Typical Caching Architectures

Objectives

After completing this lesson, you should be able to:

- Explain the different caching architectures that are typical of Coherence applications
- Describe how to distinguish each caching strategy
- Explain the purpose and function of each caching strategy

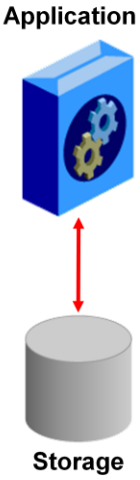
Agenda

Evolution of Data Grid Design Patterns

- Single-Application Instances
- Multiple-Application Instances
- Local Caching Pattern
- Distributed Coherent Caching Pattern
- Cache-Aside and Read-Through Patterns
- Write-Through and Write-Behind Patterns
- External Update Patterns
- Near-Caching and Client-Side Processing Patterns
- Server-Side Processing and Distributed Computing Patterns

Single-Application Instances

In the beginning, applications were simple, and communicated directly with the database.

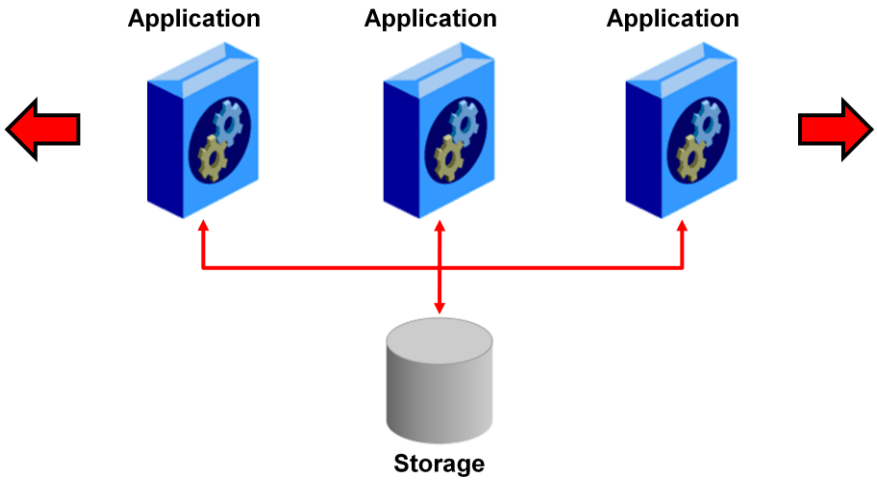


10 - 4

Applications are often written without any type of caching. These applications work directly with the database and do not involve multiple instances of programs that work together and involve many users. This type of application is very easy to write and maintain, but is very basic and is not the basis for enterprise computing. As the number of user requests grows, the application design needs to change to accommodate that growth.

Multiple-Application Instances

Then things started to scale, because customer requests needed more capacity.

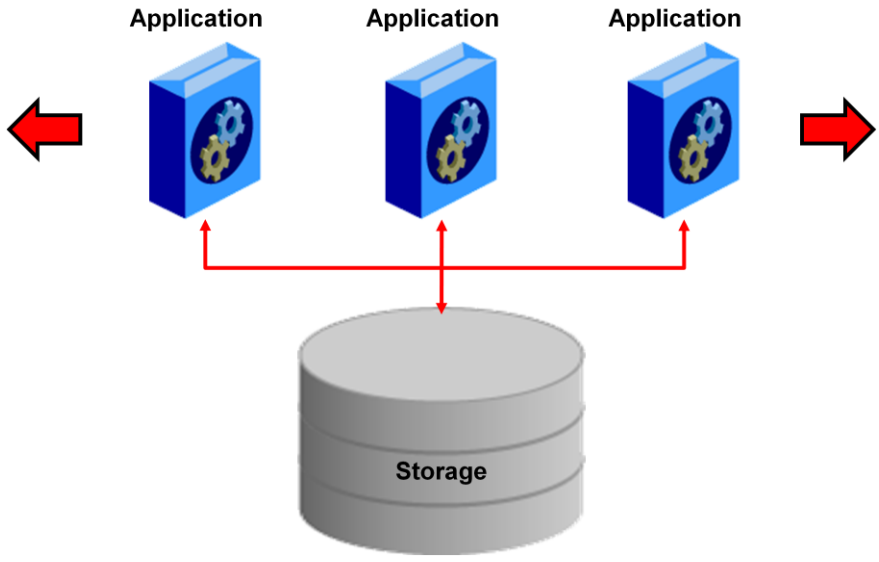


10 - 5

The first step in increasing the capacity of an application is to increase the number of instances of the application that can handle user requests. Often this is done without taking into consideration the capacity of the back-end data source. All application instances continue to use the same database.

Multiple-Application Instances

Storage and I/O was determined to be a bottleneck, so those resources were scaled up.

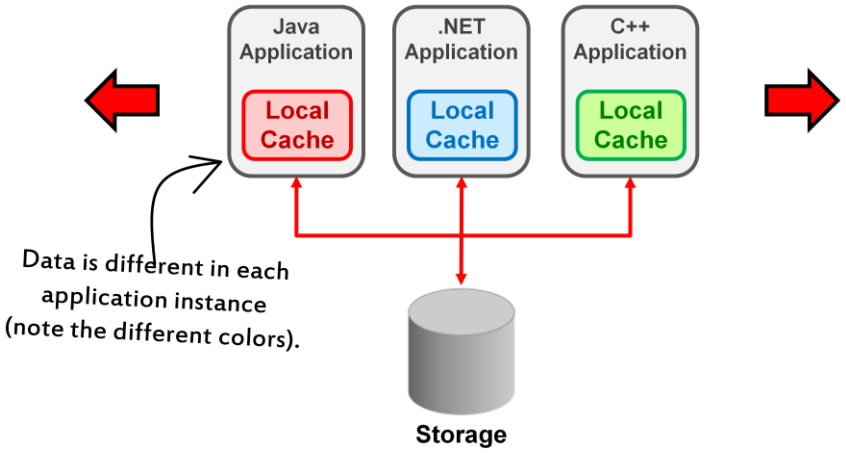


10 - 6

As user requests flow into the application instances, the back-end data source is not able to keep up with the number of requests being made by all of the application instances. This is often the time when a company upgrades their hardware and software resources for the back-end to enable it to handle the load. Eventually, the data source can be scaled up only so much until the process becomes very cost-prohibitive. If the data source is a database, then tactics such as database partitioning are employed to reduce the load, by spreading it across numerous databases that work together to form a unified data store. This can cause a maintenance issue.

Local Caching Pattern

But that was not enough, so applications started using the "local caching pattern."

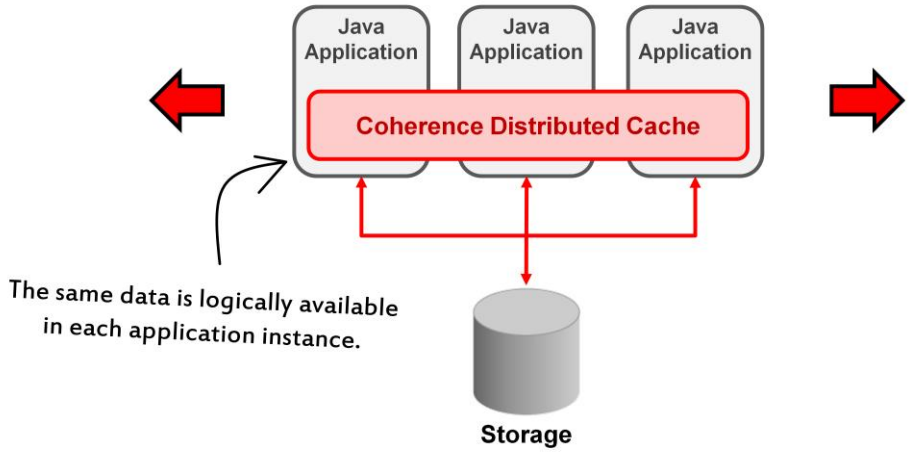


10 - 7

It becomes apparent at some point that read-intensive applications are overtaxing the data source, so a caching strategy is introduced to lower the number of requests (and thereby the load) on the data source by keeping frequently read data close to the application that is using the data. The data is reused, and the application benefits from lower latency because the data source was not involved, and the data source benefits from a lower load because it is handling fewer requests. However, each application instance contains a cache that is not necessarily in sync with the other application instances, and each instance is unaware of the data in the other.

Distributed Coherent Caching Pattern

With *local* caching, information was not consistent across servers, so caching was distributed.

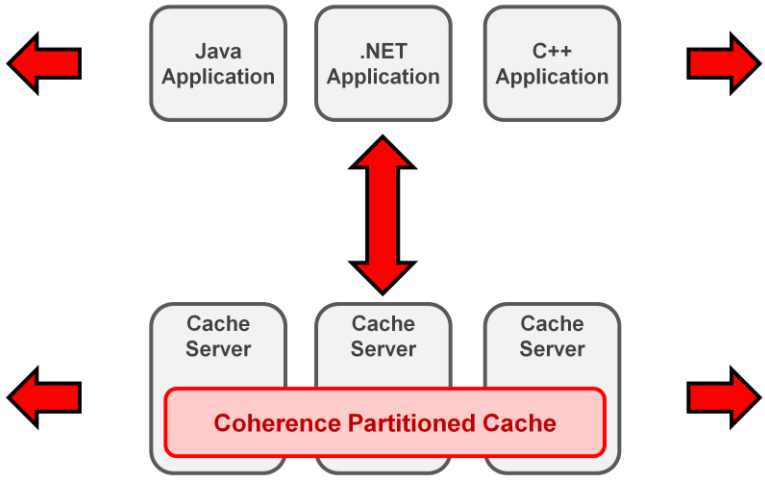


10 - 8

In order to keep data consistent across application instances, a distributed caching service is introduced that is aware of the data in each instance. This provides the ability to cache data in the applications and to keep the data consistent across each instance. In this case, it could be a replicated cache or a partitioned cache. The rest of the lesson assumes a partitioned cache.

Introducing the Caching Infrastructure Layer

Caching became a distinct *resource,* so that resource became a new layer.

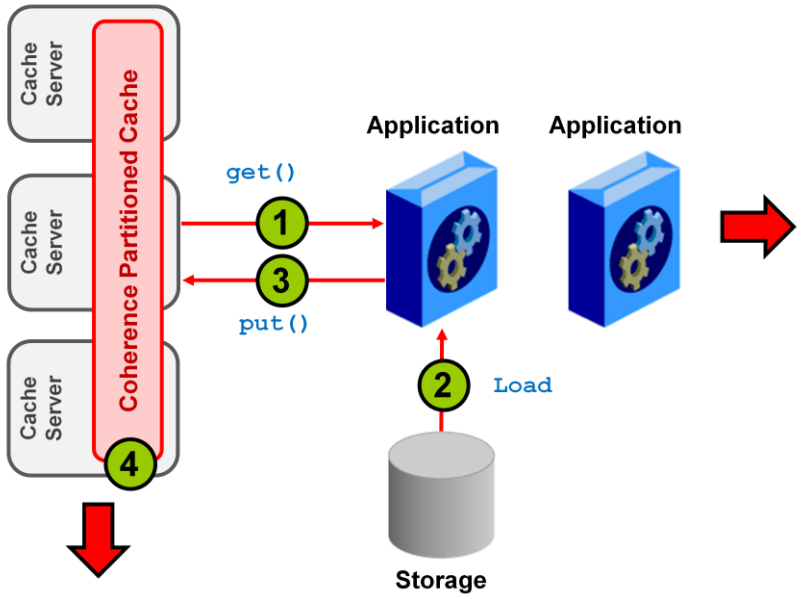


10 - 9

Because there was a service that was cache-aware, it made sense to create a proper infrastructure around the technology that could potentially be used by applications written in different languages. With this infrastructure in place, more robust caching patterns could be realized as well.

Cache-Aside Patterns

Cache as a separate resource:



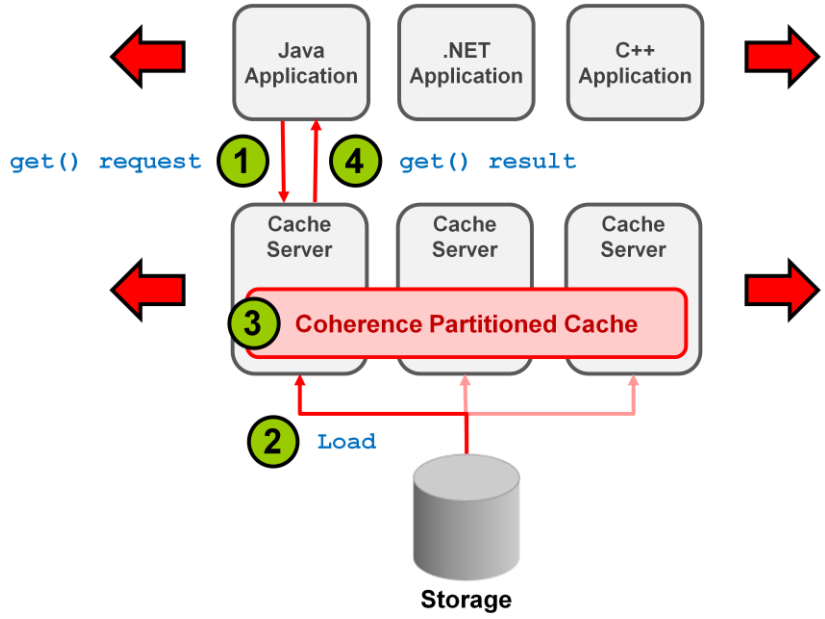
10 - 10

A typical caching pattern is to manually cache data:

1. An application requests data from the cache. If the data is in the cache, then the application has the data it needs and continues processing.
2. If the data requested is not in the cache, then the application loads the data from the data source directly.
3. After loading the data from the data source, the application places the data in the cache for future requests.
4. The distributed caching service manages the cache, and where to put the data.

Read-Through Pattern

The cache was integrated with back-end data sources.



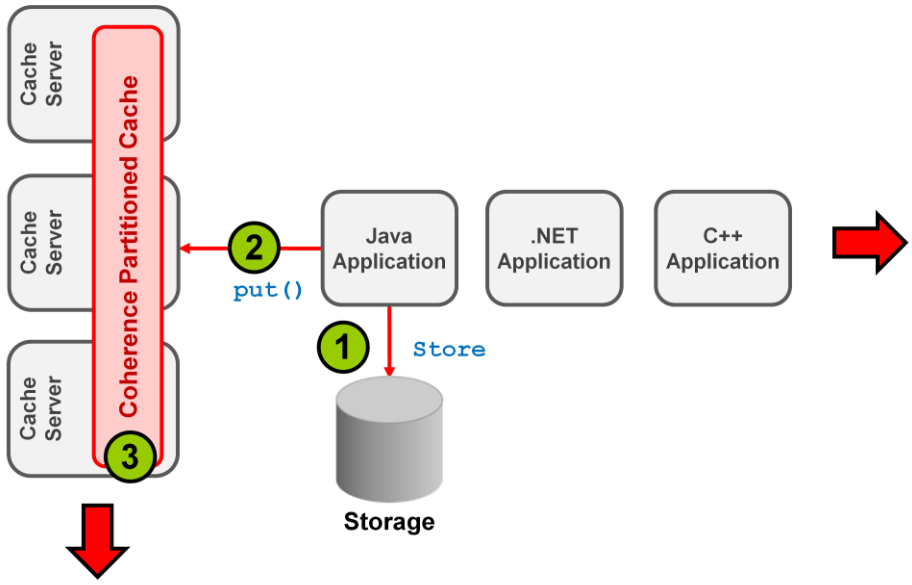
10 - 11

The read-through pattern involves using a `CacheLoader` implementation that knows how to work with the back-end data source. This abstracts the data source from the application, which works directly with the cache. The cache then manages getting data from the data source on behalf of the application:

1. The application requests data from the cache. If the data is there, then the application has the data and continues processing.
2. If the data is not in the cache, then the caching service uses a configured `CacheLoader` to load the data from the back-end data source.
3. The caching service places the data in the cache.
4. The data is returned to the application.

Write-Aside Pattern

Updates required the ability to write to the cache.



10 - 12

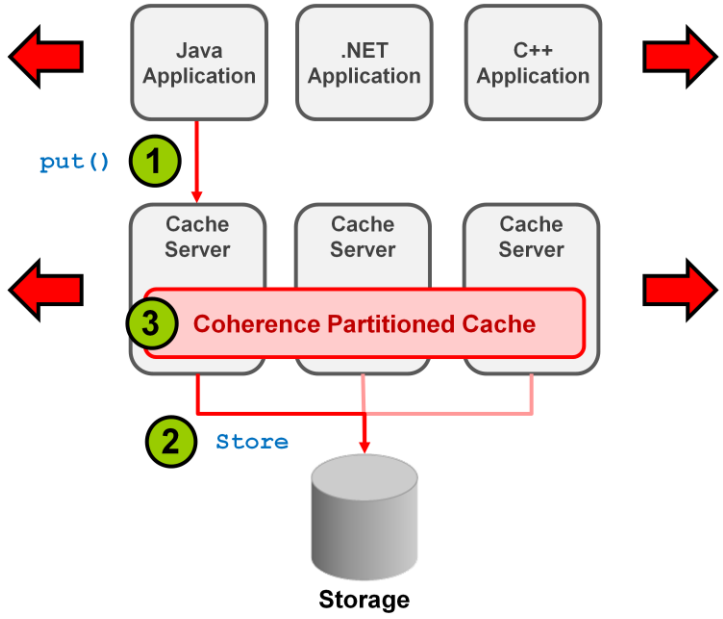
At some point, an application needs to update data that exists in the cache. A typical pattern for this is called the write-aside pattern:

1. The application stores the data directly in the database.
2. The application then separately updates the cache with the data.
3. The caching service places the data in the cache.

One potential issue for this pattern is that in the event of a failure of the application performing this task, it is possible that the data is inconsistent between the data source and the cache.

Write-Through Pattern

Writing to the cache can automatically update the database.



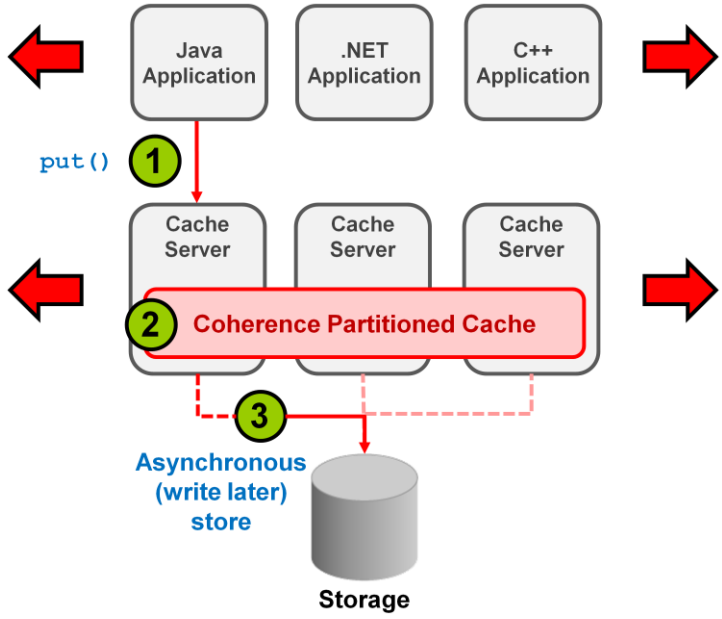
10 - 13

The write-through pattern is very similar to the read-through pattern, except it is for storing data in the data source. This pattern involves using a `CacheStore` implementation that knows how to work with the back-end data source. This abstracts the data source from the application, which works directly with the cache. The cache then takes care of putting data in the data source on behalf of the application:

1. The application puts data in the cache.
2. The caching service uses a configured `CacheStore` to store the data in the back-end data source.
3. The caching service places, or updates, the data in the cache.

Write-Behind Pattern

The cache can be the system of record for updates.



10 - 14

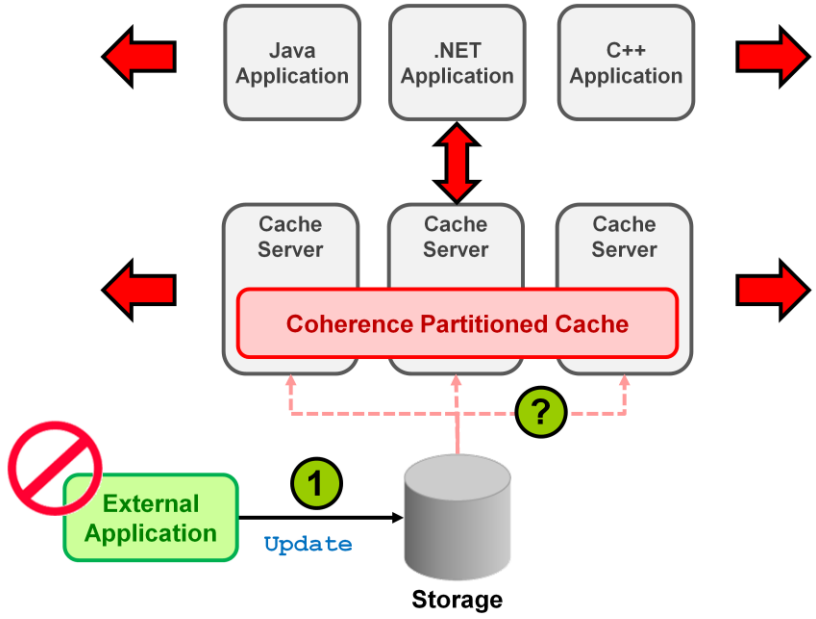
The write-behind pattern takes things a step further. This pattern also involves using a `CacheStore` implementation that knows how to work with the back-end data source. This abstracts the data source from the application, which works directly with the cache. The cache then takes care of putting data in the data source on behalf of the application. And, in the case of the write-behind strategy, the caching service becomes the system of record for the data instead of the database. The write-behind strategy involves the following:

1. The application puts data in the cache.
2. The caching service places, or updates, the data in the cache.
3. The caching service uses a configured `CacheStore` to store the data in the back-end data source. All updates are queued so that they are processed in order. There are no guarantees that the update occurs immediately, because Coherence manages the back-end update asynchronously.

Keep in mind that multiple updates on a single entry result in a single update to the database.

External Update Anti-Pattern

External applications can disrupt cache data consistency:



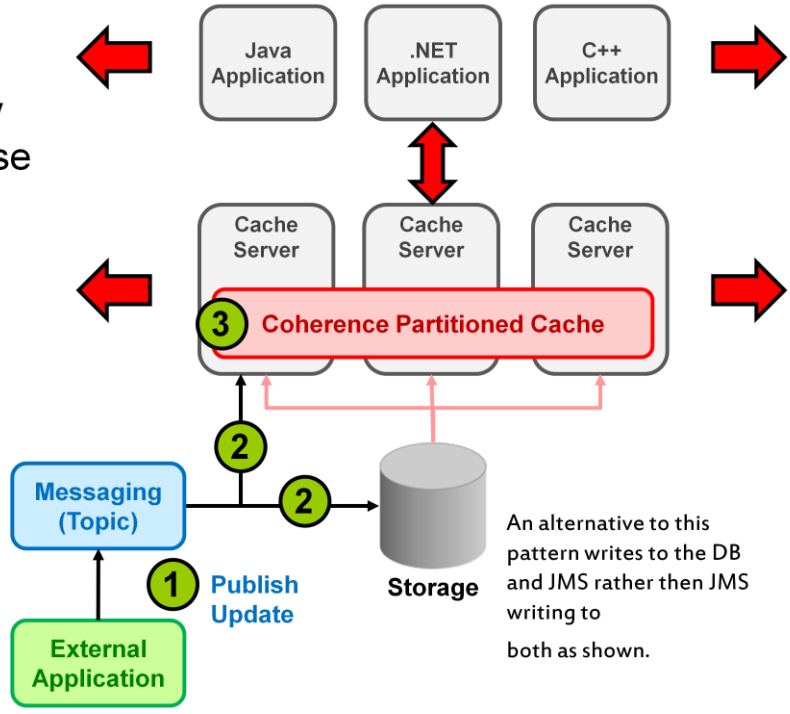
10 - 15

One potential issue that may arise with using the write-behind strategy (as well as other strategies) is that an application that is external to the caching application updates the back-end data source directly, bypassing the cache. This causes the data in the cache to be stale, and usually there is no mechanism to notify the cache that the data is invalid:

1. An external application updates the data source directly.
2. The caching service has no idea that anything has occurred with the data source.

External Update Pattern Using Messaging

Messaging can help manage consistency between the database and the cache.



10 - 16

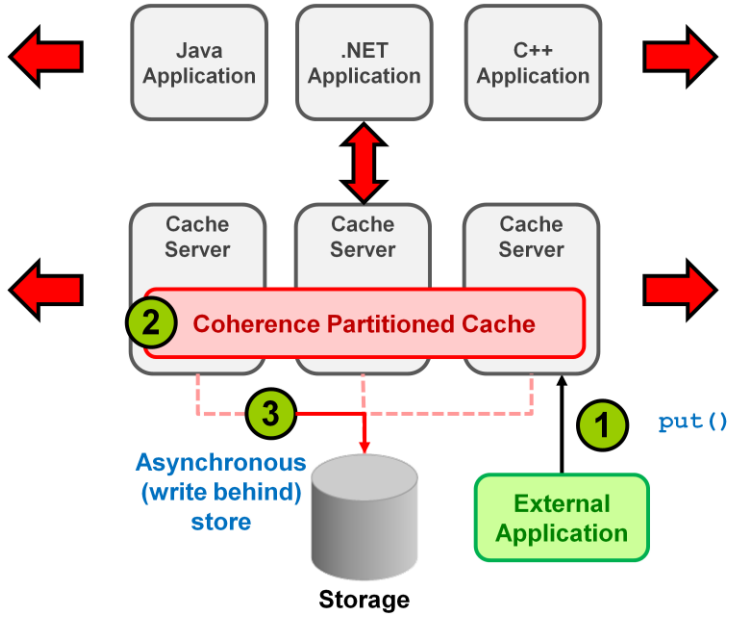
An external application can manage keeping the data in-sync between the cache and the data source by using a messaging topic:

1. External application publishes an update.
2. The messaging topic subscribers receive the update, update the data in the data source, and send the update to the caching service.
3. The caching service updates the data in the cache.

JMS is deprecated as of Coherence version 3.6, and a messaging topic or queue would have to be via a Java EE application server such as WebLogic Server.

External Update with Direct Access

External applications can update the database by using the cache.



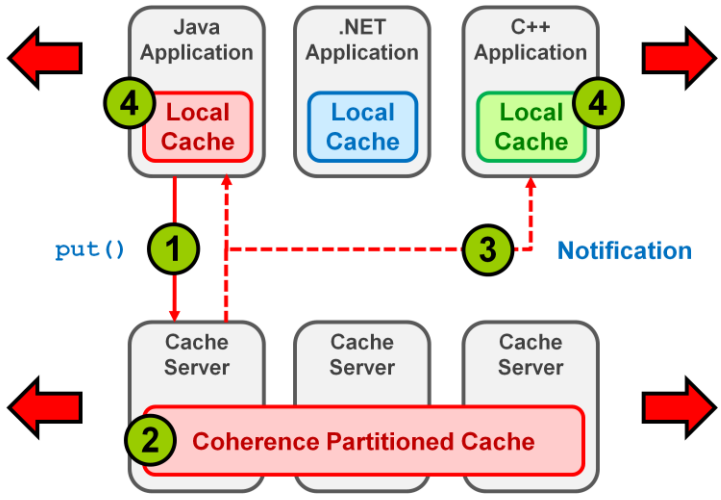
10 - 17

An external application can also manage keeping the data in-sync between the cache and the data source by using the caching service directly, as in this example that uses write-behind caching:

1. External application sends an update directly to the caching service.
2. The caching service updates the data in the cache.
3. The write-behind mechanism asynchronously manages writing the data to the data source.

Near-Caching Pattern

Near-caching provides fast local read performance for a larger back-end cache, and uses events to invalidate data when it is updated in the back cache.



10 - 18

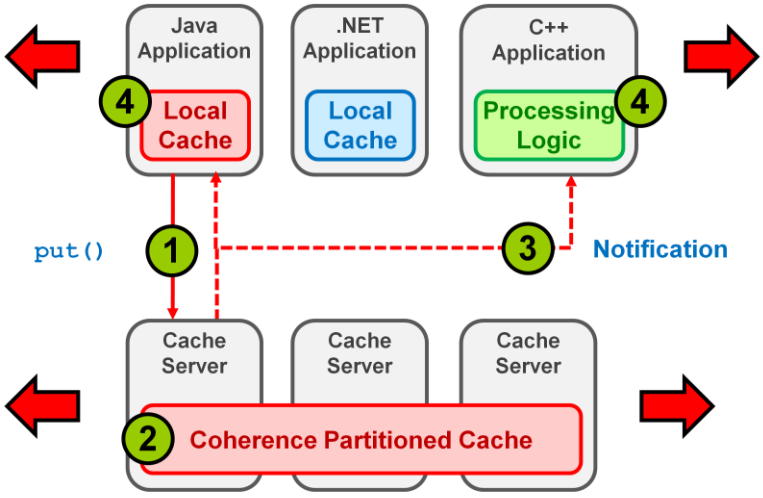
Coherence partitioned caching is able to handle very large data sets, up to the terabyte range. This cache is very scalable and provides predictable performance, but read performance is slower than in local caches. The near-caching pattern provides a size-limited local cache, which is colocated with the application, and backed by the larger data set. The local cache is a cached subset of the back cache, and provides very fast read performance for frequently read data. The near-caching pattern also uses event notification to invalidate locally cached data entries as they are updated in the back cache.

The near caching pattern:

1. The application updates the data entry by using the Coherence `put()` method, and the request is sent to the partitioned caching service for the back cache.
2. The data entry is updated in the back cache.
3. An event is published that the data entry has been updated.
4. The data entry exists in the local cache of another application instance. The local entry and that entry receive the event that the back cache entry was updated, and invalidate themselves.

Client-Side Event Processing Pattern

Clients can react to event notifications in real-time.



10 - 19

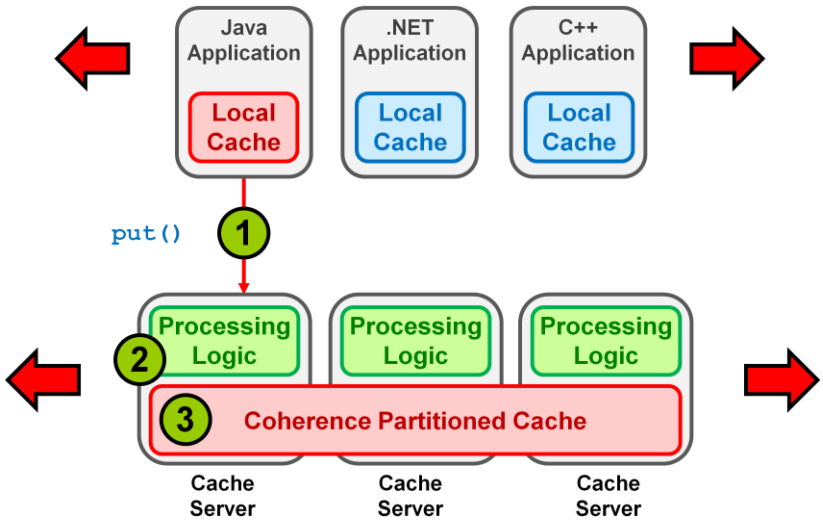
Client processes can register map listeners that can subscribe to certain events when data entries are inserted, updated, or deleted. This was discussed in detail in the lesson titled "Observing Data Grid Events."

The process works as follows:

1. The application updates data in the cache by calling `put()`, and the data is sent to the caching service.
2. The caching service updates the cache.
3. An event containing information about what data has changed is sent to all subscribing listeners.
4. The subscribers receive and handle the event programmatically.

Server-Side Event Processing Pattern

Servers can react to events in real time.



10 - 20

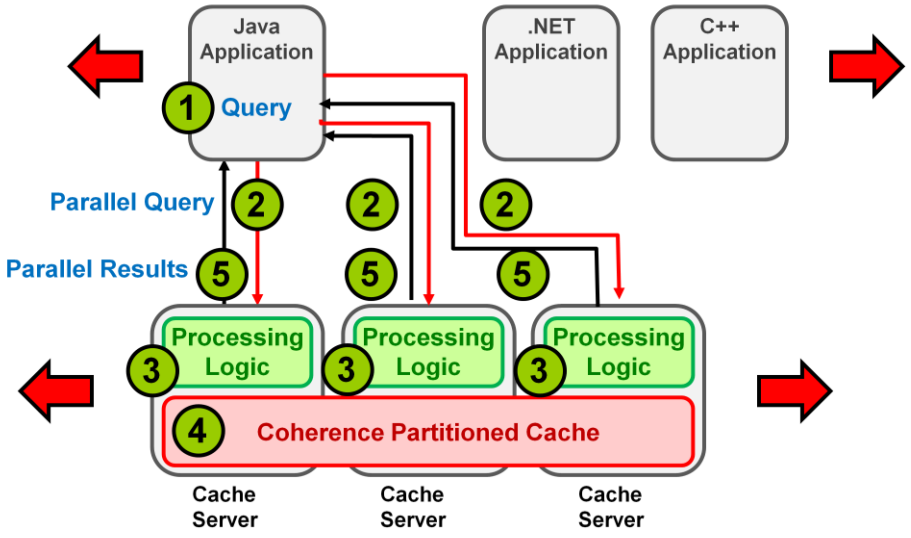
Server-side processes can register map triggers that can subscribe to certain events when data entries are inserted, updated, or deleted. This was discussed in detail in the lesson titled "Observing Data Grid Events."

The process works as follows:

1. The application updates data in the cache by calling `put()`, and the data is sent to the caching service.
2. An event is triggered before the cache is modified, and the server side executes some code that performs any type of processing before the data gets placed into the cache.
3. The caching service updates the cache.

Server-Side Processing Pattern: Queries, Map Reduction, and Computation

The cache can be queried, and process data in parallel.



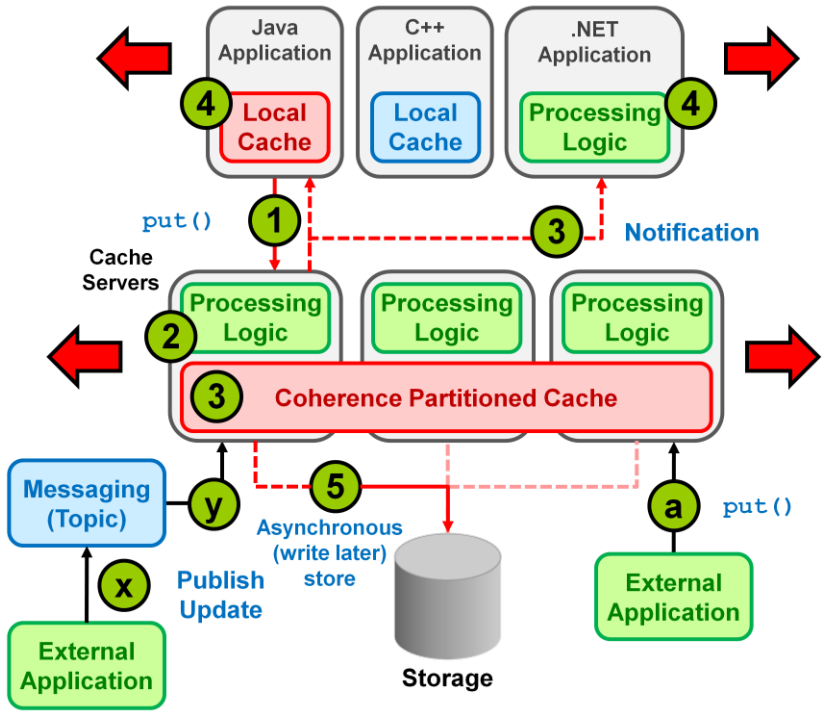
10 - 21

Using the query, `EntryProcessor`, and server-side event mechanisms, a Coherence cluster can process data entries across the grid in parallel; and if the processing involves inserting, updating, or deleting a cache entry, the server side can process the event without having to send data over the network. All processing is spread across the cluster where the cache entries exist, which maximizes CPU utilization and reduces latency.

Combining Features for a Scalable Platform

Combining these features provides the ability to:

- Process data in parallel
- Integrate with a data source
- Handle external data source updates
- Handle events in real time



Combining the various strategies described in this lesson provides a highly scalable architecture that can handle any caching needs:

1. The application updates the cache by calling `put()`, and the data is sent to the caching service.
2. Before the cache gets modified, the server side triggers an event and performs some processing on the data.
3. The cache is updated with the new data, and an event is sent to a subscribing client.
4. The subscribing clients receive and handle the event.
5. At some configured interval, the write-behind strategy in use asynchronously updates the back-end data store.

a. An external application uses the cache directly to update data in the data source via the write-behind strategy used by the cache.

x. Another external application updates the cache with a messaging strategy by publishing the update to a JMS topic.

y. The topic subscriber receives the update and updates the cache directly by using the `put()` method.

Summary

In this lesson, you should have learned how to:

- Describe the different caching strategies that are available with Coherence
- Distinguish each caching strategy
- Explain the purpose and function of each caching strategy