   <persistence strategy="distributed"/>
   <terracotta />
</cache>
```

Any other XA resource that could be involved in the transaction, such as a database, must also be configured to be XA compliant.

Local Transactions

Local transactional caches (with the `transactionalMode` attribute set to "local") write to a local store using an API that is part of the Ehcache core API. Local transactions have the following characteristics:

- Recovery occurs at the time an element is accessed.
- Updates are written to the underlying store immediately.
- Get operations on the underlying store may block during commit operations.

To use local transactions, instantiate a TransactionController instance instead of a transaction manager instance:

```
TransactionController txCtrl = cacheManager.getTransactionController();
...
txCtrl.begin();
Cache fooCache = cacheManager.getCache("Foo");
fooCache.put("1", "Bar");
txCtrl.commit();
...
```

You can use `rollback()` to roll back the transaction bound to the current thread.

Tip: You can find out if a transaction is in process on the current thread by calling `TransactionController.getCurrentTransactionContext()` and checking its return value. If the value isn't null, a transaction has started on the current thread.

Commit Failures and Timeouts

Commit operations can fail if the transaction times out. If the default timeout requires tuning, you can get and set its current value:

```
int currentDefaultTransactionTimeout = txCtrl.getDefaultTransactionTimeout();
...
txCtrl.setDefaultTransactionTimeout(30); // in seconds -- must be greater than zero.
```

You can also bypass the commit timeout using the following version of `commit()`:

```
txCtrl.commit(true); // "true" forces the commit to ignore the timeout.
```

Avoiding XA Commit Failures With Atomic Methods

If more than one transaction writes to a cache, it is possible for an XA transaction to fail. In the following example, if a second transaction writes to the same key ("1") and completes its commit first, the commit in the example may fail:

```
...
myTransactionMan.begin();
Cache fooCache = cacheManager.getCache("Foo");
fooCache.put("1", "Bar");
myTransactionMan.commit();
```

```
...
```

One approach to prevent this type of commit failure is to use one of the atomic put methods, such as `Cache.replace()`:

```
myTransactionMan.begin();
int val = cache.get(key).getValue();  // "cache" is configured to be transactional.
Element olde = new Element (key, val);
// True only if the element was successfully replaced.
if (cache.replace(olde, new Element(key, val + 1)) {
 myTransactionMan.commit();
}
else { myTransactionMan.rollback(); }
```

Another useful atomic put method is `Cache.putIfAbsent(Element element)`, which returns null on success (no previous element exists with the new element's key) or returns the existing element (the put is not executed). Atomic methods cannot be used with null elements, or elements with null keys.

Implementing an Element Comparator

For all transactional caches, the atomic methods `Cache.removeElement(Element element)` and `Cache.replace(Element old, Element element)` must compare elements for the atomic operation to complete. This requires all objects stored in the cache to override `equals()` and `hashCode()`.

If overriding these methods is not desirable for your application, a default comparator is used (`net.sf.echache.store.DefaultElementValueComparator`). You can also implement a custom comparator and specify it in the cache configuration with <elementValueComparator>:

```
<cache name="com.my.package.Foo"
     maxEntriesLocalHeap="500"
     eternal="false"
     copyOnRead="true"
     copyOnWrite="true"
     consistency="strong"
     transactionalMode="xa">
   <elementValueComparator class="com.company.xyz.MyElementComparator" />
   <persistence strategy="distributed"/>
   <terracotta />
</cache>
```

Custom comparators must implement `net.sf.ehcache.store.ElementValueComparator`.

A comparator can also be specified programmatically.

10 Working with OSGi

Working With OSGi

To allow Enterprise Ehcache to behave as an OSGi component, the following attributes should be set as shown:

```
<cache ... copyOnRead="true" ... >
...
  <terracotta ... clustered="true" ... />
...
</cache>
```

Your OSGi bundle will require the following JAR files (showing versions from a BigMemory Max 4.0.0):

- `ehcache-ee-2.7.0.jar`
- `terracotta-toolkit-runtime-ee-4.0.0.jar`
- `slf4j-api-1.6.6.jar`
- `slf4j-nop-1.6.1.jar`

  Or use another appropriate logger binding.

Use the following directory structure:

```
-- net.sf.ehcache
         |
         | - ehcache.xml
         |- ehcache-ee-2.7.0.jar
     |
     |- terracotta-toolkit-runtime-ee-4.0.0.jar
         |
         | - slf4j-api-1.6.6.jar
         |
         | - slf4j-nop-1.6.6.jar
         |
         | - META-INF/
             | - MANIFEST.MF
```

The following is an example manifest file:

```
Manifest-Version: 1.0
 Export-Package: net.sf.ehcache;version="2.7.0"
 Bundle-Vendor: Terracotta
 Bundle-ClassPath: .,ehcache-ee-2.7.0.jar,terracotta-toolkit-runtime-ee-4.0.0.jar
     ,slf4j-api-1.6.6.jar,slf4j-nop-1.6.6.jar
 Bundle-Version: 2.7.0
 Bundle-Name: EHCache bundle
 Created-By: 1.6.0_15 (Apple Inc.)
 Bundle-ManifestVersion: 2
 Import-Package: org.osgi.framework;version="1.3.0"
 Bundle-SymbolicName: net.sf.ehcache
 Bundle-RequiredExecutionEnvironment: J2SE-1.5
```

Use versions appropriate to your setup.

To create the bundle, execute the following command in the `net.sf.ehcache` directory:

```
jar cvfm net.sf.ehcache.jar MANIFEST.MF *
```

11 Working with VMware vMotion

Working with VMware vMotion

Software AG recommends against allowing live migrations of virtual machine(s) with Software AG Terracotta server processes. This is because of the downtime introduced during the live migration process which can cause other Terracotta server processes to elect a new active server which will then create data consistency problems when the migrated process becomes live again.

If you must use vMotion in your environment, please make sure that Terracotta server processes and runtime processes are excluded from live migration.

A System Properties

Special System Properties

net.sf.ehcache.disabled

Setting this system property to `true` (using `java -Dnet.sf.ehcache.disabled=true` in the Java command line) disables caching in ehcache. If disabled, no elements can be added to a cache (puts are silently discarded).

net.sf.ehcache.use.classic.lru

When LRU is selected as the eviction policy, set this system property to `true` (using `java -Dnet.sf.ehcache.use.classic.lru=true` in the Java command line) to use the older LruMemoryStore implementation. This is provided for ease of migration.