Read-Through, Write-Through, Write-Behind Caching and Refresh-Ahead

Coherence supports transparent read/write caching of any data source, including databases, web services, packaged applications and file systems, however, databases are the most common use case. As shorthand "database" will be used to describe any back-end data source. Effective caches must support both intensive read-only **and** read/write operations, and in the case of read/write operations, the cache and database must be kept fully synchronized. To accomplish this, Coherence supports **Read-Through, Write-Through, Refresh-Ahead and Write-Behind** caching.

## **9.1 Pluggable Cache Store**

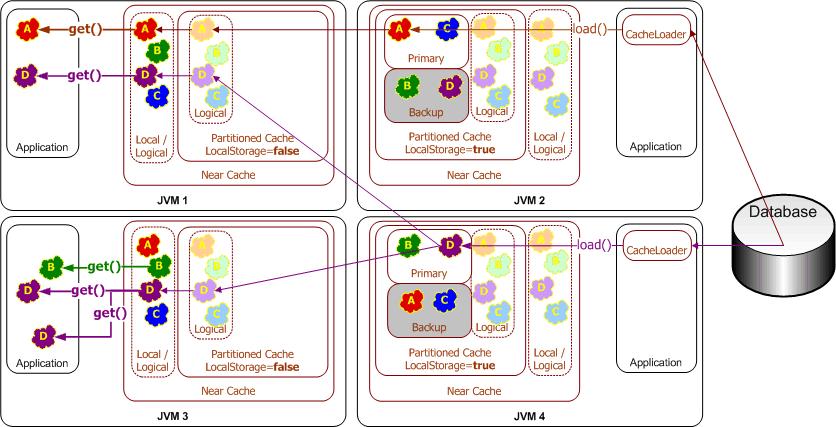
A **CacheStore** is an application-specific adapter used to connect a cache to an underlying data source. The **CacheStore** implementation accesses the data source by using a data access mechanism (for example, [Hibernate,](https://docs.oracle.com/cd/E13924_01/coh.340/e12192/usehibernatecs.htm#COHUG370) [Toplink Essentials](https://docs.oracle.com/cd/E13924_01/coh.340/e12192/configfortoplink.htm#COHUG352), [JPA](https://docs.oracle.com/cd/E13924_01/coh.340/e12192/configforjpa.htm#COHUG362), application-specific JDBC calls, another application, mainframe, another cache, and so on). The **CacheStore** understands how to build a Java object using data retrieved from the data source, map and write an object to the data source, and erase an object from the data source.

Both the data source connection strategy and the data source-to-application-object mapping information are specific to the data source schema, application class layout, and operating environment. Therefore, this mapping information must be provided by the application developer in the form of a **CacheStore** implementation. See ["Creating a CacheStore Implementation"](https://docs.oracle.com/cd/E13924_01/coh.340/e13819/readthrough.htm#CHDGHIIF) for more information.

## **9.2 Read-Through Caching**

When an application asks the cache for an entry, for example the **key** **X**, and **X** is not already in the cache, Coherence will automatically delegate to the CacheStore and ask it to load **X** from the underlying data source. If **X** exists in the data source, the **CacheStore** will load it, return it to Coherence, then Coherence will place it in the cache for future use and finally will return **X** to the application code that requested it. This is called **Read-Through** caching. Refresh-Ahead Cache functionality may further improve read performance (by reducing perceived latency). See ["Refresh-Ahead Caching"](https://docs.oracle.com/cd/E13924_01/coh.340/e13819/readthrough.htm#CHDGFFIA) for more information.

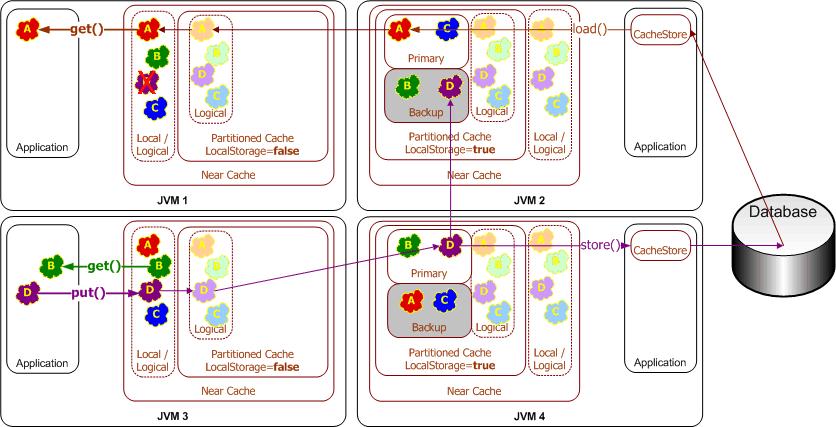
***Figure 9-1 Read Through Caching***

  
[Description of "Figure 9-1 Read Through Caching"](https://docs.oracle.com/cd/E13924_01/coh.340/e13819/img_text/read-through.htm)

## **9.3 Write-Through Caching**

Coherence can handle updates to the data source in two distinct ways, the first being **Write-Through**. In this case, when the application updates a piece of data in the cache (that is, calls **put**(...) to change a cache entry,) the operation will not complete (that is, the **put** will not return) until Coherence has gone through the **CacheStore** and successfully stored the data to the underlying data source. This does not improve write performance at all, since you are still dealing with the latency of the write to the data source. Improving the write performance is the purpose for the *Write-Behind Cache* functionality. See ["Write-Behind Caching"](https://docs.oracle.com/cd/E13924_01/coh.340/e13819/readthrough.htm#CHDIEDAA) for more information.

***Figure 9-2 Write-Through Caching***

  
[Description of "Figure 9-2 Write-Through Caching"](https://docs.oracle.com/cd/E13924_01/coh.340/e13819/img_text/write-through.htm)

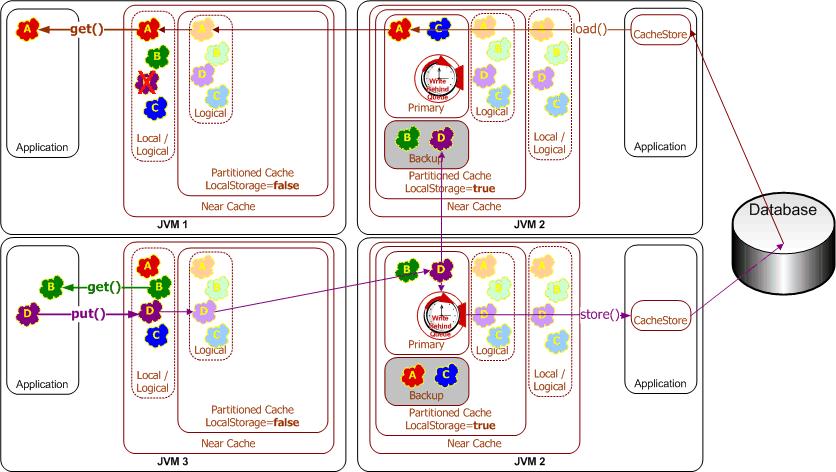
## **9.4 Write-Behind Caching**

In the **Write-Behind** scenario, modified cache entries are asynchronously written to the data source after a configurable delay, whether after 10 seconds, 20 minutes, a day or even a week or longer. For **Write-Behind** caching, Coherence maintains a write-behind queue of the data that must be updated in the data source. When the application updates **X** in the cache, **X** is added to the write-behind queue (if it isn't there already; otherwise, it is replaced), and after the specified write-behind delay Coherence will call the CacheStore to update the underlying data source with the latest state of **X**. Note that the write-behind delay is relative to the first of a series of modifications—in other words, the data in the data source will never lag behind the cache by more than the write-behind delay.

The result is a "read-once and write at a configurable interval" (that is, much less often) scenario. There are four main benefits to this type of architecture:

* The application improves in performance, because the user does not have to wait for data to be written to the underlying data source. (The data is written later, and by a different execution thread.)
* The application experiences drastically reduced database load: Since the amount of both read and write operations is reduced, so is the database load. The reads are reduced by caching, as with any other caching approach. The writes, which are typically much more expensive operations, are often reduced because multiple changes to the same object within the write-behind interval are "coalesced" and only written once to the underlying data source ("write-coalescing"). Additionally, writes to multiple cache entries may be combined into a single database transaction ("write-combining") by using the **[CacheStore.storeAll()](https://download.oracle.com/otn_hosted_doc/coherence/340/com/tangosol/net/cache/CacheStore.html" \l "storeAll(java.util.Map))** method.
* The application is somewhat insulated from database failures: The **Write-Behind** feature can be configured in such a way that a write failure will result in the object being re-queued for write. If the data that the application is using is in the Coherence cache, the application can continue operation without the database being up. This is easily attainable when using the Coherence Partitioned Cache, which partitions the entire cache across all participating cluster nodes (with local-storage enabled), thus allowing for enormous caches.
* Linear Scalability: For an application to handle more concurrent users you need only increase the number of nodes in the cluster; the effect on the database in terms of load can be tuned by increasing the write-behind interval.

***Figure 9-3 Write Behind Caching***

  
[Description of "Figure 9-3 Write Behind Caching"](https://docs.oracle.com/cd/E13924_01/coh.340/e13819/img_text/write-behind.htm)

## **9.5 Write-Behind Requirements**

While enabling write-behind caching is simply a matter of adjusting one configuration setting, ensuring that write-behind works as expected is more involved. Specifically, application design must address several design issues up-front.

The most direct implication of write-behind caching is that database updates occur outside of the cache transaction; that is, the cache transaction will (in most cases) complete before the database transaction(s) begin. This implies that the database transactions must never fail; if this cannot be guaranteed, then rollbacks must be accommodated.

As write-behind may re-order database updates, referential integrity constraints must allow out-of-order updates. Conceptually, this means using the database as ISAM-style storage (primary-key based access with a guarantee of no conflicting updates). If other applications share the database, this introduces a new challenge—there is no way to guarantee that a write-behind transaction will not conflict with an external update. This implies that write-behind conflicts must be handled heuristically or escalated for manual adjustment by a human operator.

As a rule of thumb, mapping each cache entry update to a logical database transaction is ideal, as this guarantees the simplest database transactions.

Because write-behind effectively makes the cache the system-of-record (until the write-behind queue has been written to disk), business regulations must allow cluster-durable (rather than disk-durable) storage of data and transactions.

In earlier releases of Coherence, rebalancing (due to failover/failback) would result in the re-queuing of all cache entries in the affected cache partitions (typically 1/N where N is the number of servers in the cluster). While the nature of write-behind (asynchronous queuing and load-averaging) minimized the direct impact of this, for some workloads it could be problematic. Best practice for affected applications was to use **[com.tangosol.net.cache.VersionedBackingMap](https://download.oracle.com/otn_hosted_doc/coherence/340/com/tangosol/net/cache/VersionedBackingMap.html)**. As of Coherence 3.2, backups are notified when a modified entry has been successfully written to the data source, avoiding the need for this strategy. If possible, applications should deprecate use of the **VersionedBackingMap** if it was used only for its write-queuing behavior.

## **9.6 Refresh-Ahead Caching**

In the **Refresh-Ahead** scenario, Coherence allows a developer to configure a cache to automatically and asynchronously reload (refresh) any recently accessed cache entry from the cache loader before its expiration. The result is that after a frequently accessed entry has entered the cache, the application will not feel the impact of a read against a potentially slow cache store when the entry is reloaded due to expiration. The asynchronous refresh is only triggered when an object that is sufficiently close to its expiration time is accessed—if the object is accessed after its expiration time, Coherence will perform a synchronous read from the cache store to refresh its value.

The refresh-ahead time is expressed as a percentage of the entry's expiration time. For example, assume that the expiration time for entries in the cache is set to 60 seconds and the refresh-ahead factor is set to 0.5. If the cached object is accessed after 60 seconds, Coherence will perform a *synchronous* read from the cache store to refresh its value. However, if a request is performed for an entry that is more than 30 but less than 60 seconds old, the current value in the cache is returned and Coherence schedules an *asynchronous* reload from the cache store.

Refresh-ahead is especially useful if objects are being accessed by a large number of users. Values remain fresh in the cache and the latency that could result from excessive reloads from the cache store is avoided.

The value of the refresh-ahead factor is specified by the **<refresh-ahead-factor>** sub element of the [<**read-write-backing-map-scheme**>](https://docs.oracle.com/cd/E13924_01/coh.340/e13818/appcacheelements.htm#COHDG367) element in the coherence-cache-config.xml file. Refresh-ahead assumes that you have also set an expiration time (**<expiry-delay>**) for entries in the cache.

<https://docs.oracle.com/cd/E13924_01/coh.340/e13819/readthrough.htm>