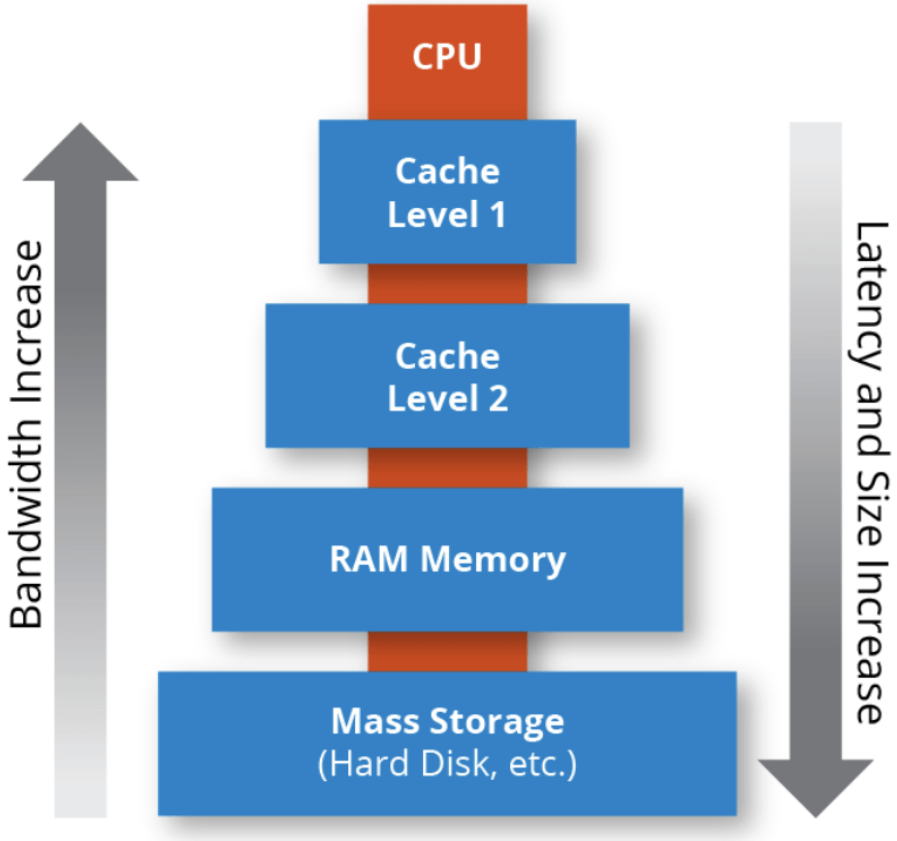
Alfresco repository Caches optimisation can have significant impact on the performance of your Alfresco deployment. This post provides an overview on how the repository caches are implemented by Alfresco.

The Alfresco repository leverages and provides **in-memory** caches. Memory caching (often simply referred to as caching) is a technique in which computer applications temporarily store data in a computer's main memory (i.e., random access memory, or RAM) to enable fast retrievals of that data. The RAM that is used for the temporary storage is known as the cache. Since accessing RAM is significantly faster than accessing other media like hard disk drives or networks, caching helps applications run faster due to faster access to data.

Caching is especially efficient when the application exhibits a common pattern in which it repeatedly accesses data that was previously accessed. Caching is also useful to store data calculations that are otherwise time-consuming to compute. By storing the calculations in a cache, the system saves time by avoiding the repetition of the calculation.



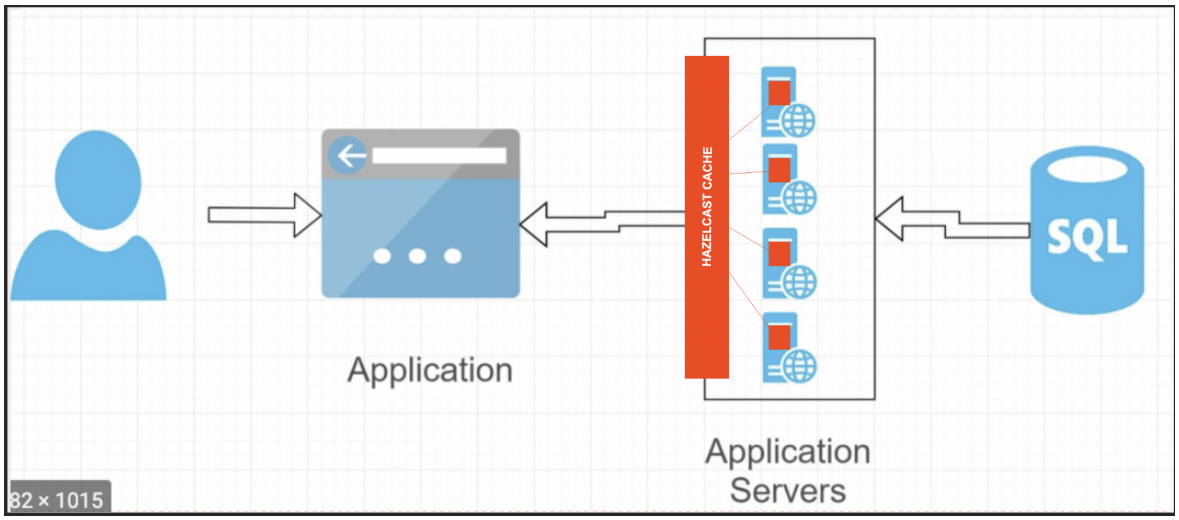
An Overview of Memory Caching

To know more on how memory caching works, refer to this URL (**https://hazelcast.com/glossary/memory-caching**/)

Well tuned repository caches can improve the overall throughput of your deployment but bear in mind that they use Java heap memory. When tuning your repository caches, make sure you have enough RAM on your machines that can meet the requirements of your tuned values.

Interesting to notice that the caches are activated in both clustered and non-clustered configurations. In clustered environments like the example below,

the L2 cache (we will talk about cache levels later on this article) works as a distributed cache and it pools together the RAM of multiple application servers (normally the alfresco nodes) into a **single in-memory data store,** represented by the Hazelcast cache on the diagram below. It's main purpose is to **provide fast access to data**.



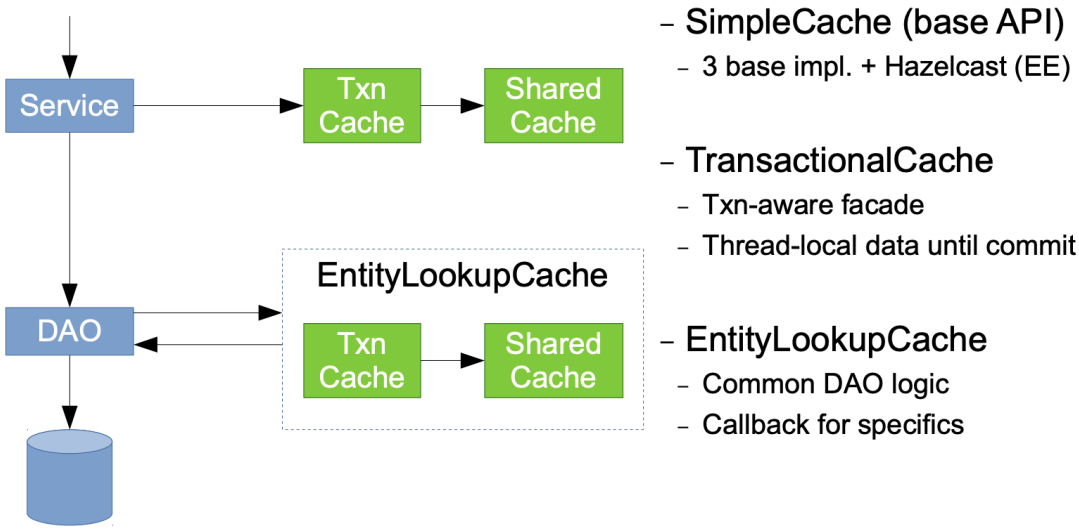
Repository Cache Levels

The repository caches are separated in 2 different levels, the L1 and L2 levels, representing the Cache Level 1 and Cache Level 2 on the diagram above.

L1 = The transactional cache (TransactionalCache.java)

Implemented directly in the Alfresco repository as a java class, the transactional cache implements a 2-level cache that maintains both a

transaction-local cache and wraps a non-transactional (shared) cache. It uses the shared **SimpleCache** for it's per-transaction. Instances of the **TransactionalCache.java**class do not require a transaction as they will work directly with the shared cache when no transaction is present. There is virtually no overhead when running out-of-transaction.



Because there is a limited amount of space available to the in-transaction caches, when either of these becomes full, the cleared flag is set. This ensures that the shared cache will not have stale data in the event of the transaction-local caches dropping items. **It is therefore important to size the transactional caches correctly.**

Alfresco provides a way to trace the transactional cache usage by enabling specific log4j classes. For example:

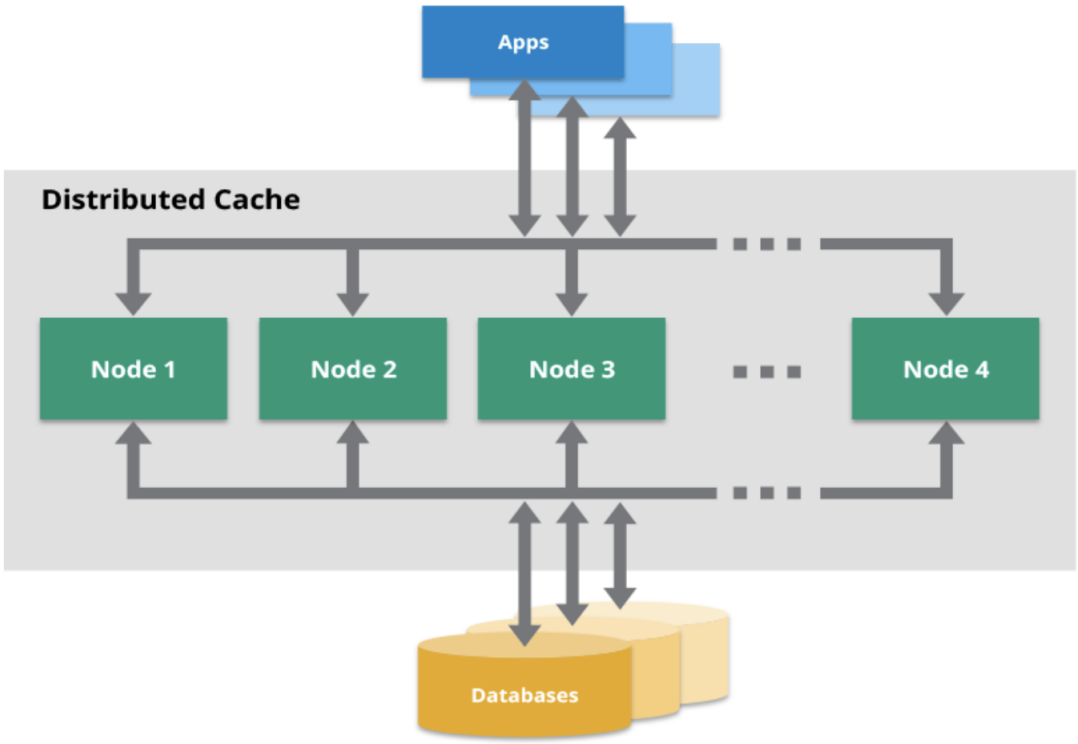
- log4j.logger.org.alfresco.repo.cache=TRACE | DEBUG

L2 = Hazelcast distributed Cache

The L1 cache described before commits to the L2 cache and the L2 cache is fully managed by Hazelcast using its distributed cache implementation.

As we've seen before , a distributed cache system pools together the RAM memory reserved for caching of multiple nodes into a **single in-memory data store** used as a data cache to **provide fast access to data**.

This type of cache can grow beyond the memory limits of a single computer by linking together multiple computers–referred to as a **distributed architecture** or a distributed cluster–for larger capacity and increased processing power.



A distributed cache pools the RAM of multiple computers into a single in-memory data store used as a data cache to provide fast access to data.

Distributed caches are especially useful in environments with high data volume and load. The distributed architecture, in line with the Alfresco architecture scaling procedures, allows **incremental expansion/horizontal scaling** by adding more computers to the cluster, allowing the cache to grow in step with the data growth. This pattern is part of the Alfresco scalability as when we add one repository node to our cluster, we are also adding a hazelcast node to the hazelcast cluster.

Using hazelcast mancenter you can trace the L2 cache usage. **https://hazelcast.com/product-features/management-center**/

Below a comprehensive list of loggers that alfresco provides that will write cache operations to the log.

- log4j.logger.org.alfresco.enterprise.repo.cluster.cache=DEBUG | TRACE
- log4j.logger.org.alfresco.repo.cache=DEBUG | TRACE
- log4j.logger.com.hazelcast=DEBUG | TRACE
- log4j.logger.org.alfresco.enterprise.repo.cluster.messenger.HazelcastMessenger=DEBUG | TRACE
- log4j.logger.org.alfresco.enterprise.repo.cluster.cache.HazelcastSimpleCache=DEBUG | TRACE
- log4j.logger.com.hazelcast.impl.TcpIpJoiner= DEBUG | TRACE

Main Advantages of using a distributed Cache

Application acceleration

Applications that rely on relational databases can't always meet today's increasingly demanding transaction performance requirements. By storing the most frequently accessed data in a distributed cache, you can dramatically reduce the I/O footprint on the database. This ensures your application can run faster, even with a large number of transactions when usage spikes.

When deploying Alfresco, you can design your solution to include your own custom caches that hold the most frequently access data for your use case. Leveraging those custom caches , you can optimise the resource usage on

your entire architecture, from avoiding un-necessary search requests to your index to alleviating the database from running constant queries that are costly and normally yield the same results.

Extreme scaling

The Alfresco platform, as any application on modern big scale deployments, will request huge volumes of data from its attached resources (Storage, Database and Memory (caches)). By leveraging more resources across multiple machines, a distributed cache helps to optimise the responses to all those requests.

Storing web session data

A site may store user session data in a cache to serve as inputs for shopping carts and recommendations. With a distributed cache, you can have a large number of concurrent web sessions that can be accessed by any of the web application servers that are running the system. This lets you load balance web traffic over several application servers and not lose session data should any application server fail.

Decreasing network usage/costs

By caching data in multiple places in your network, including on the same computers as your application, you can reduce network traffic and leave more bandwidth available for other applications that depend on the

network.Reducing the impact of interruptions. Depending on the architecture, a cache may be able to answer data requests even when the source database is unavailable. This adds another level of high availability to your system.

Getting more information from Hazelcast by adding Hazelcast jmx options

Adding the following options to your JVM will expose the jmx features of hazelcast.

- -Dhazelcast.jmx=true -Dhazelcast.jmx.detailed=true

Alfresco Repository Individual Caches Information

This section shows details on some relevant repository caches with a brief definition on their purpose.

Default Cache	
defaultCache	This cache is for when someone forgets to write a cache sizing snippet.
Store Caches	
rootNodesCache	Primary root nodes by store.
allRootNodesCache	All root nodes by store.
Nodes Caches	
nodesCache	Bi-direction mapping between NodeRef and ID.
aspectsCache	Node aspects by node ID. Size relative to nodesCache
propertiesCache	Node properties by node ID. Size relative

	to nodesCache
childByNameCache	Child node IDs cached by parent node ID and cm:name. Used for direct name lookups and especially useful for heavy CIFS use.
contentDataCache	cm:content ContentData storage. Size according to the amount content in system.
Permissions and ACLs	
authorityToChildAuthorityCache	Size according to total number of users accessing system
authorityToChildAuthorityCache	Members of each group. Size according to number of groups, incl. site groups

zoneToAuthorityCache	Authorities in each zone. Size according to zones.
authenticationCache	Scale it according to the number of users, not concurrency, because Share sites will force all of them to be hit regardless of who logs in.
authorityCache	NodeRef of the authority. Size according to total authorities
permissionsAccessCache	Size according to total authorities
readersCache	Who can read a node, Size according to total authorities.
readersDeniedCache	Who can’t read a node, Size according

	to total authorities.
nodeOwnerCache	cm:ownable mapping, needed during permission checks
personCache	Size according to the number of people accessing the system
aclCache	ACL mapped by Access control list properties
aclEntityCache	DAO level cache bi-directional ID-ACL mapping.
Other Caches	
propertyValueCache	Caches values stored for auditing. Size according to frequency and size of audit queries and audit value generation

immutableEntityCache	QNames, etc. Size according to static model sizes
tagscopeSummaryCache	Size according to tag use. Stores rollups of tags

Do i need to increase my cache values ?

An important indicator that you need to tune or increase your caches is when you see a warning message in your alfresco.log file indicating that some specific caches are full, for example:

```
2016-04-26 17:51:37,127 WARN [org.alfresco.repo.cache.TransactionalCache.org.alfresco.cache.node.nodesTransactionalCache]
[http-apr-22211-exec-42] Transactional update cache 'org.alfresco.cache.node.nodesTransactionalCache' is full (125000).
```

The repository caches are initially configured with their default values on a specific properties file (**caches.properties**). This file is embedded in the distributions of ACS and its values can be override under the common alfresco-global.properties file, that is configured outside the distribution, normally under :

- <appServerHome>/shared/classes/alfresco.

Important Factors to consider when tuning your L2 Caches

To perform a cache tuning exercise we need to analyse 3 relevant factors :

- type of data
- how often it changes
- number of gets compared to the number of writes

If we can identify caches that the correspondent values do not change often, it's worth to try and set them to invalidating, and check the performance results. Note that in distributed-caches, when we have a lot a remote gets, if the objects that are being stored are big, the remote get operation its going to be slow. This is mainly because the object is serialized and it needs to be un-serialized before its content is made available and that operation can take some time depending on the size of the object. We also need to consider, that in distributed caches, when there are a lot of remote gets the network traffic will increase.

On the other hand, if we choose and invalidation cache mechanism and the caches are changing often, the Invalidation messages can also be a single point of network stress. So overall it's all about analyzing the trade-offs of each mechanism and to choose the more appropriate for each use case.

More ? check **Individual cache settings.**

How do i tune the repository caches ?

The cache properties are used for both clustered and non-clustered configurations. To configure a specific cache, we need to override a series of properties where the property names begin with the cache name as specified in the Spring cache definition. For example, if a cache has the name “cache.texterBlueCache” then the properties **should all** start with “cache.texterBlueCache”.

The following example shows the full scope of a custom cache configuration options. We will shortly discuss each of the options individually.

- cache.texterBlueCache.tx.**maxItems**=1000
- cache.texterBlueCache.tx.**statsEnabled**=true
- cache.texterBlueCache.**maxItems**=10000
- cache.texterBlueCache.**timeToLiveSeconds**=300
- cache.texterBlueCache.**maxIdleSeconds**=0
- cache.texterBlueCache.**cluster**.type=invalidating
- cache.texterBlueCache.**backup-count**=1
- cache.texterBlueCache.**eviction-policy**=LRU
- cache.texterBlueCache.**eviction-percentage**=25
- cache.texterBlueCache.**merge-policy**=hz.ADD_NEW_ENTRY

It’s important to notice that the database is used as the central place of discovery for cluster nodes, this means that if we want to specifically remove a node from the cluster (for example the alfresco tracking node) we

need to specify **alfresco.cluster.enabled=false** on that node alfresco-global.properties file.

Defining your own Alfresco Repository Cache

You can define your own caches at the same scope as the repository caches. You can take advantage of the magic of Spring and just declare a new bean on your custom spring context file like on the following example:

```
<bean name=“texterBlueCache" factory-bean=“cacheFactory" factory-method="createCache">

<constructor-arg value="cache.customTexterBlueCache"/>

</bean>
```

- cache.customTexterBlueCache.maxItems=130000
- cache.customTexterBlueCache.timeToLiveSeconds=0
- cache.customTexterBlueCache.maxIdleSeconds=0
- cache.customTexterBlueCache.cluster.type=fully-distributed
- cache.customTexterBlueCache.backup-count=1
- cache.customTexterBlueCache.eviction-policy=LRU
- cache.customTexterBlueCache.eviction-percentage=25
- cache.customTexterBlueCache.merge-policy=hz.ADD_NEW_ENTRY

Note that there is no need for a corresponding hazelcast-tcp.xml entry for the custom cache, the factory does all the configuration programmatically using the name of the cache customTexterBlueCache as a root to discover the remaining configuration properties.

maxItems	The maxItems attribute is the maximum size a cache can reach. Use zero to set to Integer.MAX_VALUE.
timeToLiveSeconds	The timeToLiveSeconds attribute specifies that the cache items will expire once this time has passed after creation.
maxIdleSeconds	The maxIdleSeconds attribute specifies that the cache items will expire when not accessed for this period.
cluster.type	The cluster.type attribute determines what type of cache is

	created when clustering is available. The acceptable values are fully-distributed. local and invalidating. This cluster.type options are fully explained next on the article under HazelCast cache mechanisms
backup-count	The backup-count attribute controls how many cluster members should hold a backup of the key/value pair. It guarantees that a distributed cache has a specific number of backups, in case of a node holding that bit of the cache dies, those caches are still accessible. The more

	backups you have, the more memory will be consumed.
eviction-policy	When the eviction-policy attribute is set to NONE, the cache will not have a bounded capacity and the maxItems attribute will not apply. Any other value will cause the maxItems attribute to be enabled. Also, use **LRU** (Least Recently Used) or **LFU** (Least Frequently Used) algorithm with clustered caches so that the value is compatible in both

	modes (required during startup). Note that the actual value (for example, LRU) is of no consequence for the non-clustered caches and eviction is performed as for any Google Guava CacheBuilder created cache.
eviction-percentage	The percentage of the cache to evict
merge-policy	The merge-policy attribute determines how Hazelcast recovers from split brain syndrome (see **http://docs.hazelcast.org/docs/latest-**

	development/manual/html/Network_Partitioning/Split_Brain_Syndrome.html) , for example: com.hazelcast.map.merge.PassThroughMergePolicy com.hazelcast.map.merge.PutIfAbsentMapMergePolicy (the default) com.hazelcast.map.merge.HigherHitsMapMergePolicy com.hazelcast.map.merge.LatestUpdateMapMergePolicy

Hazelcast cache mechanisms

With Hazelcast the cache is distributed across the clustering members, doing a more linear distribution of the memory usage. In the alfresco implementation you have more mechanisms available to define different cache cluster types.

1 – Fully Distributed

This is the normal value for a hazelcast cache. Cache values (key value pairs) will be evenly distributed across cluster members. Leads to more remote lookups when a get request is issued and that value is present in other node (remote).

2 – Local cache (local)

Some caches you may not actually want them to be clustered at all (or distributed), so this option works as a unclustered cache.

3 – Invalidating

This is a local (cluster aware) cache that sets up a messenger that sends invalidation messages to the remaining cluster nodes if you updated an item in the cache. Can be useful to store something that its not serializable, because all the values stored in a hazelcast cache must be serializable.

That is the way hazlecast send the information to the member its going to be stored on.

If you got a cache where there is an enormous number of reads and very rare writes, using a invalidating cache can be the best approach, very likely to the way ehcache used to work. This was introduced also because there were some non serializable values in alfresco that could not reside on a fully-distributed cache.

The way we define the caches (on our cache.properties file is as follows)

- cache.aclSharedCache.tx.maxItems=40000
- cache.aclSharedCache.maxItems=100000
- cache.aclSharedCache.timeToLiveSeconds=0
- cache.aclSharedCache.maxIdleSeconds=0
- cache.aclSharedCache.cluster.type=fully-distributed
- cache.aclSharedCache.backup-count=1
- cache.aclSharedCache.eviction-policy=LRU
- cache.aclSharedCache.eviction-percentage=25
- cache.aclSharedCache.merge-policy=hz.ADD_NEW_ENTRY

How do i find the amount of memory taken by the cache

For computing how much heap memory is being taken up by the cache properties subtract the Java Memory usage which shows on your operating system from the heap memory which shows on JConsole. This equals the total repository cache size.

For caches like node property cache and node aspect cache, this is entirely dependent on the size of the arrays of properties or aspects respectively for each node entity in the cache. Estimations can be done via iterative testing.

Conclusion

Repository caches play an important role on the repository performance and they should be tuned wisely.

I recommend using a cache tracing method before you decide to tune your caches . Don’t forget to check your logs for signs that can help you on decide that you need to tune your caches. Monitoring your JVM heap and analysing your garbage collection is also recommended for cache tuning. Knowing your cache usage patters will help your to tune your caches.