An Oracle White Paper
June 2016

Oracle Coherence 12c
Planning a Successful Deployment

Disclaimer

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Executive Overview

Taking a Coherence Proof of Concept (PoC) application and making it ready for production involves a range of considerations and additional steps, which are not always obvious to architects and developers. The environment of a production application is quite different from that of development and requires additional steps to secure, monitor, and manage the application. High availability plans to meet expected SLAs need to be put in place, which in turn involves appropriate procedures to upgrade, patch, and recover the system.

This white paper addresses these issues and provides guidance about what to consider and how to plan the successful rollout of a Coherence application to a production environment.

Introduction

Planning the rollout of a Coherence application into a production environment can vary greatly in its complexity. Many of the considerations in this white paper may not be relevant for simple Coherence applications; however, each main section has recommendations that should be generally applicable.

This white paper intends to complement and extend, rather than replace or replicate, existing Coherence documentation and provides references to existing sources where appropriate. It includes guidance, tips, and experiences from "field engineers", customers, Oracle Support, and Coherence engineers. With this in mind, its structure is similar to a checklist that addresses each step of the process in sequence.

Design, Development and Testing

Best Practice

Save Time, Reuse, Don't Re-Write

Before you start writing code or architecting a new solution, check whether an existing implementation for what you want to do is available in the Coherence API or in the Coherence Incubator. The Coherence Incubator contains a number of production ready code templates to address a range of problems. Consider customizing or extending these sources if an existing implementation does not directly solve the problem. Take advantage of the existing testing, monitoring, tuning, and documentation.

Coherence is a Distributed Technology so Develop and Test in a Distributed Environment

Coherence is a distributed technology. It may be tempting to perform all development and testing on a single machine, but Coherence behaves differently in a distributed environment. Communications etc. take longer, so flush out any design and development issues early by regularly testing in a distributed environment.

Don’t waste time identifying problems that have already been fixed

Begin your testing with the latest patch release of Coherence that is available from Oracle Support. This may seem obvious, but it is quite common for developers to spend time investigating Coherence problems already fixed in a recent patch. Subscribe to the RSS feed for notifications when a new patch becomes available. Unfortunately, patches are only available from Oracle Support and not OTN. So if you are evaluating Coherence and not yet a customer, then please speak to you local Oracle contact.

Follow Best Practices

The Production Checklist, Performance Tuning guide and Best Practices for Coherence*Extend documentation can help you avoid many common problems. Read and re-read these resources while you are testing your application or setting up a production environment. Also, perform network tests using the bundled message bus, datagram, and multicast tests before you start. These tools identify networking problems before starting to test your application.

For instance, if you plan to use a virtualized environment, then not using the latest network drivers can dramatically affect network communications. Running the datagram or message bus tests can highlight such problems by showing a throughput that is significantly less than the expected ~100MB p/s - over a 1GBE network. Failing to perform these simple tests can easily waste a lot of time diagnosing error messages in Coherence log files or looking at other symptoms rather than focusing on the cause of the problem.

Avoid Anti-Patterns

These are design patterns that don’t work well in a distributed architecture. Below are just some of these patterns and practices you should consider carefully before using:

- **Distributed transactions** – requires copying multiple entries for read-consistency and rollback. This white paper discusses alternative approaches.
- **Client locks** – increases contention and hinders scalability. Executing operations where the data resides using entry processors, thus removing local locks, is usually a better alternative.
- **Client-based processing** – increases contention and hinders scalability. A better alternative is to use entry processors or Coherence Live Events to process the data where it resides. Entry processors are like database stored procedures but also allow processing to be re-run, or failed-over, if the first processing node fails. The Live Events feature, introduced in Coherence 12c, completely transfers processing to the cluster. Event Interceptors are registered to fire when the cluster or cache entries change and are executed where the event is raised – inside the cluster. This enables Coherence to scale, failover, and recover the processing that the interceptors perform. In addition, you can chain interceptors together to execute complex processing logic.
- **Un-indexed and un-targeted queries** – decreases performance. Although indexes consume memory, they can improve query performance by several orders of magnitude. Explore custom indexes to minimize their overhead or analyze index performance to balance storage and performance considerations. This white paper further discusses these approaches.

Need Help? – Search the Documentation and Check the Forum

It seems obvious but often not done. The online documentation explains the most common tasks, features, and supports searching.

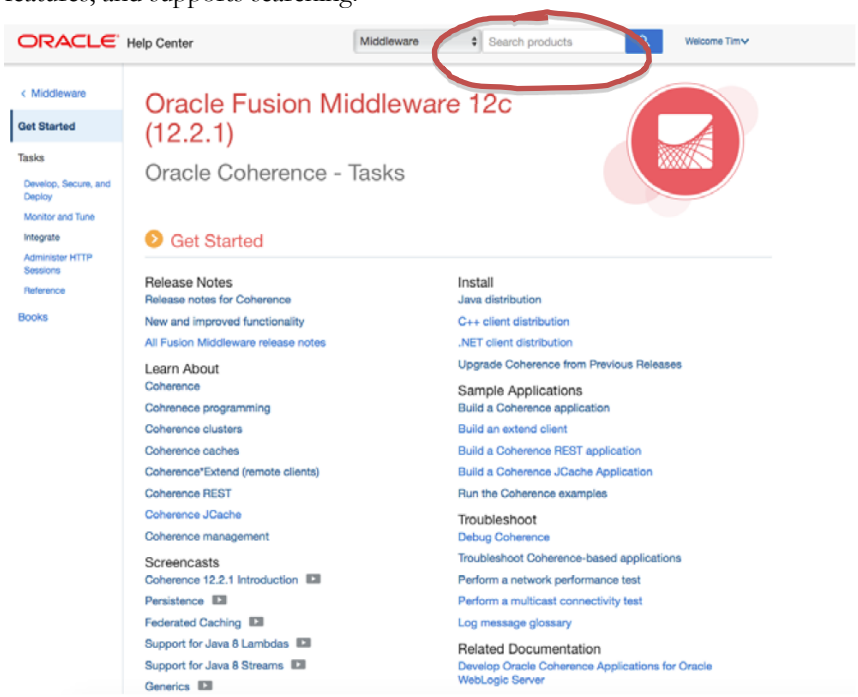


Figure 1. Oracle Documentation Search Screen

The Coherence Support Forum is also a great source of information and an easy way to reach out to experts in the Oracle Community – including the Coherence engineering team. The Oracle Technology Network (OTN) Coherence Home Page provides useful links to examples and online tutorials, and the Oracle A-Team (Architects Team) offer Coherence architectural guidance on their site. Virtual training in Coherence is available through the Oracle Learning Library, and the Oracle University offers more formal in-class courses. The Oracle Coherence YouTube Channel is also a valuable resource with a number of videos available explaining features in detail. These videos are usually presented by the Engineers who built the software so they can provide a great deal of insight into the various features. The Coherence Community on java.net is the home of many open-source projects related to Coherence and led by Coherence engineers, each of which contains project-specific message forums and issue trackers monitored by the Coherence engineers leading the project.

For dedicated and on-site help, please contact Oracle Consulting or one of our Partners. Finally, if you are already an Oracle Coherence customer and have a problem, raise a Support Request to get help or check the Oracle Support Knowledge Base for potential solutions.

The "Basics" - Design, Develop, Test (*repeat*)

Modeling Your Data

Before using Coherence to hold your data, you must decide what form the data takes in Coherence caches. The most common contents are Java domain objects (that is, Plain Old Java Objects – POJOs) from a domain model appropriate for your application (see *Domain-Driven Design*, by Eric Evans, Addison-Wesley, 2003). Other possible forms include instances of simple Java types like `Number`, `Date`, `String`, or arrays of such instances. For example, instances of `String` can hold XML or JSON documents for simple caching purposes. There are no restrictions on the classes of objects for cache keys or values, as long as they extend `java.lang.Object` and are serializable using some mechanism.

Assuming Java domain objects, which is the typical case, one of the most important design decisions is how to map your domain model to a set of caches. A Coherence cache is a Java Map, and you have to decide how many separate caches to use for your application. At one extreme is a cache for each entity type, which is what Object-Relational Mapping tools do in their second-level caches. But, that pattern can lead to chattiness when reading domain object graphs from a cache and difficulty achieving atomicity (if required) when writing multiple domain objects to cache in one logical "application transaction.". At the other extreme is a cache for each aggregate root type – in *Domain-Driven Design*, an aggregate is a graph of objects that are treated as a unit for transactional purposes, with one particular object (for example, an `Order` object) at the root of the graph. The cache per aggregate root type pattern alleviates chattiness and facilitates atomicity but can have drawbacks in terms of efficiency of data transfer and lumpiness of data distribution. Object-to-cache mapping requires balancing a set of competing forces given your application's domain model, data access patterns, and transactional requirements.

To strike a good balance, consider the following questions concerning the objects in your domain model:

- How is the application going to access objects? For instance, is the application going to serialize and store `OrderItem` objects with its owning `Order` object, or does the application need to access `OrderItem` objects separately?
- How is the application going to update objects? For instance, if you store `OrderItem` and `Order` objects in separate caches, and you embed the `Order` key as an attribute, then moving `OrderItem` objects from one `Order` object to another requires the application to modify the `Order` key in each `OrderItem` object. You should asses the overhead and likelihood of such scenarios when developing your data model to determine the consequences on entity relationships.
- Will the application need to modify entities atomically? If so, then embedding `OrderItem` objects within an `Order` object would make this easier.
- Will the application fetch both `Order` and `OrderItem` objects together? If so, then embedding the `OrderItem` objects within an `Order` object only requires one network round trip to fetch all the entities.

Coherence uses multiple cache server JVMs to partition and distribute cache contents. Therefore, joins across caches, and transactions involving multiple entities can be more complicated and expensive. If your application has these requirements, then use the Partition Affinity feature to co-locate related objects in the same cache server for fast and atomic operations. However, co-location may cause unbalanced data distribution, so ensure that the associations used for affinity do not significantly distort the otherwise random distribution of cache entries.

In summary, consider your domain model and object-to-cache mapping very carefully and take into account the points outlined above.

Key vs. Filter Based Data Access

Once you have designed your domain model and mapped it to a set of caches, how do you access the cached domain objects? It's tempting to just reach for Coherence filters to look up cache entries. However, you should index entries to make the most efficient use of filters. Queries can be several orders of magnitude slower without the use of indexes. Furthermore, even if indexes on attributes are available when filtering, you should check that the index is being used – and used in the most efficient manner.

Key-based access, in contrast, scales very well, uses a minimum amount of resources, and provides predictable response times. This is because key-based access results in requests sent only to the members (by default) that contain an entry or set of entries. Filter-based access results in requests to all members to determine which members contain an entry or set of entries. Therefore, try to use keys rather than a filter to access cache entries where possible.

If filter use is required, then follow these steps to optimize their performance, minimize resource utilization, and ensure your application scales:

- Use indexes wherever possible but remember that index structures take up memory too. Some simple tips are:

– Monitor queries through the JMX metrics that are available in the `StorageManager` MBean as demonstrated below. The `MaxQueryDescription` and `MaxQueryDurationMillis` attributes give useful information about which query is performing the worst and how long it is taking to run. Note that a query description is available only if its duration exceeds the `MaxQueryThresholdMillis` value.

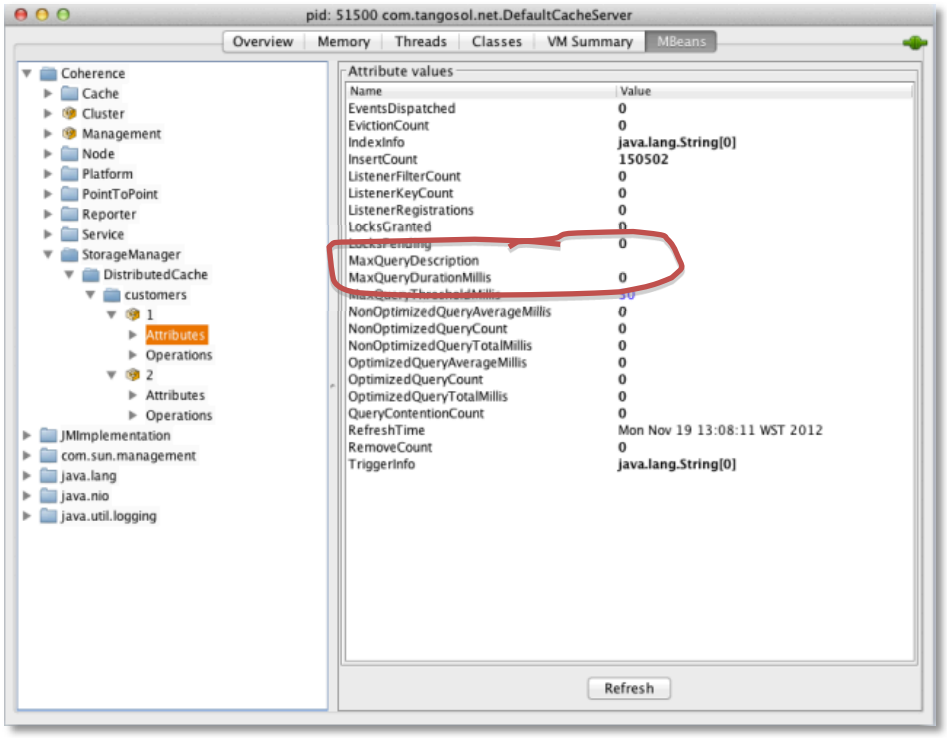
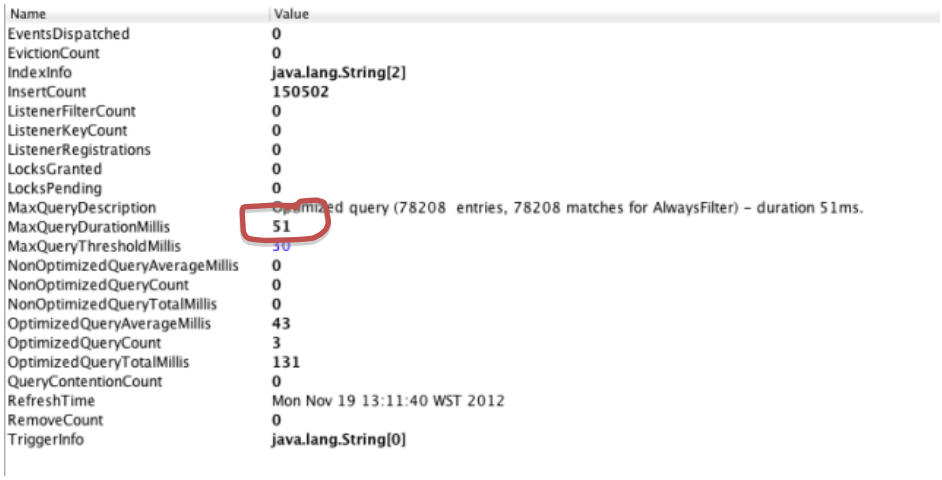


Figure 2. A cache's `MaxQueryDurationMillis` and `MaxQueryDescription` attributes can be found under its service in the `StorageManager` MBean

The difference in the metrics below illustrates just how much impact adding a couple of indexes can have when using a filter. After adding the indexes, the filter ran 20x faster.

Name	Value
EventsDispatched	0
EvictionCount	0
IndexInfo	java.lang.String[0]
InsertCount	150502
ListenerFilterCount	0
ListenerKeyCount	0
ListenerRegistrations	0
LocksGranted	0
LocksPending	0
MaxQueryDescription	Non-optimized query (149329 entries, full scan, 37302 matches for EqualsFilter(.getRegion(), South)) - duration 1161ms.
MaxQueryDurationMillis	1161
MaxQueryThresholdMillis	10
NonOptimizedQueryAverageMillis	587
NonOptimizedQueryCount	2
NonOptimizedQueryTotalMillis	1175
OptimizedQueryAverageMillis	50
OptimizedQueryCount	3
OptimizedQueryTotalMillis	151
QueryContentionCount	0
RefreshTime	Mon Nov 19 13:09:43 WST 2012
RemoveCount	0
TriggerInfo	java.lang.String[0]

Figure 3.The MaxQueryDurationMillis metric before the indexes have been added



Name	Value
EventsDispatched	0
EvictionCount	0
IndexInfo	java.lang.String[2]
InsertCount	150502
ListenerFilterCount	0
ListenerKeyCount	0
ListenerRegistrations	0
LocksGranted	0
LocksPending	0
MaxQueryDescription	Optimized query (78208 entries, 78208 matches for AlwaysFilter) - duration 51ms.
MaxQueryDurationMillis	51
MaxQueryThresholdMillis	10
NonOptimizedQueryAverageMillis	0
NonOptimizedQueryCount	0
NonOptimizedQueryTotalMillis	0
OptimizedQueryAverageMillis	43
OptimizedQueryCount	3
OptimizedQueryTotalMillis	131
QueryContentionCount	0
RefreshTime	Mon Nov 19 13:11:40 WST 2012
RemoveCount	0
TriggerInfo	java.lang.String[0]

Figure 4. The `MaxQueryDurationMillis` metric after the indexes have been added

– Be aware that the indexes for attributes with high cardinality (i.e., uniqueness of attribute values) typically consume more memory than those that have low cardinality. The amount of memory consumed by an index is measured by looking at the `IndexTotalUnits` attribute for the appropriate `StorageManager` MBean as demonstrated below:

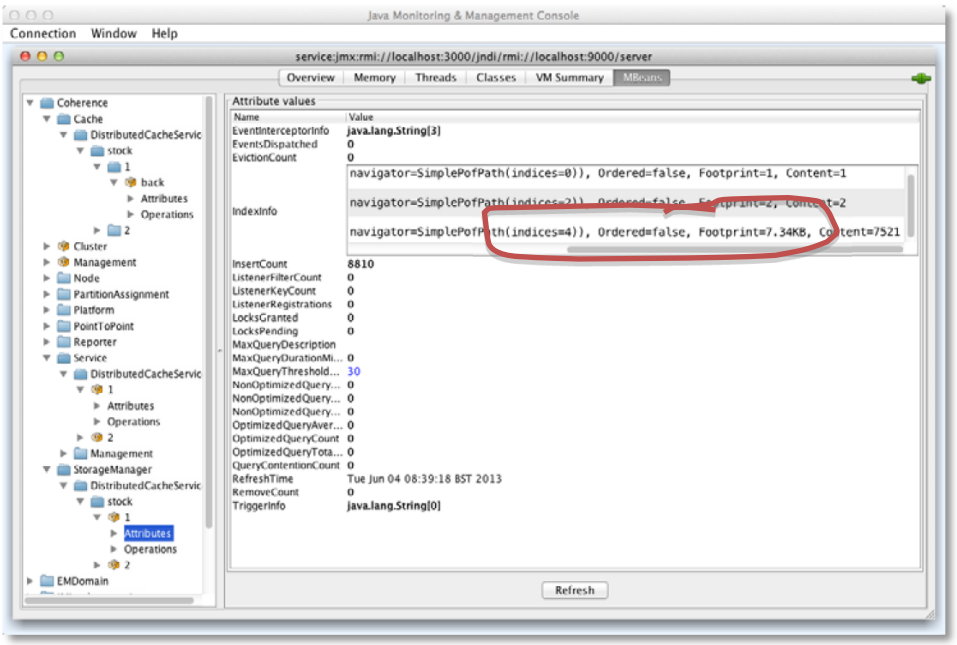


Figure 5. The `IndexInfo` attribute for a cache under its service in the `StorageManager` MBean. It shows the memory consumed by the indexes

The `IndexInfo` attribute lists the memory footprint of an index along with its associated POF index. In this case, index 4 (the `ID` attribute) has a footprint of 34K. Although the footprint of this index is small as compared with the footprint of indexes for other attributes, it is high because its cardinality is much higher.

```
@Portable
public class Stock {
  public static final int PRICE = 0;
  @PortableProperty(PRICE)
  private double price;

  public static final int QUANTITY = 1;
  @PortableProperty(QUANTITY)
  private int quantity;

  public static final int SYMBOL = 2;
  @PortableProperty(SYMBOL)
  private String symbol;

  public static final int CREATED = 3;
  @PortableProperty(CREATED)
  private Date created;

  public static final int ID = 4;
  @PortableProperty(ID)
  private int id;

  public Stock() {
  }
```

Figure 6. The attribute that corresponds to POF index 4.

– You can reduce index resource consumption by not using a forward index (key to value). However, this may not be possible if your application uses `InFilter` or Coherence's aggregation functionality, which leverage forward indexes. You can suppress forward index creation by passing `ConditionalExtractor` to the `NamedCache.addIndex` method, and passing `false` to the `ConditionalExtractor` constructor, as shown below.

```
cache.addIndex(new ConditionalExtractor(AlwaysFilter.INSTANCE, new
PofExtractor(String.class, Stock.SYMBOL), false));
```

– Make efficient use of your indexes. Consider the filter below:

```
AllFilter filter = new AllFilter(
    new Filter[] {
        new EqualsFilter(new PofExtractor(String.class, Stock.SYMBOL), "ORCL"),
        new EqualsFilter(new PofExtractor(Integer.class, Stock.ID), ENTRIES - 1),
        new EqualsFilter(new PofExtractor(Double.class, Stock.PRICE), 10.0)
        }
    );
```

Figure 7. A sample AllFilter

You can create a query explain plan and trace records to measure its effectiveness and cost. These reports provide detailed information about each step of a query. An example follows, but you can find out more about these features in the Coherence Developer's Guide.

```
Trace
Name                                    Index       Effectiveness          Duration
=============================================================================================
com.tangosol.util.filter.AllFilter  | ----       | 5012010(100%)        | 0
  EqualsFilter(PofExtractor(target= | 0          | 5012010(100%)        | 0

PartitionSet{128..256}

Trace
Name                                    Index       Effectiveness          Duration
=============================================================================================
com.tangosol.util.filter.AllFilter  | ----       | 4988011(99%)         | 70
  EqualsFilter(PofExtractor(target= | 0          | 49880125256(49%)     | 52
  EqualsFilter(PofExtractor(target= | 1          | 2525611(99%)         | 17
  EqualsFilter(PofExtractor(target= | 2          | 111(0%)              | 0

PartitionSet{0..127}

Index Lookups
Index  Description                                Extractor             Ordered
=============================================================================================
0      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
1      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
2      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
```

```
Explain Plan
Name                                    Index        Cost
=============================================================================================
com.tangosol.util.filter.AllFilter  | ----       | 0
  EqualsFilter(PofExtractor(target= | 0          | 1
  EqualsFilter(PofExtractor(target= | 1          | 1
  EqualsFilter(PofExtractor(target= | 2          | 1

PartitionSet{128..256}

Explain Plan
Name                                    Index        Cost
=============================================================================================
com.tangosol.util.filter.AllFilter  | ----       | 0
  EqualsFilter(PofExtractor(target= | 0          | 1
  EqualsFilter(PofExtractor(target= | 3          | 1
  EqualsFilter(PofExtractor(target= | 2          | 1

PartitionSet{0..127}

Index Lookups
Index  Description                              Extractor             Ordered
=============================================================================================
0      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
1      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
2      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
3      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
```

Figure 8. Sample Trace and Explain Plan output for the above query. Note that there is one for each node – in this case 2.

You can see that the first filter is not the most efficient as it only reduces the result set by 49%. In this case, performance improves tenfold by passing `Filters` to the `AllFilter` constructor in the order in which they most reduce the search space and by calling the `honorOrder` method on `AllFilter` before executing the query. For example:

```
Trace
Name                                    Index        Effectiveness          Duration
=============================================================================================
com.tangosol.util.filter.AllFilter  | ----       | 50120|0(100%)        | 0
  EqualsFilter(PofExtractor(target= | 0          | 50120|0(100%)        | 0

PartitionSet{128..256}

Trace
Name                                    Index        Effectiveness          Duration
=============================================================================================
com.tangosol.util.filter.AllFilter  | ----       | 49880|1(99%)         | 6
  EqualsFilter(PofExtractor(target= | 1          | 49880|1(99%)         | 6
  EqualsFilter(PofExtractor(target= | 2          | 1|1(0%)              | 0
  EqualsFilter(PofExtractor(target= | 3          | 1|1(0%)              | 0

PartitionSet{0..127}

Index Lookups
Index  Description                              Extractor             Ordered
=============================================================================================
0      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
1      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
2      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
3      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
```

```
Explain Plan
Name                                    Index        Cost
=============================================================================================
com.tangosol.util.filter.AllFilter   | ----       | 0
  EqualsFilter(PofExtractor(target= | 0          | 1
  EqualsFilter(PofExtractor(target= | 1          | 1
  EqualsFilter(PofExtractor(target= | 2          | 1

PartitionSet{128..256}

Explain Plan
Name                                    Index        Cost
=============================================================================================
com.tangosol.util.filter.AllFilter   | ----       | 0
  EqualsFilter(PofExtractor(target= | 3          | 1
  EqualsFilter(PofExtractor(target= | 1          | 1
  EqualsFilter(PofExtractor(target= | 2          | 1

PartitionSet{0..127}

Index Lookups
Index  Description                              Extractor             Ordered
=============================================================================================
0      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
1      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
2      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
3      SimpleMapIndex: Extractor=PofExtractor(t  PofExtractor(target=  false
```

Figure 9. Sample Trace and Explain Plan output when the order of the first 2 filters is changed

Query explain-plan and trace records can be generated without coding using the Coherence command line query tool.

- Apply the index before the entries are added to the cache. Applying indexes afterwards can take a long time and even time-out if the entry set is very large.
- Consider using `ConditionalExtractor` to create an index that excludes `null` values.
- For multi-value queries, consider using `MultiExtractor` for creating the index.
- If the cache you are querying uses a near topology, then query for the key set instead of the entry set and then call the `NamedCache.getAll` method with the key set. This approach utilizes values in the near cache, whereas querying directly for the entry set does not.
- Where possible, target filters by using `KeyAssociatedFilter` to ensure a filter only runs on the node where the results set resides – if using data affinity.
- Consider batching results by fetching the results from the partitions of one node at a time using `PartitionedFilter`. This can significantly reduce the heap utilization on the client and improve response times. Although, remember that this must be the outermost filter, so that it is executed on the client.

Further details on using Filters can be found in the Coherence Developer's Guide.

Efficient and Scalable Transactions

Coherence provides a very fast and scalable mechanism for controlling concurrent access to one or more cache entries and for performing atomic modifications. This feature, partition-level transactions, extends the functionality of an entry processor. An entry processor is similar to a database stored procedure, as it executes where a cache entry or entries reside. Entry processors effectively queue concurrent requests to modify or access a cache entry. Partition-level transactions extend them by enrolling multiple cache entries in the same, or other, caches in an atomic operation. Changes to multiple entries occur in a sandbox and get automatically committed together at the end of the entry processor execution – including the backup of any changes.

You should meet the following requirements to use partition-level transactions:

- Cache entries, accessed or modified together, need to be in caches managed by the same service and in the same partition (that is, located in the same JVM). Therefore, implement key association or data affinity between the entries.
- Equal and opposite operations should not be performed simultaneously. For instance, modifying entry X in cache A and Y in cache B at the same time as modifying entry Y in cache B and entry X in cache A.

To gain exclusive access to another cache entry in an entry processor, access the other entry from within the entry processor using the cache's `BackingMapContext.getBackingMapEntry` method. As shown below:

```
/**
 * Add a new {@link Order} to a {@link Customer}. This method is called against the
 * {@link Order} object and the {@link Customer} object will be implicitly locked.
 *
 * @param entry The customer entry to run against
 *
 * @return null
 */
@Override
public Object process(Entry entry)
{
    // entry should not be present so lets set the value. E.g. create it
    entry.setValue(newOrder);

    BinaryEntry             orderEntry = (BinaryEntry) entry;
    BackingMapManagerContext ctx       = ((BinaryEntry) entry).getContext();

    // get the customer entry as a binary - this will be in the same partition
    // by using getBackingMapEntry you are adding the changes to the sandbox

    BinaryEntry customerEntry =
    (BinaryEntry) orderEntry.getContext().getBackingMapContext(Customer.CACHENAME)
    .getBackingMapEntry(ctx.getKeyToInternalConverter()
                        .convert(new Customer.Key(newOrder.getCustomerId())));

    // get normal Object Value
    Customer thisCustomer = ((Customer) customerEntry.getValue());

    // update the customer balance
    thisCustomer.setBalance(thisCustomer.getBalance() + newOrder.getOrderTotal());
    customerEntry.setValue(thisCustomer);

    return null;
}
```

Figure 10. An entry processor that locks and updates multiple entries as part of an atomic operation

If exclusive access to other cache entries is not required (for example, read-only), then use the `BackingMapContext.getBackingMap.get` API as shown below.

```
@Override
public Object process(Entry entry)
{
    ...

    BackingMapManagerContext ctx        = ((BinaryEntry) entry).getContext();

    // get the customer as a binary by using getBackingMap().get(...)
    // because we are not going to update it and so don't need to lock it

    Binary binCustomer =
    (BinaryEntry) orderEntry.getContext().getBackingMapContext(Customer.CACHENAME)
    .getBackingMap().get(y(ctx.getKeyToInternalConverter()
                          .convert(new Customer.Key(newOrder.getCustomerId())));


    // because we are getting a Binary instead of BinaryEntry, we need to
    // deserialize with the serlializer for the Service.
    Customer c = (Customer) ExternalizableHelper
    .fromBinary(binCustomer, ctx.getCacheService().getSerializer());

    ...
}
```

Figure 11. An entry processor that locks the entry it was targeted at and then accesses an entry in another cache – without locking it

If you require transactional logic, explore partition-level transactions first, adjusting the domain model and cache mapping as required, before using the `ConcurrentMap` API (using the `lock` and `unlock` method) or transactional cache schemes. The reasons for this are all performance and scalability related: pessimistic locking and two-phase commit algorithms are well-known performance and scalability anti-patterns. It should also be noted that the different mechanisms should not be mixed, so select the one that best fits your requirements.

Finally, the 12c release includes a "mirror" partition assignment strategy that co-locates a service's partitions with those of another service. This strategy increases the likelihood that key-associated, cross-service cache access remains local to a member – though there is no guarantee. Using this strategy, in conjunction with an invocation service to perform cross-cache operations, may localize locks with cache entries to provide a scalable solution.

Ensuring Data Consistency Using Golden Gate HotCache

In many use cases, the data held in a cache is a copy of that in a database. Furthermore, the data in the database is often the master copy and updates occur outside of Coherence by third party applications. Before GoldenGate HotCache, a Coherence feature introduced in release 12c, several other approaches were available to ensure that cached data remained somewhat consistent with underlying database data:

- Setting a Time-To-Live (TTL) on cache entries and using a read-through cache store, so that cache entries regularly expire from a cache and are reloaded from the database. This technique can also be used with Coherence's refresh-ahead feature, so that entries are reloaded asynchronously before the expiry time is reached. However, some cache data may still be stale and other entries that have not changed may be needlessly reloaded.

- Database triggers can also be used to push database changes to Coherence, using messaging functionality like Oracle AQ. However, doing so requires changes to the database schema, development and operation of additional runtime components, and may not meet throughput or replication latency requirements. This approach can also add undesirable additional load to the database, and the implementation differs depending on the underlying database.
- The Database Change Notification (DCN) feature of the Oracle JDBC driver can monitor an Oracle database for changes and update Coherence when they occur. With DCN, it is possible to register listeners with the Oracle JDBC driver that receive notifications in response to Data Manipulation Language (DML) and Data Definition Language (DDL) changes affecting registered SQL query result sets, without any need to modify the database schema. Events (`DatabaseChangeEvents`) that contain values including the table that changed, the operation that was performed, and the row IDs that were affected, are returned to the registered listeners when changes are committed as part of a database transaction. However, the returned information does not include both the old and new column values of the rows that changed in the transaction, thus potentially necessitating database queries by the listener to determine exactly what happened. Designing for high availability with a DCN-based solution is also a challenge.

The HotCache feature overcomes the shortcomings of the preceding approaches to keeping caches consistent with volatile underlying databases. The feature uses GoldenGate and JPA as enabling technologies. Golden Gate is a non-invasive technology for performing real-time Change Data Capture (CDC) against many different types of relational databases. The Coherence HotCache feature uses GoldenGate, and JPA implemented by TopLink, to propagate database changes to Coherence caches. Applications already using JPA for Object Relational Mapping (ORM) in their cache stores can leverage HotCache to refresh caches from underlying database transactions, without modifying the database or application code. In addition, Golden Gate writes a copy of database changes to a trail file and keeps track of changes applied using checkpoints. Therefore, restarting the source capture or HotCache processes can overcome any failure in the change capture and cache refresh process without losing any intermediate events. HotCache can also capture and apply a high rate of database changes without significant impact to the database.

Security

Coherence provides a range of security options to secure sensitive information. Below is a list of options that are available and considerations for each.

COHERENCE SECURITY

SECURITY OPTION	SUPPORTED	NOTES
Data Encryption	Yes	Clients can use standard encryption libraries to encrypt keys and values or just part of them. A custom serializer can also incorporate encryption. However, inside a cluster these will then be opaque and not usable (for example, in a in a filter).
Secure Communications	Yes	SSL can secure TCMP over TCP or TCP Message Bus

• Inside a cluster • With extend clients	Yes	communications within a cluster. SSL can secure communications between extend clients and proxy services. Furthermore, integration between Coherence and a hardware load-balancer, like F5, allows offloading SSL termination processing.
Restricting Cluster Membership and Operations	Yes	The authorized host feature limits cluster access to new members that are in a cluster member pool. However, the list of members can be dynamically constructed if required using a custom hosts filter. An Access Controller can restrict: the operations that cluster members can perform (for example, create, join. destroy, all, and none), the caches they can control, and the services they can use.
Restricting Management Access	Yes	SSL and password based authentication can secure connections to the JMX management node. Read-only JMX access is also possible.
Restricting Client Access	Yes	Extend client access can be limited through a range of host names/IP address or a custom hosts filter.
Rogue Clients	Yes	A proxy service can disconnect extend clients that do not process their response messages fast enough in order to prevent running out of memory.
Authentication	Yes	Extend clients in all technologies can pass security tokens to Coherence proxy services for validation. Authentication is performed using a standard JAAS login module or in Coherence.
Authorization (Client)	Yes	Extend clients can use an interceptor class to wrap cache and invocation services that can then permit or disallow operations based upon the identity and permissions of a user.
Authorization (Server)	Yes	Access control authorization allows applications to define their own authorization logic to limit access to cluster operations by implementing `StorageAccessAuthorizer`. See the Coherence Java examples for sample code.
Audit Logs	Yes	The `-Dcoherence.security.log=true` system property enables audit logs and is a useful method of diagnosing application authentication issues.

Securing cluster or client operations inevitably introduces processing overhead and it is a best practice to add any necessary security measures at the beginning of any test cycles to incorporate this in any

measurements. In addition, remember that authentication only imposes an overhead at connection time where as authorization does so for each request.

Development Tooling

The free Oracle Enterprise Pack for Eclipse (OEPE) provides a set of project facets, configuration wizards and schema aware editors to accelerate development and minimize configuration errors. If your development team uses Eclipse, it is well worth investigating.

With release 12c of Coherence, OEPE allows packaging and deploying Coherence applications to as a Grid ARchive (GAR). In addition, the WLST editor and WebLogic Server runtime integration will help those new to this powerful scripting language to try it out in the same development environment.

Coherence 12c also introduces a Maven plug-in that synchronizes an Oracle home directory with a Maven repository and standardizes Maven usage and naming conventions. The Maven integration also includes an archetype and packaging plug-in for Coherence GAR modules. A Coherence GAR is a module type that packages all the artifacts required to execute a Coherence Application; this includes any class dependencies and XML configuration files (though not an override file).

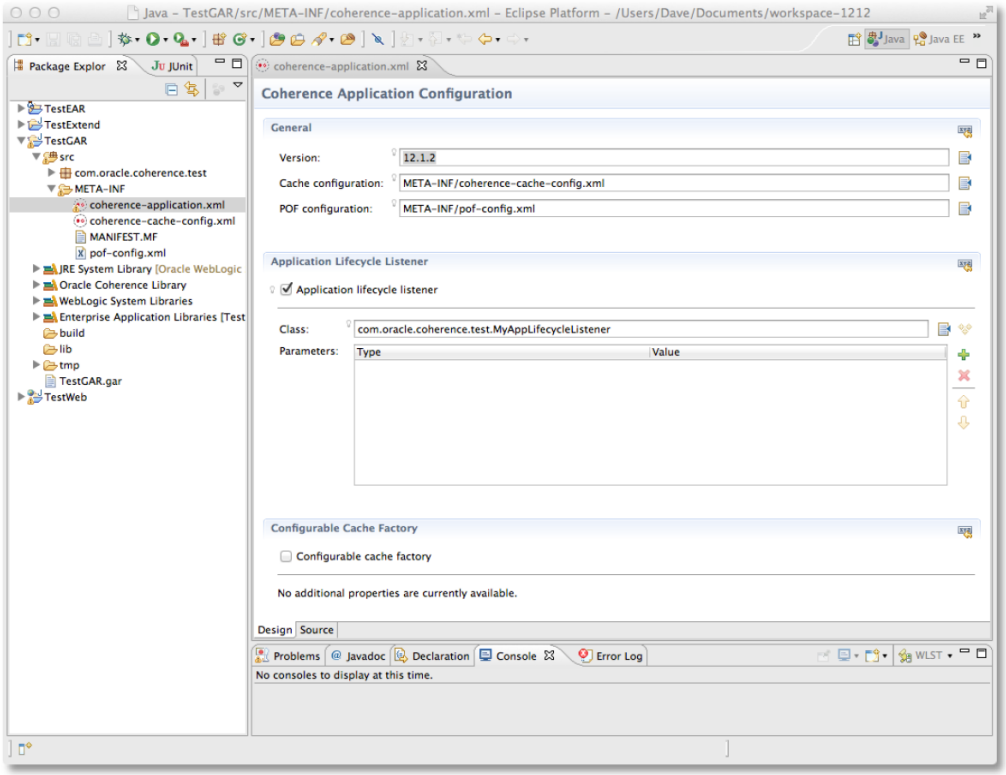


Figure 12. OEPE provides wizards for common development activities, like configuring and creating a Coherence GAR

In other development environments, like JDeveloper and Visual Studio, XSD validation of the Coherence configuration files auto-prompts users to ensure your application fails fast if it is incorrect. For this last reason alone, it is highly recommended to use schema validation.

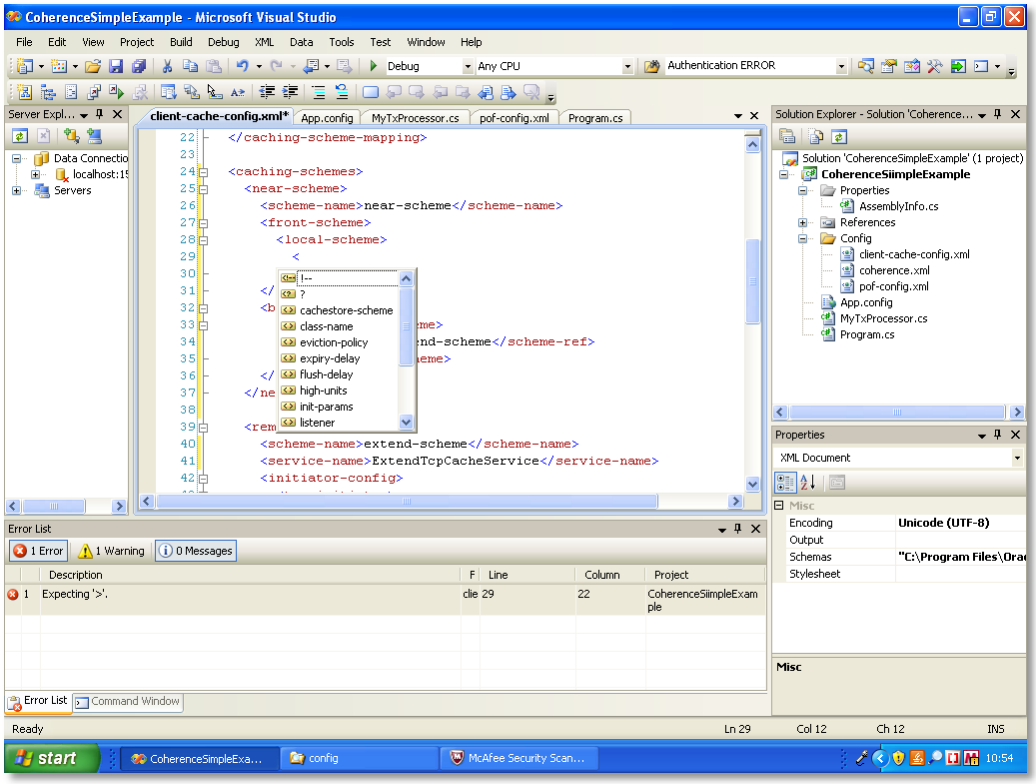


Figure 13. Schema validation can be used to prevent errors and make configuration easier.

Consider Using Java8 Features

Coherence 12.2.1 requires Java 8 and takes advantage of Lambdas, Streams, Map default methods, and `CompletableFuture`. Coherence incorporates these new features to provide alternate programming models, which can help streamline application development. More detail on Java 8 support is available in the Coherence Documentation as well as in the Examples that ship with Coherence. Some examples follow.

Replacing Value Extractors with Method References

Method references can replace traditional `ValueExtractor` implementations and in combination with static imports of new `Filter` Domain Specific Language (DSL) methods, can make queries much easier to read and write as well as ensuring that there are no "magic strings" to map to method names. The result is more type-safe code that is easier to refactor. For example:

```
import static com.tangosol.util.Filters.equal;
import static com.tangosol.util.Filters.greater;
import static com.tangosol.util.Filters.like;
import static com.tangosol.util.Filters.notEqual;
```

```
// Find all contacts who are older than 45
setResults = cache.entrySet(greater(Contact::getAge, 45));
printResults("Age > " + nAge, setResults);

// Find all contacts who are older than 45 and first name starts with M
setResults = cache.entrySet(greater(Contact::getAge, 45)
  .and(like(Contact::getFirstName, "M%")));
```

Using Lambdas as Entry Processors

You can use lambdas as entry processors from clients to introduce new logic into a running cluster without a cluster restart. When defining lambdas, the best way to ensure that all captured arguments are local variables and captured as a "closure" is to return lambdas from static methods. For example:

```
static InvocableMap.EntryProcessor<Integer, Position, Position> split(int factor)
    {
    return entry ->
        {
        Position p = entry.getValue();
        p.setQuantity(p.getQuantity() * factor);
        p.setPrice(p.getPrice() / factor);
        entry.setValue(p);
        return p;
        };
    }

...
Map<Integer, Position> afterSplit =
       positions.invokeAll(equal(Position::getSymbol, "ORCL"), split(2));
```

Utilize SimpleMapListener and Lambdas to Register Events

The `SimpleMapListener` class, introduced in 12.2.1, allows you to register events using lambdas. This can simplify code and make it more readable. For example:

```
people.addMapListener(
    new SimpleMapListener<>()
        .addEventHandler(System.out::println));

people.addMapListener(
    new SimpleMapListener<>()
        .addInsertHandler(e -> System.out.println("inserted:" + e.getNewValue()))
        .addUpdateHandler(e -> System.out.println("updated:" + e.getKey()))
        .addDeleteHandler(e -> System.out.println("deleted:" + e.getOldValue()))
    );
```

Consider Using the Asynchronous API

The `AsyncNamedCache` interface allows parallel cache operations and can improve throughput and result in more responsive user interfaces.

See the Oracle Documentation for more information on the Asynchronous API.

Streams

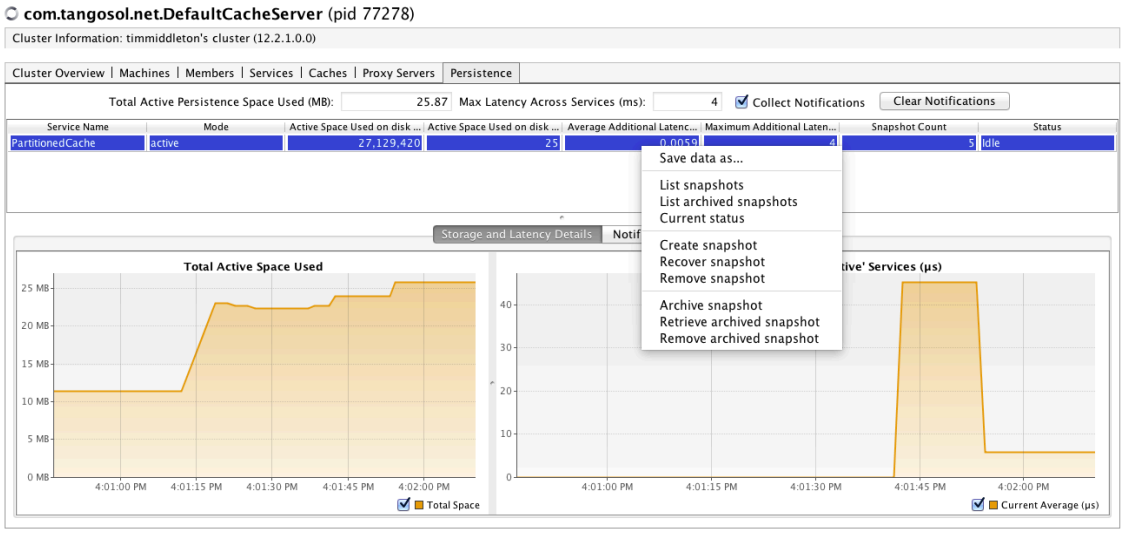
Java8 Streams provide an alternative-programming model for carrying out aggregations across a data grid. See the Oracle Documentation for more information on Streams.

Testing

A popular approach for developing Coherence applications (and others) is through Test Driven Development (TDD). As a result, a number of tools and frameworks have emerged to help developers perform continual unit testing throughout their development process. These tools and frameworks simulate complex "edge case" scenarios that involve multiple components, by running a whole cluster in a single JVM or across multiple machines. The Coherence development team uses the Oracle Tools framework to test Coherence. Take advantage of the framework to help test your Coherence-based applications.

Coherence JVisualVM Plug-In

Coherence, since 12.1.3, ships a Coherence JVisualVM Plug-In, which allows lightweight monitoring of development and test clusters. This tool gives a real-time view into a cluster and provides useful metrics to support development and testing cycles. It also supports major new functionality in 12.2.1 including Persistence and Federation. Refer to the Oracle Documentation for more information.



Plan for Change and the Worst Case Scenarios

Preventing Failures, Data Loss and Data Corruption

Coherence already has many features to minimize the chance of data loss due to a software or hardware failure. For instance, you can keep multiple copies of a cache entry:

Figure 14. Guarantee data availability during multiple simultaneous machine loss

All cache updates are synchronous by default. This means that by the time Coherence responds to a `put` call from application code, Coherence guarantees replication of the change to all copies.

Coherence also ensures that the primary copy of an entry and the backup(s) are on different physical machines – if possible.

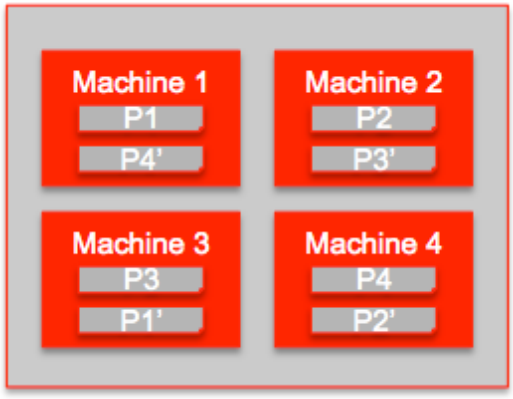


Figure 15. By default, Backup Partitions Always on Separate Machines

However, it is also possible to configure rack and site safety as well as machine safety. These additional options inform Coherence that a primary and backup copy of an entry should be on different racks or even a different site.

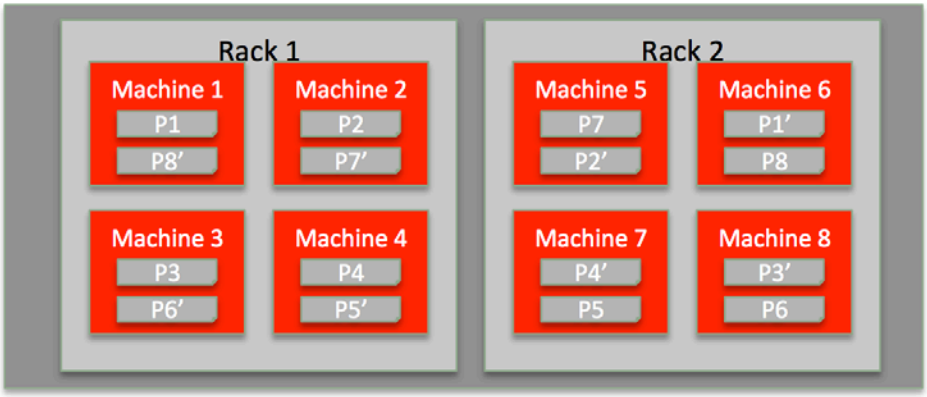


Figure 16. Coherence configured with rack safety. If a rack is lost, then the cluster falls back to machine safety.

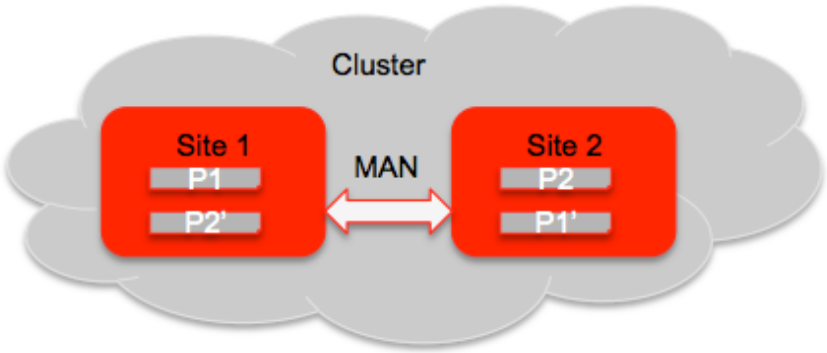


Figure 17. Coherence configured with site safety. If a site is lost, then the cluster falls back to rack safety.

When considering a site-safe configuration there are a number of implications. For instance, 50% of read and all write operations traverse the inter-site link (and 50% of all write operations do so twice). While some customers have successfully used this architecture, the synchronous nature of communications affects performance, scalability, and stability.

Consider 20 server machines connected to a single LAN switch with 1GbE links. A Coherence cluster running on that infrastructure can serve roughly 20Gb of data per second. Now consider the 20 machines split evenly over two sites separated by a fast reliable 1GbE WAN link. Sounds pretty good for a WAN, but what is the cluster throughput. Since at least half of all requests traverse the WAN and that shared 1GbE link, the effective cluster bandwidth is gated by that link, and the cluster bandwidth is going to be closer to 2Gb/s rather than 20Gb/s. This restricts the scalability and performance of the cluster. The same argument applies to request latency. As for stability, you are more likely to lose connectivity and thus face a "split-brain" scenario[1], which requires preventative action. Even if you have a better link than this, this configuration is still not scalable. That is, adding more nodes on either end does not (and cannot) improve performance. This is the reality of synchronous replication across a WAN link.

As a result, this configuration is very sensitive to the throughput, latency and reliability of the inter-site link. Consider this configuration only if it provides reasonable throughput (for example, 1GbE) and relatively low latency (for example, <10ms). Use cases where this may be suitable are:

- Caches which have a very high read-to-write ratio
- Caches for which throughput is relatively low

There are some options for improving the performance of a site-safe configuration: such as using near caching to reduce network operations and even asynchronous backups – a new feature in 12c. The Coherence quorum feature can also prevent data corruption and help Coherence overcome environmental issues. For instance, if the nodes in a cluster cannot communicate with each other, perhaps because of intermittent network issues, then the cluster quorum policy maintains a minimum number of members and prevents a cluster from breaking apart. The partitioned quorum policy can also prevent recovery and rebalancing if a temporary cluster split occurs. You can prevent updates to cluster islands to guard against corrupt data due to a split-brain scenario.

Coherence 12c also introduces a new partitioned quorum policy option to manage failover access. This moderates client request load during a failover event to give cache servers more time to recover and rebalance partition backups. It can be particularly useful where a heavy load of high-latency requests

[1]A "split-brain" scenario arises when a Coherence cluster splits into separate clusters, perhaps because of communication issues, and external clients then separately update identical entries in each cluster causing the data to diverge.

may prevent, or significantly delay, cache servers from successfully acquiring exclusive access to partitions that need to be transferred or backed up.

```
<!-- Distributed caching scheme. -->
<distributed-scheme>
    <scheme-name>example-distributed</scheme-name>
    <service-name>DistributedCacheService</service-name>

    <backing-map-scheme>
        <local-scheme>
            <unit-calculator>BINARY</unit-calculator>
        </local-scheme>
    </backing-map-scheme>

    <partitioned-quorum-policy-scheme>
        <class-name>com.tangosol.net.partition.FailoverAccessPolicy</class-name>
        <init-params>
            <!--
                Specifies the delay before the policy should start holding requests
                (after becoming endangered).
            -->
            <init-param>
                <param-name>cThresholdMillis</param-name>
                <param-value>7000</param-value>
            </init-param>
            <!--
                Specifies the delay before the policy makes a maximum effort to hold requests
                (after becoming endangered).
            -->
            <init-param>
                <param-name>cLimitMillis</param-name>
                <param-value>30000</param-value>
            </init-param>
            <!--
                Specifies the maximum amount of time to hold a request.
            -->
            <init-param>
                <param-name>cMaxDelayMillis</param-name>
                <param-value>2000</param-value>
            </init-param>
        </init-params>
    </partitioned-quorum-policy-scheme>

    <autostart>true</autostart>
</distributed-scheme>
```

Figure 18. A partition quorum policy to manage failover access to partitions and improve recovery performance

Data Source/Database Integration Options

Coherence supports both write-through and write-behind cache store options. Choosing which option to use - based upon the required level of performance, scalability, and recovery guarantees - requires weighing the benefits and drawbacks of each strategy.

SYNCRHONOUS VS. ASYNCHRONOUS CACHE STORE INTEGRATION

FEATURE	WRITE-THROUGH	WRITE-BEHIND
High Availability • **Recovery**. Guaranteed recovery from external data store.	Yes	No
• **Data Store Availability Errors**. Ability to handle availability errors and re-try later. Note: allocate additional memory to hold queued updates that need to be re-tried.	No	Yes

• **Integrity Constraint Errors**. The ability to handle integrity constraint errors.	Yes	No
Performance		
• Write Performance.	No	Yes
• Read Performance.	Yes	Yes
Scalability	Yes[2]	Yes

When using either approach for persisting cache data to a database, use a JDBC connection pool and an appropriate timeout set for database operations that is less than the cache store timeout which in turn should be less than the Service Guardian timeout. If using an Oracle database, then the JDBC `Statement` class has a `setQueryTimeout` method to set an overall timeout on the execution of the statement[3]. With JPA 2 this can also be set using the `javax.persistence.query.timeout` property in the `persistence.xml` file. Setting a query-timeout prevents Coherence threads waiting for JDBC responses from termination and left in an inconsistent state when the Service Guardian has not received a heartbeat from such threads during the configured Service Guardian timeout interval.

The following list describes the limitations with each approach.

Write-Behind Data Source Integration

- Operations need to be idempotent, that is, they must produce the same result if repeated because of a node failure.
- Operations should not fail because of errors like a referential integrity violation, as it is not possible to communicate these types of errors back to the client.
- When using a re-try queue (`<write-requeue-threshold>`), size it sufficiently to handle any failed persistence entries – for instance during a period when a database goes offline – but not so high that a node may run out of memory.
- Remove operations are always synchronous.

[2] A write-through cache store cannot batch writes like a write-behind cache store. However, a feature called operation bundling can provide pseudo write batching. It works by capturing the separate write operations across the worker threads of a service during a pre-defined time interval, grouping them into a batch. For use cases where there are frequent write operations it can improve scalability, but since write operations pause during the pre-set time window to create the batch, there is a performance impact. Another side effect is that any persistence exceptions affect all the writes in a batch.

[3] For further details see the Oracle Support Note 1531408.1. However, this option is not available on Windows.

Write-Through Data Source Integration

- Allocate a sufficient number of service threads. Both the cache put and the database operation require a service thread for the duration when using write-through. Check the `Service` MBean `ThreadIdleCount` JMX metric to monitor thread utilization.
- Ensure a read-write-backing-map-scheme has the `<rollback-cachestore-failures>` element set to `true` in order to pass back exceptions to the client.

Detecting and Reacting to Failures

Coherence has a number of failure detection mechanisms. The Service Guardian detects "stuck" threads. Guarded threads, like the worker threads of a service, issue a regular heartbeat to the Service Guardian indicating they are still active. If the Service Guardian does not receive a heartbeat for a set period of time, then it issues a soft timeout (causing a thread dump) followed by a hard timeout (resulting in a thread shutdown) if it still gets no response[4]. The failure of a cluster node can be detected in a number of ways, including the closing of a TCP socket connection that forms a ring around the cluster. It usually only takes a few milliseconds to detect a cluster node failure and for the recovery process to begin. Machine failure is detected using an IP Monitor daemon. Periodically, the `IpMonitor` daemon for the machine senior (that is, only one per machine) picks the address of a cluster member and determines if the machine it is running on is operational. The check uses the `java.net.InetAddress.isReachable` call with a default timeout setting of 5 seconds. After three unsuccessful attempts, the machine is "not reachable" and its members removed from the cluster prompting the recovery process to begin. Reducing the number or intervals of heartbeats is possible but may result in false positives.

The implementation of "isReachable" is platform specific. As of Java SE 7 (and older), it attempts to either send ICMP requests or (if ICMP requests are not allowed by the operating system) attempts to connect to TCP port 7 on the remote host. Port 7 is the default port for the Echo Protocol. This service is disabled by default on most operating systems, but the connection exception that results from a machine that rejects this connection is used to determine that the machine is running.

If a firewall prevents ICMP packets and/or connections to port 7, this may prevent the formation of a cluster since Coherence cannot verify the reachability of the machine. Therefore, if a firewall is required, then the recommendation is to open port 7 to allow the `IpMonitor` daemon to function.

When a node or machine failure occurs and the recovery and re-distribution process starts, Coherence throttles the movement of partitions at a rate determined by the transfer-threshold (by default 512KB per second, per service, per node). If a single partition exceeds the default size, then the service transfers only one partition each second. The throttling avoids the rebalancing process from starving

[4] The problem can also just be logged if a thread re-start is not desired

normal processing of network resources. The default threshold is adequate for a 1GbE network. Consider increasing the default on faster networks.

You can calculate the approximate Mean Time to Rebalancing (MTR) that is required after the failure of a machine hosting one or more cache servers. However, first consider what is possible assuming there is no throttling:

Assumptions

- Each machine is on a fully switched 1 GbE network and can transmit and receive ~100 MB of data per second. It is likely that less than 100MB per second is available if you take into account network resources that are required for ongoing processing.
- During recovery the re-creation of backups is almost instantaneous (lost primaries are created from promoted backups) and, for the most part, it is backups that are moved during rebalancing.
- The rebalancing process is network bound and re-creating indexes and firing Backing Map Listeners (triggers when entries are created) usually happen very quickly.

Calculation

- Let *M* be the number of machines in a cluster before the failure of one machine
- Let *D* be the size of the serialized backup data in megabytes
- When a machine fails the fraction of data that needs to be rebalanced is *D* / *M* because the failed machine previously owned 1 / *M* of all backup data
- This needs to be shared among *M* - 1 machines
- As rebalancing can be done in parallel (every machine can transmit backup entries), each machine can transmit 1 / (*M*-1) of the data at the same time
- The MTR equals the amount of data to transfer (*D*/*M* megabytes) divided by the rate at which it can be transferred: (*M*-1)*100 megabytes per second.

```
MTR = (D/M) / (100 * (M-1)) seconds
```

Worked Example

- 24 GB of primary data (with 1 backup) or 48 GB in total
- 3 machines, each with 64 GB memory and 2 x 4 cores
- 1 GbE network
- 1 machine fails

MTR = ((24*1024) / 3) / (100*(3-1)) = 41 seconds

Factors such as processing activity during recovery, the number of partitions, and so on can affect the accuracy of this formula. Therefore, validate any estimate through testing. But, the MTR goes up

linearly as more data is added and down linearly as network throughput is increased. For instance, in the above example the MTR would be ~4s when using a 10 GbE network.

Now compare the above calculation with the MTR when allowances are made for throttling during rebalancing:

Assumptions

- *C* is the number of cache servers per machine, 8 in this case
- *N* is the number of partitions, 2039 in this case
- Here we assume there is only 1 service
- The average partition size is (24*1024) MB / 2039 =12.05 MB
- Default transfer threshold = 0.5 MB

Calculation

Because the partition size is greater than the transfer threshold, the rebalancing time, adjusted for throttling, is:

$(N/M) / (C*(M-1)) = (2039 / 3) / (8 * (3 – 1)) = 679.67 / 16 = 42$ seconds

During recovery, Coherence throttles partition transfers so that each node can only send 1 partition per second – because the partition size exceeds the transfer threshold. Therefore, each machine sends 8 * 12.05 MB = ~96.4 MB of data per second. Here, throttling does not limit rebalancing as the threshold is near the limit of the network capacity anyway, but it can limit the network bandwidth that is used for rebalancing so that other operations can continue.

To summarize, the complete recovery time involves detecting node or machine failure, regenerating lost copies of data, and finally re-distributing/re-balancing these new copies. The first phase can take up to 15s (to detect a machine failure). The second phase should be almost instantaneous. The third phase, the MTR, depends on the network bandwidth, cluster load, and parameters affecting throttling.

Persistent Caches in Coherence 12.2.1

Coherence 12.2.1 provides an ability to persist the state of partitioned services using an innovative and consensual algorithm. Ultimately, this provides a mechanism to recover the state of a cluster after planned or unplanned shutdown of the entire cluster. This section of the document provides some best practices when using persistence.

Persistence can operate in one of two modes:

- On-Demand – an active copy of the state of the service and caches is **not** maintained on disk. Instead, on-demand operations (including creating a snapshot and recovering a snapshot) allow a point-in-time copy of the state of the service and caches. This is the default mode for partitioned services and requires no additional configuration.

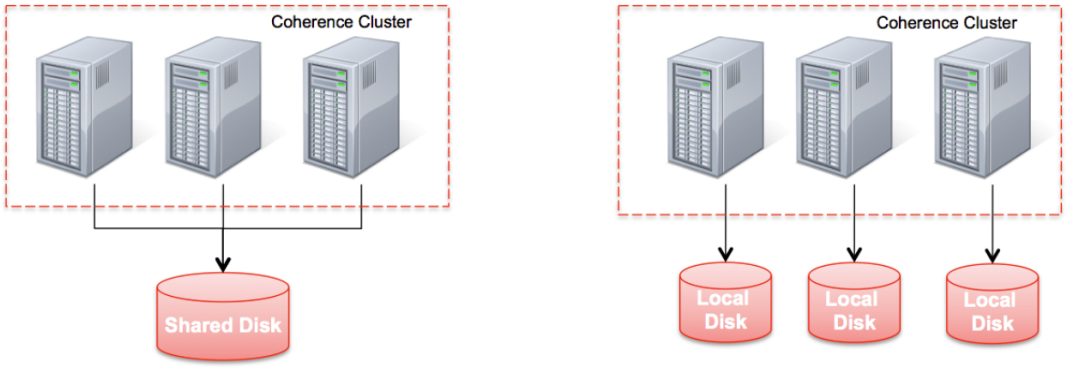
- Active – an active copy of the state of the service and caches is maintained on disk to allow automatic recovery on cluster/service startup or during simultaneous loss of primary and backup owners of a partition. Point-in-time snapshots are also permitted in this mode.

Configuring Persistent Caches

Persistence is configurable on a per service basis with some convenience shortcuts to enable global enablement (JVM argument `-Dcoherence.distributed.persistence.mode=active`). The details of configuring persistence can be found in the Coherence Documentation.

Choosing a Storage Topology

The default persistence implementation depends upon a file system which may in-turn reference a shared storage device (NFS) or a local disk. Shared storage solutions allow for all state to be accessible from any member of the service generally at the cost of increased write and read latency. The local disk topology harnesses the clustering primitives within Coherence to consensually recover the entire persisted state from the many disconnected local disks.



Persistent caches using local disk is the recommended topology for the following reasons:

- Harness the write and read speeds of all devices physically attached to each machine
- Disks on each machine can use technologies that provide faster read times such as Flash storage
- Deterministic access times when using local devices in comparison to shared storage
- Coherence makes the disconnected local disks appear as a single storage layer

A compelling reason to use shared storage is for cases where the following hold true: a backup count of zero and machine safety is required. With shared storage, all data is accessible everywhere. When using local storage, if all members on a machine depart the service, the data residing on said machine becomes inaccessible.

Perform thorough testing to determine the potential impact of either local or shared storage on individual hardware and network setup. The approximate measures of additional latency due to persistence operations are available on the `Service` MBean and exposed on the JVisualVM plugin (see below).

Setting Quorum for Recovery

Coherence uses the notion of quorums as a means to describe a minimum set of criteria prior to an action being allowed. To determine whether the action of recovery can proceed, there are two important factors: availability (access to the entire persisted state) and capacity. These have been, respectively, described below and are configurable within the partitioned-quorum-policy-scheme:

- **recovery-hosts** – a list of hosts/machines that must be available for recovery to commence
- **recover-quorum** – the minimum number of storage-enabled members that must be present for recovery to commence

When using the local disk storage topology, all machines hosting storage-enabled members prior to shutdown **must be** present within the recovery-hosts quorum policy configuration. This ensures that the action of recovery does not commence until all persisted state is available. An insufficient list of hosts is likely to result in data loss during recovery. A recommended best practice is for the recovery-hosts to reference an address provider in the operational configuration to maintain a distinction between deployment specific information (operation configuration) and application configuration (cache configuration).

In the following example, recovery only commences when eight storage-enabled members have joined the service and at least one member resides on hosts `machine1.demo.com` and `machine2.demo.com`. Within the distributed-scheme definition of the cache configuration:

```
<partitioned-quorum-policy-scheme>
        <recover-quorum>8</recover-quorum>
        <recovery-hosts>persistence-host-list</recovery-hosts>
</partitioned-quorum-policy-scheme>
```

The referenced address provider within the operational configuration:

```
<address-provider id="persistence-host-list">
         <address>machine1.demo.com</address>
         <address>machine2.demo.com</address>
 </address-provider>
```

When using the shared storage topology, only the capacity factor becomes relevant thus configuring the recover-quorum with sufficient storage-enabled members to recover the data is necessary.

Ensuring Adequate Resources for Persistent Caches

Enabling persistent caches does introduce some additional cost, specifically increased open file handles and disk usage.

Persistent caches involve partitions (because partitions are an atomic unit of data transfer), restoration from backup, scalable transactions, and recovery. As such, consider several factors when deriving the number of additional open file handles per machine: partition count (*P*), number of partitioned services (*S*) and the number of machines (*M*).

Additional Open File Handles Per Machine: $5 * (P * S) / M$

You must reserve sufficient disk space for active, snapshot, and archival locations. It is highly recommended that the amount of disk space be determined empirically based on a running application with contingency. The persistent storage mechanism asynchronously reclaims storage based on access patterns – thus the recommendation for empirical sizing. It is prudent to introduce sufficient overhead (20%) to ensure the resource is not saturated. However, the following guidelines for each storage location may be beneficial as a starting point:

- Active – an approximate overhead for active persistence data storage is an extra 10%-30% per partition. The actual overhead may vary depending upon data access patterns, the size of keys and values, and other factors such as block sizes and heavy system load.
- Snapshots - sizes are the same calculation as per active space multiplied by the number of snapshots envisaged.
- Archived snapshots – sizes are equal to the size of serialized key/value plus a 1%-2% overhead.

Note that the total size of a service can be derived using the Binary UnitCalculator and summing the Units attribute shown for each `Cache` MBean or running a report that aggregates the same metric.

Memory

When using active persistence, Coherence does require extra heap memory for persistence-related data structures. While this does vary based on the number of caches and the size of keys and values, a general guideline of 5% of the partition size should suffice. Thus a member that owns 10 partitions with 50MB of data per partition should allow an extra 2.5MB per partition or 25MB for the JVM.

Monitoring Persistence

As with all parts of the Coherence infrastructure, it is important to monitor persistent caches to ensure a smooth running system. This section describes some important areas/tools for monitoring.

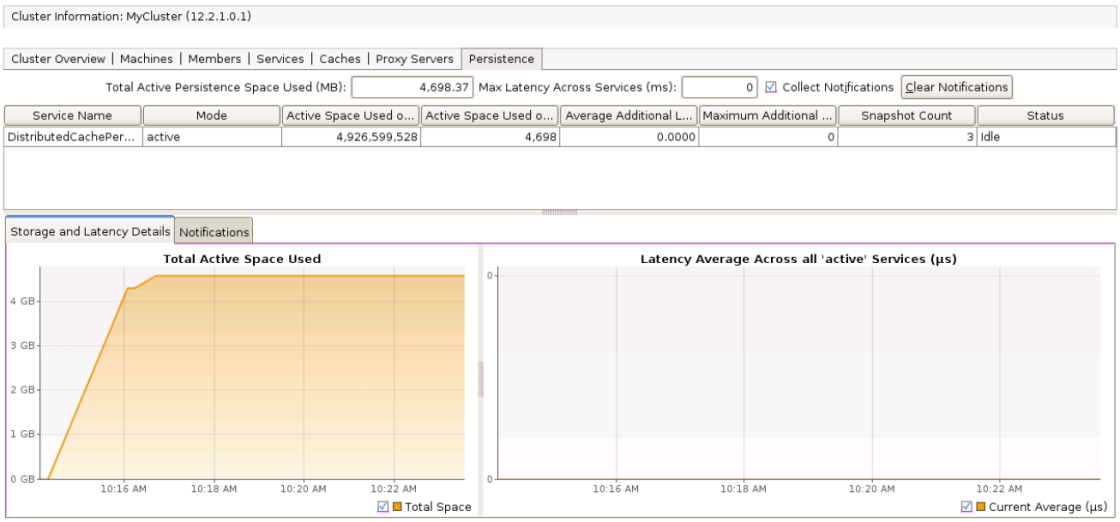
Reporter Reports

The following Reporter reports allow for monitoring of persistence-related statistics.

- reports/report-persistence.xml
- reports/report-persistence-detail.xml

Coherence JVisualVM Plug-in

The JVisualVM Plug-in provides support for all persistence snapshot operations such as create, recover, and so on, as well as tables and graphs to monitor persistence latencies and space usage, as shown below:



Service MBean Attributes

The existing `Service` MBean includes persistence-related attributes. JVisualVM uses these same attributes to show aggregate data.

Coherence Query Language (CohQL)

CohQL supports a number of persistence-related operations including list, create, recover, remove, and archive snapshots. There are also commands to retrieve archived snapshots, validate snapshots, and suspend/resume services. See the CohQL help commands for detailed usage.

Using WLST to Issue Snapshot Commands

When using persistence under Managed Coherence Servers in a WebLogic Server environment, you can use WLST to issue these commands. You must ensure you are in the `domainRuntime` tree before using the following WLST commands:

```
#
# Demonstration of calling recoverable caching API from WLST
#

serviceName   = '"ExampleGAR:PartitionedPofCache"';
snapshotName  = 'new-snapshot'

connect('weblogic','welcome1','t3://coh-dev3:7001')

# Must be in domain runtime tree otherwise no MBeans are returned
domainRuntime()

try:
   coh_listSnapshots(serviceName)
   coh_createSnapshot(snapshotName, serviceName)
   coh_listSnapshots(serviceName)
   coh_recoverSnapshot(snapshotName, serviceName)
   coh_removeSnapshot(snapshotName, serviceName)
except PersistenceException, rce:
   print 'PersistenceException: ' + str(rce)
```

```
except Exception,e:
   print 'Unknown Exception' + str(e)
else:
   print 'All operations complete'
```

Refer to the WebLogic Server Documentation for detailed information.

Managing Planned Change

Change to your production environment is inevitable. For instance, you may need to upgrade or patch Coherence or enhance your application. These are changes you can plan for and are part of the natural evolution and optimization of your application and environment. The rest of this section focuses on online changes, as offline changes tend to be much easier. An exception is data loading after a planned outage, for instance after a complete cluster shutdown. Refer to the Coherence Developer's Guide for strategies to load data into caches from a database and other sources.

Persistence

Persistence can help minimize downtime by using snapshots to save and restore caches during upgrading and patching. However, when caching a volatile database and the contents of the database evolve during the cluster downtime, then restoring a stale snapshot may not be appropriate.

Rolling re-starts

This is the process of sequentially re-starting each node in a Coherence cluster. It allows a complete cluster re-start, perhaps to introduce some code changes, without interrupting processing or availability – although it typically has some impact on performance and throughput. The pseudo code for this operation is as follows:

```
Upgrade database schema
Update Coherence nodes CLASSPATH

For each server
  For each node except management node
    For each distributed cache service
      While JMX value of StatusHA not MACHINE_SAFE
        Pause for 5s
      End while
    End For
    // Node loop
    Stop / Start node
  End For
End for

Restart management node
Introduce new version of client
```

Figure 19. Pseudo code for the rolling re-start logic

Note: A rolling restart requires sufficient capacity on *N*-1 nodes in the cluster to store all the cache data (where *N* is the number of cluster nodes or Managed Coherence Servers). Remote JMX management should also be enabled on all the cluster nodes.

Rolling restarts typically use custom scripts, but Coherence 12c can now automate this process through a bundled WLST template, `rolling_restart.py`.

Extend Client Compatibility

Starting with version 12.1.2.0.1, extend clients support both forward and backward compatibility with cluster proxies. That is, extend clients can connect to cluster proxies that have lower or higher version numbers. For example, a 12.1.2.0.2 extend client can connect to a 12.1.2.0.1 proxy. Extend client backward compatibility is not supported on proxy versions prior to 12.1.2.0.1, including 12.1.2.0.0 and proxy versions 3.7.1 or earlier.

See the Installation Guide for more information.

The Coherence binaries/JAR files

Minor upgrades to the Coherence binaries can usually be done online in a rolling fashion, for instance an upgrade from release 3.7.1.7 to 3.7.1.8 (*Major.Major.Pack:Patch*). However, check the release notes first because occasionally this may not be possible. With major and pack release upgrades, a complete cluster shutdown is required. This is because between successive pack releases or major releases the versions of internal communication protocols in Coherence may be different.

To perform a major upgrade to the Coherence binaries, for example from 3.7.1 to 12c, a parallel and identical cluster must be available. Usually this will be a Disaster Recovery (DR) site or another cluster setup in an active-active configuration using a data replication approach, so that the data in each cluster is synchronized.

Configuration Changes

Coherence cache and cluster configuration changes can be made either in the XML configuration files or through JMX. JMX enables a number of changes at runtime, such as changing the logging level, the high-units of a cache, and so on. However, these changes are made at a node or JVM level and not persisted. Therefore, changes made via JMX also required in the start-up configuration files to preserve the changes between node re-starts. With Managed Coherence Servers, WLST can perform online cluster wide cache and cluster configuration changes. These changes can be re-applied after an application re-start by applying the changes to an external cache configuration file. A GAR file references an external cache configuration file through a JNDI name.

Use a rolling restart of Coherence cluster nodes to modify cluster and cache configuration JMX parameters that are read-only.[5]

[5] For a full list of Coherence parameters that are read-only, please see the Coherence Management Guide.

Code Changes

A rolling restart of a cluster can modify custom classes, such as entry processers, classes used to represent keys or values, custom cache stores, eviction policies, event interceptors, and so on. The steps for initiating such code changes are as follows.

Prerequisites

- Classes (keys, values, custom entry processors, filters, aggregators, or invocable agents) must implement the Java `Serializable` or Portable Object Format (POF) interface to send them over the network or call them remotely.
- To support multiple client versions, value classes should support the Evolvable [6] interface and any changes to them must be additive; that is, they must add attributes not remove or change existing attributes.
- Changes to custom entry processors, event interceptors, custom filters, and so on should be backwardly compatible; that is, they should take account of Evolvable objects.
- Make changes to the database first. That is, if a database cache store is used, then these changes must be additive too.
- Lastly, make sure to meet the rolling restart prerequisites.

Steps

1. Apply any database changes.
2. Deploy your new application code changes to your cache servers or Managed Coherence Servers and modify the POF configuration file to add any new POF types.
3. Perform a rolling restart of your cluster. With Managed Coherence Servers, use the bundled WLST script to perform step 2 and 3.
4. Apply the code changes to any Extend clients that use the new functionality.

[6] See the Coherence documentation for a full description of how the `Evolvable` interface enables Coherence to support multiple versions of a cache data and cache clients. Note that evolvable objects also need to be POF objects.

Setting up the Production Environment

Capacity Planning

Capacity planning involves not only estimating and validating the capacity of a cluster but also ensuring that if limits are set they are not exceeded. Estimates should also include spare capacity to accommodate failures, peaks in demand, and growth. Before performing any calculations, it's worth answering some simple questions to save both time and resources.

Is holding the whole data set in memory required?

If the answer is no, then caching only frequently used data saves memory and makes capacity planning easier. A `CacheLoader` implementation can load entries on-demand from a database when there is a cache miss. To size-limit the cache, configure a high-units threshold and an eviction policy, which determines which entries to evict after reaching the threshold. However, remember that when configuring the high-units for a cache, it is per cache per node, not for the whole cache across the cluster.

Lastly, when caching frequently accessed data, it is important to remember that Coherence only queries data currently in a cache and cannot query un-cached data in an underlying data source.[7].

Is a backup copy of every cache entry required?

The answer is no when using the cache-aside pattern. Even when cache data changes and requires a backup, it is usually only required until the change is persisted. For instance, a backup is no longer required after a change has been persisted when using write-behind caching.[8] This reduces the memory requirements for a cache but again assumes access is only key-based so that any missing entries can be re-loaded.

Is POF serialization an option?

The Coherence Portable Object Format (POF) is a very compact and highly optimized object serialization format that can provide compression up to 5 times greater than the standard Java serialization format – though this depends on the structure of an object graph. So using POF serialization should yield a much higher density of cache data.

[7] This is also true if read-through via a `CacheLoader` implementation is configured.

[8] This can be configured by setting the `<backup-count-after-writebehind>` element to 0.

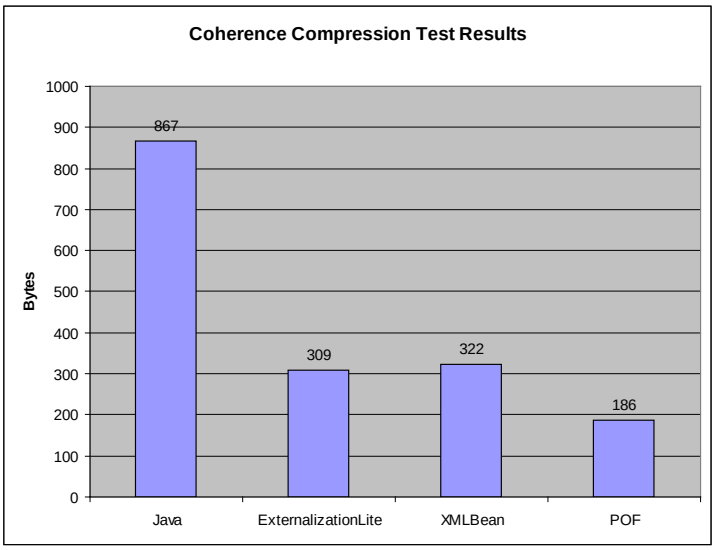


Figure 20. 5x better compression of a sample object graph using POF

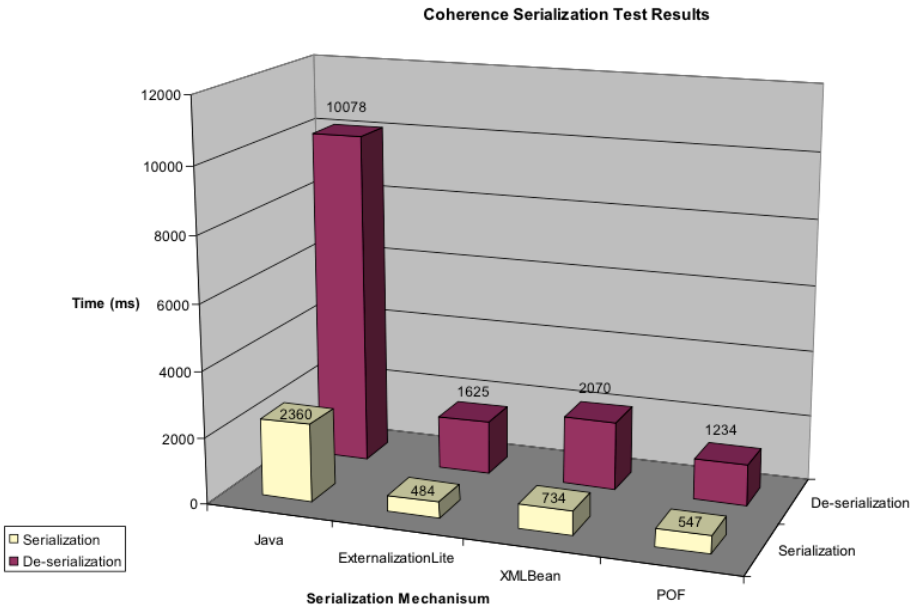


Figure 21. 10x faster de-serialization of a sample object graph using POF

Is fast disk storage, like SSD or even a local disk, available?

If fast storage is available, then the Elastic Data feature can increase storage capacity. A good starting point for exploring the benefits of Elastic Data is to consider using it initially for storing backups.

How much memory is my cache using?

Estimating capacity requirements is not an exact science. Assumptions and calculations are always a starting point for testing. Therefore, use the following recommendations as a guide.

The easiest way to answer this question is to perform some simple tests: store some representative cache entries; measure their size; and extrapolate the results to estimate the size of a planned cache. The Coherence JMX `units` metric on the `Cache` MBean reports the cache size for a particular node.

To calculate the total size of cache data held in memory (excluding indexes) as follows:

Total memory for cache data = *number of storage nodes* * *units* * (*backup count* + 1)

Since data distribution should be balanced, a sample storage node can be selected to determine average `units`.

Note: You should either specify `unit-calculator` as binary or specify `high-units` using standard KB, MB, GB suffixes.

This calculated figure includes the heap consumption of keys, values, and per-entry overhead, but not of indexes, and may be smaller than any heap utilization-based measurements due to unaccounted-for overhead.

A different approach is required for replicated or near caches. Because these entries are stored in a deserialized format[9], it is not possible to calculate their binary size using JMX metrics. Instead, use a tool based on the `java.lang.instrument` package, such as jbellis Jamm, to measure the heap consumption of a sample data set and then extrapolate the target cache size.

How many JVMs are required to store my cache data?

The required number of JVMs to store cache data (including primary copies of cache entries, backup copies if any, and index contents if any) depends on many factors:

- the amount of data being cached
- the number of backup copies
- the space consumption of indexes
- the size of JVM heaps and generations within them

Calculations for the number of JVMs needed to store cache data invariably must make assumptions about how much of a JVM heap the cache data can occupy. An old rule of thumb states that cache data should not occupy more than two thirds of a JVM's total heap. However, that rule of thumb is too basic for contemporary heap sizes. Consider, for example, a 32GB heap, which is common these

[9] Strictly speaking replicated cache entries are stored in a serialized format until they are accessed for the first time, after which they are then held in a deserialized format.

days. Using the two-thirds rule with a heap of that size, results in wasting over 10GB of usable heap space for cache data. Therefore, use the two-thirds rule of thumb only as a gross starting point if a more thorough investigation and analysis is not possible.

Likewise, it is important to explicitly size the young generation when using HotSpot with garbage collectors older than G1 (for example, the serial, parallel, and CMS collectors). By default, older garbage collectors size the young generation at one third of the declared heap, which is generally too large and wastes memory for a cache server JVM.

A more careful analysis of JVM heap capacity requires understanding the heap layout and garbage collection algorithms in the JVM. The HotSpot JVM divides the heap into tenured and young generations and the tenured generation is where cache data ultimately resides. Therefore, the question becomes how much of the tenured generation can cache data occupy for some size of tenured generation. Using a 32GB total heap as an example, explicitly sizing the young generation at 4GB leaves a 28GB tenured generation. However, do not assume that the cache data can occupy all 28GB of the tenured generation, because the garbage collector never allows the tenured generation to get 100% full. In addition, the garbage collector may run more frequently as tenured generation usage approaches 100%. A more precise, but still conservative, rule of thumb is to assume your cache data can occupy no more than the total heap minus two times the young generation size of a JVM heap (32GB – (2 * 4GB) = 24GB). In this case, cache data can occupy 75% of the heap. Note that the resulting percentage depends on the configured young generation size.

The following is a hypothetical example that illustrates how to calculate the number of cache server JVMs required to store a certain amount of cache contents, backups, and indexes in the heaps of cache server JVMs.

Assumptions

- The total amount of heap space required by cache contents, backups, and indexes is 512 GB
- The size of other application data on the heap of cache server JVM is negligible
- Each cache server has a declared heap size of 16 GB
- The CMS garbage collector is used
- The young generation is sized at 2GB
- The available tenured generation space per cache server JVM is the declared heap size minus two times the young generation size.

Use the following calculation:

Total Amount of Heap Space Required / Available Tenured Generation Space per Cache Server JVM = Number of cache server JVMs required

For the assumptions in this example:

512 GB / 12 GB = 43 cache servers

In addition, provision an additional cache server to ensure that there is sufficient "failover headroom" in the event of a cache server JVM failure. Subsequent recommendations for an additional host machine (discussed below) also meet this requirement.

Note: Detailed JVM tuning guidance is beyond the scope of this white paper, as it is usually dependent on the memory and CPU utilization profile of an application, desired application performance characteristics, and other factors.

How many machines will I need?

Although the hardware resource requirements for different Coherence applications can vary significantly, some general recommendations follow:

- Use at least 3 machines to ensure that the data is evenly balanced and that if one machine fails the remaining machines can own all failed-over data and the configuration is still resilient (i.e., "machine-safe")
- As a rule, start with one core per cluster node, whether it is a proxy, cache, or management node.
- Apportion resources according to the requirements of your application, not the resources of your machines. Having machines with balanced resources that align with the requirements of your application's workload ensures full resource utilization. For instance, if the servers have 512 GB of memory, but only 16 cores and 1 GbE network interfaces, the application may not fully utilize all the memory if its workload is compute or I/O intensive. Furthermore, the MTR after a machine failure is a lot longer in an unbalanced configuration, because a large amount of data rebalancing over a network with limited bandwidth.
- Finally, just as *n*+1 cache server JVMs should be provisioned (where *n* is the number of cache server JVMs required to support your application under normal conditions), *n*+1 machines should also be provisioned so that if a machine fails or is shut down there is sufficient capacity to store all the data.

Using these recommendations, the example below illustrates how to calculate the number of machines required to host this Coherence cluster.

Assumptions

- Oracle HotSpot 1.8 64 Bit JVM
- JVM binaries overhead for 64 bit Hotspot JVM ~512 MB
- OS memory reservation ~8 GB
- 43 cache server JVMs, each with a 16GB heap (from previous example)
- Assign approximately 1 core per cache server
- A single management node with a heap of 4 GB (this will not be included in sizing)
- Server specification of 16 cores, 256 GB memory and 1 GbE network

For this example, the memory footprint for each cache server is:

512 MB + 16 GB = 16.5 GB.

The number of cache server JVMs per machine is 15:

15 JVMs *16.5 GB heaps = 247.5 GB RAM, leaving ~8GB for OS reservation and overhead

The total number of machines required is calculated as (*number of cache server JVMs / cache server JVMs per machine*) + 1 additional machine for resilience:

(43 / 15) + 1 = 4 machines

The additional machine is required to ensure that if one fails there are still sufficient resources (memory) available to absorb failed-over data from the failed machine.

Cache server processing requirements can vary significantly. For instance, an application processing many events, queries, or performing in-grid processing typically uses more CPU resources than one handling simple key-based access operations (put, get, remove). However, as a rule of thumb, allocating 1 core per JVM is a good starting point for testing.

How do I size Proxy Servers?

A reasonable starting point for planning your deployment is to proxy-enable your cache servers. That is, use cache servers as both proxies and storage nodes. As you scale out the cache servers, you increase both cluster storage capacity as well as aggregate proxy bandwidth.

Alternatively, it is possible to run proxies and storage nodes in two separate tiers and scale them independently; although, this is generally not necessary and requires more careful planning.

Cluster Topology

The topology of a Coherence cluster can have a significant impact on its performance, reliability, and scalability. This section discusses the various components of a Coherence cluster, introduces some new features in the 12c release, and complements the documentation with some suggestions around best practices.

Ideally, all the nodes in a Coherence cluster should be located on the same network segment and connected through the same switch to ensure fast and reliable cluster communication. Members of any given role should be configured similarly and have access to similar levels of resources to maintain consistent performance within that role (tier). Members in different roles (tiers) may have significantly different requirements and thus may have different resources available to them (including running on different Operating Systems, JVM variants, or both).

Cluster Discovery

In configuring a cluster, one important decision to make is if the use of multicast or unicast-based clustering. Coherence has always defaulted to multicast based clustering as it provides the simplest mechanism for setting up a cluster.

Historically, there have been concerns about the possibility of the cluster placing excessive multicast traffic on the network and causing "multicast storms". Coherence is very judicious in its use of multicast and, in fact, sends the vast majority of cluster traffic over unicast even with multicast enabled.

Starting with 12.2.1, we further restrict the usage of multicast. By default, only cluster discovery uses multicast. In addition, only two machines in the cluster ever join the multicast group at any given time. The result of these changes means that Coherence's usage of multicast is now minimal.

During the discovery process, the discovering member does not need to join the multicast group; it just sends a multicast transmission to the group. At most two cluster members receive the transmission and only one of the two replies (except in the case of ongoing failover). That reply is a direct unicast reply to the discovering process further bypassing multicast.

For environments that do not support multicast, Coherence offers a unicast-based discovery called well-known-addresses or simply WKA. With WKA based discovery, the discovery process starts with a subset of the cluster's IP addresses and the discovering processes query each address in the list to find the cluster.

In either multicast or WKA discovery, the Coherence cluster name is now the most important identifier for a cluster and should always be set. There is no need to modify the default cluster port of 7574.

This rest of this section outlines some guidelines for configuring and deploying the various components of a Coherence cluster.

Proxy Servers

As part of a product-wide initiative to make Coherence more "self-managing", 12c introduced a dynamic thread pool for proxy services. The dynamic thread pool is the default for all services and you no longer need to specify a thread-count. This allows a proxy service to grow and shrink its thread pool to handle extend connections as needed. Also, consider using proxy-based load balancing to spread client load evenly across the proxy servers.

While proxy servers can listen on a fixed IP and port, this introduces operational challenges. Specifically, it introduces the need for specialized configuration on a per-process basis as no two proxies can listen on the same IP and port. Historically, this proxy configuration was necessary to know the IP and ports so that extend clients could configure that information in order to locate their proxy.

Proxy configuration has been improved and it is now possible and preferable to allow Coherence to manage the IP and port assignments and to have extend clients use cluster discovery to find both the cluster and desired proxy. By default, if a proxy is not configured with a specific address and port, then it will listen on the wildcard address (all local IP addresses), and an ephemeral port.

The default configuration allows all proxies to share an identical configuration without the risk of a port conflict between proxies or between other processes. Extend clients can now use the cluster's operational configuration, which they use to discover the desired cluster. The client's remote cache scheme definition does not require any specific proxy address or port and requires only a `proxy-service-name` element that specifies the name of the proxy service to which it connects. The client use these two pieces of information to find the cluster and then it is load balanced onto one of the proxies running the designated proxy service.

Very much related to the changes in proxy configuration and discovery, it is now easy to deploy proxies and the recommended default deployment model is to run Coherence servers as both proxies and storage nodes (i.e., storage-enabled proxies). By having every storage-enabled node also a proxy, both the proxy tier and data tiers automatically scale as the cluster grows. Data and client connections are spread across all servers; thus, each new server helps lower the load on all other servers.

Extend Clients

Extend clients have nearly all the functionality of those in a cluster, but can utilize a range of technologies (Java, .Net, C++, and others like JavaScript using the REST API). Since extend clients access cache data via a proxy service and are not directly involved in cluster communications, they can reside on slow or unreliable networks and do not impact the cluster when they startup or shutdown. Therefore, extend clients are the configuration of choice for clients that are desktop applications or only run for a short period– for instance to load some data or to perform a query.

To ensure extend communications are reliable and resilient, extend clients can be configured to use an out-going heartbeat, with a timeout, to make sure the proxy they are connected to is still responsive. Extend clients that are listening for events can register a "member left" event listener, to detect connection errors and re-connect by issuing a simple out-going request– for instance by calling the cache `get(null)` method.

The Management Server

Coherence 12.2.1 introduces a dynamic management mode, which automatically selects one of the cluster nodes to be the management node. This is the default management mode for Managed Coherence Servers. Cluster members no longer need explicit JMX management configuration.

To connect to the dynamically allocated node, a new discovery tool is available to print out the JMX URL for a given cluster. Use the tool from JVisualVM or other JMX tool to connect to the current management node.

```
jconsole $(java -cp coherence.jar com.tangosol.discovery.NSLookup \
       -name management/JMXServiceURL -cluster MyCluster –host
multicast_ip_or_any_cluster_member_ip)
```

See the JMX Documentation for more information on using dynamic mode.

In certain circumstances, such as very large clusters, load on the management node can become significant and may warrant specifying one or more dedicated nodes as management nodes. This is achieved by setting the `-Dcoherence.management=dynamic` system property on all nodes

that can be a management node and `-Dcoherence.management=none` on all other nodes. In this case, the management nodes should be set to storage-disabled.

When using Managed Coherence Servers in the 12.2.1 release, the WebLogic Domain Administration Server collects JMX metrics for the Coherence cluster. This can then be retrieved via the domain runtime MBean Server. See the Oracle Documentation for more information.

Cache Servers

Coherence cache servers and client applications typically run in separate JVMs to allow independently tuning and scaling the data-grid and client tiers. However, for basic cache use cases, or when caching small amounts data, running clients and cache services in the same JVM can make sense.

Cache Types

Near Cache

A near cache can improve application performance and reduce network traffic by caching a local copy of a previously requested entry. However, when deciding whether a near cache is appropriate, consider a number of factors:

- How often do clients request the same data?

  A near cache only improves performance when requesting entries more than once.

- What is the distributed cache update rate?

  A near cache only improves performance if a client has a chance to reuse its local copy.

Coherence JMX metrics provide insight into this information. If the data access profile is unknown, then configure an example near cache to measure its effectiveness by monitoring the `HitProbability` attribute of the `Cache` MBean as shown below.

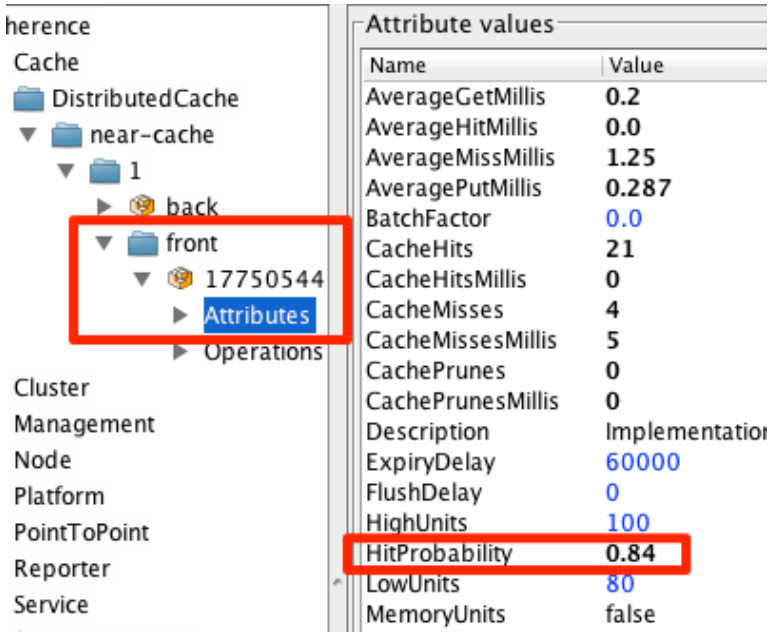


Figure 22. JConsole view of a near-cache HitProbability attribute

The flow diagram below illustrates the decision process for deciding if your client applications benefits from a near cache and, if so, what configuration provides the best performance. As always, validate your configuration through testing.

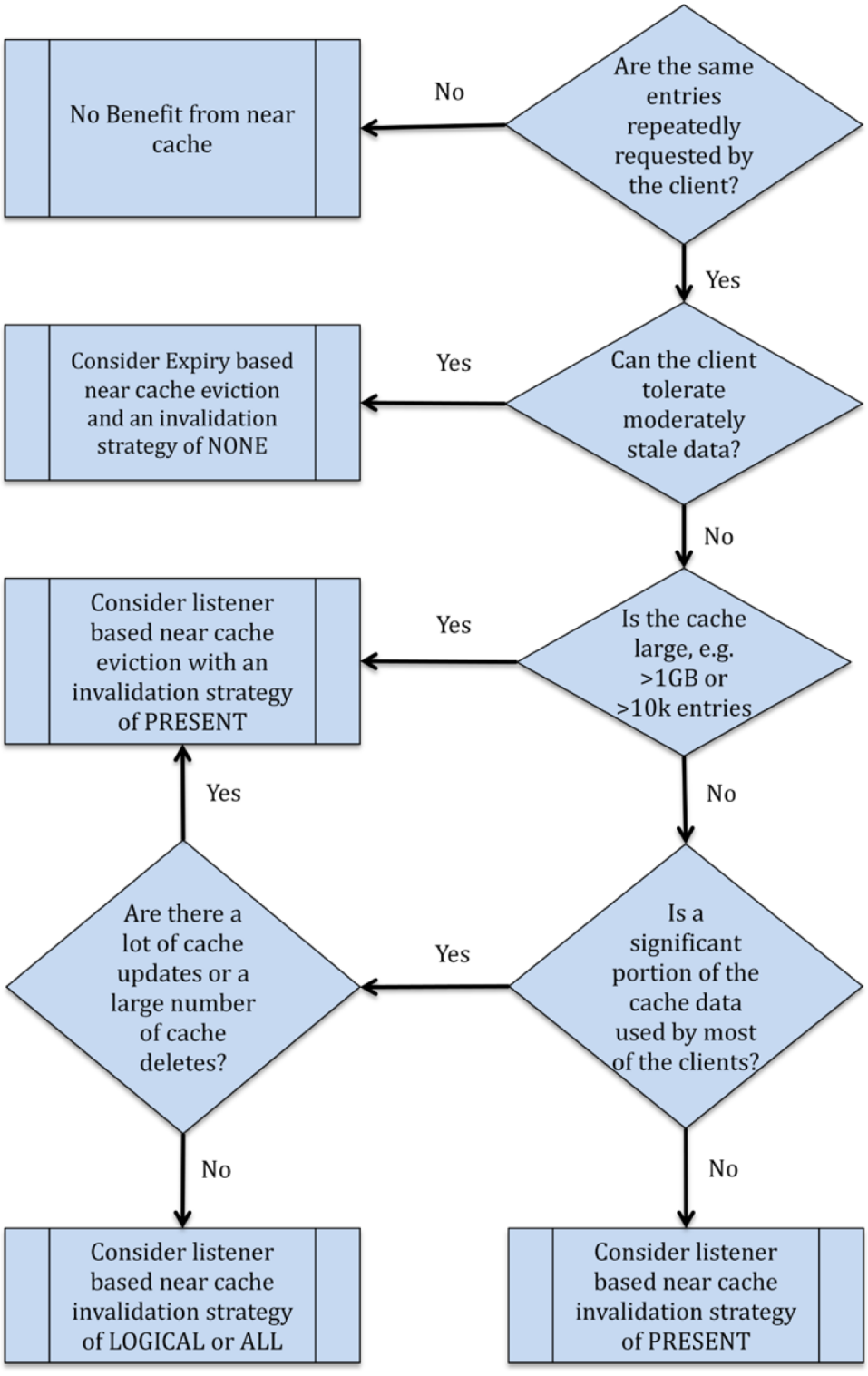


Figure 23. Decision tree for selecting the best near-cache configuration

Lastly, when configuring a near cache:

- Always size limit your near cache and be conservative. Find an optimum near cache size by setting a small near cache size and then gradually increasing its size to find a good "hit rate" that consumes the minimum amount of memory. Remember that near cache contents are unserialized, and consume more heap than their serialized form by a factor of at least four.
- Do not configure the `ALL` invalidation strategy if you intend to call the `cache.clear` method or perform a bulk update or delete of you distributed cache, because this can generate a large number of events.
- Be aware that the new default invalidation strategy for a near cache (if none is specified) in Coherence 12c is now `PRESENT`.
- Coherence 12c introduces a new invalidation strategy of `LOGICAL`. This strategy is like the `ALL` strategy except that clients do receive synthetic events from operations like cache eviction.
- If the entries in the distributed cache are periodically updated in bulk, then consider using a `NONE` invalidation strategy, to prevent a large number of events being sent to clients, in conjunction with one the following techniques:
    - After the bulk update use the invocation service to clear the clients' near caches, so that they will re-read the new entries.
    - Notify clients before the bulk update so that they can remove their near cache and notify the clients afterwards so that they can re-create their near cache. To accomplish this, use an event listener on a control cache to communicate the state changes.

Replicated Cache

A replicated cache replicates its contents wherever it is running. Although its configuration is similar to that of a distributed cache, some key differences are:

- Replicated cache services always start when they appear in a cache configuration, even on a storage-disabled node. The auto-start and storage-enabled options are only for distributed cache services.
- Entries are held in a deserialized format – after they have been accessed for the first time.

Replicated caches are particularly well-suited for caching small, relatively non-volatile data sets (e.g. "reference data") that need to be accessed with the lowest possible latency by code using that data (including code running within cache server processes, for "in-place processing" use cases) and that need to be kept immediately coherent within all using processes whenever any of that data changes. For all other caching use cases, a distributed cache combined with a near cache or Continuous Query Cache is generally a better choice and provides greater flexibility.

Distributed Cache

A distributed cache offers flexibility, scalability, performance, and reliability. The cache data managed by a distributed cache service is split into partitions. Cache entries are randomly assigned to these partitions, unless data affinity is used to store related cache entries together in the same partition. A

partition assignment strategy controls the distribution of partitions among the cluster members that run the same service.

The number of required partitions for a distributed cache service depends on the amount of cache data being stored. For details on configuring the number of partitions, see the Coherence Developers Guide.

Data Replication Between Clusters

Multi-site architectures for high availability, scalability, and performance (via geo-proximity) are commonplace among Coherence users and their prevalence in the industry continues to increase. When planning a multi-site architecture in which each site has a separate Coherence cluster, and in which at least one site has a database underlying Coherence caches, questions arise as to the "best" way to replicate data between Coherence clusters at different sites. In general, the two primary choices are replicating at the Coherence level using the Federated Caching feature introduced in version 12.2.1, or replicating at the database level using GoldenGate HotCache to get data from a database at a site to Coherence caches at that or other sites.

As a guiding principle, the data management technology mastering the data should replicate the data. For example, if an application has Coherence caches that are not integrated with a data source at all (i.e., have no read-write-backing-map or cache store), then use Federated Caching to replicate those caches' contents between sites. It does not make sense architecturally (too much complexity, too many moving parts) to introduce cache stores and an underlying database and database-level replication and HotCache just to replicate those caches' contents to a different site. On the other hand, if an application already uses HotCache at a single site to replicate transactional changes from a database at that site into Coherence caches, then it's a simple extension to use HotCache to replicate those same transactional changes to Coherence caches at another site instead of setting up Federated Caching between those caches at different sites.

A number of other factors should be considered when choosing a replication approach. For example, what is the organization's expertise with Coherence and Federated Caching (or its predecessor Push Replication), and what is the organization's expertise with GoldenGate? How much replication latency can be tolerated, and what are the expected relative latencies of different "replication paths"? Are the Coherence caches read-only, or read-write, and what are the chances of write conflicts between sites with different replication paths?

In general, simple is best. Choose the replication path with the fewest technologies, fewest moving parts, fewest "hops" for the data to take, and fewest chances of write conflicts, which usually means replicating the data from where it is mastered.

Using Federation

Starting with version 12.2.1, Federated Caching is the recommended way to connect multiple clusters in different locations together. Federation provides a high-throughput and scalable architecture for asynchronous cluster replication.

Federation provides:

- Multiple replication topologies, including active-active, active-passive, hub-spoke, central-replication, and custom topologies
- Programmable conflict resolution APIs to accept, reject, or modify cache entries both locally or remotely
- No change to application code as configuration is based on configuration files and is an extension of a Partitioned Service

See the Coherence Documentation for detailed information on Federation setup and usage.

Using Coherence 12c HotCache feature

A database is the primary data source in many use cases. Therefore, changes to cache entries held in Coherence require a write to the database. Furthermore, the database is often already being replicated to the locations where other Coherence clusters reside. In these scenarios, use HotCache for propagating database changes, applied through database replication, into a Coherence cluster at the remote location. In fact, when using Golden Gate for database replication, it can replicate to both the database and the Coherence cluster at each location through HotCache.

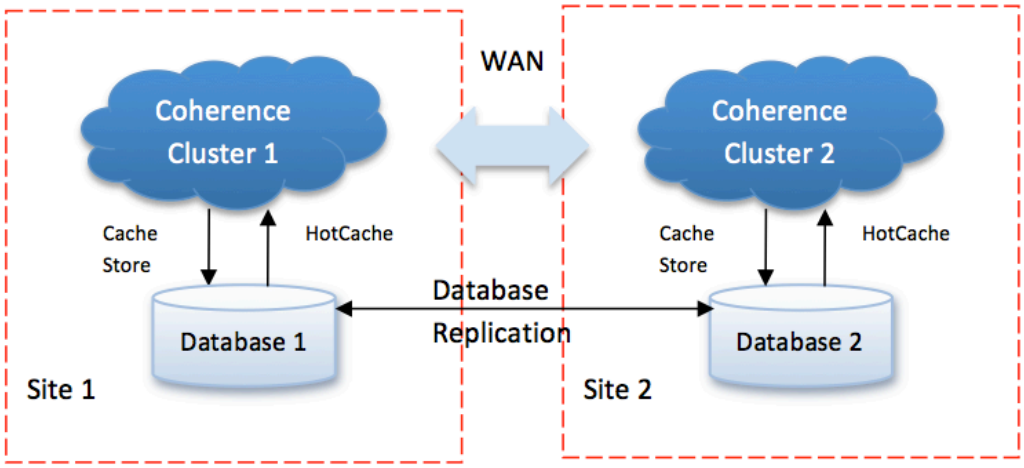


Figure 24. Inter-site replication using HotCache and Golden Gate

Hardware Considerations

It is tempting to size your hardware resources based on the normal requirements of your application. However, if failures occur, Coherence needs to perform more processing (i.e. the recovery and rebalancing of lost data) with fewer resources. So make sure you have sufficient hardware to handle potential failures within your SLAs. Platform specific considerations are outlined in the Coherence Administration Guide.

Software Considerations

Coherence has a few additional software requirements. Cluster members are based and certified on Java and not the underlying operating system. However, when setting up a test or production environment for Coherence, you should consider the following:

- **JVM** – Some platforms provide their own JVM implementation. However, if there is a choice of JVMs then select the one that is most widely used– everything else being equal.
- **Operating System** – Although Coherence runs on a range of operating systems, which each have their strengths and weaknesses, Linux is probably the most popular platform and is also used for internal testing.
- **Monitoring** – See the next section.

Monitoring

A range of monitoring tools exists from Oracle and 3rd party vendors for Coherence. Oracle Enterprise Manager is an enterprise-wide monitoring tool that many companies use to monitor Production Coherence Clusters via the Management Pack for Oracle Coherence. For developers, the Coherence JVisualVM Plugin is an ideal tool to view information from a single Coherence Cluster. Monitoring Coherence is critical in a production environment and the overhead of doing so should be factored in to any performance testing.

Metrics available through JMX are usually gathered through the Coherence Management Node. The interval at which JMX metrics are gathered usually needs to be adjusted (the default interval is 1 second). An interval of 10 seconds may be suitable for small clusters, but for larger clusters, this should be increased to 30 seconds – or even higher. The quantity of MBean metrics collected depends on the number of nodes, caches, and services in a cluster. The size of each MBean also varies depending on its attributes. Below is a list of the main Coherence MBeans and the multiplier to calculate how many MBeans are collected at each interval.

MBEAN TYPE	MULTIPLER
Cache	Service Count * Cache Count (for service) * Node Count
Node	Node Count
Partition Assignment	Service Count * Node Count
Point to Point	Node Count
Reporter	Node Count
Service	(Service Count * Node Count) + (Management Service * Node Count)
Storage Manager	Service Count * Cache Count (for service) * Node Count

MBean filters can reduce the number of Coherence and platform MBeans by preventing remote accesses to them through the Coherence management framework. Filtering can be useful to remove platform metrics that are collected through a different mechanism, for instance using Oracle JVM Diagnostics, or to exclude metrics from ancillary or temporary caches.

Larger clusters need to configure a larger heap size for the Management Node. Where as a 2 GB heap for a small cluster may be fine, a 4 GB heap may be required for a larger cluster. If the Management Node is performing excessive processing, using a lot of memory, or metrics have collection gaps, then the collection interval may be too short, or fewer MBeans may need to be collected at each interval. Further details on configuring the Coherence Management Node can be found in the Coherence Management Guide.

Although Coherence makes a wide range of metrics available through JMX, it's also important to monitor its ecosystem including the JVM, CPU, memory, OS, network, logs, and disk utilization and the application itself (i.e. application-level metrics such as response times and throughputs). The sections below provide some of the key metrics that should be monitored, along with some suggested thresholds for raising alerts. Note this is not a definitive list of metrics to monitor and thresholds may vary between applications and environments.

Data Loss

Data loss is the most important metric to monitor for obvious reasons. In Coherence 12c there is a new partition MBean notification on the `SimpleStrategy` MBean raised when a partition is lost. This event is also logged as an "orphaned partition" log message. The following graphic shows the JMX "partition-lost" notification event.

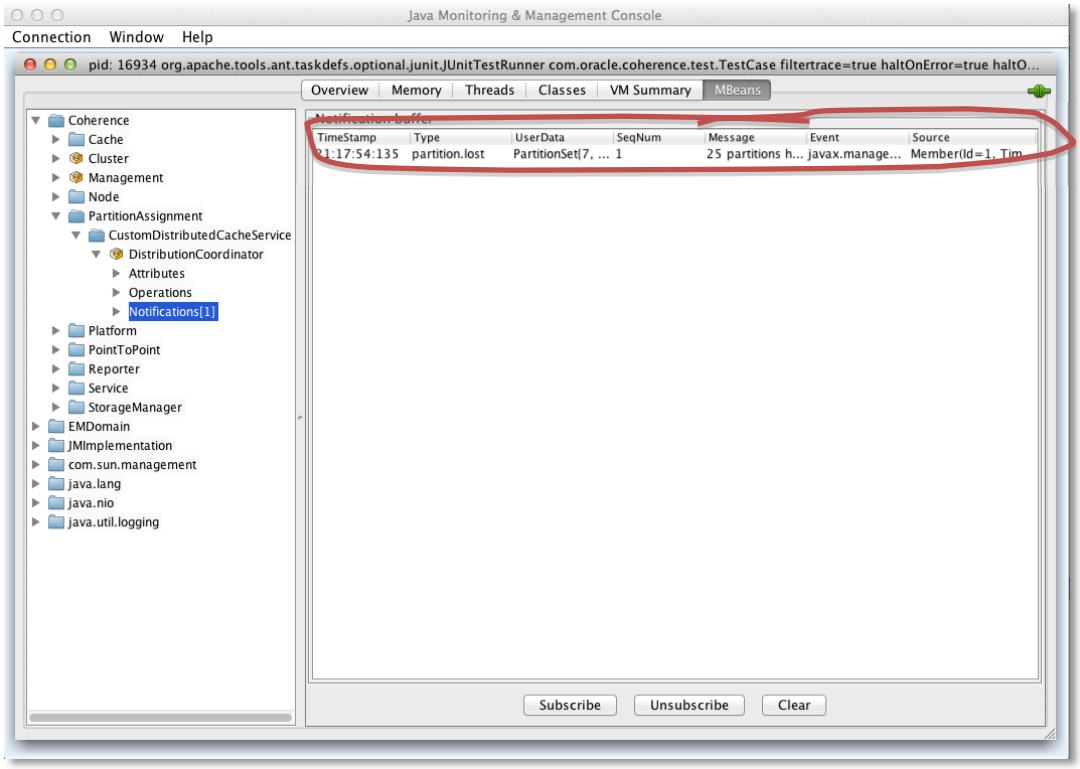


Figure 25. A partition-lost JMX notification event raised when a partition and data is lost

Partitions can be lost because two machines have failed simultaneously. An error alert should be raised if this JMX notification event is received or an "orphaned partition" error message appears in the Coherence log file. Configuring multiple backups or implementing the other HA strategies outlined earlier can mitigate the risk of this occurring.

Data at Risk

The `StatusHA` attribute on the `Service` MBean highlights the vulnerability of cache data. It indicates how safe your data is. If the HA (High Availability) status of one of the distributed cache services changes to `NODE-SAFE`[10] or even worse `ENDANGERED`,[11] then a warning alert should be raised. This can happen if a storage node leaves a cluster. If a node remains in this state for a prolonged period, for instance more than two minutes, then an error alert should be raised. The HA status of a distributed service may legitimately be in one of these states for a short period of time when re-balancing during a rolling restart. However, if this happens for a prolonged period, then there may

[10] The `NODE-SAFE` status indicates that the primaries and backups of some cache entries are only held on different nodes, not different machines

[11] The `ENDANGERED` status means that data is only held in one node

be an issue with a service thread on a node. For instance, a partition maybe "pinned" to a node because of an error state

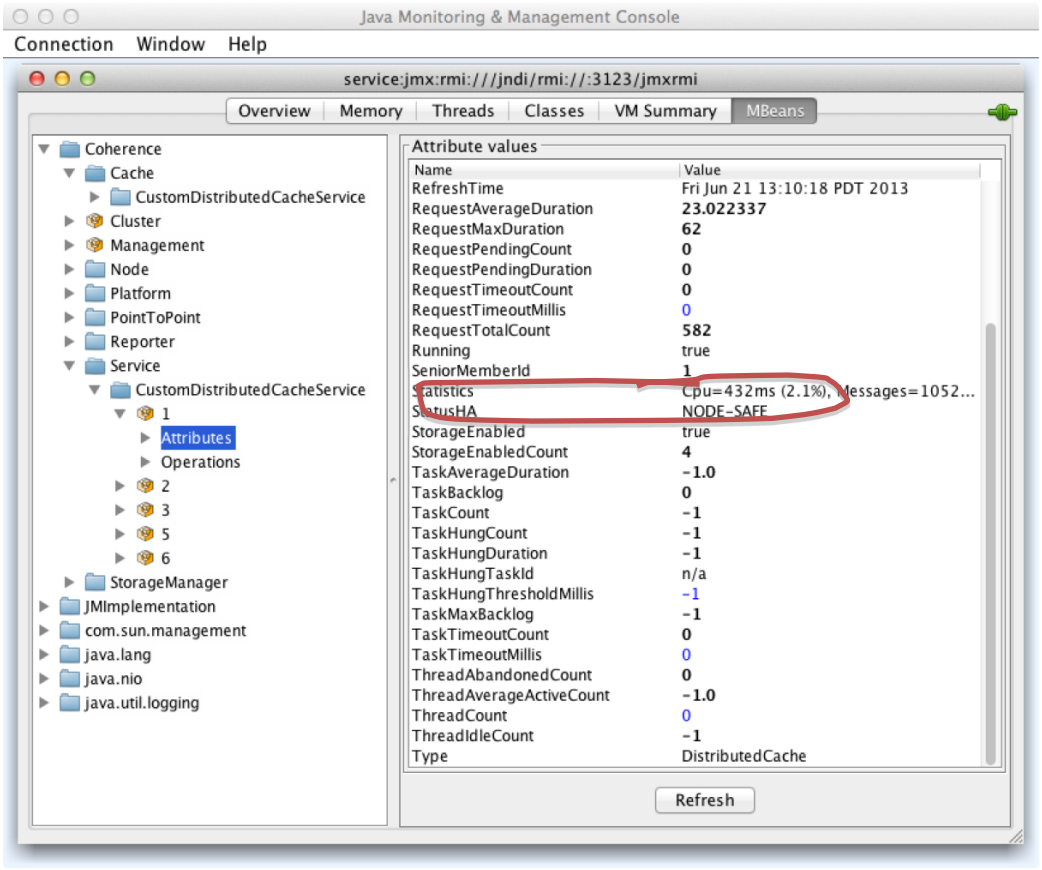


Figure 26. The `Service` MBean `StatusHA` attribute indicates here that cache data is only held on another node, not another machine

Cluster Change

A change to the number of cluster nodes can be significant. If the node that leaves holds cache data, is a proxy, or runs some important processing, then an alert should be raised. The `MembersDeparted` and `MembersDepartedCount` attributes on the `Cluster` MBean capture changes in cluster membership. Using the "role" of a node is one way to determine if this is a significant event.

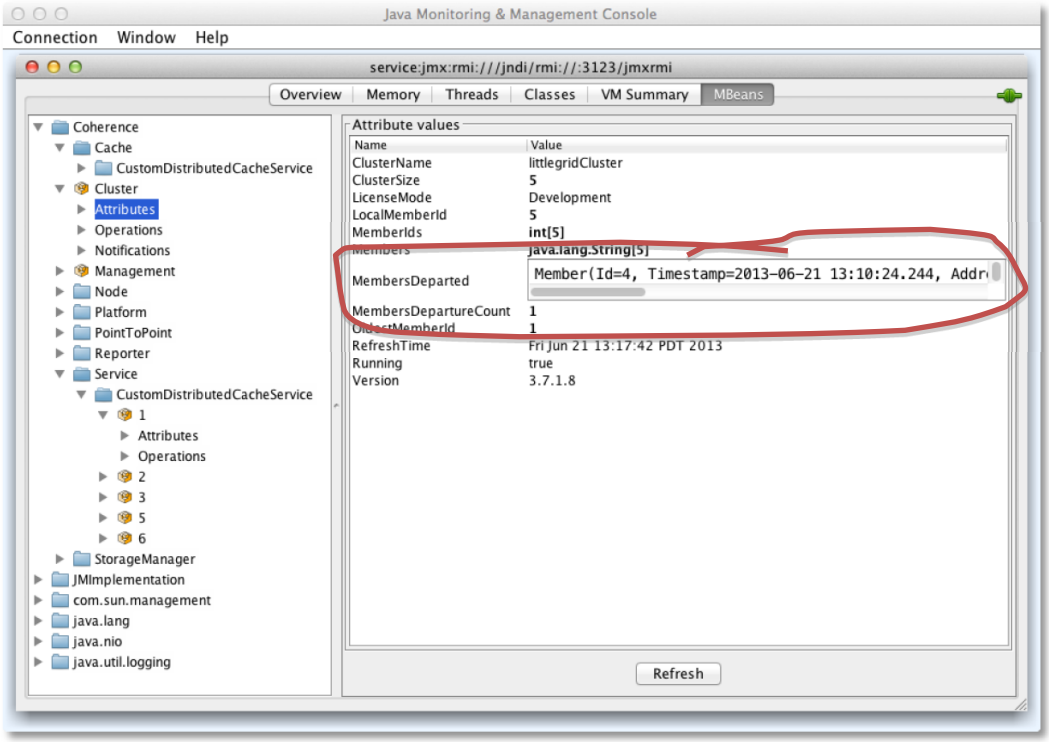


Figure 27. The `Cluster` `MembersDeparted` and `MembersDepartedCount` attributes record metrics about members that have left the cluster

A service restart also indicates that a problem has occurred. For instance, the service guardian restarts a service thread that has hung on a node. To capture this, Coherence 12c adds the service `JoinTime` attribute on the `Service` MBean. A value change for this attribute between JMX collections indicates a services restart.

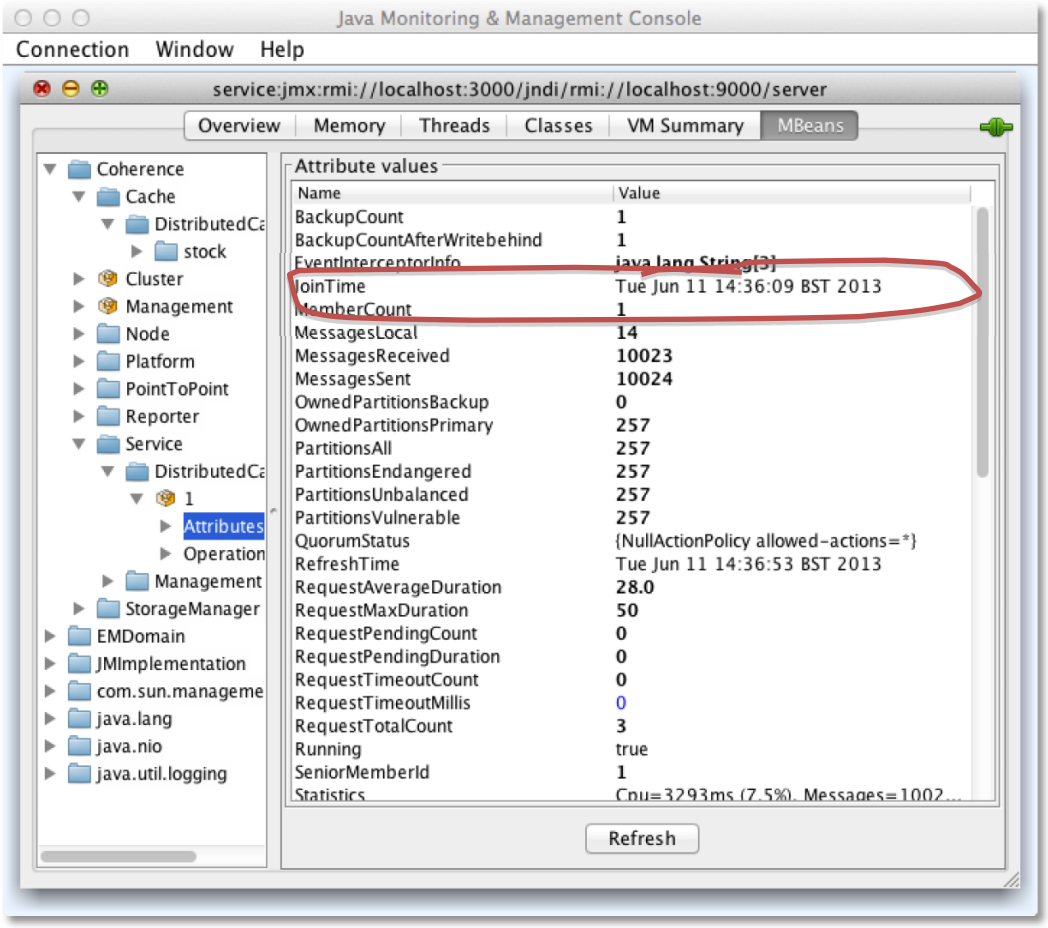


Figure 28. The new `Service` MBean `JoinTime` attribute

Low Heap Space

Closely monitor heap consumption so that if data volumes grow beyond expected levels, additional capacity can be provisioned and the unexpected growth investigated. Although careful capacity planning should ensure that there are sufficient resources to withstand the loss of a machine, processing spikes or incremental growth may not have been predictable. The threshold at which this growth becomes a risk to the cluster depends on the number of machines in the cluster. For distributed caches with one backup copy configured, when a machine fails, the surviving machines must collectively assume ownership of the primary, backup, and index data previously owned by the failed machine. Thus, there must be sufficient "headroom" in the heaps of cache server JVMs on every machine in the cluster to withstand a machine failure. The following table tabulates the minimum headroom required in each cache server JVM heap, as a function of the number of machines in the cluster.

Number of Machines Pre-Failure	Headroom Needed per Cache Server JVM Heap
3	17%
4	8%
5	5%
10	1%

This headroom refers to available tenured generation space, taking into account that the garbage collector also needs to guarantee sufficient space in the tenured generation to promote all live objects from the young generation. For example, in an 8GB heap with a 1GB young generation and 128MB survivor spaces (assuming HotSpot pre-G1), the garbage collector must in the worst case guarantee 896MB available space in the tenured generation in case it needs to promote all objects in Eden and one survivor space. This means Coherence primary and backup copies and indexes can occupy about 6.125GB of the tenured generation and 17% of that 6.125GB must be reserved as headroom for a machine failure in a three-machine cluster (without leaving any additional margin).

As well as identifying suitable heap utilization thresholds, it is also important to make sure heap utilization is accurately measured. The only reliable way to do this is to measure the heap space available after the last full GC, which is available on the following MBean as the used value:

```
Coherence:type=Platform,Domain=java.lang,subType=GarbageCollector,
name=PS MarkSweep,nodeId=<node id>
```

Between full garbage collections, the best approximation of used heap is on the following MBean as the used value:

```
Coherence:type=Platform,Domain=java.lang,subType=GarbageCollector,
name=PS Scavange,nodeId=<node id>
```

The graphic below shows the MBean.

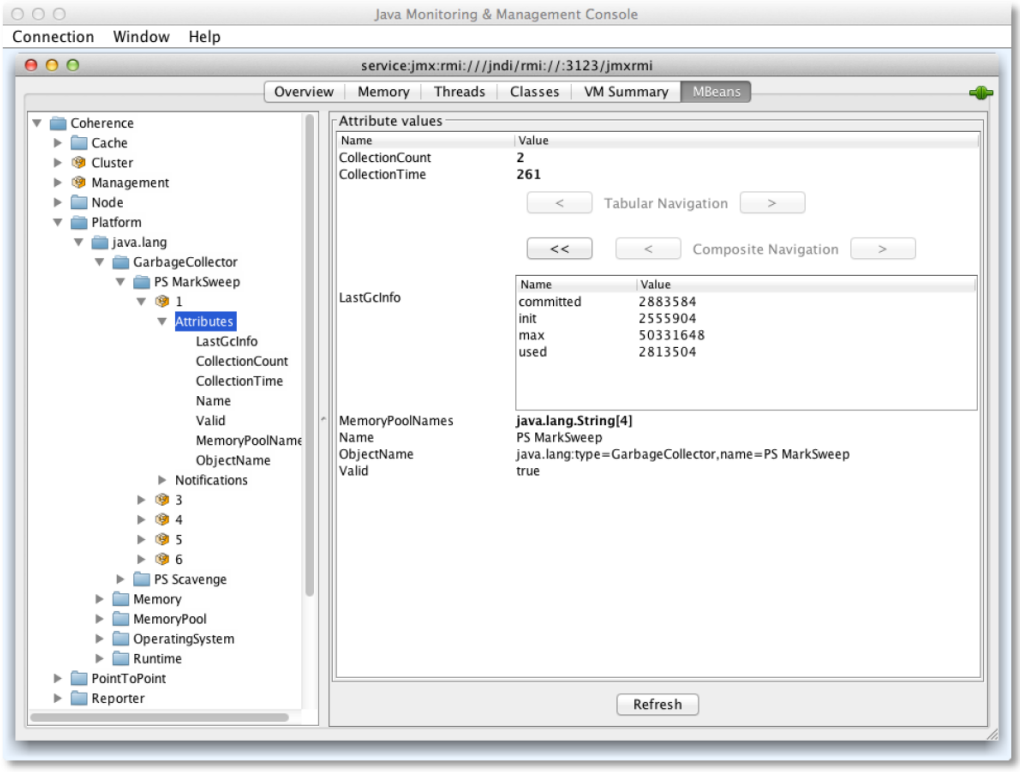


Figure 29. The heap used after last GC can be found under the `PS MarkSweep` MBean

Long Garbage Collection (GC) Pauses

Long GC pauses can affect application performance, throughput, and scalability. If a cluster node is taking a long time to perform a garbage collection, then an alert should be raised. The alert threshold can vary, but a suggested threshold for a warning alert is >1 second and >5 seconds for a critical alert. This information can be accessed on the `PS MarkSweep` MBean outlined below. Other possible reasons for these alerts, besides actual long GC pauses, can be members leaving the cluster (putting additional memory pressures on the remaining members), CPU starvation of a cache server process by other processes on a machine, network congestion, process swapping by operating systems, and moving virtual machines running Coherence cluster members (e.g. via VMotion).

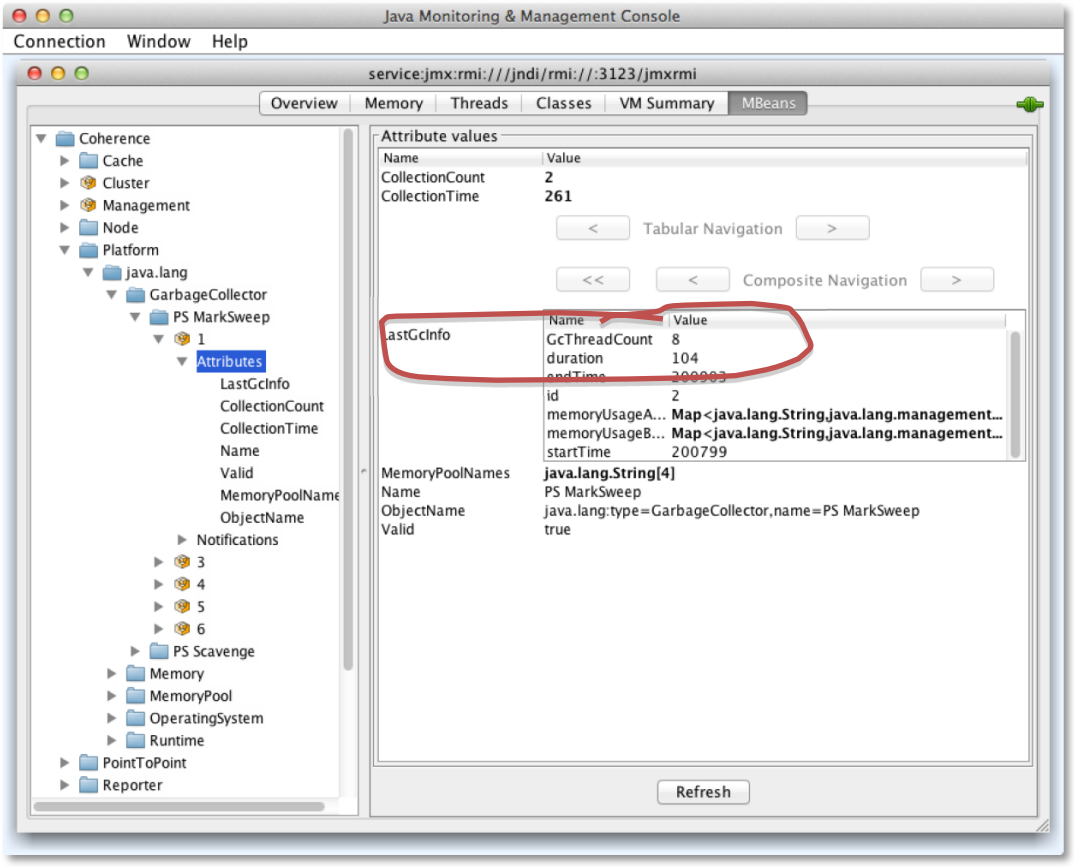


Figure 30. The last GC pause duration can be found in the in "*PS MarkSweep*" MBean GC metrics

Insufficient Processing Resources

In Coherence 12.2.1, all services use a dynamic thread pool unless a thread count value is explicitly set. In this case, Coherence manages the thread pool to ensure optimal throughput without adversely affecting performance. It is a best practice to use the dynamic thread pool; however, if you do set a thread count value, consider the following.

If the Coherence service threads cannot keep up with the number of requests being made, then the task backlog queue starts to grow and affects an application performance, throughput, and scalability. Check the `TaskBacklog` attribute on the `Service` MBean for each service. A suggested threshold for raising a warning alert is if the queue length is >10 and if >100 a critical alert should be raised. Corrective action might be to start more service threads. However, if too many service threads are allocated the `ThreadIdleCount` attribute on the `Service` MBean will be consistently >0. This indicates that the thread count for a service is too high. The `TaskBacklog` attribute is shown below.

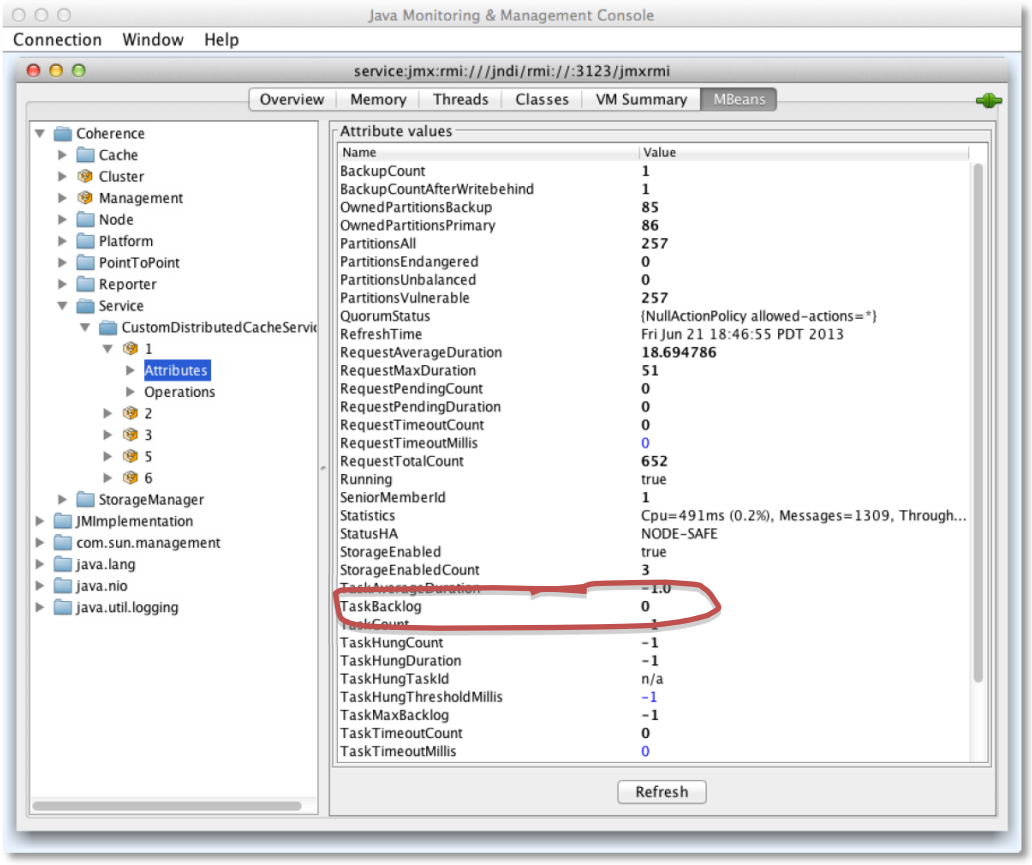


Figure 31. The `TaskBacklog` attribute for a service indicates if task are being queued before being processed

High CPU Utilization

Overloading a machine hosting Coherence may lead to unpredictable behavior and prevent recovery within the target SLAs in the event of a failure. Therefore, it is important to monitor CPU utilization. If overall CPU utilization is high this could be symptomatic of other processing being performed on a Coherence server, like a system backup, or that there are insufficient processing resources to meet the demands of the application. Alternatively, if load is high on just one or more cores, then a Coherence service can be overloaded and increasing the service `<thread-count>` may help – see the previous section. If you are using dynamic thread pool, then Coherence manages the thread pool as described above. Some suggested CPU utilization thresholds are >80% for a warning alert and >95% for a critical alert – or lower if you prefer to be more cautions. The following graphic shows CPU utilization from Enterprise Manager.

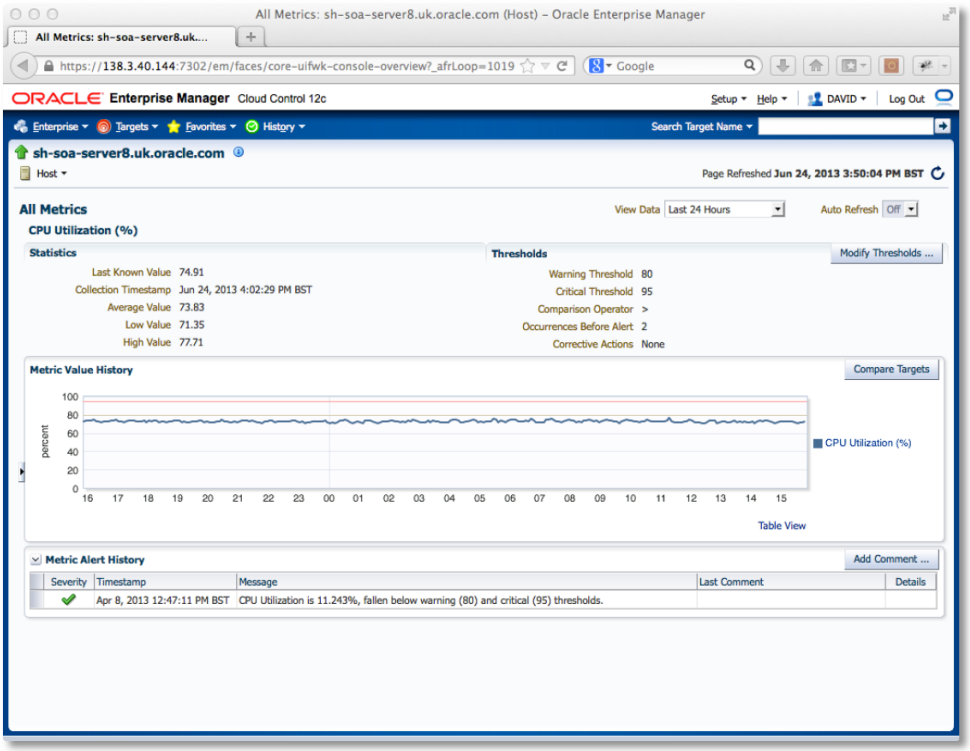


Figure 32. Oracle Cloud Control 12c can monitor and alert on host metrics, like CPU utilization.

Communication Issues

Coherence is a network centric application and sensitive to communication issues. The `PointToPoint` MBean can highlight these in its `PublisherSuccessRate` and `ReceiverSuccessRate` attributes. If either is <95% a warning alert should be raised, and if <90%, a critical alert should be raised. A possible cause could be incorrectly configured network equipment or operating system parameters; therefore, check the Production Checklist for configuration recommendations. The following graphic shows the `PublisherSuccessRate` and `RecieverSuccessRate` attributes.

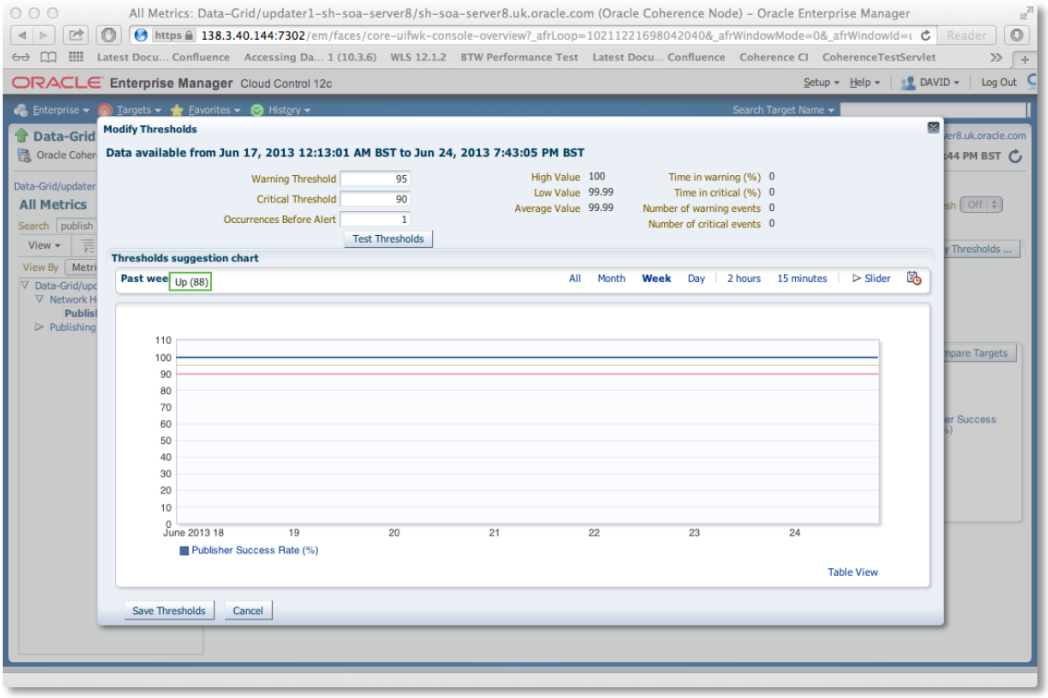


Figure 33. Thresholds can be set for both warning and critical alerts on the `PointToPoint` MBean `PublisherSuccessRate` and `RecieverSuccessRate` attributes in Oracle Cloud Control 12c

Network Saturation

The Network Interface Cards (NICs) being used by Coherence should be monitored. Some suggested thresholds for alerts on a 1GbE network are >60 MB/s for a warning and >90MB/s for a critical alert. Network saturation may not always be visible from the metrics gathered from a local NIC. For instance, if Coherence shares a switch or router with other applications that over-utilize the network, then the problem may not be visible from the local NICs.

Error Messages

Error and warning messages in log files should be monitored and appropriate alerts raised. These messages are listed in the Administrator's Guide and can be monitored using a number of tools, like Splunk, LogScape and Oracle Enterprise Manager. These tools can perform pattern matching to detect the occurrence of error and warning messages in log files.

Event Monitoring

Removing or updating a large number of cache entries can generate many events, especially if a large number of clients use a near cache invalidation strategy of ALL or are listening for changes in a large number of entries. To monitor the number of events, look at the `EventsDispatched` attribute on the `StorageManager` MBean and define alerts if the number of events over a collection interval exceeds appropriate thresholds. Note this value is the total number of events dispatched by a node since the statistic was last reset. To determine sensible thresholds it is usually necessary to monitor a

test environment first to see how an application behaves under normal conditions. The following graphic shows the `EventsDispatched` attribute.

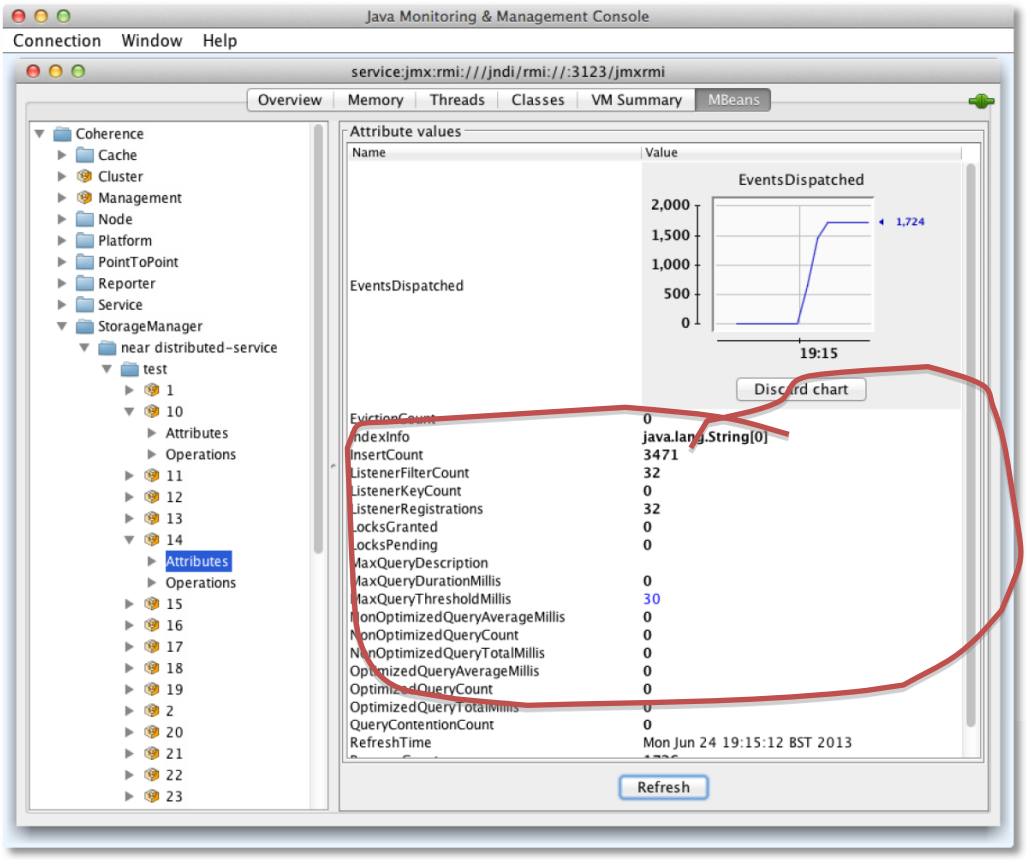


Figure 34. The `EventsDispatched` attribute on the `StorageManager` MBean shows the events since statistics were last reset

Long Response Times

When a Coherence API call takes longer than expected to return, this can be an indication of an underlying problem. For example, remote `get` calls on `NamedCaches` should generally return in less than one millisecond. `NamedCache.entrySet` calls should generally return in less than 10 milliseconds. If `entrySet` calls are taking longer than that and cache server hosts show high CPU utilization and frequent GC activity it is highly likely that a query is missing indexes. Long response times can also indicate a growing task backlog. Application code around API calls can measure and report response times

Logging

The Coherence and Java GC log files offer another source of monitoring information. Refer to the Coherence Administrator's Guide for Java GC logging parameters[12]. Setting the highest log level for Coherence (level 9) ensures the maximum amount of information is captured in the Coherence log files. This level should not create a lot of additional logging under normal conditions and provides invaluable information if problems do arise, making it easier and quicker to diagnose their cause. Since Log4J is one of the most popular Java frameworks for logging, the following guidance illustrates using its components. However, they equally apply to other logging frameworks and Coherence 12c also adds support for the Simple Logging Facade for Java (SLF4J).

A large Coherence cluster can generate a significant amount of logging information by virtue of the number of processes writing log files. Capturing and managing this effectively is critical to ensure that disks do not become full and application performance is not adversely impacted. Regularly rolling and archiving log files, using a Log4J *rolling file appender*, should prevent local disks from filling up, and a non-blocking Log4J *asynchronous appender* can discard messages in case it does. Care should be taken to tune both mechanisms to optimize the frequency with which log files are rolled, the duration for which archives are kept, and the number of messages that are buffered by a Log4J *asynchronous appender*.

If Log4J is being used, then the Coherence Log4J *logger* level needs to be set to "debug" and the logging limited by the specific Coherence log level in its override file. This is because the Coherence log levels are more fine grained than those in Log4J, for instance Log4j INFO is the same as Coherence log level 4 (INFO), but the Log4J DEBUG level includes Coherence log level 4-9.

Management

A major new feature introduced in Coherence 12c is "Managed Coherence Servers". Managed Coherence Servers allow applications to be deployed and managed inside WebLogic Server. Each Coherence application is packaged as a Grid Archive (GAR), much like a JEE application, and like JEE applications, WebLogic completely isolates each application using separate classloaders and uniquely identified cache services. Each Managed Coherence Server is a separate node in a Coherence cluster, with the WebLogic Administration server acting as the Management Node. Below is a diagram that shows a logical view of a Coherence deployment using Managed Coherence Servers.

[12] Java 6 Update 37 and Java 7 have introduced the ability to roll GC log files. For example: – `Xloggc:./gc.log -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=50M`. However, it is worth considering not splitting GC logging into its own file via `–Xloggc`. It's arguably valuable to let `–verbose:gc` output go right into the JVM's standard output stream, mixed in with other Coherence log messages. The reason for that is the contextual clues from other log messages can yield insight into what might be causing bursts of garbage generation. When GC logging is split into separate log files, you lose context and must rely on timestamp correlation between log files, which is imprecise if even possible.

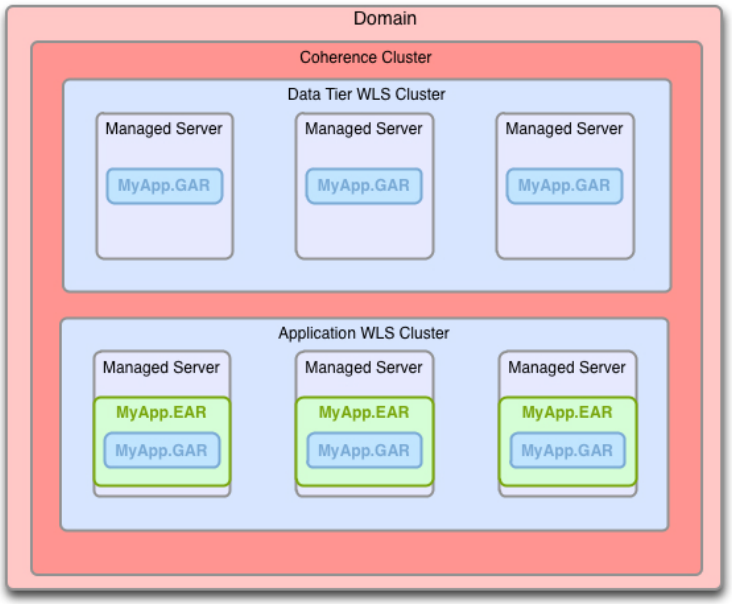


Figure 35. Logical architecture of Coherence Managed Servers in a WebLogic Domain

The overhead of running Coherence inside a WebLogic Managed server is minimal. For instance WebLogic only starts 20-30 pooled threads and only adds ~80MB to the JVM heap, or just 2% of a 4GB heap (Coherence standalone uses ~40MB).

Although optional, some of the key benefits of Managed Coherence Servers are:

- A centralized, mature, and proven Management Console for controlling the complete lifecycle of Coherence applications.
- A rich scripting framework, WLST, to setup, start-up, deploy and manage Coherence clusters and applications, and simplifying operations.
- The ability to clone a Managed Coherence Server
- More intuitive bulk management capabilities. Managed Coherence Servers can be grouped into WebLogic Clusters (as distinct from Coherence clusters) to manage them as a group. Typical groupings could be client applications, proxy servers, or storage nodes. This enables operations to be performed at a group level (or WebLogic Cluster level). For example:
    - Deploying or re-deploying a Coherence GAR
    - Making configuration changes, like increasing the high units of a near cache
    - Deploying a shared library

To effectively manage your Coherence clusters its also good practice to give each cluster a meaningful cluster name (like "PROD_APPC_CLUSTER") and explicitly label all the other facets, like role and so on. This makes management and monitoring much easier and prevents a QA or test cluster from inadvertently joining a production cluster.

Production Testing

Before you start, make sure that your environment is setup according to the recommendations outlined in the documentation. This includes:

- Production Checklist (Coherence Administrator's Guide)
- Performance Tuning (Coherence Administrator's Guide)
- Platform Specific Deployment Considerations (Coherence Administrator's Guide)
- Best Practice For Coherence*Extend (Coherence Client Guide)

Also, use the bundled datagram and multi-cast utilities to validate many of the above recommendations.

This section focuses on non-functional testing. Functional unit and end-to-end testing is not covered here.

Soak Testing

It is often overlooked, but running your application tests for a prolonged period – hours and even days – usually highlights problems that short tests do not detect. For instance, memory leaks often only manifest themselves after an application has been running for some time. Inadequate processes to handle log files or capacity-planning errors can also show up in soak tests.

Load and Stress Testing

Before deploying a Coherence application into production, you should simulate (and if possible exceed) the load and stress that it needs to support. These tests should also include simulated failures in the environment to ensure there is sufficient spare processing, network, and memory capacity to handle these and still meet your target SLAs.

Hardware Failures

Test that your hardware provides the desired level of resilience in the production environment. For instance, under load remove a network cable; fill up a local disk; shut down a server; etc... When simulating these failures, monitor application correctness, throughput, latency, and recovery time (MTR), to ensure you still meet your target SLAs.

Software Failures

As with hardware failure testing, under load try shutting down a Cache Server, the Management or Administration Server (if you are using Managed Coherence Servers) and so on to ensure your application meets your target SLAs. Also, try disabling connections to any external resources, like a database or even a DNS service. It is a best practice to configure appropriate short timeouts connecting to all such external resources (e.g. JDBC connection and statement timeouts), and code your application to detect and handle the timeout condition. This is preferable to service threads blocking indefinitely on unavailable external resources and causing task backlogs to grow.

Recovery

As well as starting your application for the first time, you may need to restart it later after maintenance or because of an unplanned outage. Ensure that you can recover if necessary from a complete system failure – and in a reasonable period – in case you need to. This requirement often places a premium on cache warming execution time. In Coherence 12cR2, the snapshot recovery behavior of the persistence feature can help address this issue.

Resolving Problems

Make sure the operational staff is familiar with your application infrastructure, monitoring dashboards, and alerts. If issues arise which seem related to Coherence, then a Service Request (SR) ticket should be raised with Oracle Support. If your operations team is not familiar with this process, it is worth raising a test SR beforehand, to make sure they are ready to do so if needed.

The typical information requested by an engineer looking into a Coherence issue is:

- The Coherence configuration files
- Thread dumps and possibly heap dumps. In Coherence 12c one of the bundled WLST scripts for performing these operations can be used – if Managed Coherence Servers are being used. To perform a remote thread dump, another option is to use the new Coherence 12c `logClusterState(roleName)` operation on the `Cluster` MBean and the `logNodeSate(nodeId)` operations on the `Node` MBean.

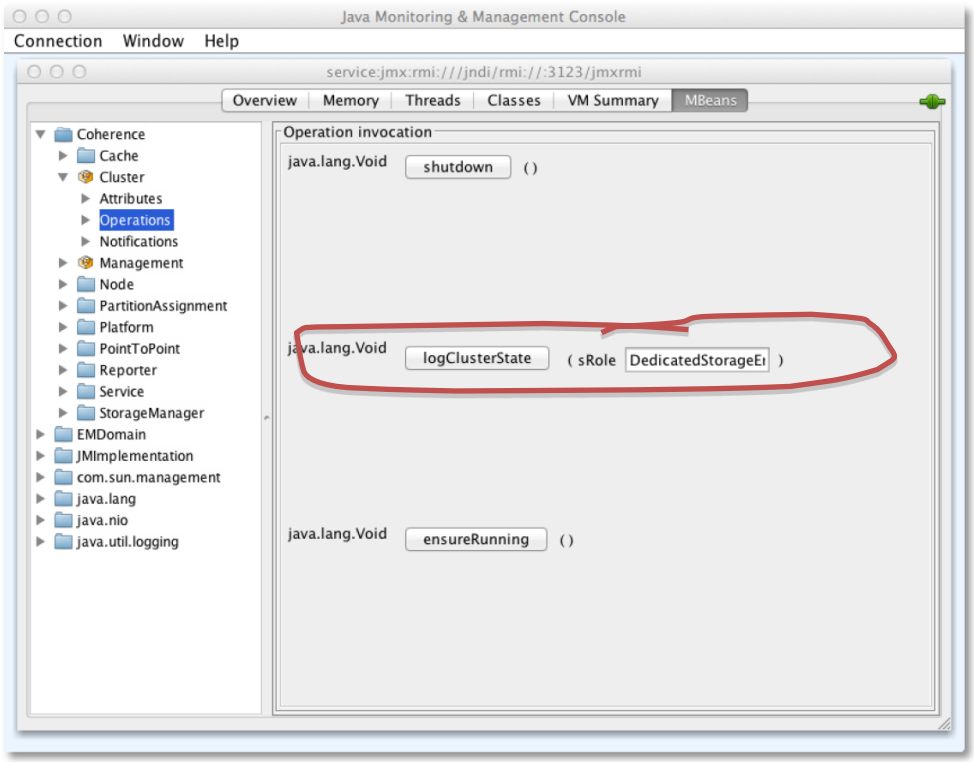


Figure 36. In Coherence 12c a thread dump of cluster nodes can be performed through JMX.

- The Coherence log files. Make sure your application logging goes to a different file from the Coherence logging. This makes the Coherence log files easier to read and share with Oracle Support, because they will not contain any confidential information. Also try not to change the Coherence log file format, so that it's easier for support engineers to examine.
- JVM startup information and parameters
- JMX Reporter files, if you are using it. The JMX reporter is a great way to capture historical JMX metrics.
- JVM GC logging information. This can be in the Coherence log files or in a separate GC log file.
- Environment metrics, like CPU and network statistics. A simple tool called OSWatcher is freely available from Oracle Support that contains standard operating system scripts to capture this kind of information in log files.

Make sure that you can easily capture this information, so any issues can be promptly investigated.

Conclusion

Following these guidelines, recommendations, and suggestions will help you have a more successful Coherence deployment. However, read the Coherence documentation first and use this white paper as an additional checklist or as input to your planning process. The formulas should give you a feel for your resource requirements and if you can meet your application SLAs. However, always validate the formulas through testing. Lastly, try the new installation and management features in Coherence 12c; they should make the setup and configuration of Coherence easier.

Coherence 12c – Planning a Successful Deployment
July 2013
Author: David Felcey
Contributing Authors: Craig Blitz, Tim Middleton, Jason Howes, Mark Falco, Harvey Raja, Randy Stafford, Jon Purdy and Patrick Peralta

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com