About BigMemory Go

Innovation Release

Version 4.3.5

April 2018

TERRACOTTA

This document applies to BigMemory Go Version 4.3.5 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: BMG-ABMG-435-20180417

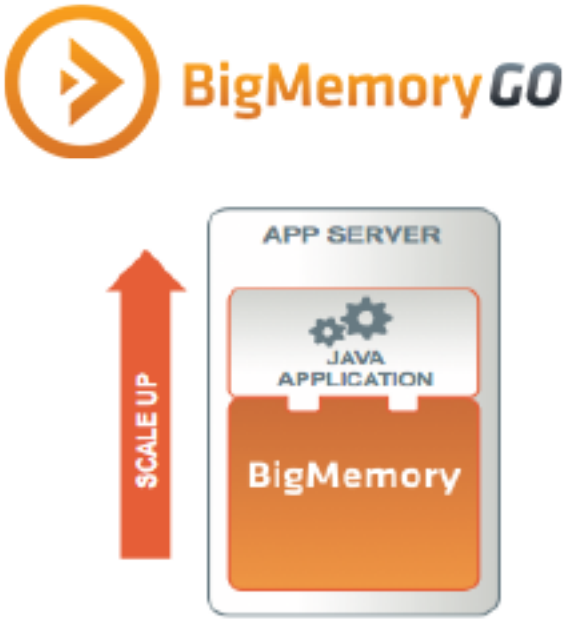
Table of Contents

1 BigMemory Go Overview

What is BigMemory Go?

BigMemory Go is a commercial extension of Ehcache, the open-source, standards-based, and most widely used Java-based caching program for boosting performance, offloading your database, and simplifying scalability.

BigMemory Go provides all of the features of Ehcache, while also enabling access to off-heap memory for caching and in-memory data storage. The use of off-heap memory enables Java applications to leverage virtually all the RAM for in-memory data storage without lengthy pauses for garbage collection.



BigMemory Go also enables you to perform SQL queries on the data in your cache or in-memory data store.

As with Ehcache, you can use BigMemory as a general-purpose cache/in-memory data store or a second-level cache for Hibernate. You can additionally integrate it with third-party products such as ColdFusion, Google App Engine, and Spring.

Basic Terms

Cache

Wiktionary defines a cache as "a store of things that will be required in the future, and can be retrieved rapidly." A cache is a collection of temporary data that either duplicates data located elsewhere or is the result of a computation. Data that is already in the cache can be repeatedly accessed with minimal costs in terms of time and resources.

Cache hit

When a data element is requested from cache and the element exists for the given key, it is referred to as a *cache hit* (or simply, "a hit").

Cache miss

When a data element is requested from cache and the element does not exist for the given key, it is referred to as a *cache miss* (or simply, "a miss").

System-of-Record

The authoritative source of truth for the data. The cache acts as a local copy of data retrieved from or stored to the system-of-record (SOR). The SOR is often a traditional database, although it might be a specialized file system or some other reliable long-term storage. For the purposes of using Ehcache, the SOR is assumed to be a database.

2 Topologies

Topology Types

- **Standalone** – The data set is held in the application node. Any other application nodes are independent with no communication between them. If a standalone topology is used where there are multiple application nodes running the same application, then there is Weak Consistency between them. They contain consistent values for immutable data or after the time-to-live on an element has completed and the element needs to be reloaded.
- **Distributed** – The data is held in a remote server (or array of servers) with a subset of recently used data held in each application node. This topology offers a rich set of consistency options.

  A distributed topology is the recommended approach in a clustered or scaled-out application environment. It provides the highest level of performance, availability, and scalability. *The distributed topology is available only with BigMemory Max.*

Many production applications are deployed in clusters of multiple instances for availability and scalability. Without a distributed topology, application clusters will experience *data drift*, meaning that updates made to the data by one application does not appear in the other instances. This also happens to web session data. Using a distributed topology ensures that the data for all the application instances is kept in sync.

3 Storage Options

Storage Tiers

You can divide a cache or in-memory data set across the following storage areas, referred to as *tiers*:

- **MemoryStore** – On-heap memory used to hold cache elements. This tier is subject to Java garbage collection.
- **OffHeapStore** – Provides overflow capacity to the MemoryStore. Limited in size only by available RAM. Not subject to Java garbage collection (GC). *Available only with Terracotta BigMemory products.*
- **DiskStore** – Backs up in-memory cache elements and provides overflow capacity to the other tiers.

MemoryStore

The memory store is always enabled and exists in heap memory. It has the following characteristics:

- It accepts all data, whether serializable or not.
- It is the fastest storage option.
- Is thread safe for use by multiple concurrent threads.

If you use OffHeapStore (available with the BigMemory products only), MemoryStore holds a copy of the hottest subset of data from the OffHeapStore.

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.

If overflow is enabled, a check for expiry is carried out. If it is expired it is deleted; if not it is spooled.

For information about sizing and configuring the MemoryStore, see "Configuring Memory Store" in the *BigMemory Go Configuration Guide*.

OffHeapStore

The OffHeapStore extends a cache to memory outside the of the Java heap. This store, which is not subject to Java garbage collection (GC), is limited only by the amount of RAM available. Using OffHeapStore, you can create extremely large local caches. *OffHeapStore is only available with the Terracotta BigMemory products.*

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

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

For the best performance, you should allocate to a cache as much heap memory as possible without triggering GC pauses. Then, use the OffHeapStore to hold the data that cannot fit in heap (without causing GC pauses).

For information about sizing and configuring OffHeapStore, see "Configuring OffHeapStore" in the *Configuration Guide* for your BigMemory product.

DiskStore

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

Note: The DiskStore tier is available only for local (standalone) instances of cache. When you use a distributed cache (available only in BigMemory Max), a Terracotta Server Array is used instead of a disk tier.

Only data that is Serializable can be placed in the DiskStore. 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. Be aware that serialization speed is affected by the size of the objects being serialized and their type. For example, it has been shown 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 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.

For more information about configuring and sizing the DiskStore, see "Configuring Fast Restart" in the *Configuration Guide* for your BigMemory Product.

4 Automatic Resource Control

Automatic Resource Control

Automatic Resource Control (ARC) gives you fine-grained controls for tuning performance and enabling trade-offs between throughput, latency and data access. Independently adjustable configuration parameters include differentiated tier-based sizing and pinning hot or eternal data in the most effective tier.

ARC offers a wealth of benefits, including:

- Sizing limitations on in-memory caches to avoid OutOfMemory errors
- Pooled sizing – no requirement to size caches individually
- Differentiated tier-based sizing for flexibility
- Sizing by bytes, entries, or percentages for more flexibility

Dynamically Sizing Stores

Tuning often involves sizing stores appropriately. There are a number of ways to size the different BigMemory Go storage tiers using simple configuration sizing attributes. For information about how to tune tier sizing by configuring dynamic allocation of memory and automatic balancing, see "Sizing Storage Tiers" in the *Configuration Guide* for BigMemory Go.

Pinning Data

One of the most important aspects of running an in-memory data store involves managing the life of the data in each tier. For more information about managing life of data in a tier using pinning, expiration, and eviction, see "Managing Data Life" in the *Configuration Guide* for BigMemory Go.