About Ehcache

Version 2.10.3

October 2016

EHCACHE

This document applies to Ehcache Version 2.10.3 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: EHC-AE-2103-20161018

Table of Contents

1 Ehcache Overview

What is Ehcache?

Ehcache is an open-source, standards-based cache for boosting performance, offloading your database, and simplifying scalability. As a robust, proven, and full-featured solution, it is today's most widely used Java-based cache. You can use Ehcache as a general-purpose cache or a second-level cache for Hibernate. You can additionally integrate it with third-party products such as ColdFusion, Google App Engine, and Spring.

Ehcache provides in-process cache, which you can replicate across multiple nodes. It is also at the core of BigMemory Go and BigMemory Max, Terracotta's commercial caching and in-memory data-storage products. The Terracotta Server Array provided with BigMemory Max enables mixed in-process/out-of-process configurations with terabyte-size caches. For information about Terracotta's BigMemory offerings, see the BigMemory Go and BigMemory Max product documentation at http://terracotta.org/documentation.

Note: BigMemory Max is available in trial and full versions. The open-source version of the Terracotta Server Array can be found at http://www.terracotta.org/downloads/open-source/catalog.

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.

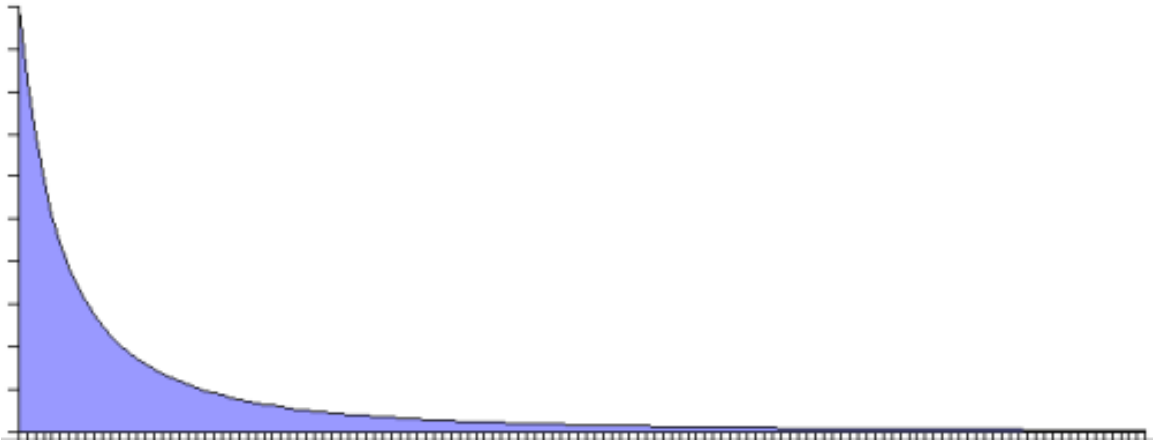
2 Why Caching Works

Locality of Reference

While Ehcache concerns itself with Java objects, caching is used throughout computing, from CPU caches to the Internet Domain Name System (DNS) system. Why? Because many computer systems exhibit "locality of reference." Data that is near other data or has recently been used is more likely to be used again.

The Long Tail

Chris Anderson, of Wired Magazine, coined the term "the long tail" to refer to cases in e-commerce systems where a small number of items can make up the bulk of sales (or a small number of blogs can get the most hits) and there is a long "tail" of less popular items.



The long tail is an example of a power -law probability distribution, such as the Pareto distribution or 80:20 rule. If 20% of objects are used 80% of the time and a way can be found to reduce the cost of obtaining that 20%, system performance will improve.

Will an Application Benefit from Caching?

Often the answer is yes, especially if the application is I/O bound. If an application is I/O bound, it depends on the rate at which data can be obtained. If it is CPU bound, then the time taken principally depends on the speed of the CPU and main memory. Caching can improve performance and also reduce the load on a web server.

Speeding up CPU-bound Applications

CPU bound applications are often sped up by:

- Improving algorithm performance
- Parallelizing the computations across multiple CPUs (SMP) or multiple machines (clusters).

- Upgrading the CPU speed.

A cache can temporarily store computations for reuse, including but not limited to:

- Large web pages that have a high rendering cost
- Authentication status, where authentication requires cryptographic transforms

Speeding up I/O-bound Applications

Many applications are I/O bound, either by disk or network operations. In the case of databases they can be limited by both.

Network operations can be bound by a number of factors:

- Time to set up and tear down connections
- Latency, or the minimum round trip time
- Throughput limits
- Overhead for marshalling and unmarshalling

The caching of data can often help significantly with I/O bound applications. For example, you might use Ehcache for:

- Data Access Object caching for Hibernate
- Web page caching, for pages generated from databases

Increased Application Scalability

The corollary to increased performance is increased scalability. Suppose you have a database that can perform up to 100 expensive queries per second. Beyond that threshold, the database backs up and if addition connections occur, the database slowly dies.

In this case, caching is likely to reduce the workload. If caching can cause 90% of those 100 queries to be cache hits and not impact the database, the database can scale 10 times higher.

How Much Will an Application Speed up with Caching?

In applications that are I/O bound, which is most business applications, most of the response time is getting data from a database. In a system where each piece of data is used only one time, there is no benefit. In a system where a high proportion of the data is reused, the speed up is significant.

Applying Amdahl's Law

Amdahl's law finds the total system speedup from a speedup in part of the system.

```
1 / ((1 - Proportion Sped Up) + Proportion Sped Up / Speed up)
```

The following examples show how to apply Amdahl's law to common situations. In the interests of simplicity, we assume:

- A single server.
- A system with a single thing in it, which when cached, gets 100% cache hits and lives forever.

Persistent Object Relational Caching

A Hibernate Session.load() for a single object is about 1000 times faster from cache than from a database. .

A typical Hibernate query will return a list of IDs from the database, and then attempt to load each. If Session.iterate() is used, Hibernate goes back to the database to load each object.

Imagine a scenario where we execute a query against the database that returns a hundred IDs and then loads each one. The query takes 20% of the time and the round trip loading takes the rest (80%). The database query itself is 75% of the time that the operation takes. The proportion being sped up is thus 60% (75% * 80%).

The expected system speedup is thus:

```
1 / ((1 - .6) + .6 / 1000)
= 1 / (.4 + .0006)
= 2.5 times system speedup
```

Web Page Caching

An observed speed up from caching a web page is 1000 times. Ehcache can retrieve a page from its SimplePageCachingFilter in a few milliseconds.

Because the web page is the result of a computation, it has a proportion of 100%.

The expected system speedup is thus:

```
1 / ((1 - 1) + 1 / 1000)
 = 1 / (0 + .0001)
 = 1000 times system speedup
```

Web Page Fragment Caching

Caching the entire page is a big win. Sometimes the liveness requirements vary in different parts of the page. Here the SimplePageFragmentCachingFilter can be used.

Let's say we have a 1000 fold improvement on a page fragment that takes 40% of the page render time.

The expected system speedup is thus:

```
1 / ((1 - .4) + .4 / 1000)
 = 1 / (.6 + .0004)
 = 1.6 times system speedup
```

Cache Efficiency

Cache entries do not live forever. Some examples that come close are:

- Static web pages or web page fragments, like page footers.
- Database reference data, such as the currencies in the world.

Factors that affect the efficiency of a cache are:

- **Liveliness** — How live the data needs to be. High liveliness means that the data can change frequently, so the value in cache will soon be out of date. Low liveliness means that the data changes only rarely, so the value in cache will often match the current real value of the non-cached data. In general, the less live an item of data is, the more it can be cached.
- **Proportion of data cached** — What proportion of the data can fit into the resource limits of the machine. For 32-bit Java systems, there was a hard limit of 2 GB of address space. 64-bit systems do not have that constraint, but garbage collection issues often make it impractical to have the Java heap be large. Various eviction algorithms are used to evict excess entries.
- **Shape of the usage distribution** — If only 300 out of 3000 entries can be cached, but the Pareto (80/20 rule) distribution applies, it might be that 80% of the time, those 300 will be the ones requested. This drives up the average request lifespan.
- **Read/Write ratio** — The proportion of times data is read compared with how often it is written. Things such as the number of empty rooms in a hotel change often, and will be written to frequently. However the details of a room, such as number of beds, are immutable, and therefore a maximum write of 1 might have thousands of reads.

Ehcache keeps these statistics for each cache and each element, so they can be measured directly rather than estimated.

Cluster Efficiency

Assume a round-robin load balancer where each hit goes to the next server. The cache has one entry which has a variable lifespan of requests, say caused by a time to live (TTL) setting. The following table shows how that lifespan can affect hits and misses.

```
Server 1    Server 2   Server 3    Server 4
 M             M          M           M
 H             H          H           H
 H             H          H           H
 H             H          H           H
 H             H          H           H
 ...           ...        ...         ...
```

The cache hit ratios for the system as a whole are as follows:

```
Entry
Lifespan  Hit Ratio   Hit Ratio  Hit Ratio   Hit Ratio
in Hits   1 Server    2 Servers  3 Servers   4 Servers
2          1/2           0/2        0/2         0/2
4          3/4           2/4        1/4         0/4
10         9/10          8/10       7/10        6/10
20         19/20         18/20      17/20       16/10
50         49/50         48/50      47/20       46/50
```

The efficiency of a cluster of standalone caches is generally:

```
(Lifespan in requests - Number of Standalone Caches) / Lifespan in requests
```

Where the lifespan is large relative to the number of standalone caches, cache efficiency is not much affected. However when the lifespan is short, cache efficiency is dramatically affected. This problem can be solved using the distributed caching capability provided in Terracotta BigMemory Max. With distributed cache, entries put into a local cache are propagated to other servers in the cluster.

A Cache Version of Amdahl's Law

Applying Amdahl's law to caching, we now have:

```
1 / ((1 - Proportion Sped Up * effective cache efficiency) +
(Proportion Sped Up  * effective cache efficiency)/ Speed up)
where:
effective cache efficiency = (cache efficiency) * (cluster efficiency)
```

Web Page Example

Applying this formula to the earlier web page cache example where we have cache efficiency of 35% and average request lifespan of 10 requests and two servers:

```
 cache efficiency = .35
 cluster efficiency = .(10 - 1) / 10
                    = .9
 effective cache efficiency = .35 * .9
                            = .315
system speedup:
    1 / ((1 - 1 * .315) + 1 * .315 / 1000)
    = 1 / (.685 + .000315)
    = 1.45 times system speedup
```

If the cache efficiency is 70% (two servers):

```
 cache efficiency = .70
 cluster efficiency = .(10 - 1) / 10
                    = .9
 effective cache efficiency = .70 * .9
                            = .63
system speedup:
    1 / ((1 - 1 * .63) + 1 * .63 / 1000)
    = 1 / (.37 + .00063)
    = 2.69 times system speedup
```

If the cache efficiency is 90% (two servers):

```
 cache efficiency = .90
 cluster efficiency = .(10 - 1) / 10
                    = .9
 effective cache efficiency = .9 * .9
                            = .81
system speedup:
    1 / ((1 - 1 * .81) + 1 * .81 / 1000)
    = 1 / (.19 + .00081)
    = 5.24 times system speedup
```

The benefit is dramatic because Amdahl’s law is most sensitive to the proportion of the system that is sped up.

3 Caching 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.*
- **Replicated** – The cached data set is held in each application node and data is copied or invalidated across the nodes without locking. Replication can be either asynchronous or synchronous, where the writing thread blocks while propagation occurs. The only consistency mode supported in this topology is Weak Consistency.

Many production applications are deployed in clusters of multiple instances for availability and scalability. However, without a distributed or replicated cache, application clusters exhibit a number of undesirable behaviors, such as:

- **Cache Drift** - If each application instance maintains its own cache, updates made to one cache will not appear in the other instances. This also happens to web session data. A distributed or replicated cache ensures that all of the cache instances are kept in sync with each other.
- **Database Bottlenecks** - In a single-instance application, a cache effectively shields a database from the overhead of redundant queries. However, in a distributed application environment, each instance must load and keep its own cache fresh. The overhead of loading and refreshing multiple caches leads to database bottlenecks as more application instances are added. A distributed or replicated cache eliminates the per-instance overhead of loading and refreshing multiple caches from a database.

4 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 *Ehcache 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 the "Persistence and Restartability" section in the *Ehcache Configuration Guide*.

5 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 Ehcache 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 Ehcache.

Pinning Data

One of the most important aspects of running a cache 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 Ehcache.