WHITE PAPER

# Caching Strategies Explained


## By Christoph Engelbert

Technical Evangelist
Senior Solutions Architect
Hazelcast

# Caching Strategies

This document aims to describe different strategies for application caching strategies. It will explain the advantages and disadvantages, and when to apply the appropriate strategy.

Additionally, it will give a short introduction to JCache, the standard Java Caching API, as well as insight into the characteristics of Hazelcast's JCache implementation and how it helps to integrate the different caching strategies into your application landscape.

This white paper is designed as a general overview of caching, its purpose, and use cases. While this paper is not specialized to Hazelcast, it uses Hazelcast in its examples.

# Caching Strategies

## TABLE OF CONTENTS

# 1. What is a Cache?

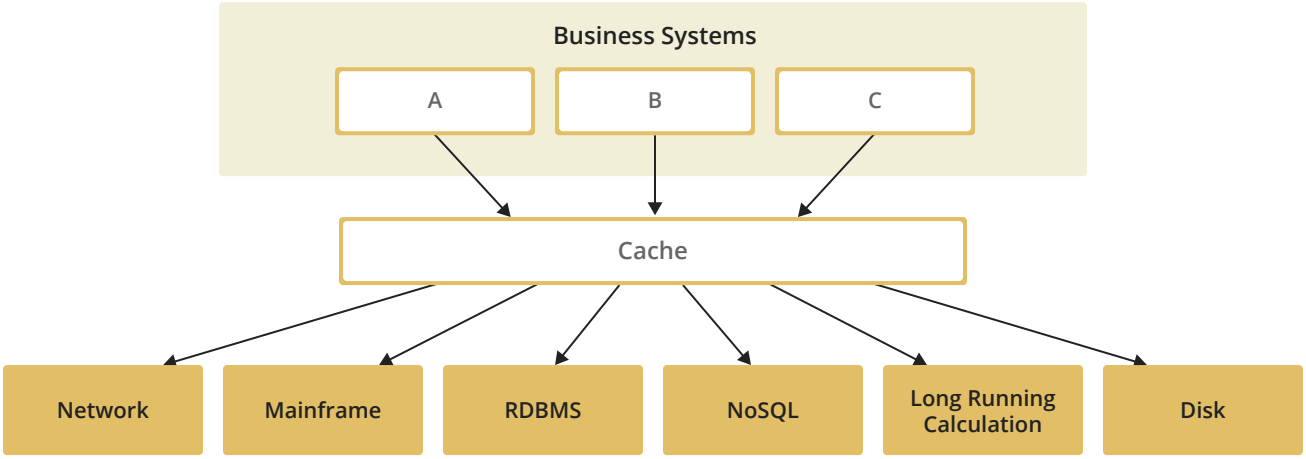
In computing, cache is a component to store portions of datasets which would otherwise either take a long time to calculate / process or originate from another underlying backend system, where caching is used to prevent additional request round trips for frequently used data. In both cases, caching could be used to gain performance or decrease application latencies.

A *cache hit* happens whenever data is already available in the cache and can be returned without any other operation, otherwise the cache responds with a *cache miss* or, if available, can transparently retrieve the value from another underlying backend system and cache it before returning it to the requestor.

Caches are designed to respond to cache requests in near real-time and therefore are implemented as simple key-value stores. The underlying data structure, however, can still be different and depends on the storage backend. In addition, caches can most often be used by multiple front end consumers such as web applications.

Caches can have multiple levels, so-called tiered storages, which are ordered by their speed factor. Frequently or recently used data are typically held in memory, whereas other data (depending on the tiered storage implementations) can be written to SSD, slower spinning disk systems and later on other even slower systems, or can be evicted completely - if reproducible.

Typical use cases are in-memory caches for database, slowly retrievable disk-based data, data stored remotely behind slow network connections or results of previous calculations.



# 2. Introducing Caching

This section will give an introduction into why caching sometimes is necessary, what to cache and most importantly, what not to cache.

## 2.1. CACHING FIRST – THE FIRST-CLASS-CITIZEN

Often people start to think about caching when response times of their web servers, databases or other backend resources start to grow exponentially. Their first reaction to optimize the situation is to quickly integrate some cache into the application landscape, but a real caching strategy is commonly missing or not defined upfront.

Most often the main problem area is a central system acting not only as a single point of failure but also as the limiting factor. A common example would be all data being stored inside a relational database. Every page request not only does a single query to the database but possibly multiple queries, sometimes also complex joins.

As a result, users, at peak times, start to suffer from long loading times, as well as high latencies depending on the round-trip times between their location and the datacenter.

*Caching First* is a term we came up to describe the situation where you start thinking about Caching itself as one of the main domains of your application, just like database, application server, hardware specifications or user experience.

Systems these days have to reach a critical mass to be successful. They are unlikely to reach this critical mass within the range of a single database or web server. Systems need to integrate certain caches and this paper will provide the necessary information to build your own caching strategy the same way you select your database or programing language.

## 2.2. WHY CACHING?

As mentioned above, caching is used to speed up certain requests or parts of those requests. Web pages that are dynamically rendered but act as static content because they almost never change can be cached extensively. The same is true for page fragments that will be pre-rendered and only enriched with user-specific data before sending them out. Also queries that won't result in dynamic datasets (because of in-query calculations) can be very easily cached and result in much higher request speeds.

The overall benefits of caching is to aid both the content consumer and content providers. A good caching strategy can offer lots of advantages and reduces the amount of money spent on hardware to achieve similar goals.

- **Improved responsiveness:** Caches provide faster retrieval of content and prevent additional network roundtrips. Good caching strategies (for example, on websites) also integrate clear static content to be cached inside the user's web browser directly.
- **Decreased network costs:** Depending on the caching strategy, content is available in multiple regions inside the network path and/or world. That way, content moves closer to the user and network activity beyond the cache is reduced.
- **Improved performance with same hardware:** Caches store already retrieved content and deliver it as long as it is valid. Costly calculations, page rendering or other long-running operations are reduced or eliminated and allowed to run more requests on the same hardware.
- **Availability of content with interruption on network or backend resources:** Caches can provide cached data also in case of unavailability of backend systems or problems with network connection.

## 2.3. WHAT TO CACHE?

A good starting point to find out what to cache in your application is to imagine everything where multiple executions of some request result in the same outcome. This can be database queries, HTML fragments, complete web pages or an output of a heavy computation.

It also makes sense to store any kind of language related data geographically local to the content consumer. Common elements of this type of data is translations, user data for people living in a given region. Only one rule applies: data should not change too often or fast but should be read very frequently.

## 2.4. WHAT NOT TO CACHE?

A common misconception is if you cache everything,  you'll automatically benefit from it. What often works in the first place delivers another problem during high data peaks.

Data that changes often is generally not very good for caches. Whenever data changes, the cache must be invalidated and, depending on the chosen caching strategy, this can be a costly operation. Imagine a caching system that is located around the world and your data change with a rate of more than 100 changes per second. The benefit of having those data cached will be nullified by the fact that all caches need to invalidate and maybe re-retrieve that changed data record for every single change.

Another point to think about is to not cache data that is fast to retrieve anyways. Caching those elements will introduce an additional round-trip while filling the cache, requiring additional memory. The benefit might not show the expected results or be worth the overhead of bringing in another layer into the architecture.

# 3. Cache Types

Caches can mostly be separated into three distinct categories, depending on the stored data. This section will give a quick overview over those categories and a few examples of when they are used.

## 3.1. HTTP CACHE

A HTTP Cache is mostly used in browsers. This kind of cache keeps information about the last modification date of a resource or a content hash to identify changes to its content. Web servers are expected to deliver useful information about the state of an element to prevent retrieval of an already cached element from the server.

This kind of cache is used to reduce network traffic, minimize cost and offer the user an instant experience for multiple visits.

## 3.2. FRAGMENT CACHE

A Fragment Cache caches parts of a response or result. This could be a database query outcome or a part of an HTML page. Whatever it is, it should not change often.

A common use case for a Fragment Cache is a web page known to contain user specific and user unspecific content. The user-independent content can be cached as a fragment and augmented with user specific content on retrieval. This process is called *Content Enrichment*.

This caching type is used to reduce operation cost and hardware by providing the same throughput with less computational overhead.

## 3.3. OBJECT CACHE

An Object Cache stores any sort of objects that otherwise need to be read from other data representations. A cache of this type can be used in front of a database to speed up queries and store the resulting objects (e.g. Object Relational Mapping, ORM), or store un-marshalled results of XML, JSON or other general data representations transformed into objects.

These caches often act as a proxy between some external resource, like a database or webservice, and they speed up transformation processes or prevent additional network round-trips between the consumer and producer systems.

# 4. Caching Strategies

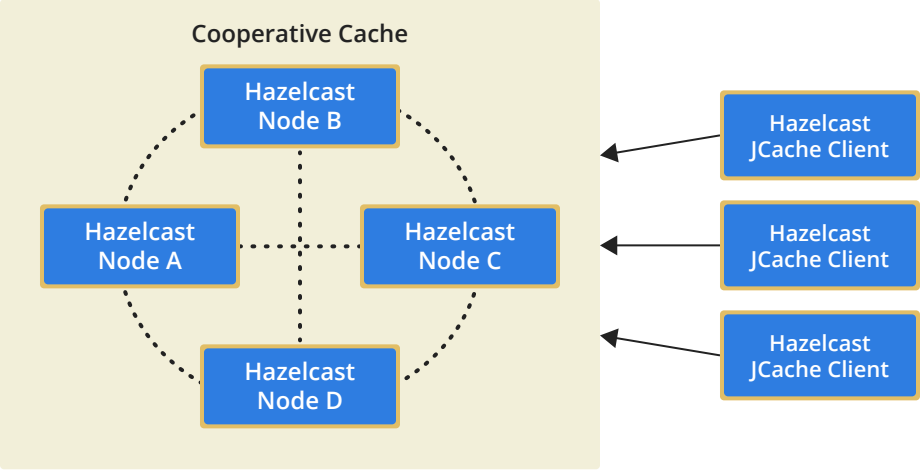
This section describes the different types of caching strategies to be found in the wild. Those strategies can be seen as general descriptions rather than special Hazelcast use cases.

## 4.1. COOPERATIVE / DISTRIBUTED CACHING

In Cooperative Caching, also known as Distributed Caching, multiple distinct systems (normally referred to as cluster-nodes) work together to build a huge, shared cache.

Implementations of Cooperative Caches can either be simple and route requests to all nodes. They look for someone to respond, or provide a more intelligent partitioning system, normally based on consistent hashing to route the request to a node containing the requested key.

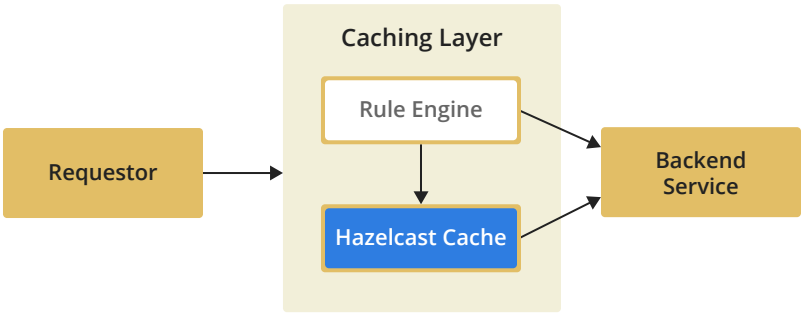
Nowadays, Cooperative Caching is the common way for caching in bigger systems when large amounts of data need to be cached.




## 4.2. PARTIAL CACHING

Partial Caching describes a type of caching where not all data is stored inside the cache. Depending on certain criteria, responses might not be cacheable or are not expected to be cached (like temporary failures).

A typical example for data where not everything is cacheable is websites. Some pages are “static” and only change if some manual or regular action happens. Those pages can easily be cached and invalidated whenever this particular action happened. Apart from that, other pages consist of mostly dynamic content or frequently updated content (like stock market tickers) and shouldn’t be cached at all.



## 4.3. GEOGRAPHICAL CACHING

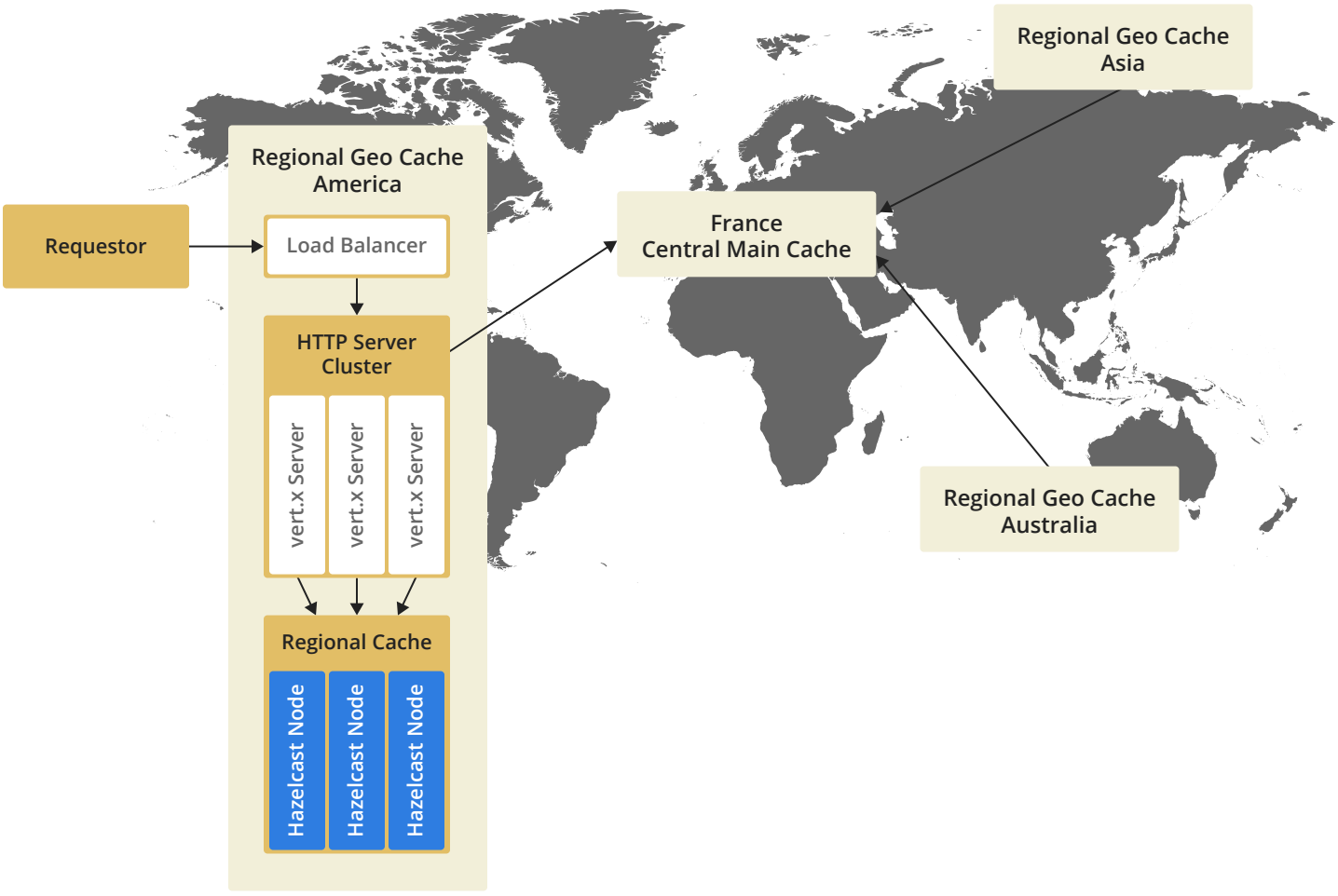
Geographical Caches are located in strategically chosen locations to optimize latency on requests, therefore this kind of cache will mostly be used for website content. It is also known as CDN (Content Delivery Network).

Typically a Geographical Cache will transparently go down to a central cache which acts as the main source for content and cache retrieved data locally. This works great for static content or content that changes less often.

The regional caches can normally be smaller than the central cache due to the fact that certain caches will handle regions with different languages or cultural differences or preferences.

In addition, Geographical Caching is often used in conjunction with Partial Caching to cache static content like assets (images, sounds, movies) as near as possible to the requestor but create dynamic content on the fly using different routes or rule engines (forwarders).

The geographically nearest cache is most often determined by geo-routing based on IP addresses (DNS routing) or using proxies which redirect to certain IP addresses using HTTP status 302.

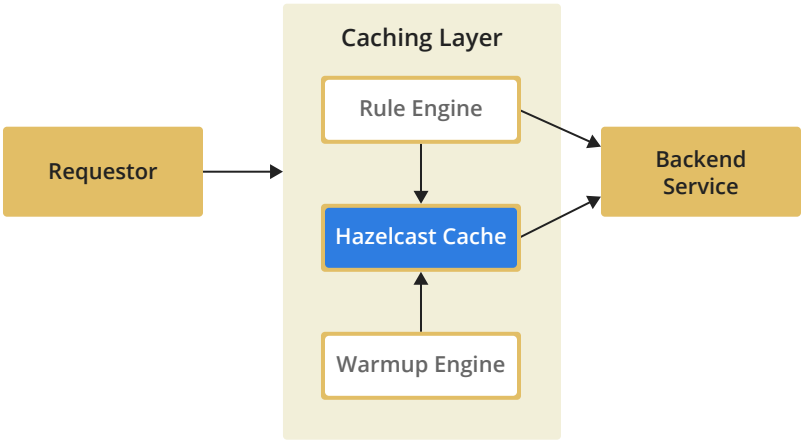


## 4.4. PREEMPTIVE CACHING

A *Preemptive Cache* itself is not a caching type like the others above but is mostly used in conjunction with a *Geographical Cache*.

Using a warm-up engine a *Preemptive Cache* is populated on startup and tries to update itself based on rules or events. The idea behind this cache addition is to reload data from any backend service or central cluster even before a requestor wants to retrieve the element. This keeps access time to the cached elements constant and prevents accesses to single elements from becoming unexpectedly long.

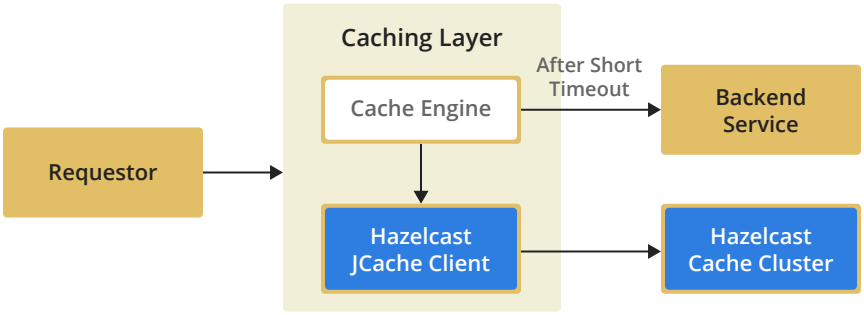
Building a *Preemptive Cache* is not easy and requires a lot of knowledge of the cached domain and the update workflows.



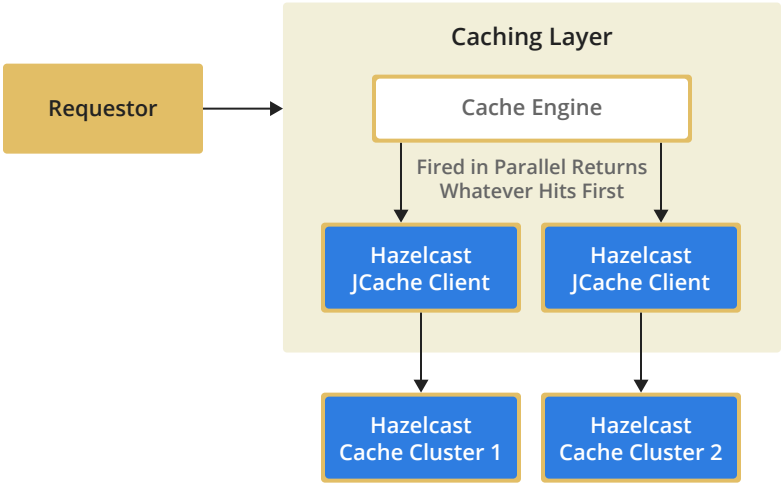
## 4.5. LATENCY SLA CACHING

A *Latency SLA Cache* is able to maintain latency SLAs even if the cache is slow or overloaded. This type of cache can be build in two different ways.

The first option is to have a timeout to exceed before the system either requests the potentially cached element from the original source (in parallel to the already running cache request) or provides a simple default answer, and uses whatever returns first.




The other option is to always fire both requests in parallel and take whatever returns first. This option is not the preferred way of implementation since it mostly works against the idea of caching and won't reduce load on the backend system. This option might still make sense if multiple caching layers are available, like always try the first and second nearest caches in parallel.



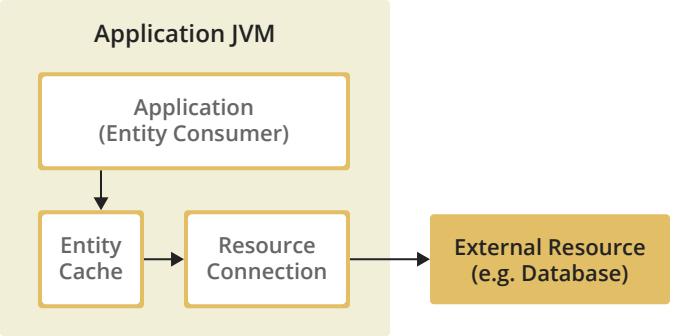
# 5. Caching Topologies

Modern caches manifest in three distinct types of systems. This section will shortly explain those types and show advantages/disadvantages of them.

## 5.1. IN-PROCESS CACHES

The *In-Process Cache* is most oftenly used in non-distributed systems. The cache is kept inside the application's memory space itself and offers the fastest possible access speed.

This type of cache is used for caching database entities but can also be used as some kind of an object pool, for instance pooling most recently used network connections to be reused at a later point.




**Advantages:**

- Highest access speed
- Data available locally
- Easy to maintain

**Disadvantages:**

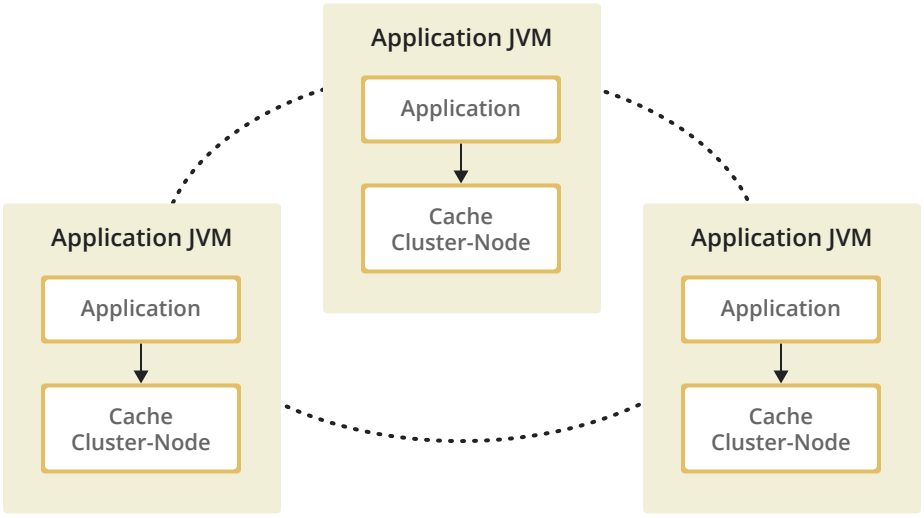
- Data duplication if multiple applications
- High memory consumption on a single node
- Data cached inside the applications memory
- Seems easy to build but has a lot of hidden challenges

## 5.2. EMBEDDED NODE CACHES

Using an *Embedded Node Cache* the application itself will be part of the cluster. This caching topology is a kind of combination between an *In-Process Cache* and the *Cooperative Caching* and it can either use partitioning or full dataset replication.

Using full replication the application will get all the benefits of an *In-Process Cache* since all data is available locally (highest access speed) but for the sake of memory consumption and heap size.

By using data partitioning the application knows about the owner of a requested record and will ask directly using an existing data stream. Speed is lower than locally available data but still accessible quickly.




**Advantages:**

- Data can be replicated for highest access speed
- Data can be partitioned to create bigger datasets
- Cached data might be used a shared state lookup between applications
- Possible to scale out the application itself

**Disadvantages:**

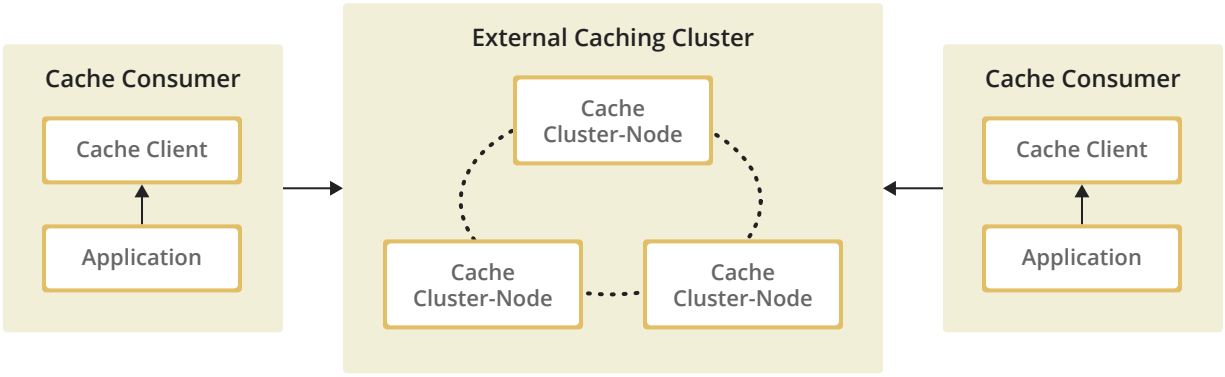
- High duplication rate on replication
- Application and cache cannot be scaled independently
- Data cached inside the applications memory

## 5.3. CLIENT-SERVER CACHES

A *Client-Server Cache* is one of the most typical setups these days (next to a pure In-Process Cache). In general these systems tend to be *Cooperative Caches* by having a multi-server architecture to scale out and have the same feature set as the *Embedded Node Caches* but with the client layer on top.

This architecture keeps separate clusters of the applications using the cached data and the data itself, offering the possibility to scale the application cluster and the caching cluster independently. Instead of a caching cluster it is also possible to have a single caching server however this situation slowly losing traction.

Having a *Client-Server Cache* architecture is quite similar to the common usage patterns of an external relational database or other network-connected backend resources.




**Advantages:**

- Data can be replicated for highest access speed
- Data can be partitioned to create bigger datasets
- Cached data might be used a shared state lookup between applications
- Applications and cache can be scaled out independently
- Applications can be restarted without losing data

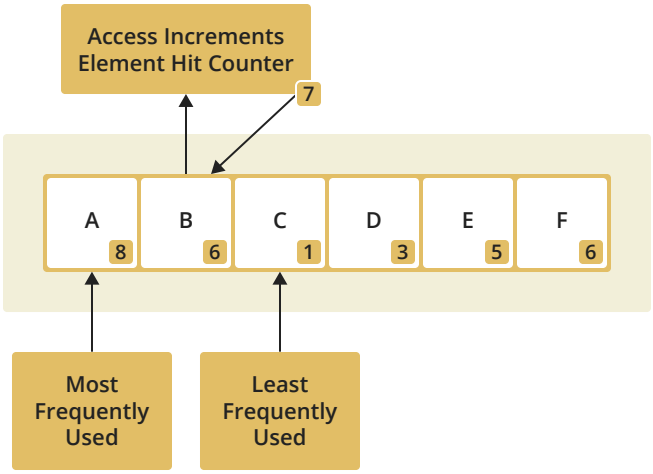
**Disadvantages:**

- High duplication rate on replication
- Always an additional network round trip (fast network)

# 6. Eviction Strategies

## 6.1. LEAST FREQUENTLY USED (LFU)

The *Least Frequently Used* eviction strategy removes values that are accessed the least amount of times. To do the analysis each record keeps track of its accesses using a counter which is increment only. This counter can then be compared to those of other records to find the least frequently used element.




**Advantages:**

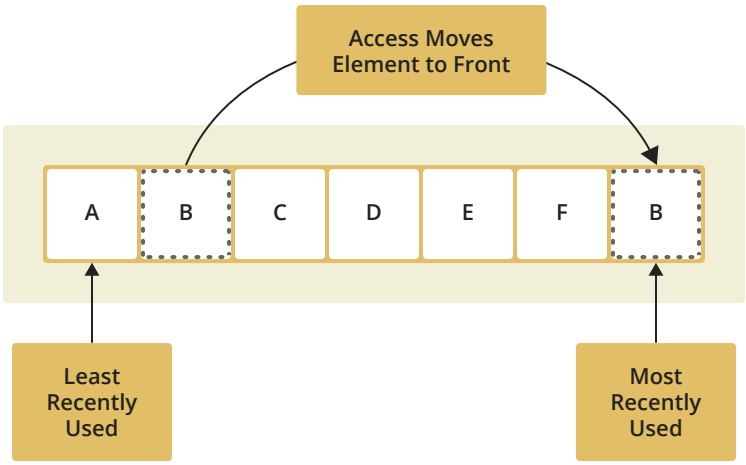
- Takes age of the element into account
- Takes reference frequency of the element into account
- Works better under high load when quickly a lot of elements is requested (less false eviction)

**Disadvantages:**

- A frequently accessed element will only be evicted after lots of misses
- More important to have invalidation on elements that can change

## 6.2. LEAST RECENTLY USED (LRU)

The *Least Recently Used* eviction strategy removes values that were last used most far back in terms of time. To do the analysis, each record keeps track of its last access timestamp that can be compared to other records to find the last least recently used element.




**Advantages:**

- Is nearest to the most optimal algorithm
- Selects and removes elements that are not used recently

**Disadvantages:**

- Only mild success rate since often it is more important how often an element was accessed than when it was last accessed

## 6.3. OTHER EVICTION STRATEGIES

Apart from LRU and LFU (previously explained), a couple of other eviction strategies were invented over the past decade. However their goals can diverge. Some try to find better matching eviction candidates but require more processing or metadata; others try to combine the two prior strategies or, in the simplest case, just select random entries.

As to their goals, the outcome and efficiency of evicted elements can differ. A full overview of other common eviction strategies can be found on Wikipedia.

# 7. Java Temporary Caching – JCache

The Java world uses caching APIs. Different caching vendors and APIs were seen over the past and some of them disappeared. Having proprietary APIs or vendors that come and go can be a big issue for users of the corresponding APIs. Refactoring or bigger rewrites can be time consuming and expensive and users often hold back as long as possible to prevent those penalties from applying.

The Java Temporary Caching API, shortly referred to as JCache or JSR 107, is a common standard for Java SE and Java EE to provide a general interface for caching purposes. It was defined by the Java Community Process (JCP) and was finalized in 2014.

## 7.1. WHY JCACHE?

JCache, based on a Map-like API, provides access to data but optimizes for caching only. For example, it removes the mandatory return value from put operations to minimize traffic in distributed systems.

Having a standard API removes the need for vendor proprietary integrations and offers a fast way to use different vendors interchangeably. It also provides the user with the power to use a local-only cache in unit tests but scale out in production using a *Cooperative Cache* solution like Hazelcast.

Per specification JCache already supports integration into familiar frameworks like Spring or similar ones using a set of predefined annotations. It also easily integrates into applications that used a custom cache, based on ConcurrentMap, by mostly just replacing calls to the map with calls to a JCache cache. It is expected to be part of the Java EE specification, offering out-of-the-box support on application servers.

## 7.2. JCACHE 1.0 DRAWBACKS

The current version 1.0 of the specification might have some drawbacks, depending on the application to be integrated into. It doesn't offer asynchronous operations and lacks some features to support certain caching strategies (like Latency SLA Caches). Furthermore, transactional support is not yet in the specification.

For some of those features, if necessary, vendors provide proprietary extensions, but the expert group behind JCache (which Hazelcast is part of) is eager to fix them as soon as possible by providing a standard way to prevent users from using vendor specifics.

# 8. Hazelcast Caching

## 8.1. ARCHITECTURE

Hazelcast is a data partitioning and distributed computing platform, also known as In-Memory Data Grid.

A Hazelcast cluster, once established, forms a Peer-to-Peer network with connections between all members for highest speed. Hazelcast clients, by default, will also connect to all members and know about the cluster partition table to route requests to the correct node, preventing additional round trips. Therefore a Hazelcast cluster used as a cache is always a *Cooperative Cache* based on multiple members.

The Hazelcast JCache layer is the preferred way to support caching in Java applications, as it implements the new Java Temporary Caching API, defined in JSR 107. Using a defined Java standard offers the advantage to exchange

## 8.2. HAZELCAST AND JCACHE

Starting with Hazelcast release 3.3.1, a specification compliant JCache implementation is offered. To show our commitment to this important specification the Java world was waiting for over a decade, we do not just provide a simple wrapper around our existing APIs: we implement a caching structure from ground up to optimize the behavior to the needs of JCache. As mentioned before, the Hazelcast JCache implementation is 100% TCK (Technology Compatibility Kit) compliant and therefore passes all specification requirements.

In addition to the given specification, we added some features like asynchronous versions of almost all operations to give the user extra power and further possibilities like an explicit expiration policy per value.

For more information please see the *ICache* chapter of the Hazelcast documentation.

## 8.3. EVICTION ALGORITHM

The Hazelcast JCache implementation supports a fully distributed eviction implementation based on random data samples. Using the sampling algorithm Hazelcast is able to evict values based on heuristics with a sanitized O(1) runtime behavior.

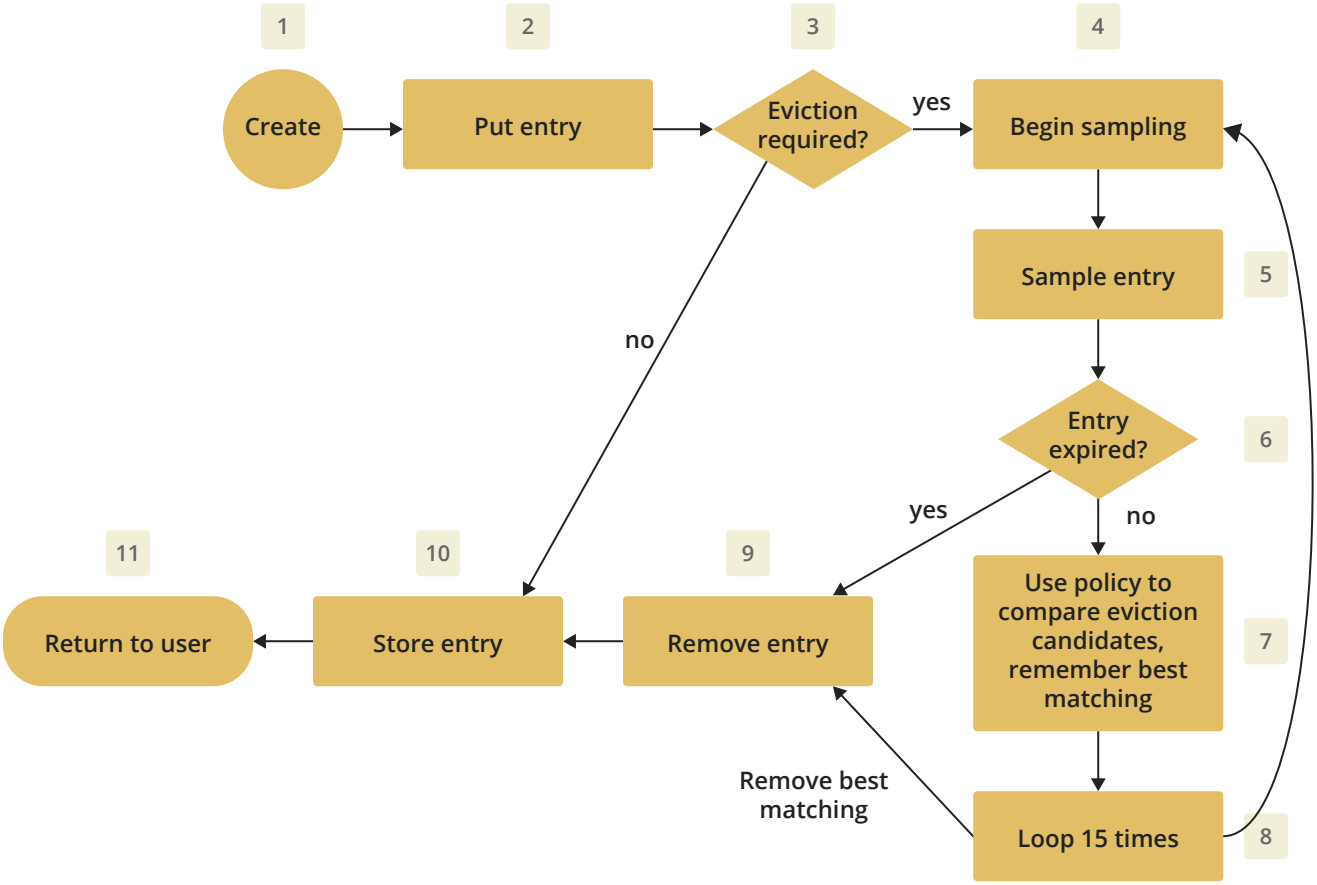
The underlying data storage implementation, inside the Hazelcast JCache nodes, provides a constant runtime selection algorithm to sample those random elements.

Typically two types of caches can be found in applications:

- **Reference Caches:** Caches for reference data are normally small and are used to speed up the de-referencing as a lookup table. Those caches tend to be small and contain a previously known, fixed number of elements (e.g. states of the USA or abbreviations of elements).
- **Active DataSet Caches:** The other type of caches normally caches an active data set. These caches run to their maximum size and evict the oldest or not frequently used entries to keep in memory bounds. They sit in front of a database or HTML generators to cache the latest requested data.

Hazelcast JCache eviction supports both types of caches using a slightly different approach based on the configured maximum size of the cache. The exact heuristics are described below.




1. A new cache is created. Without any special settings, the eviction is configured to kick in when the **cache** exceeds 10,000 elements and an LRU (Less Recently Used) policy is set up.
2. The user puts in a new entry (e.g. key-value pair).
3. For every put, the eviction strategy evaluates the current cache size and decides if an eviction is necessary or not, if not the entry is stored in step 10.
4. If eviction is required, a new sampling is started. To make things efficient, built-in sampler is implemented as a lazy iterator.
5. The sampling algorithm selects a random sample from the underlying data storage.
6. The eviction strategy tests the sampled entry to already be expired (lazy expiration), if expired the sampling stops and the entry is removed in step 9.
7. If not yet expired, the entry (eviction candidate) is compared to the last best matching candidate (based on the eviction policy) and the new best matching candidate is remembered.
8. The sampling is repeated 15 times and then the best matching eviction candidate is returned to the eviction strategy.
9. The expired or best matching eviction candidate is removed from the underlying data storage.
10. The new put entry is stored.
11. The put operation returns to the user.

As seen by the flowchart, the general eviction operation is easy. As long as the cache does not reach its maximum capacity or you execute updates (put/replace), no eviction is executed.

To prevent network operations and concurrent access, the cache size is estimated based on the size of the currently handled partition. Due to the imbalanced partitions, the single partitions might start to evict earlier than the other partitions.

As mentioned before, typically two types of caches are found in the production systems. For small caches, referred to as Reference Caches, the eviction algorithm has a special set of rules depending on the maximum configured cache size. Please see the Reference Caches section for details. The other type of cache is referred to as Active DataSet Caches, which in most cases makes heavy use of the eviction to keep the most active data set in the memory. Those kinds of caches using a very simple but efficient way to estimate the cluster-wide cache size.

All of the following calculations have a well known set of fixed variables:

- *GlobalCapacity:* The user defined maximum cache size (cluster-wide)
- *PartitionCount:* The number of partitions in the cluster (defaults to 271)
- *BalancedPartitionSize:* The number of elements in a balanced partition state, *BalancedPartitionSize* := *GlobalCapacity* / *PartitionCount*
- *Deviation:* An approximated standard deviation (tests proved it to be pretty near), *Deviation* := *sqrt(BalancedPartitionSize)*

## 8.4. REFERENCE CACHES

A *Reference Cache* is typically small and the number of elements to store in the reference caches is normally known prior to creating the cache. Typical examples of reference caches are lookup tables for abbreviations or the states of a country. They tend to have a fixed but small element number and the eviction is an unlikely event and a rather undesirable behavior.

Since an imbalanced partition is the worst problem in the small and mid-sized caches, rather than for the caches with millions of entries, the normal estimation rule (as will be discussed in a bit) is not applied to these kinds of caches. To prevent unwanted eviction on the small and mid-sized caches, Hazelcast implements a special set of rules to estimate the cluster size.

To adjust the imbalance of partitions as found in the typical runtime, the actual calculated maximum cache size (as known as the eviction threshold) is slightly higher than the user defined size. That means more elements can be stored into the cache than expected by the user. This needs to be taken into account especially for large objects, since those can easily exceed the expected memory consumption!

**Small caches:**

If a cache is configured with no more than *4,000* elements, this cache is considered to be a small cache. The actual partition size is derived from the number of elements (GlobalCapacity) and the deviation using the following formula:

```
MaxPartitionSize := Deviation * 5 + BalancedPartitionSize
```

This formula ends up with big partition sizes which, summed up, exceed the expected maximum cache size (set by the user). However, since the small caches typically have a well known maximum number of elements, this is not a big issue. Only if the small caches are used for a use case other than using it as a reference cache does this need to be taken into account.

**Mid-sized caches:**

A mid-sized cache is defined as a cache with a maximum number of elements that is bigger than *4,000* but not bigger than *1,000,000* elements. The calculation of mid-sized caches is similar to that of the small caches but with a different multiplier. To calculate the maximum number of elements per partition, the following formula is used:

```
MaxPartitionSize := Deviation * 3 + BalancedPartitionSize
```

## 8.5. ACTIVE DATASET CACHES

For large caches, where the maximum cache size is bigger than *1,000,000* elements, there is no additional calculation needed. The maximum partition size is considered to be equal to *BalancedPartitionSize* since statistically big partitions are expected to almost balance themselves. Therefore, the formula is as easy as the following:

```
MaxPartitionSize := BalancedPartitionSize
```

## 8.6. EVICTION NECESSARY HEURISTICS

The provided estimation algorithm is used to prevent cluster-wide network operations, concurrent access to other partitions and background tasks. It also offers a highly predictable operation runtime when the eviction is necessary.

The estimation algorithm is based on the previously calculated maximum partition size and is calculated against the current partition only.

The heuristic algorithm to reckon the number of stored entries in the cache (cluster-wide) and to decide if the eviction is necessary or not is shown using the following pseudo-code example:

```
RequiresEviction[Boolean] := CurrentPartitionSize >= MaxPartitionSize
```

## 8.7. LFU AND LRU HEURISTICS

Given the above description of the sampling based eviction, Hazelcast JCache eviction doesn't implement LFU and LRU as *Least Frequently Used* or *Least Recently Used* but as a heuristic based on the sampling space of random elements. In Hazelcast, LFU and LRU are therefore implemented as **Less Frequently Used** and **Less Recently Used**.

Internal tests have shown that this sampling space has a good tradeoff between false eviction of still needed elements and the factor of calculation time and memory overhead.

Even though it is a best-guess algorithm, in almost all cases the decision based on the sampling space is scored efficiently enough to be safe to remove.

hazelcast

**350 Cambridge Ave, Suite 100, Palo Alto, CA 94306 USA**
**Email:** sales@hazelcast.com **Phone:** +1 (650) 521-5453
**Visit us at** www.hazelcast.com