Oracle® Coherence

Getting Started Guide

Release 3.6

E15724-01

July 2010

Provides conceptual information about Coherence distributed caching technology.

ORACLE®

Oracle Coherence Getting Started Guide, Release 3.6

E15724-01

Primary Author: Joseph Ruzzi

Contents

6 Cluster Your Objects and Data

7 Cluster Services Overview

8 Partitioned Cache Service

9 Replicated Cache Service

10 Local Cache

11 Near Cache

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

List of Examples

List of Figures

List of Tables

Preface

This book provides conceptual information on the caching technology behind Oracle Coherence. It describes the various types of caches that can be employed, caching strategies, and the features of clients that interact with caching services.

Audience

This document is intended for software developers who want to become familiar with the concepts behind Oracle Coherence caching technology.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/support/contact.html` or visit `http://www.oracle.com/accessibility/support.html` if you are hearing impaired.

Related Documents

For more information, see the following documents that are included in the Oracle Coherence documentation set:

- *Oracle Coherence Client Guide*
- *Oracle Coherence Developer's Guide*
- *Oracle Coherence Integration Guide for Oracle Coherence*
- *Oracle Coherence Tutorial for Oracle Coherence*
- *Oracle Coherence User's Guide for Oracle Coherence*Web*
- *Oracle Coherence Java API Reference*
- *Oracle Coherence C++ API Reference*
- *Oracle Coherence .NET API Reference*
- *Oracle Coherence Release Notes for Oracle Coherence*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Defining a Data Grid

The Oracle Coherence **In-Memory Data Grid** is a data management system for application objects that are shared across multiple servers, require low response time, very high throughput, predictable scalability, continuous availability and information reliability. For clarity, each of these terms and claims is explained:

As a result of these capabilities, Oracle Coherence is ideally suited for use in computational intensive, stateful middle-tier applications. Coherence is targeted to run in the application tier, and is often run in-process with the application itself, for example in an Application Server Cluster.

The combination of these four capabilities results is the information within the Data Grid being reliable for use by transactional applications.

- A **Data Grid** is a system composed of multiple servers that work together to manage information and related operations - such as computations - in a distributed environment.
- An **In-Memory Data Grid** is a Data Grid that stores the information *in memory* to achieve very high performance, and uses redundancy—by keeping copies of that information synchronized across multiple servers—to ensure the resiliency of the system and the availability of the data in the event of server failure.
- The **application objects** are the actual components of the application that contain the information shared across multiple servers. These objects must survive a possible server failure in order for the application to be continuously available. These objects are typically built in an object-oriented language such as Java (for example, POJOs), C++, C#, or VB.NET. Unlike a relational schema, the application objects are often hierarchical and may contain information that is pulled from any database.
- The application objects must be **shared across multiple servers** because a middleware application (such as eBay and Amazon.com) is horizontally scaled by adding servers - each server running an instance of the application. Since the application instance running on one server may read and write some of the same information as an application instance on another server, the information must be shared. The alternative is to always access that information from a shared resource, such as a database, but this will lower performance by requiring both remote coordinated access and Object/Relational Mapping (ORM), and decrease scalability by making that shared resource a bottleneck.
- Because an application object is not relational, to retrieve it from a relational database the information must be mapped from a relational query into the object. This is known as **Object/Relational Mapping (ORM)**. Examples of ORM include Java EJB 3.0, JPA, and ADO.NET. The same ORM technology allows the object to be stored in a relational database by deconstructing the object (or changes to the

object) into a series of SQL inserts, updates and deletes. Since a single object may be composed of information from many tables, the cost of accessing objects from a database using Object/Relational Mapping can be significant, both in terms of the load on the database and the latency of the data access.

- An In-Memory Data Grid achieves **low response times** for data access by keeping the information in-memory and in the application object form, and by sharing that information across multiple servers. In other words, applications may be able to access the information that they require without any network communication and without any data transformation step such as ORM. In cases where network communication is required, the Oracle Coherence avoids introducing a Single Point of Bottleneck (SPOB) by partitioning—spreading out—information across the grid, with each server being responsible for managing its own fair share of the total set of information.
- **High throughput** of information access and change is achieved through four different aspects of the In-Memory Data Grid:
  - Oracle Coherence employs a sophisticated clustering protocol that can achieve wire speed throughput of information on each server. This allows the aggregate flow of information to increase linearly with the number of servers.
  - By partitioning the information, as servers are added each one assumes responsibility for its fair share of the total set of information, thus load-balancing the data management responsibilities into smaller and smaller portions.
  - By combining the wire speed throughput and the partitioning with automatic knowledge of the location of information within the Data Grid, Oracle Coherence routes all read and write requests directly to the servers that manage the targeted information, resulting in true linear scalability of both read and write operations; in other words, high throughput of information access and change.
  - For queries, transactions and calculations, particularly those that operate against large sets of data, Oracle Coherence can route those operations to the servers that manage the target data and execute them in parallel.
- By using dynamic partitioning to eliminate bottlenecks and achieving predictably low latency regardless of the number of servers in the Data Grid, Oracle Coherence provides **predictable scalability** of applications. While certain applications can use Coherence to achieve linear scalability, that is largely determined by the nature of the application, and thus varies from application to application. More important is the ability of a customer to examine the nature of their application and to be able to predict how many servers will be required to achieve a certain level of scale, such as supporting a specified number of concurrent users on a system or completing a complex financial calculation within a certain number of minutes. One way that Coherence accomplishes this is by executing large-scale operations, such as queries, transactions and calculations, in parallel using all of the servers in the Data Grid.
- One of the ways that Coherence can **eliminate bottlenecks** is to queue up transactions that have occurred in memory and asynchronously write the result to a system of record, such as an Oracle database. This is particularly appropriate in systems that have extremely high rates of change due to the processing of many small transactions, particularly when only the result must be made persistent. Coherence both coalesces multiple changes to a single application object and batches multiple modified application objects into a single database transaction, meaning that a hundred different changes to each of a hundred different

application objects could be persisted to a database in a single, large—and thus highly efficient—transaction. Application objects pending to be written are safeguarded from loss by being managed in a continuously available manner.

- **Continuous availability** is achieved by a combination of four capabilities.
  - First, the clustering protocol used by Oracle Coherence can rapidly detect server failure and achieve consensus across all the surviving servers about the detected failure.
  - Second, information is synchronously replicated across multiple servers, so no Single Point of Failure (SPOF) exists.
  - Third, each server knows where the synchronous replicas of each piece of information are located, and automatically re-routes information access and change operations to those replicas.
  - Fourth, Oracle Coherence ensures that each operation executes in a Once-and-Only-Once manner, so that operations that are being executed when a server fails do not accidentally corrupt information during failover.
- **Failover** is the process of switching over automatically to a redundant or standby computer server, system, or network upon the failure or abnormal termination of the previously active server, system, or network. Failover happens without human intervention and generally without warning. (As defined by Wikipedia: http://en.wikipedia.org/wiki/Failover)
- **Information reliability** is achieved through a combination of four capabilities.
  - Oracle Coherence uses cluster consensus to achieve unambiguous ownership of information within the Data Grid. At all times, exactly one server is responsible for managing the master copy of each piece of information in the Data Grid.
  - Because that master copy is owned by a specific server, that server can order the operations that are occurring to that information and synchronize the results of those operations with other servers.
  - Because the information is continuously available, these qualities of service exist even during and after the failure of a server.
  - By ensuring Once-and-Only-Once operations, no operations are lost or accidentally repeated when server failure does occur.

2

Provide a Data Grid

Coherence provides the ideal infrastructure for building Data Grid services, and the client and server-based applications that use a Data Grid. At a basic level, Coherence can manage an immense amount of data across a large number of servers in a grid; it can provide close to zero latency access for that data; it supports parallel queries across that data; and it supports integration with database and EIS systems that act as the system of record for that data. Additionally, Coherence provides several services that are ideal for building effective data grids.

For more information on the infrastructure for the Data Grid features in Coherence, see Chapter 3, "Provide a Queryable Data Fabric".

Note: All of the Data Grid capabilities described in the following sections are features of Coherence Enterprise Edition and higher.

Targeted Execution

Coherence provides for the ability to execute an agent against an entry in any map of data managed by the Data Grid:

```
map.invoke(key, agent);
```

In the case of partitioned data, the agent executes on the grid node that owns the data to execute against. This means that the queuing, concurrency management, agent execution, data access by the agent and data modification by the agent all occur on that grid node. (Only the synchronous backup of the resultant data modification, if any, requires additional network traffic.) For many processing purposes, it is much more efficient to move the serialized form of the agent (usually only a few hundred bytes, at most) than to handle distributed concurrency control, coherency and data updates.

For request/response processing, the agent returns a result:

```
Object oResult = map.invoke(key, agent);
```

In other words, Coherence as a Data Grid will determine the location to execute the agent based on the configuration for the data topology, move the agent there, execute the agent (automatically handling concurrency control for the item while executing the agent), back up the modifications if any, and return a result.

Parallel Execution

Coherence additionally provides for the ability to execute an agent against an entire collection of entries. In a partitioned Data Grid, the execution occurs in parallel,

meaning that the more nodes that are in the grid, the broader the work is load-balanced across the Data Grid:

```
map.invokeAll(collectionKeys, agent);
```

For request/response processing, the agent returns one result for each key processed:

```
Map mapResults = map.invokeAll(collectionKeys, agent);
```

In other words, Coherence determines the optimal location(s) to execute the agent based on the configuration for the data topology, moves the agent there, executes the agent (automatically handling concurrency control for the item(s) while executing the agent), backing up the modifications if any, and returning the coalesced results.

Query-Based Execution

As discussed in Chapter 3, "Provide a Queryable Data Fabric", Coherence supports the ability to query across the entire data grid. For example, in a trading system it is possible to query for all open `Order` objects for a particular trader:

Example 2–1 Querying Across a Data Grid

```
NamedCache map    = CacheFactory.getCache("trades");
Filter     filter = new AndFilter(new EqualsFilter("getTrader", traderid),
                                  new EqualsFilter("getStatus", Status.OPEN));
Set setOpenTradeIds = mapTrades.keySet(filter);
```

By combining this feature with Parallel Execution in the data grid, Coherence provides for the ability to execute an agent against a query. As in the previous section, the execution occurs in parallel, and instead of returning the identities or entries that match the query, Coherence executes the agents against the entries:

```
map.invokeAll(filter, agent);
```

For request/response processing, the agent returns one result for each key processed:

```
Map mapResults = map.invokeAll(filter, agent);
```

In other words, Coherence combines its Parallel Query and its Parallel Execution together to achieve query-based agent invocation against a Data Grid.

Data-Grid-Wide Execution

Passing an instance of `AlwaysFilter` (or a null) to the `invokeAll` method will cause the passed agent to be executed against all entries in the `InvocableMap`:

```
map.invokeAll((Filter) null, agent);
```

As with the other types of agent invocation, request/response processing is supported:

```
Map mapResults = map.invokeAll((Filter) null, agent);
```

An application can process all the data spread across a particular map in the Data Grid with a single line of code.

Agents for Targeted, Parallel and Query-Based Execution

An agent implements the `EntryProcessor` interface, typically by extending the `AbstractProcessor` class.

Several agents are included with Coherence, including:

- `AbstractProcessor` - an abstract base class for building an `EntryProcessor`
- `ExtractorProcessor` - extracts and returns a specific value (such as a property value) from an object stored in an `InvocableMap`
- `CompositeProcessor` - bundles together a collection of `EntryProcessor` objects that are invoked sequentially against the same Entry
- `ConditionalProcessor` - conditionally invokes an `EntryProcessor` if a `Filter` against the Entry-to-process evaluates to true
- `PropertyProcessor` - an abstract base class for `EntryProcessor` implementations that depend on a `PropertyManipulator`
- `NumberIncrementor` - pre- or post-increments any property of a primitive integral type, and `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `BigInteger`, `BigDecimal`
- `NumberMultiplier` - multiplies any property of a primitive integral type, and `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `BigInteger`, `BigDecimal`, and returns either the previous or new value

The `EntryProcessor` interface (contained within the `InvocableMap` interface) contains only two methods:

Example 2–2 Methods in the EntryProcessor Interface

```
/**
* An invocable agent that operates against the Entry objects within a
* Map.
*/
public interface EntryProcessor
        extends Serializable
    {
    /**
    * Process a Map Entry.
    *
    * @param entry  the Entry to process
    *
    * @return the result of the processing, if any
    */
    public Object process(Entry entry);

    /**
    * Process a Set of InvocableMap Entry objects. This method is
    * semantically equivalent to:
    * <pre>
    *   Map mapResults = new ListMap();
    *   for (Iterator iter = setEntries.iterator(); iter.hasNext(); )
    *       {
    *       Entry entry = (Entry) iter.next();
    *       mapResults.put(entry.getKey(), process(entry));
    *       }
    *   return mapResults;
    * </pre>
    *
```

```
    * @param setEntries  a read-only Set of InvocableMap Entry objects to
    *                    process
    *
    * @return a Map containing the results of the processing, up to one
    *         entry for each InvocableMap Entry that was processed, keyed
    *         by the keys of the Map that were processed, with a
    *         corresponding value being the result of the processing for
    *         each key
    */
    public Map processAll(Set setEntries);
    }
```

(The `AbstractProcessor` implements the `processAll` method as described in the previous example.)

The `InvocableMap.Entry` that is passed to an `EntryProcessor` is an extension of the `Map.Entry` interface that allows an `EntryProcessor` implementation to obtain the necessary information about the entry and to make the necessary modifications in the most efficient manner possible:

Example 2–3 InvocableMap.Entry API

```
/**
* An InvocableMap Entry contains additional information and exposes
* additional operations that the basic Map Entry does not. It allows
* non-existent entries to be represented, thus allowing their optional
* creation. It allows existent entries to be removed from the Map. It
* supports several optimizations that can ultimately be mapped
* through to indexes and other data structures of the underlying Map.
*/
public interface Entry
        extends Map.Entry
    {
    // ----- Map Entry interface ------------------------------------

    /**
    * Return the key corresponding to this entry. The resultant key does
    * not necessarily exist within the containing Map, which is to say
    * that <tt>InvocableMap.this.containsKey(getKey)</tt> could return
    * false. To test for the presence of this key within the Map, use
    * {@link #isPresent}, and to create the entry for the key, use
    * {@link #setValue}.
     *
    * @return the key corresponding to this entry; may be null if the
    *         underlying Map supports null keys
    */
    public Object getKey();

    /**
    * Return the value corresponding to this entry. If the entry does
    * not exist, then the value will be null. To differentiate between
    * a null value and a non-existent entry, use {@link #isPresent}.
    * <p/>
    * <b>Note:</b> any modifications to the value retrieved using this
    * method are not guaranteed to persist unless followed by a
    * {@link #setValue} or {@link #update} call.
    *
    * @return the value corresponding to this entry; may be null if the
    *         value is null or if the Entry does not exist in the Map
    */
```

```
public Object getValue();

/**
* Store the value corresponding to this entry. If the entry does
* not exist, then the entry will be created by invoking this method,
* even with a null value (assuming the Map supports null values).
*
* @param oValue  the new value for this Entry
*
* @return the previous value of this Entry, or null if the Entry did
*         not exist
*/
public Object setValue(Object oValue);

// ----- InvocableMap Entry interface ---------------------------

/**
* Store the value corresponding to this entry. If the entry does
* not exist, then the entry will be created by invoking this method,
* even with a null value (assuming the Map supports null values).
* <p/>
* Unlike the other form of {@link #setValue(Object) setValue}, this
* form does not return the previous value, and consequently may be
* significantly less expensive (in terms of cost of execution) for
* certain Map implementations.
*
* @param oValue      the new value for this Entry
* @param fSynthetic  pass true only if the insertion into or
*                    modification of the Map should be treated as a
*                    synthetic event
*/
public void setValue(Object oValue, boolean fSynthetic);

/**
* Extract a value out of the Entry's value. Calling this method is
* semantically equivalent to
* <tt>extractor.extract(entry.getValue())</tt>, but this method may
* be significantly less expensive because the resultant value may be
* obtained from a forward index, for example.
*
* @param extractor  a ValueExtractor to apply to the Entry's value
*
* @return the extracted value
*/
public Object extract(ValueExtractor extractor);

/**
* Update the Entry's value. Calling this method is semantically
* equivalent to:
* <pre>
*   Object oTarget = entry.getValue();
*   updater.update(oTarget, oValue);
*   entry.setValue(oTarget, false);
* </pre>
* The benefit of using this method is that it may allow the Entry
* implementation to significantly optimize the operation, such as
* for purposes of delta updates and backup maintenance.
*
* @param updater  a ValueUpdater used to modify the Entry's value
*/
```

```
    public void update(ValueUpdater updater, Object oValue);

    /**
    * Determine if this Entry exists in the Map. If the Entry is not
    * present, it can be created by calling {@link #setValue} or
    * {@link #setValue}. If the Entry is present, it can be destroyed by
    * calling {@link #remove}.
    *
    * @return true iff this Entry is existent in the containing Map
    */
    public boolean isPresent();

    /**
    * Remove this Entry from the Map if it is present in the Map.
    * <p/>
    * This method supports both the operation corresponding to
    * {@link Map#remove} and synthetic operations such as
    * eviction. If the containing Map does not differentiate between
    * the two, then this method will always be identical to
    * <tt>InvocableMap.this.remove(getKey())</tt>.
    *
    * @param fSynthetic  pass true only if the removal from the Map
    *                    should be treated as a synthetic event
    */
    public void remove(boolean fSynthetic);
    }
```

Data Grid Aggregation

While the agent discussion in the previous section corresponds to *scalar* agents, the `InvocableMap` interface also supports aggregation:

Example 2–4 Aggregation in the InvocableMap API

```
/**
* Perform an aggregating operation against the entries specified by the
* passed keys.
*
* @param collKeys  the Collection of keys that specify the entries within
*                  this Map to aggregate across
* @param agent     the EntryAggregator that is used to aggregate across
*                  the specified entries of this Map
*
* @return the result of the aggregation
*/
public Object aggregate(Collection collKeys, EntryAggregator agent);

/**
* Perform an aggregating operation against the set of entries that are
* selected by the given Filter.
* <p/>
* <b>Note:</b> calling this method on partitioned caches requires a
* Coherence Enterprise Edition (or higher) license.
*
* @param filter  the Filter that is used to select entries within this
*                Map to aggregate across
* @param agent   the EntryAggregator that is used to aggregate across
*                the selected entries of this Map
*
```

```
* @return the result of the aggregation
*/
public Object aggregate(Filter filter, EntryAggregator agent);
```

A simple `EntryAggregator` processes a set of `InvocableMap.Entry` objects to achieve a result:

Example 2–5 EntryAggregator API

```
/**
* An EntryAggregator represents processing that can be directed to occur
* against some subset of the entries in an InvocableMap, resulting in a
* aggregated result. Common examples of aggregation include functions
* such as min(), max() and avg(). However, the concept of aggregation
* applies to any process that must evaluate a group of entries to
* come up with a single answer.
*/
public interface EntryAggregator
        extends Serializable
    {
    /**
    * Process a set of InvocableMap Entry objects to produce an
    * aggregated result.
    *
    * @param setEntries  a Set of read-only InvocableMap Entry objects to
    *                    aggregate
    *
    * @return the aggregated result from processing the entries
    */
    public Object aggregate(Set setEntries);
    }
```

For efficient execution in a Data Grid, an aggregation process must be designed to operate in a parallel manner.

Example 2–6 ParallelAwareAggregator API for running Aggregation in Parallel

```
/**
* A ParallelAwareAggregator is an advanced extension to EntryAggregator
* that is explicitly capable of being run in parallel, for example in a
* distributed environment.
*/
public interface ParallelAwareAggregator
        extends EntryAggregator
    {
    /**
    * Get an aggregator that can take the place of this aggregator in
    * situations in which the InvocableMap can aggregate in parallel.
    *
    * @return the aggregator that will be run in parallel
    */
    public EntryAggregator getParallelAggregator();

    /**
    * Aggregate the results of the parallel aggregations.
    *
    * @return the aggregation of the parallel aggregation results
    */
    public Object aggregateResults(Collection collResults);
    }
```

Coherence comes with all of the natural aggregation functions, including:

- `Count`
- `DistinctValues`
- `DoubleAverage`
- `DoubleMax`
- `DoubleMin`
- `DoubleSum`
- `LongMax`
- `LongMin`
- `LongSum`

> **Note:** All aggregators that come with Coherence are parallel-aware.

See the `com.tangosol.util.aggregator` package for a list of Coherence aggregators. To implement your own aggregator, see the `AbstractAggregator` abstract base class.

Node-Based Execution

Coherence provides an Invocation Service which allows execution of single-pass agents (called Invocable objects) anywhere within the grid. The agents can be executed on any particular node of the grid, in parallel on any particular set of nodes in the grid, or in parallel on all nodes of the grid.

An invocation service is configured using the `<invocation-scheme>` element in the cache configuration file. Using the name of the service, the application can easily obtain a reference to the service:

```
InvocationService service = CacheFactory.getInvocationService("agents");
```

Agents are simply runnable classes that are part of the application. An example of a simple agent is one designed to request a GC from the JVM:

Example 2–7 Simple Agent to Request Garbage Collection

```
/**
* Agent that issues a garbage collection.
*/
public class GCAgent
        extends AbstractInvocable
    {
    public void run()
        {
        System.gc();
        }
    }
```

To execute that agent across the entire cluster, it takes one line of code:

```
service.execute(new GCAgent(), null, null);
```

Here is an example of an agent that supports a grid-wide request/response model:

Example 2–8 Agent to Support a Grid-Wide Request and Response Model

```
/**
* Agent that determines how much free memory a grid node has.
*/
public class FreeMemAgent
        extends AbstractInvocable
    {
    public void run()
        {
        Runtime runtime = Runtime.getRuntime();
        int cbFree  = runtime.freeMemory();
        int cbTotal = runtime.totalMemory();
        setResult(new int[] {cbFree, cbTotal});
        }
    }
```

To execute that agent across the entire grid and retrieve all the results from it, **it still takes only one line of code**:

```
Map map = service.query(new FreeMemAgent(), null);
```

While it is easy to do a grid-wide request/response, it takes a bit more code to print out the results:

Example 2–9 Printing the Results from a Grid-Wide Request or Response

```
Iterator iter = map.entrySet().iterator();
while (iter.hasNext())
    {
    Map.Entry entry  = (Map.Entry) iter.next();
    Member    member = (Member) entry.getKey();
    int[]     anInfo = (int[]) entry.getValue();
    if (anInfo != null) // nullif member died
        System.out.println("Member " + member + " has "
            + anInfo[0] + " bytes free out of "
            + anInfo[1] + " bytes total");
    }
```

The agent operations can be stateful, which means that their invocation state is serialized and transmitted to the grid nodes on which the agent is to be run.

Example 2–10 Stateful Agent Operations

```
/**
* Agent that carries some state with it.
*/
public class StatefulAgent
        extends AbstractInvocable
    {
    public StatefulAgent(String sKey)
        {
        m_sKey = sKey;
        }

    public void run()
        {
        // the agent has the key that it was constructed with
        String sKey = m_sKey;
```

```
        // ...
        }

    private String m_sKey;
    }
```

Work Manager

Coherence provides a grid-enabled implementation of the *CommonJ Work Manager*. Using a Work Manager, an application can submit a collection of work that must be executed. The Work Manager distributes that work in such a way that it is executed in parallel, typically across the grid. In other words, if there are ten work items submitted and ten servers in the grid, then each server will likely process one work item. Further, the distribution of work items across the grid can be tailored, so that certain servers (for example, one that acts as a gateway to a particular mainframe service) will be the first choice to run certain work items, for sake of efficiency and locality of data.

The application can then wait for the work to be completed, and can provide a timeout for how long it is willing to wait. The API for this purpose is quite powerful, allowing an application to wait for the first work item to complete, or for a specified set of the work items to complete. By combining methods from this API, it is possible to do things like "Here are 10 items to execute; for these 7 unimportant items, wait no more than 5 seconds, and for these 3 important items, wait no more than 30 seconds".

Example 2–11 Using a Work Manager

```
Work[] aWork = ...
Collection collBigItems = new ArrayList();
Collection collAllItems = new ArrayList();
for (int i = 0, c = aWork.length; i < c; ++i)
    {
    WorkItem item = manager.schedule(aWork[i]);

    if (i < 3)
        {
        // the first three work items are the important ones
        collBigItems.add(item);
        }

    collAllItems.add(item);
    }

Collection collDone = manager.waitForAll(collAllItems, 5000L);
if (!collDone.containsAll(collBigItems))
    {
    // wait the remainder of 30 seconds for the important work to finish
    manager.waitForAll(collBigItems, 25000L);
    }
```

Oracle Coherence Work Manager: Feedback from a Major Financial Institution

Our primary use case for the Work Manager is to allow our application to serve coarse-grained service requests using our blade infrastructure in a standards-based way. We often have what appears to be a simple request, like "give me this family's information." In reality, however, this request expands into a large number of requests to several diverse back-end data sources consisting of web services, RDMBS calls, and so on. This use case expands into two different but related problems that we are looking to the distributed version of the work manager to solve.

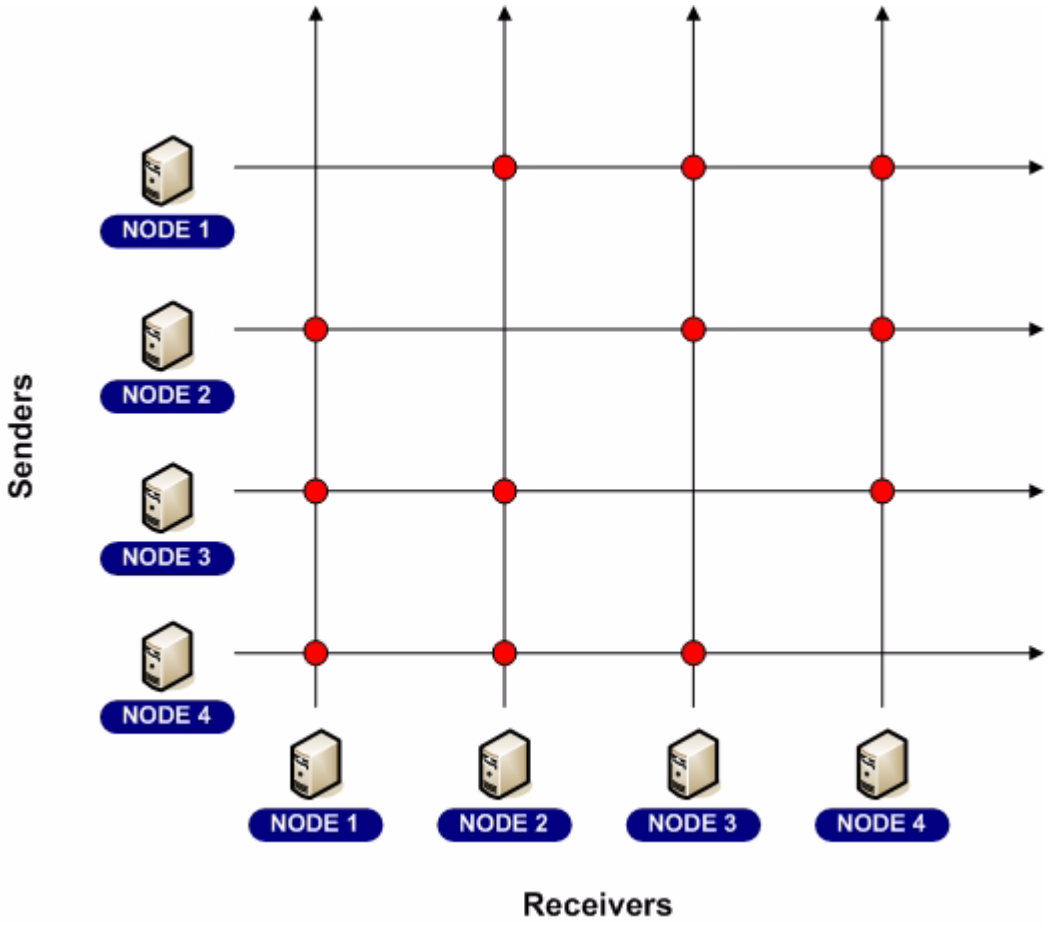
- How do we take a coarse-grained request that expands into several fine-grained requests and execute them in parallel to avoid blocking the caller for an unreasonable time? In the previous example, we may have to make upwards of 100 calls to various places to retrieve the information. Since Java EE has no legal threading model, and since the threading we observed when trying a message-based approach to this was unacceptable, we decided to use the Coherence Work Manager implementation.
- Given that we want to make many external system calls in parallel while still leveraging low-cost blades, we are hoping that fanning the required work across many dual processor (logically 4-processor because of hyperthreading) machines allows us to scale an inherently vertical scalability problem with horizontal scalability at the hardware level. We think this is reasonable because the cost to marshall the request to a remote Work Manager instance is small compared to the cost to execute the service, which usually involves dozens or hundreds of milliseconds.

3

Provide a Queryable Data Fabric

Oracle invented the concept of a data fabric with the introduction of the Coherence partitioned data management service in 2002. Since then, Forrester Research has labeled the combination of data virtualization, transparent and distributed EIS integration, queryability and uniform accessibility found in Coherence as an *information fabric*. The term *fabric* comes from a 2-dimensional illustration of interconnects, as in a *switched fabric*. The purpose of a fabric architecture is that all points within a fabric have a direct interconnect with all other points.

Figure 3–1 Data fabric illustrating which senders and receivers are connected



Data Fabric

An information fabric, or the more simple form called a *data fabric* or *data grid*, uses a switched fabric concept as the basis for managing data in a distributed environment. Also referred to as a *dynamic mesh architecture*, Coherence automatically and dynamically forms a reliable, increasingly resilient switched fabric composed of any number of servers within a grid environment. Consider the attributes and benefits of this architecture:

- The aggregate data throughput of the fabric is linearly proportional to the number of servers;
- The in-memory data capacity and data-indexing capacity of the fabric is linearly proportional to the number of servers;
- The aggregate I/O throughput for disk-based overflow and disk-based storage of data is linearly proportional to the number of servers;
- The resiliency of the fabric increases with the extent of the fabric, resulting in each server being responsible for only 1/n of the failover responsibility for a fabric with an extent of n servers;
- If the fabric is servicing clients, such as trading systems, the aggregate maximum number of clients that can be served is linearly proportional to the number of servers.

Coherence accomplishes these technical feats through a variety of algorithms:

- Coherence dynamically partitions data across all data fabric nodes;
- Since each data fabric node has a configurable maximum amount of data that it will manage, the capacity of the data fabric is linearly proportional to the number of data fabric nodes;
- Since the partitioning is automatic and load-balancing, each data fabric node ends up with its fair share of the data management responsibilities, allowing the throughput (in terms of network throughput, disk I/O throughput, query throughput, and so on) to scale linearly with the number of data fabric nodes;
- Coherence maintains a configurable level of redundancy of data, automatically eliminating single points of failure (SPOFs) by ensuring that data is kept synchronously up-to-date in multiple data fabric nodes;
- Coherence spreads out the responsibility for data redundancy in a dynamically load-balanced manner so that each server backs up a small amount of data from many other servers, instead of backing up all of the data from one particular server, thus amortizing the impact of a server failure across the entire data fabric;
- Each data fabric node can handle a large number of client connections, which can be load-balanced by a hardware load balancer.

EIS and Database Integration

The Coherence information fabric can automatically load data on demand from an underlying database or EIS using automatic read-through functionality. If data in the fabric are modified, the same functionality allows that data to be synchronously updated in the database, or queued for asynchronous write-behind. For more information on read-through and write behind functionality, see Chapter 12, "Read-Through, Write-Through, Write-Behind, and Refresh-Ahead Caching".

Coherence automatically partitions data access across the data fabric, resulting in load-balanced data accesses and efficient use of database and EIS connectivity.

Furthermore, the read-ahead and write-behind capabilities can cut data access latencies to near-zero levels and insulate the application from temporary database and EIS failures.

Note: Coherence solves the data bottleneck for large-scale compute grids.

In large-scale compute grids, such as in DataSynapse financial grids and biotech grids, the bottleneck for most compute processes is in loading a data set and making it available to the compute engines that require it. By layering a Coherence data fabric onto (or beside) a compute grid, these data sets can be maintained in memory at all times, and **Coherence can feed the data in parallel at close to wire speed to all of the compute nodes**. In a large-scale deployment, Coherence can provide several thousand times the aggregate data throughput of the underlying data source.

Queryable

The Coherence information fabric supports querying from any server in the fabric or any client of the fabric. The queries can be performed using any criteria, including custom criteria such as XPath queries and full text searches. When Coherence partitioning is used to manage the data, **the query is processed in parallel across the entire fabric** (that is, the query is also partitioned), resulting in a data query engine that can scale its throughput up to fabrics of thousands of servers. For example, in a trading system it is possible to query for all open `Order` objects for a particular trader:

Example 3–1 Querying the Cache for a Particular Object

```
NamedCache mapTrades = ...
Filter filter = new AndFilter(new EqualsFilter("getTrader", traderid),
                              new EqualsFilter("getStatus", Status.OPEN));
Set setOpenTrades = mapTrades.entrySet(filter);
```

When an application queries for data from the fabric, the result is a point-in-time snapshot. Additionally, the query results can be kept up-to-date by placing a query on the listener itself or by using the Coherence Continuous Query feature.

Continuous Query

While it is possible to obtain a point in time query result from a Coherence data fabric, and it is possible to receive events that would change the result of that query, Coherence provides a feature that combines a query result with a continuous stream of related events that maintain the query result in a real-time fashion. This capability is called *continuous query* because it has the same effect as if the desired query had zero latency *and* the query were repeated several times every millisecond. See

Coherence implements Continuous Query using a combination of its *data fabric parallel query* capability and its *real-time event-filtering and streaming*. The result is support for thousands of client application instances, such as trading desktops. Using the previous trading system example, it can be converted to a Continuous Query with only one a single line of code changed:

Example 3–2 Implementing a Continuous Query

```
NamedCache mapTrades = ...
```

```
Filter filter = new AndFilter(new EqualsFilter("getTrader", traderid),
                              new EqualsFilter("getStatus", Status.OPEN));
NamedCache mapOpenTrades = new ContinuousQueryCache(mapTrades, filter);
```

The result of the Continuous Query is maintained locally, and optionally all of corresponding data can be cached locally as well.

4

Clustering in Coherence

Coherence is built on a fully *clustered* architecture. Since "clustered" is an overused term in the industry, it is worth stating exactly what it means to say that Coherence is *clustered*. Coherence is based on a peer-to-peer clustering protocol, using a conference room model, in which servers are capable of:

- **Speaking to Everyone:** When a party enters the conference room, it is able to speak to all other parties in a conference room.
- **Listening:** Each party present in the conference room can hear messages that are intended for *everyone*, and messages that are intended for that particular party. It is also possible that a message is not heard the first time, thus a message may need to be repeated until it is heard by its intended recipients.
- **Discovery:** Parties can only communicate by speaking and listening; there are no other senses. Using only these means, the parties must determine exactly who is in the conference room at any given time, and parties must detect when new parties enter the conference room.
- **Working Groups and Private Conversations:** Although a party can talk to everyone, after a party is introduced to the other parties in the conference room (that is, after discovery has completed), the party can communicate directly to any set of parties, or directly to an individual party.
- **Death Detection:** Parties in the conference room must quickly detect when parties leave the conference room - or die.

Using the conference room model provides the following benefits:

- There is no configuration required to add members to a cluster. Subject to configurable security restrictions, any JVM running Coherence will automatically join the cluster and be able to access the caches and other services provided by the cluster. This includes Java EE application servers, Cache Servers, dedicated cache loader processes, or any other JVM that is running with the Coherence software. When a JVM joins the cluster, it is called a *cluster node*, or alternatively, a *cluster member*.
- Since all cluster members are known, it is possible to provide redundancy within the cluster, such that the death of any one JVM or server machine does not cause any data to be lost.
- Since the death or departure of a cluster member is automatically and quickly detected, failover occurs very rapidly, and more importantly, it occurs transparently, which means that the application does not have to do any extra work to handle failover.
- Since all cluster members are known, it is possible to load balance responsibilities across the cluster. Coherence does this automatically with its Distributed Cache

Service, for example. Load balancing automatically occurs to respond to new members joining the cluster, or existing members leaving the cluster.

- Communication can be very well optimized, since some communication is multi-point in nature (for example, messages for everyone), and some is between two members.

Two of the terms used here describe processing for failed servers:

- **Failover:** Failover refers to the ability of a server to assume the responsibilities of a failed server. For example, "When the server died, its processes failed over to the backup server."
- **Failback:** Failback is an extension to failover that allows a server to reclaim its responsibilities after it restarts. For example, "When the server came back up, the processes that it was running previously were failed back to it."

All of the Coherence clustered services, including cache services and grid services, provide automatic and transparent failover and failback. While these features are transparent to the application, it should be noted that the application can sign up for events to be notified of **all** *comings and goings* in the cluster.

5

Network Protocols

This chapter describes the network protocols supported by Coherence.

Coherence and the TCMP Protocol

Coherence uses TCMP, a clustered IP-based protocol, for server discovery, cluster management, service provisioning and data transmission. To ensure true scalability, the TCMP protocol is completely asynchronous, meaning that communication is never blocking, even when many threads on a server are communicating at the same time. Further, the asynchronous nature also means that the latency of the network (for example, on a routed network between two different sites) does not affect cluster *throughput*, although it will affect the speed of certain operations.

TCMP uses a combination of UDP/IP multicast, UDP/IP unicast and TCP/IP as follows:

- **Multicast**
  - Cluster discovery: Is there a cluster already running that a new member can join?
  - Cluster heartbeat: The most senior member in the cluster issues a periodic heartbeat through multi-cast; the rate can be configured and defaults to once per second.
  - Message delivery: Messages that need to be delivered to multiple cluster members will often be sent through multicast, instead of unicasting the message one time to each member.
- **Unicast**
  - Direct member-to-member ("point-to-point") communication, including messages, asynchronous acknowledgments (ACKs), asynchronous negative acknowledgments (NACKs) and peer-to-peer heartbeats.
  - Under some circumstances, a message may be sent through unicast even if the message is directed to multiple members. This is done to shape traffic flow and to reduce CPU load in very large clusters.
- **TCP**
  - An optional TCP/IP ring is used as an additional "death detection" mechanism, to differentiate between actual node failure and an unresponsive node, such as when a JVM conducts a full GC.
  - TCP/IP is **not** used as a data transfer mechanism due to the intrinsic overhead of the protocol and its synchronous nature.

Protocol Reliability

The TCMP protocol provides fully reliable, in-order delivery of all messages. Since the underlying UDP/IP protocol does not provide for either reliable or in-order delivery, TCMP uses a queued, fully asynchronous ACK- and NACK-based mechanism for reliable delivery of messages, with unique integral identity for guaranteed ordering of messages.

Protocol Resource Utilization

The TCMP protocol requires only two UDP/IP sockets (one multicast, one unicast) and six threads per JVM, regardless of the cluster size. This is a key element in the scalability of Coherence, in that regardless of the number of servers, each node in the cluster can still communicate either point-to-point or with collections of cluster members without requiring additional network connections.

The optional TCP/IP ring will use a few additional TCP/IP sockets, and a total of one additional thread.

Protocol Tunability

The TCMP protocol is very tunable to take advantage of specific network topologies, or to add tolerance for low-bandwidth and/or high-latency segments in a geographically distributed cluster. Coherence comes with a pre-set configuration. Some TCMP attributes are dynamically self-configuring at runtime, but can also be overridden and locked down for deployment purposes.

Multicast Scope

Multicast UDP/IP packets are configured with a time-to-live value (TTL) that designates how far those packets can travel on a network. The TTL is expressed in terms of how many "hops" a packet will survive; each network interface, router and managed switch is considered one hop. Coherence provides a TTL setting to limit the scope of multicast messages.

Disabling Multicast

In most WAN environments, and some LAN environments, multicast traffic is disallowed. To prevent Coherence from using multicast, configure a list of `well-known-addresses` (WKA). This will disable multicast discovery, and also disable multicast for all data transfer. Coherence is designed to use point-to-point communication as much as possible, so most application profiles will not see a substantial performance impact.

6

Cluster Your Objects and Data

Coherence is an essential ingredient for building reliable, high-scale clustered applications. The term **clustering** refers to the use of more than one server to run an application, usually for reliability and scalability purposes. Coherence provides all of the necessary capabilities for applications to achieve the maximum possible availability, reliability, scalability and performance. Virtually any clustered application will benefit from using Coherence.

Coherence and Clustered Data

One of the primary uses of Coherence is to **cluster an application's objects and data**. In the simplest sense, this means that all of the objects and data that an application delegates to Coherence are automatically available to and accessible by all servers in the application cluster. None of the objects or data will be lost in the event of server failure.

By clustering the application's objects and data, Coherence solves many of the difficult problems related to achieving availability, reliability, scalability, performance, serviceability and manageability of clustered applications.

Availability

Availability refers to the percentage of time that an application is operating. High Availability refers to achieving availability close to 100%. Coherence is used to achieve High Availability in several different ways:

Supporting Redundancy in Java Applications

Coherence makes it possible for an application to run on more than one server, which means that the servers are *redundant*. Using a load balancer, for example, an application running on redundant servers will be available if one server is still operating. Coherence enables redundancy by allowing an application to share, coordinate access to, update and receive modification events for critical runtime information across all of the redundant servers. Most applications cannot operate in a redundant server environment unless there architecture is designed to run in such an environment; Coherence is a key enabler of such an architecture.

Enabling Dynamic Cluster Membership

Coherence tracks exactly what servers are available at any given moment. When the application is started on an additional server, Coherence is instantly aware of that server coming online, and automatically joins it into the cluster. This allows redundancy (and thus availability) to be dynamically increased by adding servers.

Exposing Knowledge of Server Failure

Coherence reliably detects most types of server failure in less than a second, and immediately fails over all of the responsibilities of the failed server without losing any data. Consequently, server failure does not impact availability.

Part of an availability management is *Mean Time To Recovery* (MTTR), which is a measurement of how much time it takes for an unavailable application to become available. Since server failure is detected and handled in less than a second, and since redundancy means that the application is available even when that server goes down, the MTTR due to server failure is zero from the point of view of application availability, and typically sub-second from the point of view of a load-balancer re-routing an incoming request.

Eliminating Other Single Points Of Failure (SPOFs)

Coherence provides insulation against failures in other infrastructure tiers. For example, Coherence write-behind caching and Coherence distributed parallel queries can insulate an application from a database failure; in fact, using these capabilities, two different Coherence customers have had database failure during operational hours, yet their production Coherence-based applications maintained their availability and their operational status.

Providing Support for Disaster Recovery (DR) and Contingency Planning

Coherence can even insulate against failure of an entire data center, by clustering across multiple data centers and failing over the responsibilities of an entire data center. Again, this capability has been proven in production, with a Coherence customer running a mission-critical real-time financial system surviving a complete data center outage.

Reliability

Reliability refers to the percentage of time that an application is able to process *correctly*. In other words, an application may be available, yet unreliable if it cannot correctly handle the application processing. An example that we use to illustrate high availability but low reliability is a *mobile phone network*: While most mobile phone networks have very high uptimes (referring to *availability*), dropped calls tend to be relatively common (referring to *reliability*).

Coherence is explicitly build to achieve very high levels of reliability. For example, server failure does not impact "in flight" operations, since each operation is atomically protected from server failure, and will internally re-route to a secondary node based on a dynamic pre-planned recovery strategy. In other words, every operation has a backup plan ready to go!

Coherence is designed based on the assumption that *failures are always about to occur*. Consequently, the algorithms employed by Coherence are carefully designed to assume that each step within an operation could fail due to a network, server, operating system, JVM or other resource outage. An example of how Coherence plans for these failures is the synchronous manner in which it maintains redundant copies of data; in other words, Coherence does not gamble with the application's data, and that ensures that the application will continue to work correctly, even during periods of server failure.

Scalability

Scalability refers to the ability of an application to predictably handle more load. An application exhibits linear scalability if the maximum amount of load that an application can sustain is directly proportional to the hardware resources that the application is running on. For example, if an application running on 2 servers can handle 2000 requests per second, then linear scalability would imply that 10 servers would handle 10000 requests per second.

Linear scalability is the *goal* of a scalable architecture, but it is difficult to achieve. The measurement of how well an application scales is called the *scaling factor* (SF). A scaling factor of 1.0 represents linear scalability, while a scaling factor of 0.0 represents no scalability. Coherence provides several capabilities designed to help applications achieve linear scalability.

When planning for extreme scale, the first thing to understand is that **application scalability is limited by any necessary shared resource that does not exhibit linear scalability**. The limiting element is referred to as a *bottleneck*, and in most applications, the bottleneck is the data source, such as a database or an EIS.

Coherence helps to solve the scalability problem by targeting obvious bottlenecks, and by completely eliminating bottlenecks whenever possible. It accomplishes this through a variety of capabilities, including:

Distributed Caching

Coherence uses a combination of replication, distribution, partitioning and invalidation to reliably maintain data in a cluster in such a way that **regardless of which server is processing, the data that it obtains from Coherence is the same**. In other words, Coherence provides a *distributed shared memory* implementation, also referred to as *Single System Image* (SSI) and Coherent Clustered Caching.

Any time that an application can obtain the data it needs from the application tier, it is eliminating the data source as the Single Point Of Bottleneck (SPOB).

Partitioning

Partitioning refers to the ability for Coherence to load-balance data storage, access and management across all of the servers in the cluster. For example, when using Coherence data partitioning, if there are four servers in a cluster then each will manage 25% of the data, and if another server is added, each server will dynamically adjust so that each of the five servers will manage 20% of the data, and this data load balancing will occur without any application interruption and without any lost data or operations. Similarly, if one of those five servers were to die, each of the remaining four servers would be managing 25% of the data, and this data load balancing will occur without any application interruption and without any lost data or operations - including the 20% of the data that was being managed on the failed server.

Coherence accomplishes failover without data loss by synchronously maintaining a configured number of copies of the data within the cluster. Just as the data management responsibility is spread out over the cluster, so is the responsibility for backing up data, so in the previous example, each of the remaining four servers would have roughly 25% of the failed server's data backed up on it. This *mesh architecture* guarantees that on server failure, no particular remaining server is inundated with a massive amount of additional responsibility.

Coherence prevents loss of data even when multiple instances of the application run on a single physical server within the cluster. It does so by ensuring that backup copies of data are being managed on different physical servers, so that if a physical

server fails or is disconnected, all of the data being managed by the failed server has backups ready to go on a different server. This assumes that the cluster has been provisioned for reliability and high availability (typically at least 4 physical servers with the number of JVMs roughly equivalent on all physical servers). See "Does it matter how JVMs are distributed among servers?" in the *Oracle Coherence Developer's Guide* for more information on setting up Coherence for high availability.

Lastly, **partitioning supports linear scalability of both data capacity and throughput**. It accomplishes the scalability of data capacity by evenly balancing the data across all servers, so four servers can naturally manage two times as much data as two servers. Scalability of throughput is also a direct result of load-balancing the data across all servers, since as servers are added, each server is able to use its full processing power to manage a smaller and smaller percentage of the overall data set. For example, in a ten-server cluster each server has to manage 10% of the data operations, and - since Coherence uses a peer-to-peer architecture - 10% of those operations are coming *from* each server. With ten times that many servers (that is, 100 servers), each server is managing only 1% of the data operations, and only 1% of those operations are coming from each server - but there are ten times as many servers, so the cluster is accomplishing ten times the total number of operations! In the 10-server example, if each of the ten servers was issuing 100 operations per second, they would each be sending 10 of those operations to each of the other servers, and the result would be that each server was receiving 100 operations (10x10) that it was responsible for processing. In the 100-server example, each would still be issuing 100 operations per second, but each would be sending only one operation to each of the other servers, so the result would be that each server was receiving 100 operations (100x1) that it was responsible for processing. This linear scalability is made possible by modern switched network architectures that provide backplanes that scale linearly to the number of ports on the switch, providing each port with dedicated fully-duplexed (upstream and downstream) bandwidth. Since each server is only sending and receiving 100 operations (in both the 10-server and 100-server examples), the **network bandwidth utilization is roughly constant per port regardless of the number of servers in the cluster**.

Session Management

One common use case for Coherence clustering is to manage user sessions (*conversational state*) in the cluster. This capability is provided by the Coherence*Web module, which is a built-in feature of Coherence. Coherence*Web provides linear scalability for HTTP Session Management in clusters of hundreds of production servers. It can achieve this linear scalability because at its core it is built on Coherence dynamic partitioning.

Session management highlights the scalability problem that typifies shared data sources: If an application could not share data across the servers, it would have to delegate that data management entirely to the shared store, which is typically the application's database. If the HTTP session were stored in the database, each HTTP request (in the absence of sticky load-balancing) would require a read from the database, causing the desired reads-per-second from the database to increase linearly with the size of the server cluster. Further, each HTTP request causes an update of its corresponding HTTP session, so regardless of sticky load balancing, to ensure that HTTP session data is not lost when a server fails the desired writes-per-second to the database will also increase linearly with the size of the server cluster. In both cases, the actual reads and writes per second that a database is capable of, does not scale in relation to the number of servers requesting those reads and writes, and the database quickly becomes a bottleneck, forcing availability, reliability (for example, asynchronous writes) and performance compromises. Additionally, related to

performance, each read from a database has an associated latency, and that latency increases dramatically as the database experiences increasing load.

Coherence*Web, however, has the same latency in a 2-server cluster as it has in a 200-server cluster, since all HTTP session read operations that cannot be handled locally (for example, locality as the result of the sticky load balancing) are spread out evenly across the rest of the cluster, and all update operations (which must be handled remotely to ensure survival of the HTTP sessions) are likewise spread out evenly across the rest of the cluster. The result is linear scalability with constant latency, regardless of the size of the cluster.

Performance

Performance is the inverse of latency, and latency is the measurement of how long something takes to complete. If increasing performance is the goal, then getting rid of anything that has any latency is the solution. Obviously, it is impossible to get rid of all latencies, since the High Availability and reliability aspects of an application are counting on the underlying infrastructure, such as Coherence, to maintain reliable up-to-date back-ups of important information, which means that some operations (such as data modifications and pessimistic transactions) have unavoidable latencies. However, every remaining operation that could possibly have any latency must be targeted for elimination, and Coherence provides a large number of capabilities designed to do just that.

Replication

Just as *partitioning* dynamically load-balances data evenly across the entire server cluster, *replication* ensures that a desired set of data is up-to-date on every single server in the cluster at all times. Replication allows operations running on any server to obtain the data that they need locally, at basically no cost, because that data has already been replicated to that server. In other words, **replication is a tool to guarantee locality of reference**, and the result is zero-latency access to replicated data.

Near Caching

Since replication works best for data that *should* be on all servers, it follows that replication is inefficient for data that an application would want to avoid copying to all servers. For example, data that changes all of the time and very large data sets are both poorly suited to replication, but both are excellently suited to partitioning, since it exhibits linear scale of data capacity and throughput.

The only downside of partitioning is that it introduces latency for data access, and in most applications the data access rate far out-weighs the data modification rate. To eliminate the latency associated with partitioned data access, *near caching* maintains frequently- and recently-used data from the partitioned cache on the specific servers that are accessing that data, and it keeps that data coherent with event-based invalidation. In other words, **near caching keeps the most-likely-to-be-needed data near to where it will be used**, thus providing good locality of access, yet backed up by the linear scalability of partitioning.

Write-Behind, Write-Coalescing and Write-Batching

Since the transactional throughput in the cluster is linearly scalable, the cost associated with data changes can be a fixed latency, typically in the range of a few milliseconds, and the total number of transactions per second is limited only by the size of the cluster. In one application, Coherence was able to achieve transaction rates close to a

half-million transactions per second - and that on a cluster of commodity two-CPU servers.

Often, the data being managed by Coherence is actually a temporary copy of data that exists in an official *System Of Record* (SOR), such as a database. To avoid having the database become a transaction bottleneck, and to eliminate the latency of database updates, Coherence provides a *Write-Behind* capability, which allows the application to change data in the cluster, and those changes are asynchronously replayed to the application's database (or EIS). By managing the changes in a clustered cache (which has all of the High Availability, reliability and scalability attributes described previously,) the pending changes are immune to server failure and the total rate of changes scales linearly with the size of the cluster.

The Write-Behind functionality is implemented by queuing each data change; the queue contains a list of what changes must be written to the System Of Record. The duration of an item within the queue can be configured, and is referred to as the *Write-Behind Delay*. When data changes, it is added to the write-behind queue (if it is not already in the queue), and the queue entry is set to *ripen* after the configured *Write-Behind Delay* has passed. When the queue entry has ripened, the latest copy of the corresponding data is written to the System Of Record.

To avoid overwhelming the System Of Record, Coherence will replay only the latest copies of data to the database, thus coalescing many updates that occur to the same piece data into a single database operation. The longer the Write-Behind Delay, the more coalescing may occur. Additionally, if many different pieces of data have changed, all of those updates can be batched (for example, using JDBC statement batching) into a single database operation. In this way, a massive breadth of changes (number of pieces of data changed) and depth of changes (number of times each was changed) can be bundled into a single database operation, which results in dramatically reduced load on the database. The batching can also be fully configured; one option, called the *Write Batch Factor*, even allows some of the queue entries that have not yet ripened to be included in the batched update.

Serviceability

Serviceability refers to the ease and extent of changes that can be affected without affecting availability. Coherence helps to increase an application's serviceability by allowing servers to be taken off-line without impacting the application availability. Those servers can be serviced and brought back online without any end-user or processing interruptions. Many configuration changes related to Coherence can also be made on a node-by-node basis in the same manner. With careful planning, even major application changes can be rolled into production—again, one node at a time—without interrupting the application.

Manageability

Manageability refers to the level of information that a running system provides, and the capability to tweak settings related to that information. For example, Coherence provides a cluster wide view of management information through the standard JMX API, so that the entire cluster can be managed from a single server. The information provided includes hit and miss rates, cache sizes, read-, write- and write-behind statistics, and detailed information all the way down to the network packet level.

Additionally, Coherence allows applications to place their own management information—and expose their own configuration settings—through the same clustered JMX implementation. The result is an application infrastructure that makes

managing and monitoring a clustered application as simple as managing and monitoring a single server, and all through Java's standard management API.

7

Cluster Services Overview

Coherence functionality is based on the concept of cluster services. Each cluster node can participate in (which implies both the ability to *provide* and to *consume*) any number of named services. These named services may already exist, which is to say that they may already be running on one or more other cluster nodes, or a cluster node can register new named services. Each named service has a service name that uniquely identifies the service within the cluster, and a service type, which defines what the service can do. There may be multiple named instances of each service type (other than the root Cluster service). By way of analogy, a service instance corresponds roughly to a database schema, and in the case of data services, a hosted NamedCache corresponds roughly to a database table. While services can be fully configured, many applications will only need to use the default set of services shipped with Coherence. There are several service types that are supported by Coherence.

Connectivity Services

- **Cluster Service:** This service is automatically started when a cluster node must join the cluster; each cluster node always has exactly one service of this type running. This service is responsible for the detection of other cluster nodes, for detecting the failure (death) of a cluster node, and for registering the availability of other services in the cluster. In other words, the Cluster Service keeps track of the membership and services in the cluster.
- **Proxy Service:** While many applications are configured so that all clients are also cluster members, there are use cases where it is desirable to have clients running outside the cluster, especially in cases where there will be hundreds or thousands of client processes, where the clients are not running on the Java platform, or where a greater degree of coupling is desired. This service allows connections (using TCP) from clients that run outside the cluster.

Processing Services

- **Invocation Service:** This service provides clustered invocation and supports grid computing architectures. Using the Invocation Service, application code can invoke agents on any node in the cluster, or any group of nodes, or across the entire cluster. The agent invocations can be request/response, fire and forget, or an asynchronous user-definable model.

Data Services

- **Distributed Cache Service:** This is the distributed cache service, which allows cluster nodes to distribute (partition) data across the cluster so that each piece of data in the cache is managed (held) by only one cluster node. The Distributed Cache Service supports pessimistic locking. Additionally, to support failover without any data loss, the service can be configured so that each piece of data will

be backed up by one or more other cluster nodes. Lastly, some cluster nodes can be configured to hold no data at all; this is useful, for example, to limit the Java heap size of an application server process, by setting the application server processes to not hold any distributed data, and by running additional cache server JVMs to provide the distributed cache storage.

- **Replicated Cache Service:** This is the synchronized replicated cache service, which fully replicates all of its data to all cluster nodes that run the service. Furthermore, it supports pessimistic locking so that data can be modified in a cluster without encountering the classic missing update problem. With the introduction of near caching and continuous query caching, almost all of the functionality of replicated caches is available on top of the Distributed cache service (and with better robustness). But replicated caches are often used to manage internal application metadata.
- **Optimistic Cache Service:** This is the optimistic-concurrency version of the Replicated Cache Service, which fully replicates all of its data to all cluster nodes, and employs an optimization similar to optimistic database locking to maintain coherency. Coherency refers to the fact that all servers will end up with the same "current" value, even if multiple updates occur at the same exact time from different servers. The Optimistic Cache Service does not support pessimistic locking, so in general it should only be used for caching "most recently known" values for read-only uses. This service is rarely used.

Regarding resources, a clustered service typically uses one daemon thread, and optionally has a thread pool that can be configured to provide the service with additional processing bandwidth. For example, the invocation service and the distributed cache service both fully support thread pooling to accelerate database load operations, parallel distributed queries, and agent invocations.

It is important to note that these are only the basic clustered services, and not the full set of types of caches provided by Coherence. By combining clustered services with cache features such as backing maps and overflow maps, Coherence can provide an extremely flexible, easily configured and powerful set of options for clustered applications. For example, the Near Cache functionality uses a Distributed Cache as one of its components.

Within a cache service, there exists any number of named caches. A named cache provides the standard JCache API, which is based on the Java collections API for key-value pairs, known as `java.util.Map`. The `Map` interface is the same API that is implemented by the Java `Hashtable` class, for example.

8

Partitioned Cache Service

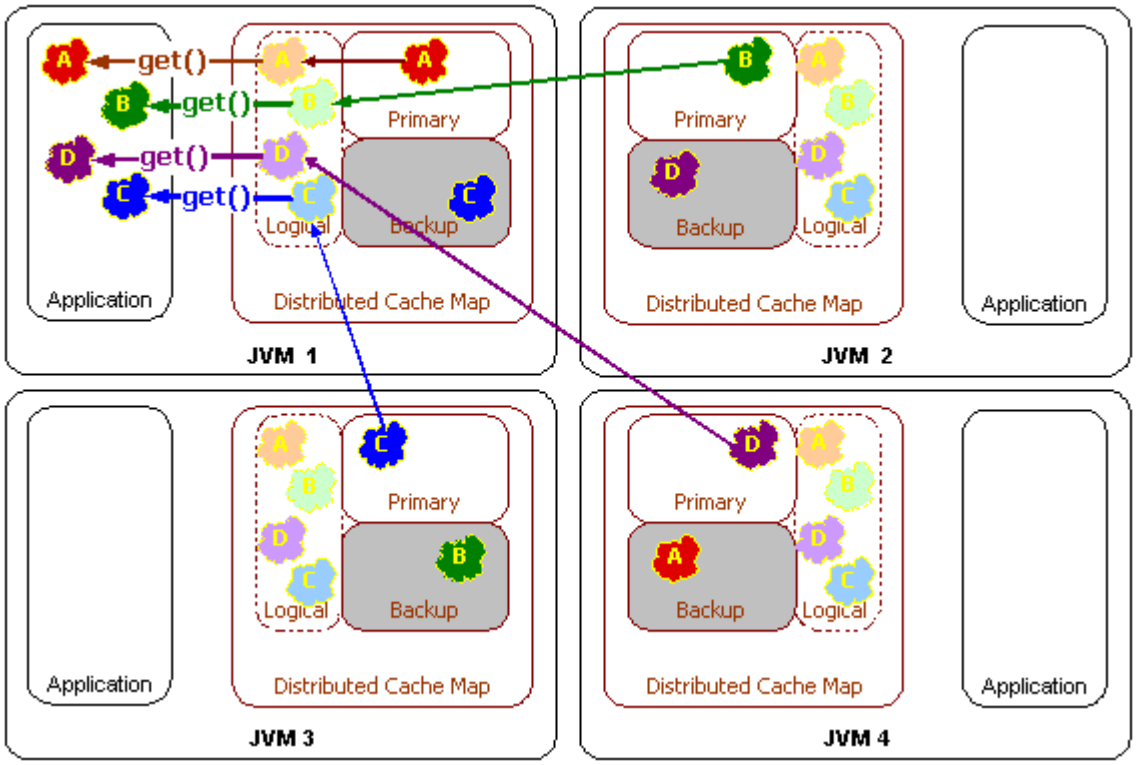
To address the potential scalability limits of the replicated cache service, both in terms of memory and communication bottlenecks, Coherence has provided a distributed cache service since release 1.2. Many products have used the term distributed cache to describe their functionality, so it is worth clarifying exactly what is meant by that term in Coherence. Coherence defines a distributed cache as a collection of data that is distributed (or, *partitioned*) across any number of cluster nodes such that exactly one node in the cluster is responsible for each piece of data in the cache, and the responsibility is distributed (or, load-balanced) among the cluster nodes.

There are several key points to consider about a distributed cache:

- **Partitioned:** The data in a distributed cache is spread out over all the servers in such a way that no two servers are responsible for the same piece of cached data. This means that the size of the cache and the processing power associated with the management of the cache can grow linearly with the size of the cluster. Also, it means that operations against data in the cache can be accomplished with a "single hop," in other words, involving at most one other server.
- **Load-Balanced:** Since the data is spread out evenly over the servers, the responsibility for managing the data is automatically load-balanced across the cluster.
- **Location Transparency:** Although the data is spread out across cluster nodes, the exact same API is used to access the data, and the same behavior is provided by each of the API methods. This is called location transparency, which means that the developer does not have to code based on the topology of the cache, since the API and its behavior will be the same with a local JCache, a replicated cache, or a distributed cache.
- **Failover:** All Coherence services provide failover and failback without any data loss, and that includes the distributed cache service. The distributed cache service allows the number of backups to be configured; if the number of backups is one or higher, any cluster node can fail without the loss of data.

Access to the distributed cache will often need to go over the network to another cluster node. All other things equals, if there are n cluster nodes, (n - 1) / n operations will go over the network:

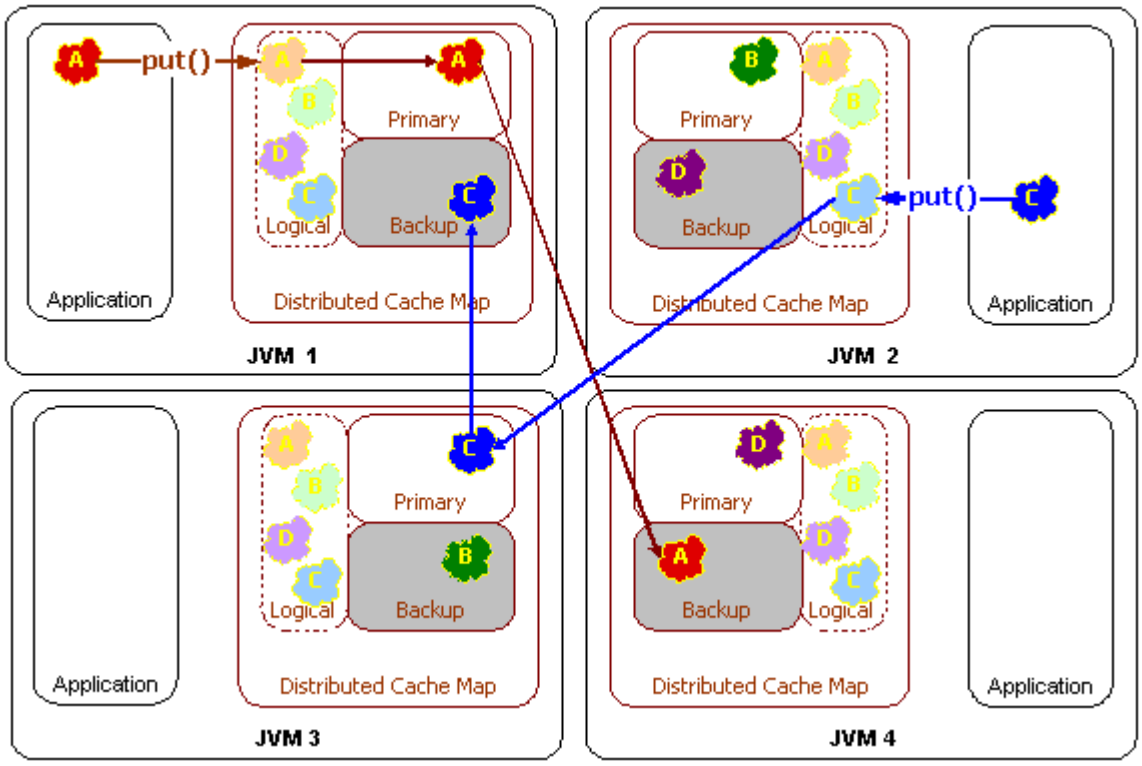
Figure 8–1 Get Operations in a Partitioned Cache Environment



Since each piece of data is managed by only one cluster node, an access over the network is only a "single hop" operation. This type of access is extremely scalable, since it can use point-to-point communication and thus take optimal advantage of a switched network.

Similarly, a cache update operation can use the same single-hop point-to-point approach, which addresses one of the two known limitations of a replicated cache, the need to push cache updates to all cluster nodes.

Figure 8–2 Put Operations in a Partitioned Cache Environment

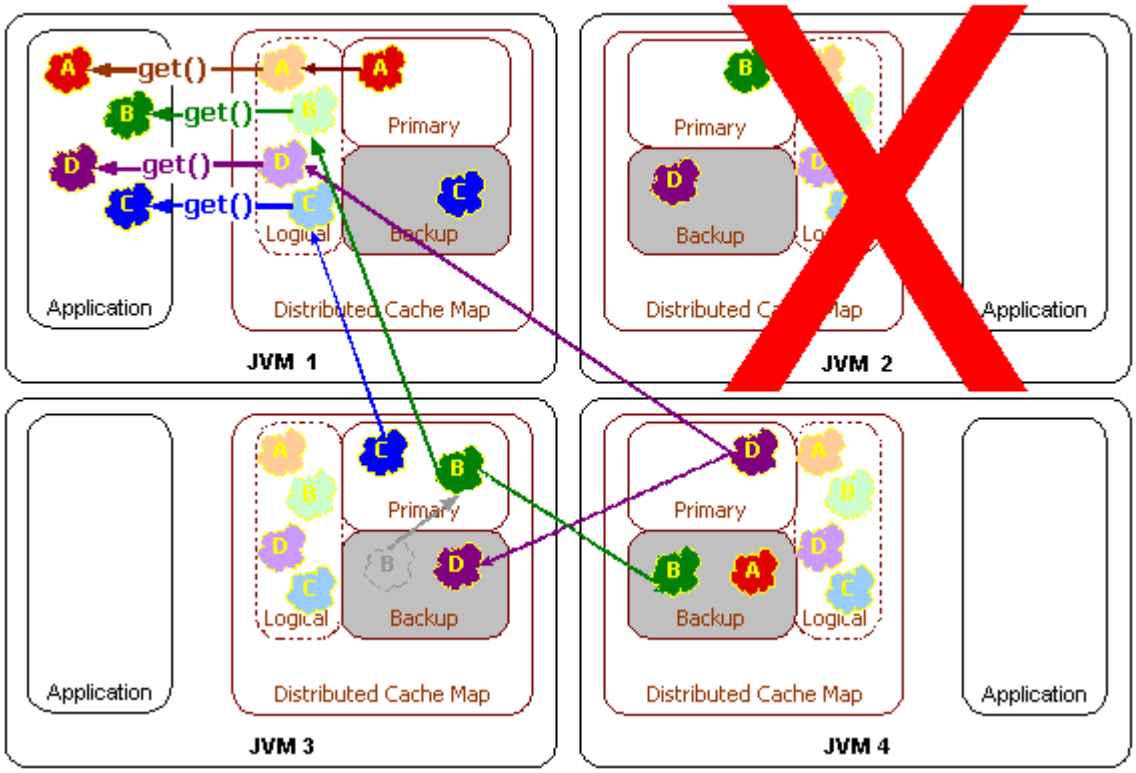


In the figure above, the data is being sent to a primary cluster node and a backup cluster node. This is for failover purposes, and corresponds to a backup count of one. (The default backup count setting is one.) If the cache data were not critical, which is to say that it could be re-loaded from disk, the backup count could be set to zero, which would allow some portion of the distributed cache data to be lost in the event of a cluster node failure. If the cache were extremely critical, a higher backup count, such as two, could be used. The backup count only affects the performance of cache modifications, such as those made by adding, changing or removing cache entries.

Modifications to the cache are not considered complete until all backups have acknowledged receipt of the modification. This means that there is a slight performance penalty for cache modifications when using the distributed cache backups; however it guarantees that if a cluster node were to unexpectedly fail, that data consistency is maintained and no data will be lost.

Failover of a distributed cache involves promoting backup data to be primary storage. When a cluster node fails, all remaining cluster nodes determine what data each holds in backup that the failed cluster node had primary responsible for when it died. Those data becomes the responsibility of whatever cluster node was the backup for the data:

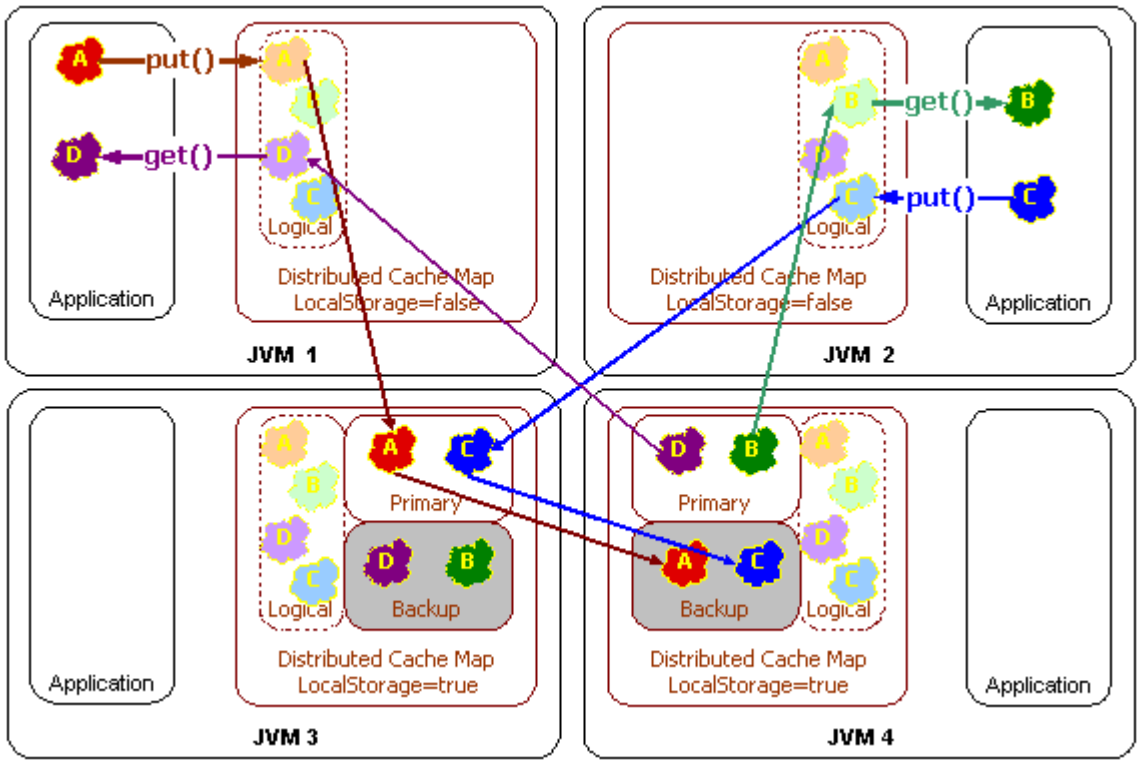
Figure 8–3 Failover in a Partitioned Cache Environment



If there are multiple levels of backup, the first backup becomes responsible for the data; the second backup becomes the new first backup, and so on. Just as with the replicated cache service, lock information is also retained in the case of server failure; the sole exception is when the locks for the failed cluster node are automatically released.

The distributed cache service also allows certain cluster nodes to be configured to store data, and others to be configured to not store data. The name of this setting is *local storage enabled*. Cluster nodes that are configured with the local storage enabled option will provide the cache storage and the backup storage for the distributed cache. Regardless of this setting, all cluster nodes will have the same exact view of the data, due to location transparency.

Figure 8–4 Local Storage in a Partitioned Cache Environment



There are several benefits to the local storage enabled option:

- The Java heap size of the cluster nodes that have turned off local storage enabled will not be affected at all by the amount of data in the cache, because that data will be cached on other cluster nodes. This is particularly useful for application server processes running on older JVM versions with large Java heaps, because those processes often suffer from garbage collection pauses that grow exponentially with the size of the heap.
- Coherence allows each cluster node to run any supported version of the JVM. That means that cluster nodes with local storage enabled turned on could be running a newer JVM version that supports larger heap sizes, or Coherence's off-heap storage using the Java NIO features.
- The local storage enabled option allows some cluster nodes to be used just for storing the cache data; such cluster nodes are called Coherence cache servers. Cache servers are commonly used to scale up Coherence's distributed query functionality.

9

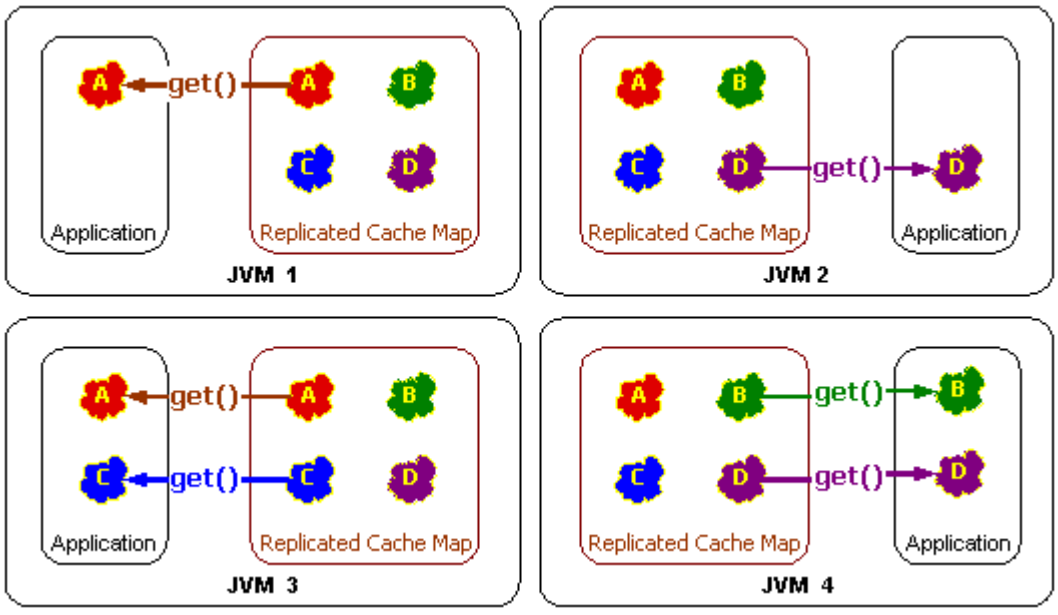
Replicated Cache Service

The replicated cache excels in its ability to handle data replication, concurrency control and failover in a cluster, all while delivering in-memory data access speeds. A clustered replicated cache is exactly what it says it is: a cache that replicates its data to all cluster nodes.

There are several challenges to building a reliable replicated cache. The first is how to get it to scale and perform well. Updates to the cache have to be sent to all cluster nodes, and all cluster nodes have to end up with the same data, even if multiple updates to the same piece of data occur at the same time. Also, if a cluster node requests a lock, it should not have to get all cluster nodes to agree on the lock, otherwise it will scale extremely poorly; yet in the case of cluster node failure, all of the data and lock information must be kept safely. Coherence handles all of these scenarios transparently, and provides the most scalable and highly available replicated cache implementation available for Java applications.

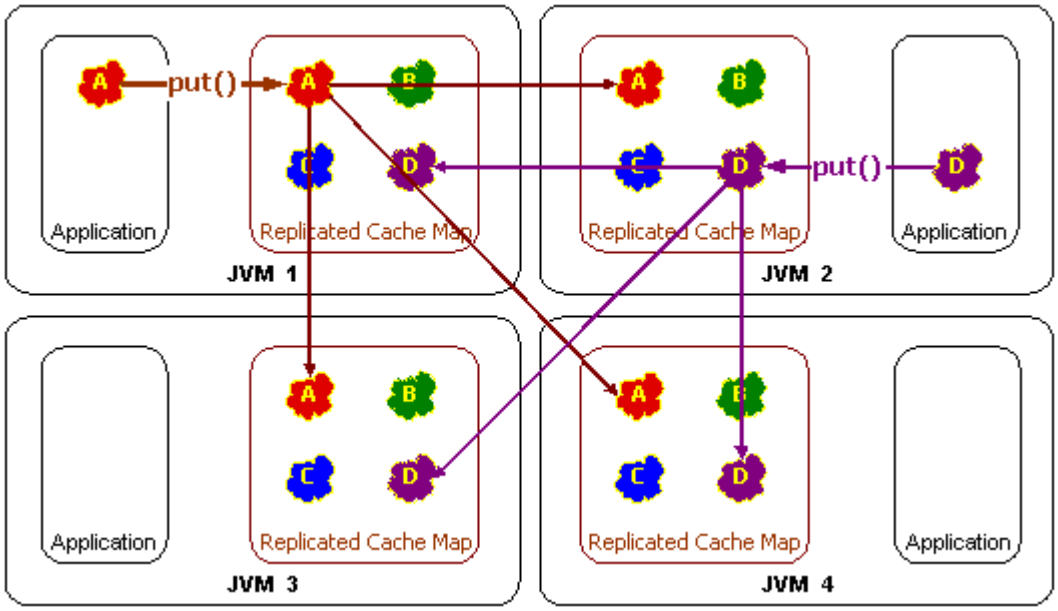
The best part of a replicated cache is its access speed. Since the data is replicated to each cluster node, it is available for use without any waiting. This is referred to as "zero latency access," and is perfect for situations in which an application requires the highest possible speed in its data access. Each cluster node (JVM) accesses the data from its own memory:

Figure 9–1 Get Operation in a Replicated Cache Environment



In contrast, updating a replicated cache requires pushing the new version of the data to all other cluster nodes:

Figure 9–2 Put Operation in a Replicated Cache Environment



Coherence implements its replicated cache service in such a way that all read-only operations occur locally, all concurrency control operations involve at most one other cluster node, and only update operations require communicating with all other cluster nodes. The result is excellent scalable performance, and as with all of the Coherence services, the replicated cache service provides transparent and complete failover and failback.

The limitations of the replicated cache service should also be carefully considered. First, however much data is managed by the replicated cache service is on each and every cluster node that has joined the service. That means that memory utilization (the Java heap size) is increased for each cluster node, which can impact performance. Secondly, replicated caches with a high incidence of updates will not scale linearly as the cluster grows; in other words, the cluster will suffer diminishing returns as cluster nodes are added.

10

Local Cache

While it is not a clustered service, the Coherence local cache implementation is often used in combination with various Coherence clustered cache services. The Coherence local cache is just that: a cache that is local to (completely contained within) a particular cluster node. There are several attributes of the local cache that are particularly interesting:

- The local cache implements the same standard collections interface that the clustered caches implement, meaning that there is no programming difference between using a local or a clustered cache. Just like the clustered caches, the local cache is tracking to the JCache API, which itself is based on the same standard collections API that the local cache is based on.
- The local cache can be size-limited. This means that the local cache can restrict the number of entries that it caches, and automatically evict entries when the cache becomes full. Furthermore, both the sizing of entries and the eviction policies can be customized, for example allowing the cache to be size-limited based on the memory used by the cached entries. The default eviction policy uses a combination of Most Frequently Used (MFU) and Most Recently Used (MRU) information, scaled on a logarithmic curve, to determine what cache items to evict. This algorithm is the best general-purpose eviction algorithm because it works well for short duration and long duration caches, and it balances frequency versus recentness to avoid cache thrashing. The pure LRU and pure LFU algorithms are also supported, and the ability to plug in custom eviction policies.
- The local cache supports automatic expiration of cached entries, meaning that each cache entry can be assigned a time to live in the cache.
- The local cache is thread safe and highly concurrent, allowing many threads to simultaneously access and update entries in the local cache.
- The local cache supports cache notifications. These notifications are provided for additions (entries that are put by the client, or automatically loaded into the cache), modifications (entries that are put by the client, or automatically reloaded), and deletions (entries that are removed by the client, or automatically expired, flushed, or evicted.) These are the same cache events supported by the clustered caches.
- The local cache maintains hit and miss statistics. These runtime statistics can be used to accurately project the effectiveness of the cache, and adjust its size-limiting and auto-expiring settings accordingly while the cache is running.

The local cache is important to the clustered cache services for several reasons, including as part of Coherence's near cache technology, and with the modular backing map architecture.

Configuring the Local Cache

The key element for configuring the Local Cache is `<local-scheme>`. Local caches are generally nested within other cache schemes, for instance as the front-tier of a near-scheme. Thus, this element can appear as a sub-element of any of these elements in the `coherence-cache-config` file: `<caching-schemes>`, `<distributed-scheme>`, `<replicated-scheme>`, `<optimistic-scheme>`, `<near-scheme>`, `<versioned-near-scheme>`, `<overflow-scheme>`, `<read-write-backing-map-scheme>`, and `<versioned-backing-map-scheme>`.

The `<local-scheme>` provides several optional sub-elements that let you define the characteristics of the cache. For example, the `<low-units>` and `<high-units>` sub-elements allow you to limit the cache in terms of size. When the cache reaches its maximum allowable size it prunes itself back to a specified smaller size, choosing which entries to evict according to a specified eviction-policy (`<eviction-policy>`). The entries and size limitations are measured in terms of units as calculated by the scheme's unit-calculator (`<unit-calculator>`).

You can also limit the cache in terms of time. The `<expiry-delay>` sub-element specifies the amount of time from last update that entries will be kept by the cache before being marked as expired. Any attempt to read an expired entry will result in a reloading of the entry from the configured cache store (`<cachestore-scheme>`).

If a `<cache-store-scheme>` is not specified, then the cached data will only reside in memory, and only reflect operations performed on the cache itself. See `<local-scheme>` for a complete description of all of the available sub-elements.

The XML code Example 10–1 illustrates the configuration of a Local Cache. See *Sample Cache Configurations* for additional examples.

Example 10–1 Local Cache Configuration

```
<?xml version="1.0"?>

<cache-config>
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>example-local-cache</cache-name>
      <scheme-name>example-local</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>
  <caching-schemes>
    <local-scheme>
      <scheme-name>example-local</scheme-name>
      <eviction-policy>LRU</eviction-policy>
      <high-units>32000</high-units>
      <low-units>10</low-units>
      <unit-calculator>FIXED</unit-calculator>
      <expiry-delay>10ms</expiry-delay>
      <cachestore-scheme>
        <class-scheme>
          <class-name>ExampleCacheStore</class-name>
        </class-scheme>
      </cachestore-scheme>
      <pre-load>true</pre-load>
    </local-scheme>
  </caching-schemes>
</cache-config>
```

For more information, see "Configuring a Local Cache for C++ Clients" and "Configuring a Local Cache for .NET Clients" in the *Oracle Coherence Client Guide*.

11

Near Cache

The objective of a Near Cache is to provide the best of both worlds between the extreme performance of the Replicated Cache Service and the extreme scalability of the Partitioned Cache Service by providing fast read access to Most Recently Used (MRU) and Most Frequently Used (MFU) data. To achieve this, the Near Cache is an implementation that wraps two caches: a "front cache" and a "back cache" that automatically and transparently communicate with each other by using a read-through/write-through approach.

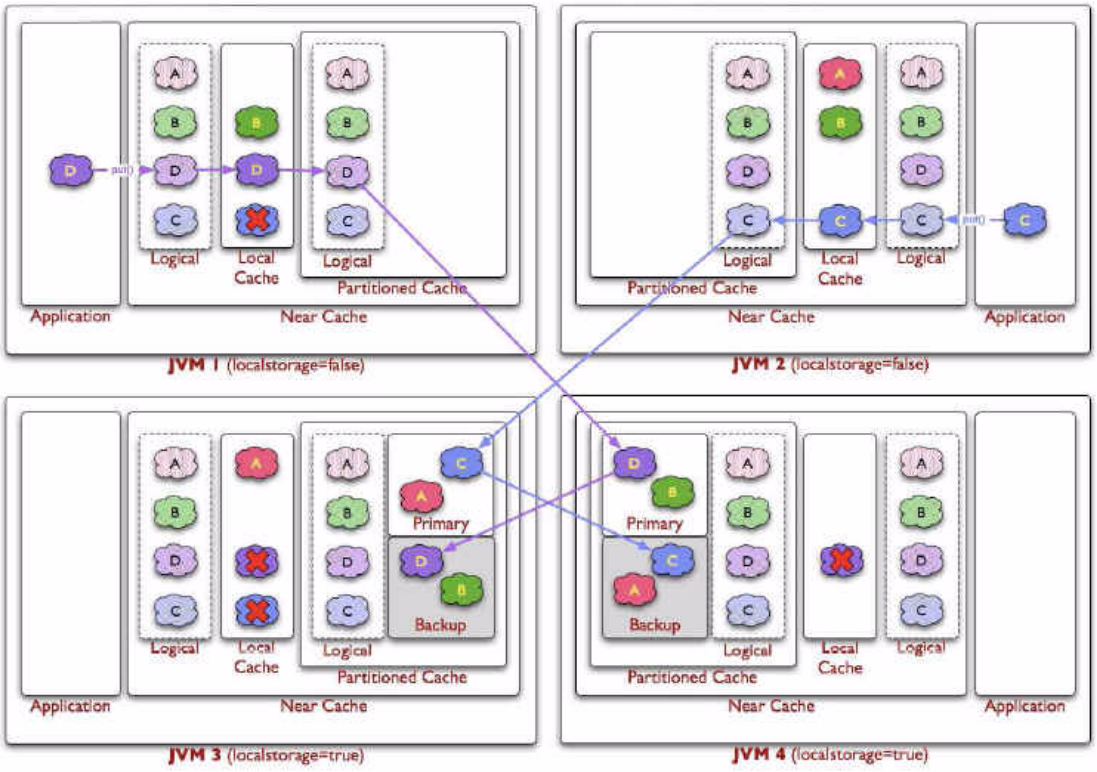
The "front cache" provides local cache access. It is assumed to be inexpensive, in that it is fast, and is limited in terms of size. The "back cache" can be a centralized or multi-tiered cache that can load-on-demand in case of local cache misses. The "back cache" is assumed to be complete and correct in that it has much higher capacity, but more expensive in terms of access speed. The use of a Near Cache is not confined to Coherence*Extend; it also works with TCMP.

This design allows Near Caches to configure levels of cache coherency, from the most basic expiry-based caches and invalidation-based caches, up to advanced data-version caches that can provide guaranteed coherency. The result is a tunable balance between the preservation of local memory resources and the performance benefits of truly local caches.

The typical deployment uses a Local Cache for the "front cache". A Local Cache is a reasonable choice because it is thread safe, highly concurrent, size-limited and/or auto-expiring and stores the data in object form. For the "back cache", a remote, partitioned cache is used.

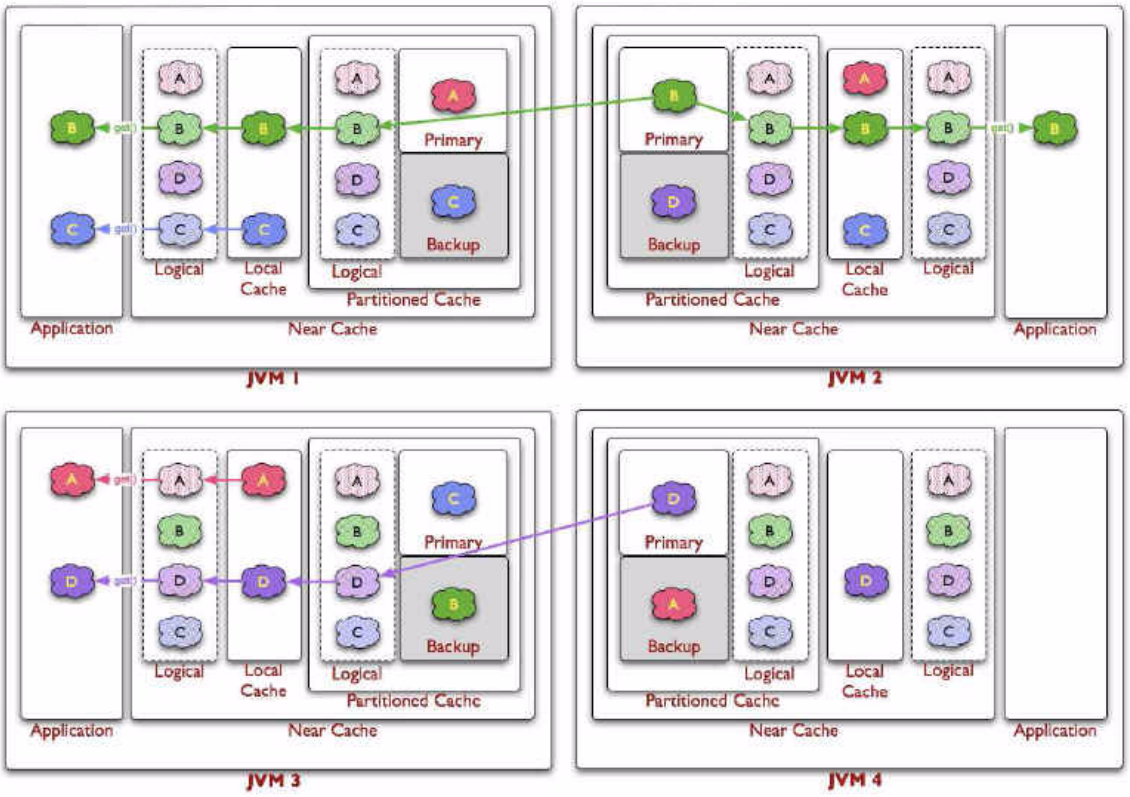
The following figure illustrates the data flow in a Near Cache. If the client writes an object D into the grid, the object is placed in the local cache inside the local JVM and in the partitioned cache which is backing it (including a backup copy). If the client requests the object, it can be obtained from the local, or "front cache", in object form with no latency.

Figure 11–1 Put Operations in a Near Cache Environment



If the client requests an object that has been expired or invalidated from the "front cache", then Coherence will automatically retrieve the object from the partitioned cache. The updated object will be written to the "front cache" and then delivered to the client.

Figure 11–2 Get Operations in a Near Cache Environment



Near Cache Invalidation Strategies

An invalidation strategy keeps the "front cache" of the Near Cache synchronized with the "back cache." The Near Cache can be configured to listen to certain events in the back cache and automatically update or invalidate entries in the front cache. Depending on the interface that the back cache implements, the Near Cache provides four different strategies of invalidating the front cache entries that have changed by other processes in the back cache

Table 11–1 describes the invalidation strategies. You can find more information on the invalidation strategies and the read-through/write-through approach in Chapter 12, "Read-Through, Write-Through, Write-Behind, and Refresh-Ahead Caching".

Table 11–1 Near Cache Invalidation Strategies

Strategy Name	Description
None	This strategy instructs the cache not to listen for invalidation events at all. This is the best choice for raw performance and scalability when business requirements permit the use of data which might not be absolutely current. Freshness of data can be guaranteed by use of a sufficiently brief eviction policy for the front cache.
Present	This strategy instructs the Near Cache to listen to the back cache events related only to the items currently present in the front cache. This strategy works best when each instance of a front cache contains distinct subset of data relative to the other front cache instances (for example, sticky data access patterns).

Table 11–1 (Cont.) Near Cache Invalidation Strategies

Strategy Name	Description
All	This strategy instructs the Near Cache to listen to all back cache events. This strategy is optimal for read-heavy tiered access patterns where there is significant overlap between the different instances of front caches.
Auto	This strategy instructs the Near Cache to switch automatically between `Present` and `All` strategies based on the cache statistics.

Configuring the Near Cache

A Near Cache is configured by using the `<near-scheme>` element in the `coherence-cache-config` file. This element has two required sub-elements: front-scheme for configuring a local (front-tier) cache and a back-scheme for defining a remote (back-tier) cache. While a local cache (`<local-scheme>`) is a typical choice for the front-tier, you can also use schemes based on Java Objects (`<class-scheme>`) and, other than for .Net and C++ clients, non-JVM heap-based caches (`<external-scheme>` or `<paged-external-scheme>`).

The remote or back-tier cache is described by the `<back-scheme>` element. A back-tier cache can be either a distributed cache (`<distributed-scheme>`) or a remote cache (`<remote-cache-scheme>`). The `<remote-cache-scheme>` element enables you to use a clustered cache from outside the current cluster.

Optional sub-elements of `<near-scheme>` include `<invalidation-strategy>` for specifying how the front-tier and back-tier objects will be kept synchronous and `<listener>` for specifying a listener which will be notified of events occurring on the cache.

For an example configuration, see "Sample Near Cache Configuration". The elements in the file are described in the `<near-scheme>` topic.

Obtaining a Near Cache Reference

Coherence provides methods in the `com.tangosol.net.CacheFactory` class to obtain a reference to a configured Near Cache by name. For example:

Example 11–1 Obtaining a Near Cache Reference

```
NamedCache cache = CacheFactory.getCache("example-near-cache");
```

Coherence also enables you to configure a Near Cache for Java, C++, or for .NET clients.

Cleaning Up Resources Associated with a Near Cache

Instances of all `NamedCache` implementations, including `NearCache`, should be explicitly released by calling the `NamedCache.release()` method when they are no longer needed. This frees any resources they might hold.

If the particular `NamedCache` is used for the duration of the application, then the resources will be cleaned up when the application is shut down or otherwise stops. However, if it is only used for a period, the application should call its `release()` method when finished using it.

Sample Near Cache Configuration

The following sample code illustrates the configuration of a Near Cache. Sub-elements of <near-scheme> define the Near Cache. Note the use of the <front-scheme> element for configuring a local (front) cache and a <back-scheme> element for defining a remote (back) cache. See the <near-scheme> topic for a description of the Near Cache elements.

Example 11–2 Sample Near Cache Configuration

```
<?xml version="1.0"?>

<cache-config>
  <caching-scheme-mapping>
    <cache-mapping>
      <cache-name>example-near-cache</cache-name>
      <scheme-name>example-near</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>
  <caching-schemes>
    <local-scheme>
      <scheme-name>example-local</scheme-name>
    </local-scheme>

    <near-scheme>
      <scheme-name>example-near</scheme-name>
      <front-scheme>
        <local-scheme>
          <scheme-ref>example-local</scheme-ref>
        </local-scheme>
      </front-scheme>
      <back-scheme>
        <remote-cache-scheme>
          <scheme-ref>example-remote</scheme-ref>
        </remote-cache-scheme>
      </back-scheme>
    </near-scheme>

    <remote-cache-scheme>
      <scheme-name>example-remote</scheme-name>
      <service-name>ExtendTcpCacheService</service-name>
      <initiator-config>
        <tcp-initiator>
          <remote-addresses>
            <socket-address>
              <address>localhost</address>
              <port>9099</port>
            </socket-address>
          </remote-addresses>

          <connect-timeout>5s</connect-timeout>
        </tcp-initiator>

        <outgoing-message-handler>
          <request-timeout>30s</request-timeout>
        </outgoing-message-handler>
      </initiator-config>
    </remote-cache-scheme>
  </caching-schemes>
</cache-config>
```

12

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

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

> **Note:** For use with Partitioned (Distributed) and Near cache topologies: Read-through/write-through caching (and variants) are intended for use only with the Partitioned (Distributed) cache topology (and by extension, Near cache). Local caches support a subset of this functionality. Replicated and Optimistic caches should not be used.

Pluggable Cache Store

A `CacheStore` is an application-specific adapter used to connect a cache to a underlying data source. The `CacheStore` implementation accesses the data source by using a data access mechanism (for example, *Hibernate*, *Toplink* Essentials, *JPA*, application-specific JDBC calls, another application, mainframe, another cache, and so on). The `CacheStore` understands how to build a Java object using data retrieved from the data source, map and write an object to the data source, and erase an object from the data source.

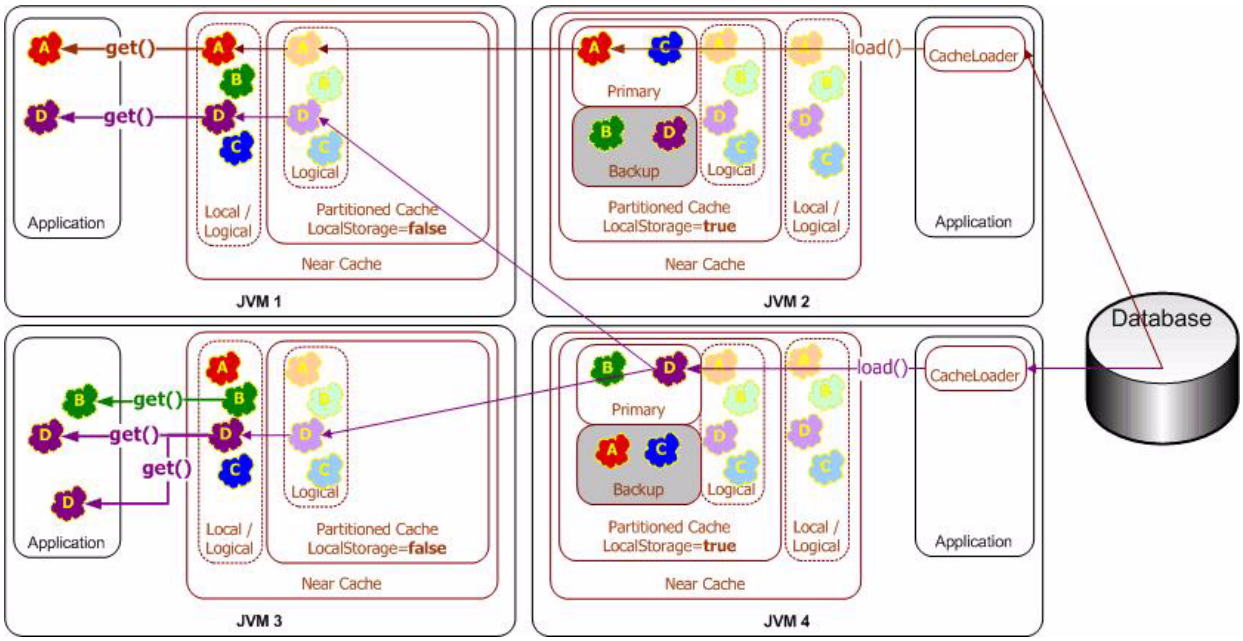
Both the data source connection strategy and the data source-to-application-object mapping information are specific to the data source schema, application class layout, and operating environment. Therefore, this mapping information must be provided by the application developer in the form of a `CacheStore` implementation. See "Creating a CacheStore Implementation" for more information.

Read-Through Caching

When an application asks the cache for an entry, for example the `key X`, and `X` is not already in the cache, Coherence will automatically delegate to the CacheStore and ask it to load `X` from the underlying data source. If `X` exists in the data source, the `CacheStore` will load it, return it to Coherence, then Coherence will place it in the cache for future use and finally will return `X` to the application code that requested it. This is called **Read-Through** caching. Refresh-Ahead Cache functionality may further

improve read performance (by reducing perceived latency). See "Refresh-Ahead Caching" for more information.

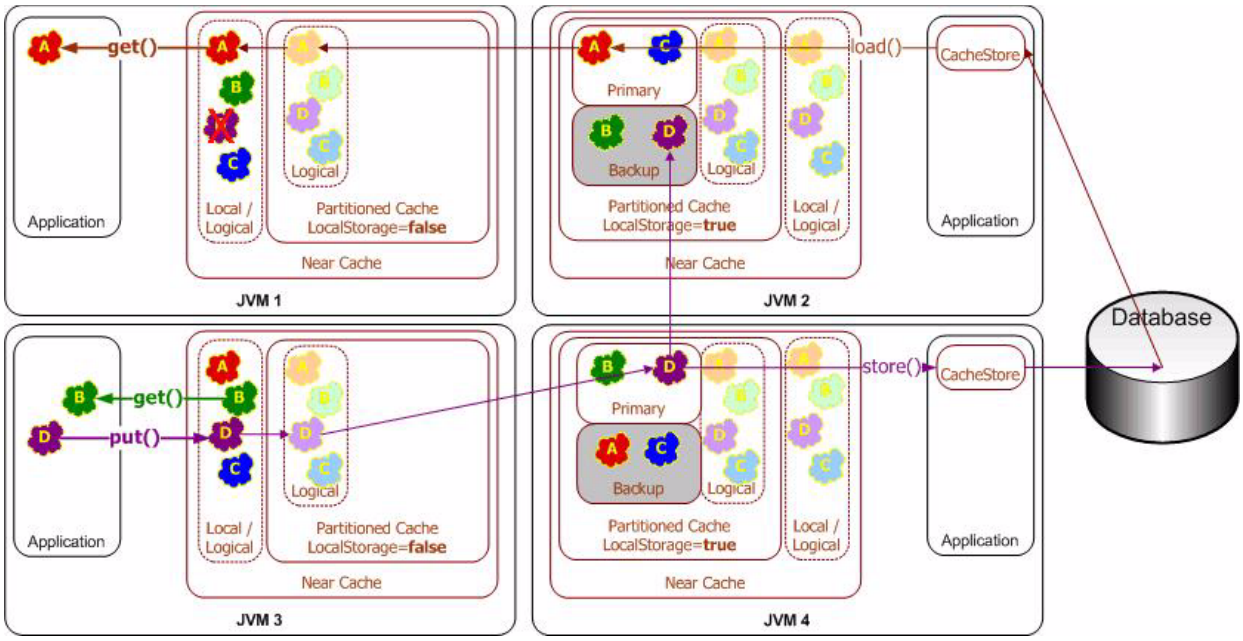
Figure 12–1 Read Through Caching



Write-Through Caching

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

Figure 12–2 Write-Through Caching



Write-Behind Caching

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

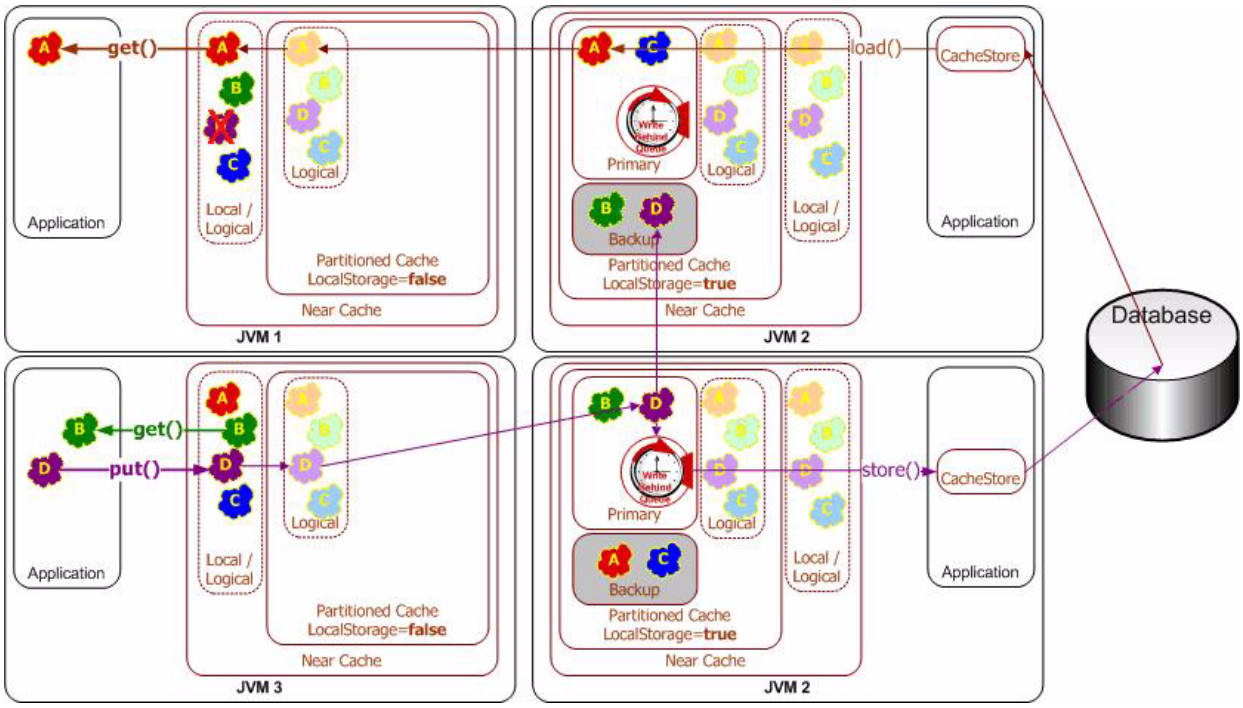
The result is a "read-once and write at a configured interval" (that is, much less often) scenario. There are four main benefits to this type of architecture:

- The application improves in performance, because the user does not have to wait for data to be written to the underlying data source. (The data is written later, and by a different execution thread.)
- The application experiences drastically reduced database load: Since the amount of both read and write operations is reduced, so is the database load. The reads are reduced by caching, as with any other caching approach. The writes, which are typically much more expensive operations, are often reduced because multiple changes to the same object within the write-behind interval are "coalesced" and only written once to the underlying data source ("write-coalescing"). Additionally, writes to multiple cache entries may be combined into a single database transaction ("write-combining") by using the `CacheStore.storeAll()` method.
- The application is somewhat insulated from database failures: the **Write-Behind** feature can be configured in such a way that a write failure will result in the object being re-queued for write. If the data that the application is using is in the Coherence cache, the application can continue operation without the database being up. This is easily attainable when using the Coherence Partitioned Cache,

which partitions the entire cache across all participating cluster nodes (with local-storage enabled), thus allowing for enormous caches.

- Linear Scalability: For an application to handle more concurrent users you need only increase the number of nodes in the cluster; the effect on the database in terms of load can be tuned by increasing the write-behind interval.

Figure 12–3 Write Behind Caching



Write-Behind Requirements

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

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

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

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

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

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

Refresh-Ahead Caching

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

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

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

The value of the refresh-ahead factor is specified by the `<refresh-ahead-factor>` subelement of the `<read-write-backing-map-scheme>` element in the coherence-cache-config.xml file. Refresh-ahead assumes that you have also set an expiration time (`<expiry-delay>`) for entries in the cache.

The XML code fragment in Example 12–1 configures a refresh-ahead factor of 0.5 and an expiration time of 20 seconds for entries in the local cache. This means that if an entry is accessed within 10 seconds of its expiration time, it will be scheduled for an asynchronous reload from the cache store.

Example 12–1 Cache Configuration Specifying a Refresh-Ahead Factor

```
<cache-config>
...

   <distributed-scheme>
      <scheme-name>categories-cache-all-scheme</scheme-name>
      <service-name>DistributedCache</service-name>
      <backing-map-scheme>

      <!--
      Read-write-backing-map caching scheme.
```

```
      -->
      <read-write-backing-map-scheme>
         <scheme-name>categoriesLoaderScheme</scheme-name>
         <internal-cache-scheme>
            <local-scheme>
               <scheme-ref>categories-eviction</scheme-ref>
            </local-scheme>
         </internal-cache-scheme>

         <cachestore-scheme>
            <class-scheme>

<class-name>com.demo.cache.coherence.categories.CategoryCacheLoader</class-name>
            </class-scheme>
         </cachestore-scheme>
         <refresh-ahead-factor>0.5</refresh-ahead-factor>
      </read-write-backing-map-scheme>
      </backing-map-scheme>
      <autostart>true</autostart>
   </distributed-scheme>
...

    <!--
    Backing map scheme definition used by all the caches that require
    size limitation and/or expiry eviction policies.
    -->
   <local-scheme>
      <scheme-name>categories-eviction</scheme-name>
      <expiry-delay>20s</expiry-delay>
   </local-scheme>
...
</cache-config>
```

Selecting a Cache Strategy

This section compares and contrasts the benefits of several caching strategies.

- Read-Through/Write-Through versus Cache-Aside
- Refresh-Ahead versus Read-Through
- Write-Behind versus Write-Through

Read-Through/Write-Through versus Cache-Aside

There are two common approaches to the cache-aside pattern in a clustered environment. One involves checking for a cache miss, then querying the database, populating the cache, and continuing application processing. This can result in multiple database visits if different application threads perform this processing at the same time. Alternatively, applications may perform double-checked locking (which works since the check is atomic with respect to the cache entry). This, however, results in a substantial amount of overhead on a cache miss or a database update (a clustered lock, additional read, and clustered unlock - up to 10 additional network hops, or 6-8ms on a typical gigabit Ethernet connection, plus additional processing overhead and an increase in the "lock duration" for a cache entry).

By using inline caching, the entry is locked only for the 2 network hops (while the data is copied to the backup server for fault-tolerance). Additionally, the locks are maintained locally on the partition owner. Furthermore, application code is fully managed on the cache server, meaning that only a controlled subset of nodes will

directly access the database (resulting in more predictable load and security). Additionally, this decouples cache clients from database logic.

Refresh-Ahead versus Read-Through

Refresh-ahead offers reduced latency compared to read-through, but only if the cache can accurately predict which cache items are likely to be needed in the future. With full accuracy in these predictions, refresh-ahead will offer reduced latency and no added overhead. The higher the rate of misprediction, the greater the impact will be on throughput (as more unnecessary requests will be sent to the database) - potentially even having a negative impact on latency should the database start to fall behind on request processing.

Write-Behind versus Write-Through

If the requirements for write-behind caching can be satisfied, write-behind caching may deliver considerably higher throughput and reduced latency compared to write-through caching. Additionally write-behind caching lowers the load on the database (fewer writes), and on the cache server (reduced cache value deserialization).

Idempotency

All `CacheStore` operations should be designed to be idempotent (that is, repeatable without unwanted side-effects). For write-through and write-behind caches, this allows Coherence to provide low-cost fault-tolerance for partial updates by re-trying the database portion of a cache update during failover processing. For write-behind caching, idempotency also allows Coherence to combine multiple cache updates into a single `CacheStore` invocation without affecting data integrity.

Applications that have a requirement for write-behind caching but which must avoid write-combining (for example, for auditing reasons), should create a "versioned" cache key (for example, by combining the natural primary key with a sequence id).

Write-Through Limitations

Coherence does not support two-phase `CacheStore` operations across multiple `CacheStore` instances. In other words, if two cache entries are updated, triggering calls to `CacheStore` modules sitting on separate cache servers, it is possible for one database update to succeed and for the other to fail. In this case, it may be preferable to use a cache-aside architecture (updating the cache and database as two separate components of a single transaction) with the application server transaction manager. In many cases it is possible to design the database schema to prevent logical commit failures (but obviously not server failures). Write-behind caching avoids this issue as "puts" are not affected by database behavior (and the underlying issues will have been addressed earlier in the design process). This limitation will be addressed in an upcoming release of Coherence.

Cache Queries

Cache queries only operate on data stored in the cache and will not trigger the `CacheStore` to load any missing (or potentially missing) data. Therefore, applications that query `CacheStore`-backed caches should ensure that all necessary data required for the queries has been pre-loaded. For efficiency, most bulk load operations should be done at application startup by streaming the dataset directly from the database into the cache (batching blocks of data into the cache by using `NamedCache.putAll()`.

The loader process will need to use a "Controllable Cachestore" pattern to disable circular updates back to the database. The `CacheStore` may be controlled by using an Invocation service (sending agents across the cluster to modify a local flag in each JVM) or by setting the value in a Replicated cache (a different cache service) and reading it in every `CacheStore` method invocation (minimal overhead compared to the typical database operation). A custom MBean can also be used, a simple task with Coherence's clustered JMX facilities.

Creating a CacheStore Implementation

`CacheStore` implementations are pluggable, and depending on the cache's usage of the data source you will need to implement one of two interfaces:

- `CacheLoader` for read-only caches
- `CacheStore` which extends `CacheLoader` to support read/write caches

These interfaces are located in the `com.tangosol.net.cache` package. The `CacheLoader` interface has two main methods: `load(Object key)` and `loadAll(Collection keys)`, and the `CacheStore` interface adds the methods `store(Object key, Object value)`,`storeAll(Map mapEntries)`, `erase(Object key)`, and `eraseAll(Collection colKeys)`.

See"*Sample CacheStores*" in the *Oracle Coherence Developer's Guide* for an example implementation.

Plugging in a CacheStore Implementation

To plug in a `CacheStore` module, specify the `CacheStore` implementation class name within the `distributed-scheme`, `backing-map-scheme`, `cachestore-scheme`, or `read-write-backing-map-scheme`, cache configuration element.

The `read-write-backing-map-scheme` configures a `com.tangosol.net.cache.ReadWriteBackingMap`. This backing map is composed of two key elements: an internal map that actually caches the data (see `internal-cache-scheme`), and a `CacheStore` module that interacts with the database (see `cachestore-scheme`).

Example 12–2 illustrates a cache configuration that specifies a `CacheStore` module. The `<init-params>` element contains an ordered list of parameters that will be passed into the `CacheStore` constructor. The `{cache-name}` configuration macro is used to pass the cache name into the `CacheStore` implementation, allowing it to be mapped to a database table. For a complete list of available macros, see *Cache Configuration Parameter Macros*.

For more detailed information on configuring write-behind and refresh-ahead, see the `read-write-backing-map-scheme`, taking note of the `write-batch-factor`, `refresh-ahead-factor`, `write-requeue-threshold`, and `rollback-cachestore-failures` elements.

Example 12–2 A Cache Configuration with a Cachestore Module

```
<?xml version="1.0"?>

<!DOCTYPE cache-config SYSTEM "cache-config.dtd">

<cache-config>
  <caching-scheme-mapping>
```

```
    <cache-mapping>
      <cache-name>com.company.dto.*</cache-name>
      <scheme-name>distributed-rwbm</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>


  <caching-schemes>
    <distributed-scheme>
      <scheme-name>distributed-rwbm</scheme-name>
      <backing-map-scheme>
        <read-write-backing-map-scheme>

          <internal-cache-scheme>
            <local-scheme/>
          </internal-cache-scheme>

          <cachestore-scheme>
            <class-scheme>
              <class-name>com.company.MyCacheStore</class-name>
              <init-params>
                <init-param>
                  <param-type>java.lang.String</param-type>
                  <param-value>{cache-name}</param-value>
                </init-param>
              </init-params>
            </class-scheme>
          </cachestore-scheme>
        </read-write-backing-map-scheme>
      </backing-map-scheme>
    </distributed-scheme>
  </caching-schemes>
</cache-config>
```

Note: Thread Count: The use of a `CacheStore` module will substantially increase the consumption of cache service threads (even the fastest database select is orders of magnitude slower than updating an in-memory structure). Consequently, the cache service thread count will need to be increased (typically in the range 10-100). The most noticeable symptom of an insufficient thread pool is increased latency for cache requests (without corresponding behavior in the backing database).

Implementation Considerations

Please keep the following in mind when implementing a `CacheStore`.

Re-entrant Calls

The `CacheStore` implementation must not call back into the hosting cache service. This includes OR/M solutions that may internally reference Coherence cache services. Note that calling into another cache service instance is allowed, though care should be taken to avoid deeply nested calls (as each call will "consume" a cache service thread and could result in deadlock if a cache service threadpool is exhausted).

Cache Server Classpath

The classes for cache entries (also known as Value Objects, Data Transfer Objects, and so on) must be in the cache server classpath (as the cache server must serialize-deserialize cache entries to interact with the `CacheStore` module.

CacheStore Collection Operations

The `CacheStore.storeAll` method is most likely to be used if the cache is configured as write-behind and the `<write-batch-factor>` is configured. The `CacheLoader.loadAll` method is also used by Coherence. For similar reasons, its first use will likely require refresh-ahead to be enabled.

Connection Pools

Database connections should be retrieved from the container connection pool (or a 3rd party connection pool) or by using a thread-local lazy-initialization pattern. As dedicated cache servers are often deployed without a managing container, the latter may be the most attractive option (though the cache service thread-pool size should be constrained to avoid excessive simultaneous database connections).

13

Managing an Object Model

This document describes best practices for managing an object model whose state is managed in a collection of Coherence named caches. Given a set of entity classes, and a set of entity relationships, what is the best means of expressing and managing the object model across a set of Coherence named caches?

Cache Usage Paradigms

The value of a clustered caching solution depends on how it is used. Is it simply caching database data in the application tier, keeping it ready for instant access? Is it taking the next step to move transactional control into the application tier? Or does it go a step further by aggressively optimizing transactional control?

Simple Data Caching

Simple data caches are common, especially in situations where concurrency control is not required (for example, content caching) or in situations where transactional control is still managed by the database (for example, plug-in caches for Hibernate and JDO products). This approach has minimal impact on application design, and is often implemented transparently by the Object/Relational Mapping (ORM) layer or the application server (EJB container, Spring, to name a few). However, it still does not completely solve the issue of overloading the database server; in particular, while non-transactional reads are handled in the application tier, all transactional data management still requires interaction with the database.

It is important to note that caching is not an orthogonal concern when data access requirements go beyond simple access by primary key. In other words, to truly benefit from caching, applications must be designed with caching in mind.

Transactional Caching

Applications that need to scale and operate independently of the database must start to take greater responsibility for data management. This includes using Coherence features for read access (read-through caching, cache queries, aggregations), features for minimizing database transactions (write-behind), and features for managing concurrency (locking, cache transactions).

Transaction Optimization

Applications that need to combine fault-tolerance, low latency and high scalability will generally need to optimize transactions even further. Using traditional transaction control, an application might need to specify SERIALIZABLE isolation when managing an Order object. In a distributed environment, this can be a very expensive operation. Even in non-distributed environments, most databases and caching products will often use a table lock to achieve this. This places a hard limit on

scalability regardless of available hardware resources; in practice, this may limit transaction rate to hundreds of transactions per second, even with exotic hardware. However, locking "by convention" can help - for example, requiring that all acessors lock only the "parent" Order object. Doing this can reduce the scope of the lock from table-level to order-level, enabling far higher scalability. (Of course, some applications already achieve similar results by partitioning event processing across multiple JMS queues to avoid the need for explicit concurrency control.) Further optimizations include using an EntryProcessor to avoid the need for clustered coordination, which can dramatically increase the transaction rate for a given cache entry.

Techniques to Manage the Object Model

The term "relationships" refers to how objects are related to each other. For example, an Order object may contain (exclusively) a set of LineItem objects. It may also refer to a Customer object that is associated with the Order object.

The data access layer can generally be broken down into two key components, Data Access Objects (DAOs) and Data Transfer Objects (DTOs) (in other words, behavior and state). DAOs control the behavior of data access, and generally will contain the logic that manages the database or cache. DTOs contain data for use with DAOs, for example, an Order record. Also, note that a single object may (in some applications) act as both a DTO and DAO. These terms describe patterns of usage; these patterns will vary between applications, but the core principles will be applicable. For simplicity, the examples in this document follow a "Combined DAO/DTO" approach (behaviorally-rich Object Model).

Managing entity relationships can be a challenging task, especially when scalability and transactionality are required. The core challenge is that the ideal solution must be capable of managing the complexity of these inter-entity relationships with a minimum of developer involvement. Conceptually, the problem is one of taking the relationship model (which could be represented in any of several forms, including XML or Java source) and providing runtime behavior which adheres to that description.

Present solutions can be categorized into a few groups:

- Code generation (`.java` or `.class` files)
- Runtime byte-code instrumentation (`ClassLoader` interception)
- Predefined DAO methods

Code Generation

Code generation is a popular option, involving the generation of .java or .class files. This approach is commonly used with several Management and Monitoring, AOP and ORM tools (AspectJ, Hibernate). The primary challenges with this approach are the generation of artifacts, which may need to be managed in a software configuration management (SCM) system.

Byte-code instrumentation

This approach uses ClassLoader interception to instrument classes as they are loaded into the JVM. This approach is commonly used with AOP tools (AspectJ, JBossCacheAop, TerraCotta) and some ORM tools (very common with JDO implementations). Due to the risks (perceived or actual) associated with run-time modification of code (including a tendency to break the hot-deploy options on application servers), this option is not viable for many organizations and as such is a non-starter.

Developer-implemented classes

The most flexible option is to have a runtime query engine. ORM products shift most of this processing off onto the database server. The alternative is to provide the query engine inside the application tier; but again, this leads toward the same complexity that limits the manageability and scalability of a full-fledged database server.

The recommended practice for Coherence is to map out DAO methods explicitly. This provides deterministic behavior (avoiding dynamically evaluated queries), with some added effort of development. This effort will be directly proportional to the complexity of the relationship model. For small- to mid-size models (up to ~50 entity types managed by Coherence), this will be fairly modest development effort. For larger models (or for those with particularly complicated relationships), this may be a substantial development effort.

As a best practice, all state, relationships and atomic transactions should be handled by the Object Model. For more advanced transactional control, there should be an additional Service Layer which coordinates concurrency (allowing for composable transactions).

Composable Transactions:

See "Performing Transactions" in Oracle Coherence Developer's Guide for details and instructions on using transaction in Coherence.

Domain Model

A `NamedCache` should contain one type of entity (in the same way that a database table contains one type of entity). The only common exception to this are directory-type caches, which often may contain arbitrary values.

Each additional `NamedCache` consumes only a few dozen bytes of memory per participating cluster member. This will vary, based on the backing map. Caches configured with the `<read_write_backing_map_scheme>` for transparent database integration will consume additional resources if write-behind caching is enabled, but this will not be a factor until there are hundreds of named caches.

If possible, cache layouts should be designed so that business transactions map to a single cache entry update. This simplifies transactional control and can result in much greater throughput.

Most caches should use meaningful keys (as opposed to the "meaningless keys" commonly used in relational systems whose only purpose is to manage identity). The one drawback to this is limited query support (as Coherence queries currently apply only to the entry value, not the entry key); to query against key attributes, the value must duplicate the attributes.

Best Practices for Data Access Objects in Coherence

DAO objects must implement the getter/setter/query methods in terms of `NamedCache` access. The `NamedCache` API makes this very simple for the most types of operations, especially primary key lookups and simple search queries.

Example 13–1 Implementing Methods for NamedCache Access

```
public class Order
    implements Serializable
    {
    // static"Finder" method
    public static Order getOrder(OrderId orderId)
```

```
        {
        return (Order)m_cacheOrders.get(orderId);
        }

    // ...
    // mutator/accessor methods for updating Order attributes
    // ...

    // lazy-load an attribute
    public Customer getCustomer()
        {
        return (Customer)m_cacheCustomers.get(m_customerId);
        }

    // lazy-load a collection of child attributes
    public Collection getLineItems()
        {
        // returns map(LineItemId -> LineItem); just return the values
        return ((Map)m_cacheLineItems.getAll(m_lineItemIds)).values();
        }

    // fields containing order state
    private CustomerId m_customerId;
    private Collection m_lineItemIds;

    // handles to caches containing related objects
    private static final NamedCache m_cacheCustomers =
CacheFactory.getCache("customers");
    private static final NamedCache m_cacheOrders    =
CacheFactory.getCache("orders");
    private static final NamedCache m_cacheLineItems =
CacheFactory.getCache("orderlineitems");
    }
```

Service Layer

Applications that require composite transactions should use a Service Layer. This accomplishes two things. First, it allows for proper composing of multiple entities into a single transaction without compromising ACID characteristics. Second, it provides a central point of concurrency control, allowing aggressively optimized transaction management.

Automatic Transaction Management

Basic transaction management consists of ensuring clean reads (based on the isolation level) and consistent, atomic updates (based on the concurrency strategy). The Transaction Framework API (accessible either through the J2CA adapter or programmatically) handles these issues automatically.

See "Performing Transactions" in *Oracle Coherence Developer's Guide*.

Explicit Transaction Management

Unfortunately, the transaction characteristics common with database transactions (described as a combination of isolation level and concurrency strategy for the entire transaction) provide very coarse-grained control. This coarse-grained control is often unsuitable for caches, which are generally subject to far greater transaction rates. By

manually controlling transactions, applications can gain much greater control over concurrency and therefore dramatically increase efficiency.

The general pattern for pessimistic transactions is `lock -> read -> write -> unlock`. For optimistic transactions, the sequence is `read -> lock & validate -> write -> unlock`. When considering a two-phase commit, "locking" is the first phase, and "writing" is the second phase. Locking individual objects will ensure `REPEATABLE_READ` isolation semantics. Dropping the locks will be equivalent to `READ_COMMITTED` isolation.

By mixing isolation and concurrency strategies, applications can achieve higher transaction rates. For example, an overly pessimistic concurrency strategy will reduce concurrency, but an overly optimistic strategy may cause excessive transaction rollbacks. By intelligently deciding which entities will be managed pessimistically, and which optimistically, applications can balance the trade-offs. Similarly, many transactions may require strong isolation for some entities, but much weaker isolation for other entities. Using only the necessary degree of isolation can minimize contention, and thus improve processing throughput.

Optimized Transaction Processing

There are several advanced transaction processing techniques that can best be applied in the Service Layer. Proper use of these techniques can dramatically improve throughput, latency and fault-tolerance, at the expense of some added effort.

The most common solution relates to minimizing the need for locking. Specifically, using an ordered locking algorithm can reduce the number of locks required, and also eliminate the possibility of deadlock. The most common example is to lock a parent object before locking the child object. In some cases, the Service Layer can depend on locks against the parent object to protect the child objects. This effectively makes locks coarse-grained (slightly increasing contention) and substantially minimizes the lock count.

Example 13–2 Using an Ordered Locking Algorithm

```
public class OrderService
    {
    // ...

    public void executeOrderIfLiabilityAcceptable(Order order)
        {
        OrderId orderId = order.getId();

        // lock the parent object; by convention, all accesses
        // will lock the parent object first, guaranteeing
        // "SERIALIZABLE" isolation with respect to the child
        // objects.
        m_cacheOrders.lock(orderId, -1);
        try
            {
            BigDecimal outstanding = new BigDecimal(0);

            // sum up child objects
            Collection lineItems = order.getLineItems();
            for (Iterator iter = lineItems.iterator(); iter.hasNext(); )
                {
                LineItem item = (LineItem)iter.next();
                outstanding = outstanding.add(item.getAmount());
                }
```

```
            // get the customer information; no locking, so
            // it is effectively READ_COMMITTED isolation.
            Customer customer = order.getCustomer();

            // apply some business logic
            if (customer.isAcceptableOrderSize(outstanding))
                {
                order.setStatus(Order.REJECTED);
                }
            else
                {
                order.setStatus(Order.EXECUTED);
                }

            // update the cache
            m_cacheOrders.put(order);
            }
        finally
            {
            m_cacheOrders.unlock(orderId);
            }
        }

    // ...
    }
```

Managing Collections of Child Objects

Shared Child Objects

For shared child objects (for example, two parent objects may both refer to the same child object), the best practice is to maintain a list of child object identifiers (aka foreign keys) in the parent object. Then use the `NamedCache.get()` or `NamedCache.getAll()` methods to access the child objects. In many cases, it may make sense to use a Near cache for the parent objects and a Replicated cache for the referenced objects (especially if they are read-mostly or read-only).

If the child objects are read-only (or stale data is acceptable), and the entire object graph is often required, then including the child objects in the parent object may be beneficial in reducing the number of cache requests. This is less likely to make sense if the referenced objects are already local, as in a Replicated, or in some cases, Near cache, as local cache requests are very efficient. Also, this makes less sense if the child objects are large. However, if fetching the child objects from another cache is likely to result in additional network operations, the reduced latency of fetching the entire object graph immediately might outweigh the cost of in-lining the child objects inside the parent object.

Owned Child Objects

If the objects are owned exclusively, then there are a few additional options. Specifically, it is possible to manage the object graph "top-down" (the normal approach), "bottom-up", or both. Generally, managing "top-down" is the simplest and most efficient approach.

If the child objects are inserted into the cache before the parent object is updated (an "ordered update" pattern), and deleted after the parent object's child list is updated, the application will never see missing child objects.

Similarly, if all Service Layer access to child objects locks the parent object first, SERIALIZABLE-style isolation can be provided very inexpensively (with respect to the child objects).

Bottom-Up Management of Child Objects

To manage the child dependencies "bottom-up", tag each child with the parent identifier. Then use a query (semantically, "find children where parent = ?") to find the child objects (and then modify them if needed). Note that queries, while very fast, are slower than primary key access. The main advantage to this approach is that it reduces contention for the parent object (within the limitations of `READ_COMMITTED` isolation). Of course, efficient management of a parent-child hierarchy could also be achieved by combining the parent and child objects into a single composite object, and using a custom "Update Child" `EntryProcessor`, which would be capable of hundreds of updates per second against each composite object.

Bi-Directional Management of Child Objects

Another option is to manage parent-child relationships bi-directionally. An advantage to this is that each child "knows" about its parent, and the parent "knows" about the child objects, simplifying graph navigation (for example, allowing a child object to find its sibling objects). The biggest drawback is that the relationship state is redundant; for a given parent-child relationship, there is data in both the parent and child objects. This complicates ensuring resilient, atomic updates of the relationship information and makes transaction management more difficult. It also complicates ordered locking/update optimizations.

Colocating Owned Objects

Denormalization

Exclusively owned objects may be managed as normal relationships (wrapping getters/setters around `NamedCache` methods), or the objects may be embedded directly (roughly analogous to "denormalizing" in database terms). Note that by denormalizing, data is not being stored redundantly, only in a less flexible format. However, since the cache schema is part of the application, and not a persistent component, the loss of flexibility is a non-issue if there is not a requirement for efficient ad hoc querying. Using an application-tier cache allows for the cache schema to be aggressively optimized for efficiency, while allowing the persistent (database) schema to be flexible and robust (typically at the expense of some efficiency).

The decision to inline child objects is dependent on the anticipated access patterns against the parent and child objects. If the bulk of cache accesses are against the entire object graph (or a substantial portion thereof), it may be optimal to embed the child objects (optimizing the "common path").

To optimize access against a portion of the object graph (for example, retrieving a single child object, or updating an attribute of the parent object), use an `EntryProcessor` to move as much processing to the server as possible, sending only the required data across the network.

Affinity

Affinity can be used to optimize collocation of parent and child objects (ensuring that the entire object graph is always located within a single JVM). This will minimize the number of servers involved in processing a multiple-entity request (queries, bulk operations, and so on). Affinity offers much of the benefit of denormalization without having any impact on application design. However, denormalizing structures can further streamline processing (for example, turning graph traversal into a single network operation).

Managing Shared Objects

Shared objects should be referenced by using a typical "lazy getter" pattern. For read-only data, the returned object may be cached in a transient (non-serializable) field for subsequent access. As usual, multiple-entity updates (for example, updating both the parent and a child object) should be managed by the service layer.

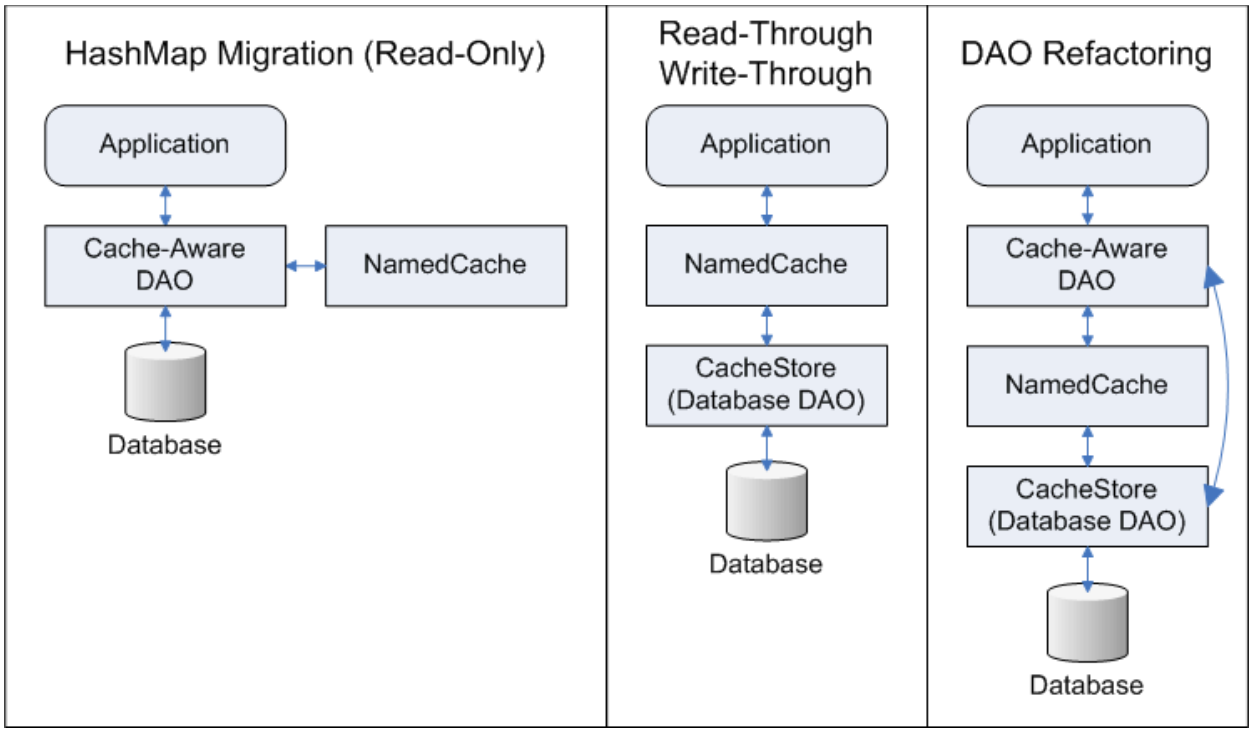
Example 13–3 Using a "Lazy Getter" Pattern

```
public class Order
    {
    // ...

    public Customer getCustomer()
        {
        return (Customer)m_cacheCustomers.get(m_customerId);
        }

    // ...
    }
```

Refactoring Existing DAOs

Generally, when refactoring existing DAOs, the pattern is to split the existing DAO into an interface and two implementations (the original database logic and the new cache-aware logic). The application will continue to use the (extracted) interface. The database logic will be moved into a `CacheStore` module. The cache-aware DAO will access the `NamedCache` (backed by the database DAO). All DAO operations that cannot be mapped onto a `NamedCache` will be passed directly to the database DAO.

Figure 13–1 Processes for Refactoring DAOs



14

Storage and Backing Map

This chapter provides information on coherence storage using backing maps. The following sections are included in this chapter:

Cache Layers

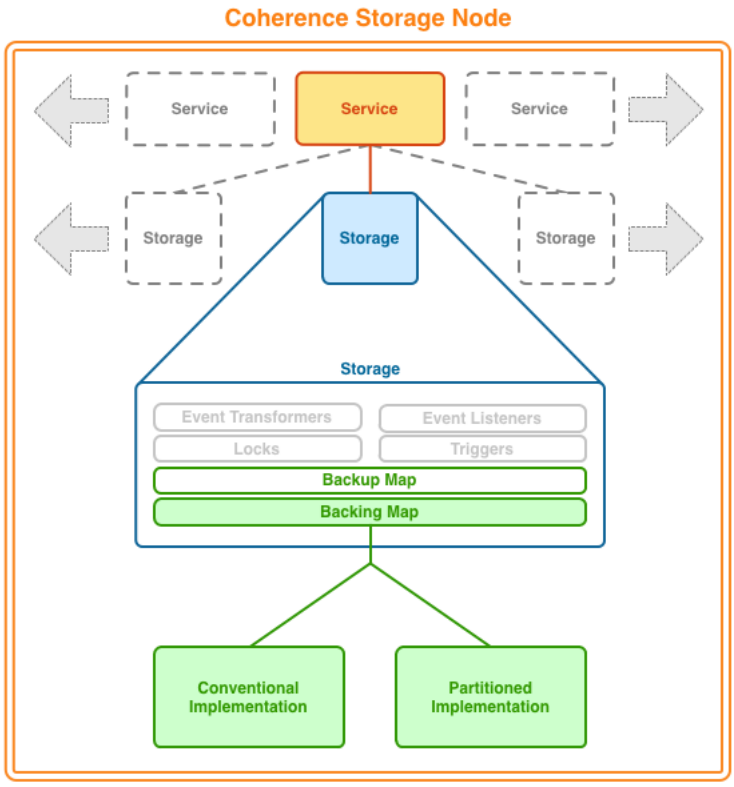
Partitioned (Distributed) cache service in Coherence has three distinct layers:

- **Client View** – The client view represents a virtual layer that provides access to the underlying partitioned data. Access to this tier is provided using the `NamedCache` interface. In this layer you can also create synthetic data structures such as `NearCache` or `ContinuousQueryCache`.
- **Storage Manager** – The storage manager is the server-side tier that is responsible for processing cache-related requests from the client tier. It manages the data structures that hold the actual cache data (primary and backup copies) and information about locks, event listeners, map triggers etc.
- **Backing Map** – The Backing Map is the server-side data structure that holds actual data.

Coherence 3.x allows users to configure a number of out-of-the-box backing map implementations as well as custom ones. Basically, the only constraint that all these Map implementation have to be aware of, is the understanding that the Storage Manager provides all keys and values in internal (Binary) format. To deal with conversions of that internal data to and from an Object format, the Storage Manager can supply Backing Map implementations with a `BackingMapManagerContext` reference.

Figure 14–1 shows a conceptual view of backing maps.

Figure 14–1 Backing Map Storage



Operations

There are number of operation types performed against the Backing Map:

- Natural access and update operations caused by the application usage. For example, `NamedCache.get()` call naturally causes a `Map.get()` call on a corresponding Backing Map; the `NamedCache.invoke()` call may cause a sequence of `Map.get()` followed by the `Map.put()`; the `NamedCache.keySet(filter)` call may cause an `Map.entrySet().iterator()` loop, and so on.
- Remove operations caused by the time-based expiry or the size-based eviction. For example, a `NamedCache.get()` or `NamedCache.size()` call from the client tier could cause a `Map.remove()` call due to an entry expiry timeout; or `NamedCache.put()` call causing a number of `Map.remove()` calls (for different keys) caused by the total amount data in a backing map reaching the configured high water-mark value.
- Insert operations caused by a `CacheStore.load()` operation (for backing maps configured with read-through or read-ahead features)
- Synthetic access and updates caused by the partition distribution (which in turn could be caused by cluster nodes fail-over or fail-back). In this case, without any application tier call, a number of entries could be inserted or removed from the backing map.

Capacity Planning

Depending on the actual implementation, the Backing Map stores the cache data in one of the following ways:

- on-heap memory
- off-heap memory
- disk (memory-mapped files or in-process DB)
- combination of any of the above

Keeping data in memory naturally provides dramatically smaller access and update latencies and is most commonly used.

More often than not, applications need to ensure that the total amount of data placed into the data grid does not exceed some predetermined amount of memory. It could be done either directly by the application tier logic or automatically using size- or expiry-based eviction. Quite naturally, the total amount of data held in a Coherence cache is equal to the sum of data volume in all corresponding backing maps (one per each cluster node that runs the corresponding partitioned cache service in a storage enabled mode).

Consider following cache configuration excerpts:

```
<backing-map-scheme>
  <local-scheme/>
</backing-map-scheme>
```

The backing map above is an instance of `com.tangosol.net.cache.LocalCache` and does not have any pre-determined size constraints and has to be controlled explicitly. Failure to do so could cause the JMV to go out-of-memory.

```
<backing-map-scheme>
  <local-scheme>
    <high-units>100m</high-units>
    <unit-calculator>BINARY</unit-calculator>
    <eviction-policy>LRU</eviction-policy>
  </local-scheme>
</backing-map-scheme>
```

This backing map is also a `com.tangosol.net.cache.LocalCache` and has a capacity limit of 100MB. As the total amount of data held by this backing map exceeds that high watermark, some entries will be removed from the backing map, bringing the volume down to the low watermark value (`<low-units>` configuration element, witch defaults to 75% of the `<high-units>`). The choice of the removed entries will be based on the LRU (Least Recently Used) eviction policy. Other options are LFU (Least Frequently Used) and Hybrid (a combination of the LRU and LFU). The value of `<high-units>` is limited to 2GB. To overcome that limitation (but maintain backward compatibility) Coherence 3.5 introduces a `<unit-factor>` element. For example, the `<high-units>` value of `8192` with a `<unit-factor>` of `1048576` will result in a high watermark value of `8GB` (see a configuration excerpt below).

```
<backing-map-scheme>
  <local-scheme>
    <expiry-delay>1h</expiry-delay>
  </local-scheme>
</backing-map-scheme>
```

This backing map will automatically evict any entries that were not updated for more than an hour. Note, that such an eviction is a "lazy" one and can happen any time after

an hour since the last update happens; the only guarantee Coherence provides is that entires more than one hour old will not be returned to a caller.

```
<backing-map-scheme>
  <external-scheme>
    <high-units>100</high-units>
    <unit-calculator>BINARY</unit-calculator>
    <unit-factor>1048576</unit-factor>
    <nio-memory-manager>
      <initial-size>1MB</initial-size>
      <maximum-size>100MB</maximum-size>
    </nio-memory-manager>
  </external-scheme>
</backing-map-scheme>
```

This backing map is an instance of `com.tangosol.net.cache.SerializationCache` which stores values in the extended (nio) memory and has a capacity limit of 100MB (100*1048576). Quite naturally, you would configure a backup storage for this cache being `off-heap` (or `file-mapped`):

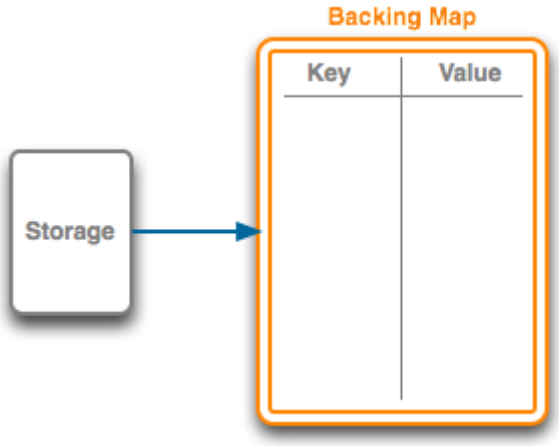
```
<backup-storage>
  <type>off-heap</type>
  <initial-size>1MB</initial-size>
  <maximum-size>100MB</maximum-size>
</backup-storage>
```

Partitioned Backing Maps

Prior to Coherence 3.5, a backing map would contain entries for all partitions owned by the corresponding node. (During partition transfer, it could also hold "in flight" entries that from the clients' perspective are temporarily not owned by anyone).

Figure 14–2 shows a conceptual view of the conventional backing map implementation.

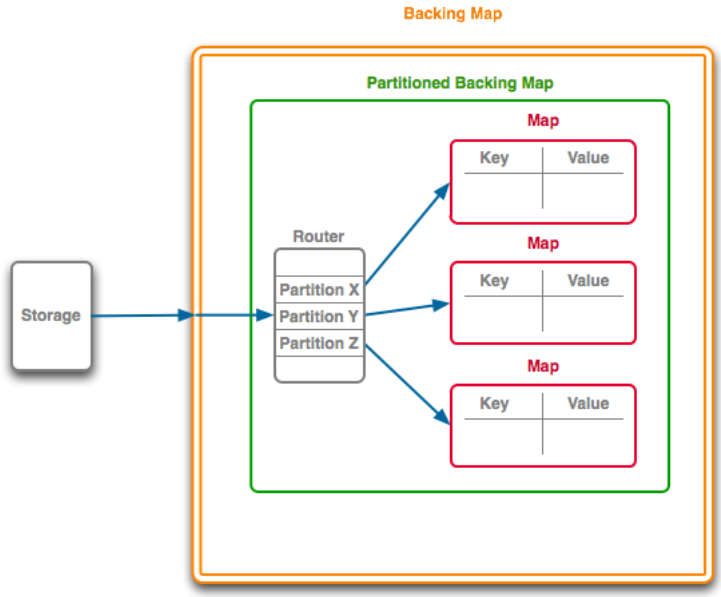
Figure 14–2 Conventional Backing Map Implementation



Coherence 3.5 introduced a concept of partitioned backing map, which is basically a multiplexer of actual Map implementations, each of which would contain only entries that belong to the same partition.

Figure 14–3 shows a conceptual view of the partitioned backing map implementation.

Figure 14–3 Partitioned Backing Map Implementation



To configure a partitioned backing map, add a `<partitioned>` element with a value of `true`. For example:

```
<backing-map-scheme>
  <partitioned>true</partitioned>
  <external-scheme>
    <high-units>8192</high-units>
    <unit-calculator>BINARY</unit-calculator>
    <unit-factor>1048576</unit-factor>
    <nio-memory-manager>
      <initial-size>1MB</initial-size>
      <maximum-size>50MB</maximum-size>
    </nio-memory-manager>
  </external-scheme>
</backing-map-scheme>
```

This backing map is an instance of `com.tangosol.net.partition.PartitionSplittingBackingMap`, with individual partition holding maps being instances of `com.tangosol.net.cache.SerializationCache` that each store values in the extended (nio) memory. The individual nio buffers have a limit of 50MB, while the backing map as whole has a capacity limit of 8GB (8192*1048576). Again, you would need to configure a backup storage for this cache being `off-heap` or `file-mapped`.

15

Local Storage

The Coherence architecture is modular, allowing almost any piece to be extended or even replaced with a custom implementation; this includes local storage. Local storage refers to the data structures that actually store or cache the data that is managed by Coherence. For an object to provide local storage, it must support the same standard collections interface, `java.util.Map`. When a local storage implementation is used by Coherence to store replicated or distributed data, it is called a backing map, because Coherence is actually backed by that local storage implementation. The other common uses of local storage is in front of a distributed cache and as a backup behind the distributed cache.

Typically, Coherence uses one of the following local storage implementations:

- **Safe HashMap:** This is the default lossless implementation. A lossless implementation is one, like Java's Hashtable class, that is neither size-limited nor auto-expiring. In other words, it is an implementation that never evicts ("loses") cache items on its own. This particular `HashMap` implementation is optimized for extremely high thread-level concurrency. (For the default implementation, use class `com.tangosol.util.SafeHashMap`; when an implementation is required that provides cache events, use `com.tangosol.util.ObservableHashMap`. These implementations are thread-safe.)
- **Local Cache:** This is the default size-limiting and/or auto-expiring implementation. The local cache is covered in more detail below, but the primary points to remember about it are that it can limit the size of the cache, and it can automatically expire cache items after a certain period. (For the default implementation, use `com.tangosol.net.cache.LocalCache`; this implementation is thread safe and supports cache events, `com.tangosol.net.CacheLoader`, `CacheStore` and configurable/pluggable eviction policies.)
- **Read/Write Backing Map:** This is the default backing map implementation for caches that load from a database on a cache miss. It can be configured as a read-only cache (consumer model) or as either a write-through or a write-behind cache (for the consumer/producer model). The write-through and write-behind modes are intended only for use with the distributed cache service. If used with a near cache and the near cache must be kept synchronous with the distributed cache, it is possible to combine the use of this backing map with a Seppuku-based near cache (for near cache invalidation purposes); however, given these requirements, it is suggested that the versioned implementation be used. (For the default implementation, use class `com.tangosol.net.cache.ReadWriteBackingMap`.)
- **Versioned Backing Map:** This is an optimized version of the read/write backing map that optimizes its handling of the data by utilizing a data versioning

technique. For example, to invalidate near caches, it simply provides a version change notification, and to determine whether cached data must be written back to the database, it can compare the persistent (database) version information with the transient (cached) version information. The versioned implementation can provide very balanced performance in large scale clusters, both for read-intensive and write-intensive data. (For the default implementation, use class `com.tangosol.net.cache.VersionedBackingMap`; with this backing map, you can optionally use the `com.tangosol.net.cache.VersionedNearCache` as a near cache implementation.)

- **Binary Map (Java NIO):** This is a backing map implementation that can store its information in memory but outside of the Java heap, or even in memory-mapped files, which means that it does not affect the Java heap size and the related JVM garbage-collection performance that can be responsible for application pauses. This implementation is also available for distributed cache backups, which is particularly useful for read-mostly and read-only caches that require backup for high availability purposes, because it means that the backup does not affect the Java heap size yet it is immediately available in case of failover.
- **Serialization Map:** This is a backing map implementation that translates its data to a form that can be stored on disk, referred to as a serialized form. It requires a separate `com.tangosol.io.BinaryStore` object into which it stores the serialized form of the data; usually, this is the built-in LH disk store implementation, but the Serialization Map supports any custom implementation of `BinaryStore`. (For the default implementation of Serialization Map, use `com.tangosol.net.cache.SerializationMap`.)
- **Serialization Cache:** This is an extension of the SerializationMap that supports an LRU eviction policy. This can be used to limit the size of disk files, for example. (For the default implementation of Serialization Cache, use `com.tangosol.net.cache.SerializationCache`.)
- **Overflow Map:** An overflow map doesn't actually provide storage, but it deserves mention in this section because it can tie together two local storage implementations so that when the first one fills up, it will overflow into the second. (For the default implementation of `OverflowMap`, use `com.tangosol.net.cache.OverflowMap`.)

16

The Portable Object Format

The following sections are included in this chapter:

Overview

Serialization is the process of encoding an object into a binary format. It is a critical component to working with Coherence as data needs to be moved around the network. The Portable Object Format (also referred to as POF) is a language agnostic binary format. POF was designed to be incredibly efficient in both space and time and has become a cornerstone element in working with Coherence. For more information on the POF binary stream, see "The PIF-POF Binary Format" in *Oracle Coherence Developer's Guide*. See "Using the Portable Object Fromat" for details on using the POF API.

Why Should I Use POF

There are several options available with respect to serialization including standard Java serialization, POF, and your own custom serialization routines. Each has their own trade-offs. Standard Java serialization is easy to implement, supports cyclic object graphs and preserves object identity. Unfortunately, it's also comparatively slow, has a verbose binary format, and restricted to only Java objects.

The Portable Object Format on the other hand has the following advantages:

- It's language independent with current support for Java, .NET, and C++.
- It's very efficient, in a simple test class with a `String`, a `long`, and three `ints`, (de)serialization was seven times faster, and the binary produced was one sixth the size compared with standard Java serialization.
- It's versionable, objects can evolve and have forward and backward compatibility.
- It supports the ability to externalize your serialization logic.
- It's indexed which allows for extracting values without deserializing the whole object.

Working with POF

POF requires you to implement serialization routines that know how to serialize and deserialize your objects. There are two ways to do this:

- Have your objects implement the `com.tangosol.io.pof.PortableObject` interface.
- Implement a serializer for your objects using the `com.tangosol.io.pof.PofSerializer` interface.

Implementing the PortableObject interface

The `PortableObject` interface is a simple interface made up of two methods:

- `public void readExternal(PofReader reader)`
- `public void writeExternal(PofWriter writer)`

As mentioned above, POF elements are indexed. This is accomplished by providing a numerical index for each element that you write or read from the POF stream. It's important to keep in mind that the indexes must be unique to each element written and read from the POF stream, especially when you have derived types involved because the indexes must be unique between the super class and the derived class.

Example 16–1 is a simple example of implementing the `PortableObject` interface:

Example 16–1 Implementation of the PortableObject Interface

```
public void readExternal(PofReader in)
        throws IOException
    {
    m_symbol    = (Symbol) in.readObject(0);
    m_ldtPlaced = in.readLong(1);
    m_fClosed   = in.readBoolean(2);
    }

public void writeExternal(PofWriter out)
        throws IOException
    {
    out.writeObject(0, m_symbol);
    out.writeLong(1, m_ldtPlaced);
    out.writeBoolean(2, m_fClosed);
    }
```

Implementing the PofSerializer interface:

The `PofSerializer` interface provide you with a way to externalize your serialization logic from the classes you want to serialize. This is particularly useful when you don't want to change the structure of your classes to work with POF and Coherence. The `PofSerializer` interface is also made up of two methods:

- `public Object deserializer(PofReader in)`
- `public void writeObject(PofWriter out, Object o)`

As with the `PortableObject` interface, all elements written to or read from the POF stream must be uniquely indexed. Below is an example implementation of the `PofSerializer` interface:

Example 16–2 Implementation of the PofSerializer Interface

```
public Object deserialize(PofReader in)
        throws IOException
    {
    Symbol symbol    = (Symbol)in.readObject(0);
    long   ldtPlaced = in.readLong(1);
    bool   fClosed   = in.readBoolean(2);

    // mark that we're done reading the object
    in.readRemainder(null);

    return new Trade(symbol, ldtPlaced, fClosed);
    }

public void serialize(PofWriter out, Object o)
        throws IOException
    {
    Trade trade = (Trade) o;
    out.writeObject(0, trade.getSymbol());
    out.writeLong(1, trade.getTimePlaced());
    out.writeBoolean(2, trade.isClosed());

    // mark that we're done writing the object
    out.writeRemainder(null);
    }
```

Assigning POF indexes

When assigning POF indexes to your object's attributes, it's important to keep a few things in mind:

- Order your reads and writes: you should start with the lowest index value in your serialization routine and finish with the highest. When deserializing a value, you want to read in the same order you've written.
- It's ok to have non-contiguous indexes but you must read/write sequentially.
- When Subclassing reserve index ranges: index's are cumulative across derived types. As such, each derived type must be aware of the POF index range reserved by its super class.
- Don't re-purpose your indexes: if you ever want Evolvable support, it's imperative that you don't re-purpose the indexes of your attributes across class revisions.
- Label your indexes: if you label your indexes with a `public static final int`, it will be much easier to work with the object, especially when using PofExtractors and PofUpdaters. Once you've labeled your indexes, you want to still make sure that they are read and written out in order as mentioned above.

The ConfigurablePofContext

Coherence provides a `ConfigurablePofContext` class which is responsible for mapping a POF serialized object to an appropriate serialization routine (either a PofSerializer or by calling through the `PortableObject` interface). Each class is given a unique type-id in POF that can be mapped to an optional PofSerializer. Once your classes have serialization routines, they must be registered with the `ConfigurablePofContext`. Custom user types are registered with the `ConfigurablePofContext` using a POF configuration file. This is an XML file which has a `<user-type-list>` element that contains a list of classes that implement `PortableObject` or have a PofSerializer associated with them. The

type-id for each class must be unique, and must match across all cluster instances (including extend clients).

The following is an example of what a `pof-config.xml` file would look like:

```
<pof-config>
  <user-type-list>
    <include>coherence-pof-config.xml</include>
    ...
    <!-- User types must be above 1000 -->
    <user-type>
      <type-id>1001</type-id>
      <class-name>com.examples.MyTrade</class-name>
      <serializer>
        <class-name>com.examples.MyTradeSerializer</class-name>
      </serializer>
    </user-type>

    <user-type>
      <type-id>1002</type-id>
      <class-name>com.examples.MyPortableTrade</class-name>
    </user-type>
  ...
</pof-config>
```

Note: Coherence reserves the first 1000 type-id's for internal use. If you look closely you'll see that the user-type-list includes the `coherence-pof-config.xml` file. This is where the Coherence specific user types are defined and should be included in all of your POF configuration files.

Configuring Coherence to use the ConfigurablePofContext

In order to start using POF, you must also configure each service to use the `ConfigurablePofContext`. This is accomplished using the `<serializer>` element of your cache scheme in your cache configuration file. The `ConfigurablePofContext` takes a string based `<init-param>` that points to your pof-configuration file.

Below is an example of a distributed cache scheme configured to use POF:

```
 <distributed-scheme>
    <scheme-name>example-distributed</scheme-name>
    <service-name>DistributedCache</service-name>
    <serializer>
      <class-name>com.tangosol.io.pof.ConfigurablePofContext</class-name>
      <init-params>
        <init-param>
          <param-value>my-pof-config.xml</param-value>
          <param-type>String</param-type>
        </init-param>
      </init-params>
    </serializer>
    ...
 </distributed-scheme>
```

Alternatively you can configure an entire JVM instance to use POF using the following system properties:

- `tangosol.pof.enabled=true` - This will turn on POF for the entire JVM instance.
- `tangosol.pof.config=`*`CONFIG_FILE_PATH`* - The path to the POF configuration file you want to use. Note that if this is not in the class path it must be presented as a file resource (for example `file:///opt/home/coherence/mycustom-pof-config.xml`).

Summary

Using POF has many advantages ranging from performance benefits to language independence. It's recommended that you look closely at POF as your serialization solution when working with Coherence. For information on how to work with POF in C++, see "Building Integration Objects for C++ Clients" in *Oracle Coherence Client Guide*. For information on how to work with POF in .NET, see "Building Integration Objects for .NET Clients" in *Oracle Coherence Client Guide*.

17

Coherence*Extend

Coherence*Extend extends the reach of the core Coherence TCMP cluster to a wider range of consumers, including desktops, remote servers and machines located across WAN connections. It also provides a wider range of language support including .NET and C++ clients. Typical uses of Coherence*Extend include providing desktop applications with access to Coherence caches (including support for Near Cache and Continuous Query) and Coherence cluster "bridges" that link together multiple Coherence clusters connected through a high-latency, unreliable WAN.

Coherence*Extend consists of two basic components: a client running outside the cluster, and a proxy service running in the cluster. The client API includes implementations of both the `CacheService` and `InvocationService` interfaces which route all requests to a proxy running within the Coherence cluster. The proxy service in turn responds to client requests by delegating to an actual Coherence clustered service (for example, a Partitioned or Replicated cache service).

There are three Coherence*Extend clients available:

- Java
- .NET
- C++

See *Oracle Coherence Client Guide* for detailed information on setting up Coherence*Extend and creating clients.

Types of Clients

Coherence*Extend clients provide the same rich API support as the standard Coherence API without being full data members of the cluster. There are two categories of clients:

- Data client—The base client that allows for:
  - Key-based cache access through the `NamedCache` interface
  - Attribute-based cache access using Filters
  - Custom processing and aggregation of cluster side entries using the `InvocableMap` interface
  - In process caching through `LocalCache`
  - Remote invocation of custom tasks in the cluster through the Invocation Service
- Real Time Client (described in Chapter 18, "Real Time Client—RTC")

Proxy Service Overview

The proxy service is responsible for dispatching requests from Extend clients to actual clustered services and returning the results of these requests to the appropriate client. It is hosted by one or more `DefaultCacheServer` processes running within a cluster. Clients communicate with a proxy service using a low-level messaging protocol that has TCP/IP transport binding, Extend-TCP which uses a high performance, scalable TCP/IP-based communication layer to connect to the cluster. This protocol is supported by all available clients.

18

Real Time Client—RTC

The Coherence Real Time Client provides secure and scalable access from client applications into a Coherence Data Grid. Coherence RTC extends the Data Grid to the desktop, providing the same core API as the rest of the Coherence product line.

Connectivity into the Coherence Data Grid is achieved through Coherence*Extend technology, which enables a client application to connect to a particular server within the Data Grid. Since the connections are load-balanced across all of the servers in the Data Grid, this approach to connectivity can scale to support tens of thousands of clients. See *Configuring and Using Coherence*Extend* for more information.

Uses

The primary use for Coherence RTC is to provide clients with read-only/read-mostly access to data held in a Coherence cluster. Clients can query clustered caches and receive real-time updates as the data changes. Clients may also initiate server-side data manipulation tasks, including aggregations and processing. For more information on aggregations and processing, see the following API:

- Java: `com.tangosol.util.InvocableMap.EntryAggregator` and `com.tangosol.util.InvocableMap.EntryProcessor`
- C++: `coherence::util::InvocableMap::EntryAggregator` and `coherence::util::InvocableMap::EntryProcessor`

Cache Access

Normally, client applications are granted only read access to the data being managed by the Data Grid (delegating cache updates to Data Grid Agents), although it is possible to enable direct read/write access.

Local Caches

While the client application can directly access the caches managed by the Data Grid, that may be inefficient depending on the network infrastructure. For efficiency, the client application can use both Near Caching (Java, C++, or .NET) and Continuous Query Caching (Java, C++, or .NET) to maintain cache data locally.

For more information on near caching and continuous query, see:

- Java: "Near Cache", *"Perform Continuous Query"*
- C++: *"Configuring a Near Cache for C++ Clients"*, *"Perform Continuous Query for C++ Clients"*

- .NET: *"Configuring a Near Cache for .NET Clients"*, *"Continuous Query Cache for .NET Clients"*

Event Notification

Using the standard Coherence event model, data changes that occur within the Data Grid are visible to the client application. Since the client application indicates the exact events that it is interested in, only those events are actually delivered over the wire, resulting in efficient use of network bandwidth and client processing.

For more information on the event model, see: "Using Cache Events" in *Oracle Coherence Developer's Guide*.

Agent Invocation

Since the client application will likely only have read-only access, any manipulation of data is done within the Data Grid itself; the mechanism for this is the Data Grid Agent, which is presented by these API:.

- Java: `com.tangosol.util.InvocableMap`
- C++: `coherence::util::InvocableMap`

Clients may invoke tasks, aggregators and processors for server-side cached objects using the InvocableMap methods.

For more information on the Data Grid Agent, see Chapter 2, "Provide a Data Grid".

Connection Failover

If the server to which the client application is attached happens to fail, the connection is automatically reestablished to another server, and then any locally cached data is re-synced with the cluster.

19

Session Management for Clustered Applications

Clustered applications require reliable and performant HTTP session management. Unfortunately, moving to a clustered environment introduces several challenges for session management. This article discusses those challenges and proposes solutions and recommended practices. The included session management features of Oracle Coherence*Web are examined here.

See *Oracle Coherence User's Guide for Oracle Coherence*Web* for detailed information on using Coherence*Web.

Basic Terminology

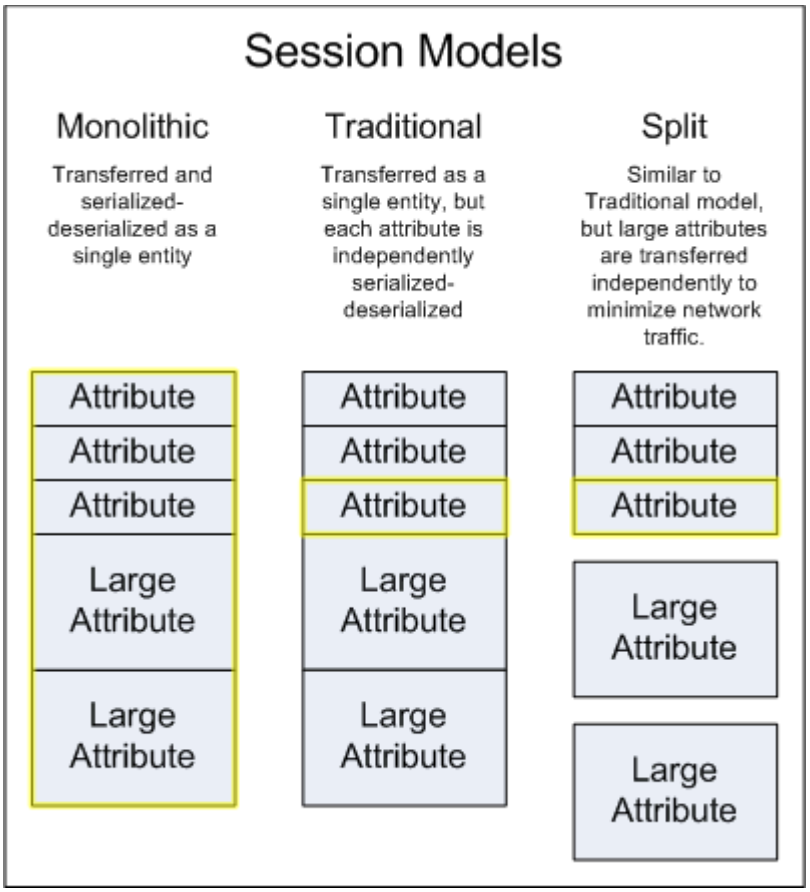
An HTTP session ("session") spans a sequence of user interactions within a Web application. "Session state" is a collection of user-specific information. This session state is maintained for a period, typically beginning with the user's first interaction and ending a short while after the user's last interaction, perhaps thirty minutes later. Session state consists of an arbitrary collection of "session attributes," each of which is a Java object and is identified by name. "Sticky load balancing" describes the act of distributing user requests across a set of servers in such a way that requests from a given user are consistently sent to the same server.

Coherence is a data management product that provides real-time, fully coherent data sharing for clustered applications. Coherence*Web is a session management module that is included as part of Coherence. An HTTP session model ("session model") describes how Coherence*Web physically represents session state. Coherence*Web includes three session models. The Monolithic model stores all session state as a single entity, serializing and deserializing all attributes as a single operation. The Traditional model stores all session state as a single entity but serializes and deserializes attributes individually. The Split model extends the Traditional model but separates the larger session attributes into independent physical entities. The applications of these models are described in later sections of this article.

"*Select the Appropriate Session Model*" in the Coherence FAQ provides more information on the Monolithic, Traditional, and Split Session models. It also describes how to configure Coherence to use a particular model.

Figure 19–1 illustrates the Monolithic, Traditional, and Split Session models.

Figure 19–1 Session Models Supported by Coherence



Sharing Data in a Clustered Environment

Session attributes must be serializable if they are to be processed across multiple JVMs, which is a requirement for clustering. It is possible to make some fields of a session attribute non-clustered by declaring those fields as transient. While this eliminates the requirement for all fields of the session attributes to be serializable, it also means that these attributes will not be fully replicated to the backup server(s). Developers who follow this approach should be very careful to ensure that their applications are capable of operating in a consistent manner even if these attribute fields are lost. In most cases, this approach ends up being more difficult than simply converting all session attributes to serializable objects. However, it can be a useful pattern when very large amounts of user-specific data are cached in a session.

The J2EE Servlet specification (versions 2.2, 2.3, and 2.4) states that the servlet context should not be shared across the cluster. Non-clustered applications that rely on the servlet context as a singleton data structure will have porting issues when moving to a clustered environment. Coherence*Web does support the option of a clustered context, though generally it should be the goal of all development teams to ensure that their applications follow the J2EE specifications.

A more subtle issue that arises in clustered environments is the issue of object sharing. In a non-clustered application, if two session attributes reference a common object, changes to the shared object will be visible as part of both session attributes. However, this is not the case in most clustered applications. To avoid unnecessary use of compute resources, most session management implementations serialize and

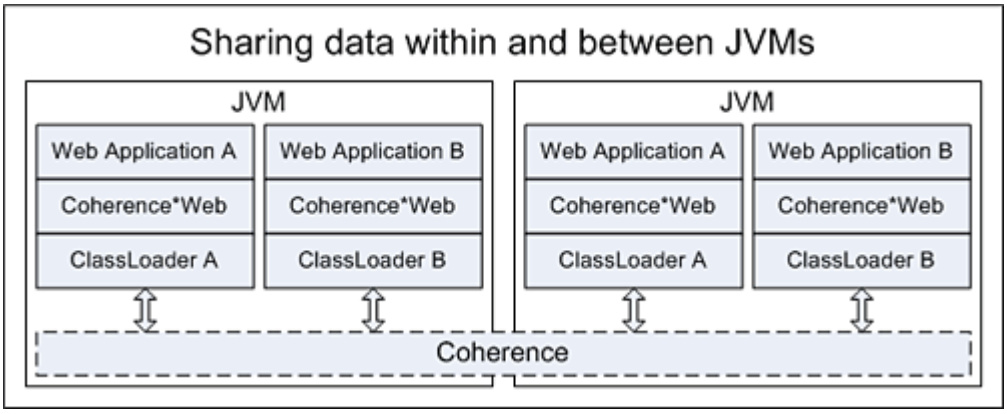
deserialize session attributes individually on demand. Coherence*Web (Traditional and Split session models) normally operates in this manner. If two session attributes that reference a common object are separately deserialized, the shared common object will be instantiated twice. For applications that depend on shared object behavior and cannot be readily corrected, Coherence*Web provides the option of a Monolithic session model, which serializes and deserializes the entire session object as a single operation. This provides compatibility for applications that were not originally designed with clustering in mind.

Many projects require sharing session data between different Web applications. The challenge that arises is that each Web application typically has its own class loader. Consequently, objects cannot readily be shared between separate Web applications. There are two general methods for working around this, each with its own set of trade-offs.

- Place common classes in the Java CLASSPATH, allowing multiple applications to share instances of those classes at the expense of a slightly more complicated configuration.
- Use Coherence*Web to share session data across class loader boundaries. Each Web application is treated as a separate cluster member, even if they run within the same JVM. This approach provides looser coupling between Web applications (assuming serialized classes share a common serial Version UID), but suffers from a performance impact because objects must be serialized-deserialized for transfer between cluster members.

Figure 19–2 illustrates the sharing of data between Web applications or portlets by clustering (serializing-deserializing session state).

Figure 19–2 Sharing Data Between Web Applications



Reliability and Availability

An application must guarantee that a user's session state is properly maintained to exhibit correct behavior for that user. Some availability considerations occur at the application design level and apply to both clustered and non-clustered applications. For example, the application should ensure that user actions are idempotent: the application should be capable of handling a user who accidentally submits an HTML form twice.

With sticky load balancing, issues related to concurrent session updates are normally avoided, as all updates to session state are made from a single server (which dramatically simplifies concurrency management). This has the benefit of ensuring no

overlap of user requests occurs even in cases where a user submits a new request before the previous request has been fully processed. Use of HTML frames complicates this, but the same general pattern applies: Simply ensure that only one display element is modifying session state.

In cases where there may be concurrent requests, Coherence*Web manages concurrent changes to session state (even across multiple servers) by locking sessions for exclusive access by a single server. With Coherence*Web, developers can specify whether session access is restricted to one server at a time (the default), or even one thread at a time.

As a general rule, all session attributes should be treated as immutable objects if possible. This ensures that developers are consciously aware when they change attributes. With mutable objects, modifying attributes often requires two steps: modifying the state of the attribute object, and then manually updating the session with the modified attribute object by calling `javax.servlet.http.HttpSession.setAttribute()`. This means that your application should always call `setAttribute()` if the attribute value has been changed, otherwise, the modified attribute value will not replicate to the backup server. Coherence*Web tracks all mutable attributes retrieved from the session, and so will automatically update these attributes, even if `setAttribute()` has not been called. This can help applications that were not designed for clustering to work in a clustered environment.

Session state is normally maintained on two servers, one primary and one backup. A sticky load balancer will send each user request to the specified primary server, and any local changes to session state will be copied to the backup server. If the primary server fails, the next request will be rerouted to the backup server, and the user's session state will be unaffected. While this is a very efficient approach (among other things, it ensures that the cluster is not overwhelmed with replication activity after a server failure), there are a few drawbacks. Because session state is copied when the session is updated, failure (or cycling) of both the primary and backup servers between session updates will result in a loss of session state. To avoid this problem, wait thirty minutes between each server restart when cycling a cluster of server instances. The thirty-minute interval increases the odds of a return visit from a user, which can trigger session replication. Additionally, if the interval is at least as long as the session timeout, the session state will be discarded anyway if the user has not returned.

This cycling interval is not required with Coherence*Web, which will automatically redistribute session data when a server fails or is cycled. Coherence's "location transparency" ensures that node failure does not affect data visibility. However, node failure does impact redundancy, and therefore fresh backup copies must be created. With most Coherence*Web configurations, two machines (primary and backup) are responsible for managing each piece of session data, regardless of cluster size. With this configuration, Coherence can handle one failover transition at any time. When a server fails, no data will be lost if the next server failure occurs after the completion of the current failover process. The worst-case scenario is a small cluster with large amounts of session data on each server, which may require a minute or two to rebalance. Increasing the cluster size, or reducing the amount of data storage per server, will improve failover performance. In a large cluster of commodity servers, the failover process may require less than a second to complete. For particularly critical applications, increasing the number of backup machines will increase the number of simultaneous failures that Coherence can manage.

The need for serialization in clustered applications introduces a new opportunity for failure. Serialization failure of a single session attribute will ordinarily prevent the remaining session attributes from being copied to the backup server and can result in

the loss of the entire session. Coherence*Web works around this by replicating only serializable objects, while maintaining non-serializable objects in the local server instance.

One last issue to be aware of is that under heavy load, a server can lose session attribute modifications due to network congestion. The log will contain information about lost attributes, which brings up the most critical aspect of high-availability planning: Be sure to test all of your application components under full load to ensure that failover and failback operate as expected. While many applications will see no difficulties even at 99-percent load, the real test of application availability occurs when the system is fully saturated.

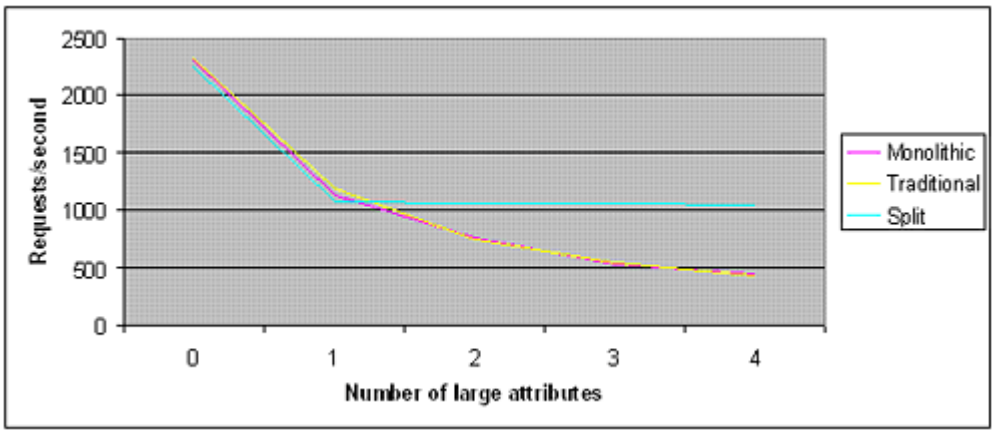
Scalability and Performance

Moving to a clustered environment makes session size a critical consideration. Memory usage is a factor regardless of whether an application is clustered or not, but clustered applications also need to consider the increased CPU and network load that larger sessions introduce. While non-clustered applications using in-memory sessions do not need to serialize-deserialize session state, clustered applications must do this every time session state is updated. Serializing session state and then transmitting it over the network becomes a critical factor in application performance. For this reason and others, a server should generally limit session size to no more than a few kilobytes.

While the Traditional and Monolithic session models for Coherence*Web have the same limiting factor, the Split session model was explicitly designed to efficiently support large HTTP sessions. Using a single clustered cache entry to contain all of the small session attributes means that network traffic is minimized when accessing and updating the session or any of its smaller attributes. Independently deserializing each attribute means that CPU usage is minimized. By splitting out larger session attributes into separate clustered cache entries, Coherence*Web ensures that the application only pays the cost for those attributes when they are actually accessed or updated. Additionally, because Coherence*Web leverages the data management features of Coherence, all of the underlying features are available for managing session attributes, such as near caching, NIO buffer caching, and disk-based overflow.

Figure 19–3 illustrates performance as a function of session size. Each session consists of ten 10-character Strings and from zero to four 10,000-character Strings. Each HTTP request reads a single small attribute and a single large attribute (for cases where there are any in the session), and 50 percent of requests update those attributes. Tests were performed on a two-server cluster. Note the similar performance between the Traditional and Monolithic models; serializing-deserializing Strings consumes minimal CPU resources, so there is little performance gain from deserializing only the attributes that are actually used. The performance gain of the Split model increases to over 37:1 by the time session size reaches one megabyte (100 large Strings). In a clustered environment, it is particularly true that application requests that access only essential data have the opportunity to scale and perform better; this is part of the reason that sessions should be kept to a reasonable size.

Figure 19–3 Performance as a Function of Session Size



Another optimization is the use of transient data members in session attribute classes. Because Java serialization routines ignore transient fields, they provide a very convenient means of controlling whether session attributes are clustered or isolated to a single cluster member. These are useful in situations where data can be "lazy loaded" from other data sources (and therefore recalculated in the event of a server failover process), and also in scenarios where absolute reliability is not critical. If an application can withstand the loss of a portion of its session state with zero (or acceptably minimal) impact on the user, then the performance benefit may be worth considering. In a similar vein, it is not uncommon for high-scale applications to treat session loss as a session timeout, requiring the user to log back in to the application (which has the implicit benefit of properly setting user expectations regarding the state of their application session).

Sticky load balancing plays a critical role because session state is not globally visible across the cluster. For high-scale clusters, user requests normally enter the application tier through a set of stateless load balancers, which redistribute (more or less randomly) these requests across a set of sticky load balancers, such as Microsoft IIS or Apache HTTP Server. These sticky load balancers are responsible for the more computationally intense act of parsing the HTTP headers to determine which server instance will be processing the request (based on the server ID specified by the session cookie). If requests are misrouted for any reason, session integrity will be lost. For example, some load balancers may not parse HTTP headers for requests with large amounts of POST data (for example, more than 64KB), so these requests will not be routed to the appropriate server instance. Other causes of routing failure include corrupted or malformed server IDs in the session cookie. Most of these issues can be handled with proper selection of a load balancer and designing tolerance into the application whenever possible (for example, ensuring that all large POST requests avoid accessing or modifying session state).

Sticky load balancing aids the performance of Coherence*Web but is not required. Because Coherence*Web is built on the Coherence data management platform, all session data is globally visible across the cluster. A typical Coherence*Web deployment places session data in a near cache topology, which uses a partitioned cache to manage huge amounts of data in a scalable and fault-tolerant manner, combined with local caches in each application server JVM to provide instant access to commonly used session state. While a sticky load balancer is not required when Coherence*Web is used, there are two key benefits to using one. Due to the use of near cache technology, read access to session attributes will be instant if user requests are consistently routed to the same server, as using the local cache avoids the cost of deserialization and network transfer of session attributes. Additionally, sticky load

balancing allows Coherence to manage concurrency locally, transferring session locks only when a user request is rebalanced to another server.

Conclusion

Clustering can boost scalability and availability for applications. Clustering solutions such as Coherence*Web solve many problems for developers, but successful developers must be aware of the limitations of the underlying technology, and how to manage those limitations. Understanding what the platform provides, and what users require, gives developers the ability to eliminate the gap between the two.

20

The Coherence Ecosystem

The purpose of this document is to describe the:

- Breakdown of Coherence editions
- Coherence Client and Server Connections
- Coherence Modules Involved in Connecting Client and Server Editions

Breakdown of Coherence editions

The Coherence ecosystem is divided into two subsections: Coherence client editions and Coherence server editions. There are two different client editions:

- Data Client
- Real Time Client—RTC

And there are three different server editions:

- Standard Edition
- Enterprise Edition
- Grid Edition

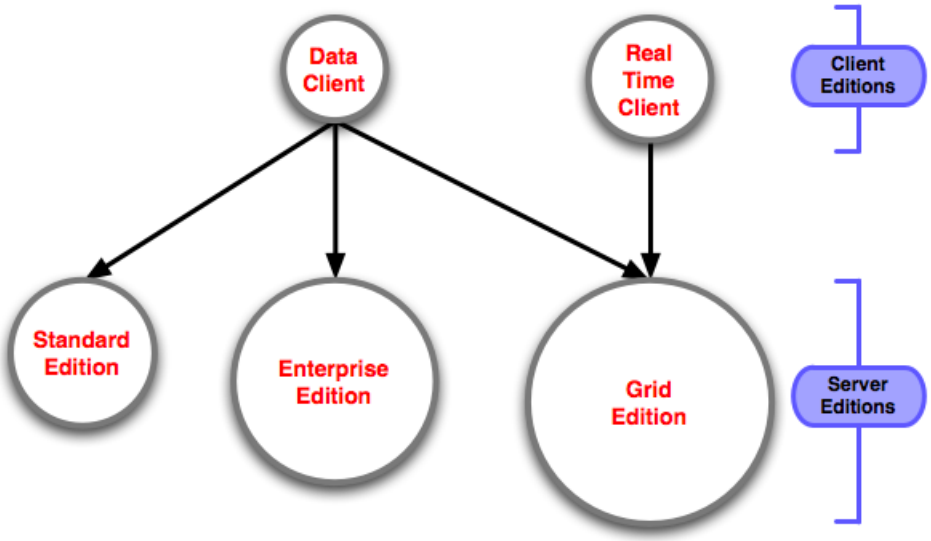
Each edition has a different (graduated) list of features which can be found in "Oracle Coherence" in the *Oracle Fusion Middleware Licensing Information* book.

Coherence Client and Server Connections

Figure 20–1 illustrates which client editions can connect to which server editions. It illustrates two important points:

- Coherence Data Client can connect **to any** of the Coherence Server Editions.
- Coherence Real Time Client can **only** connect to Coherence Grid Edition.

Figure 20–1 Client/Server Features by Edition



Coherence Modules Involved in Connecting Client and Server Editions

There are two Coherence modules involved in connecting Coherence client and server editions:

- Coherence*Extend—which is the protocol (built on TCP/IP) that is used between the Coherence client and server processes.
- Coherence*Extend TCP Proxy—which is the module that sits within a Coherence server edition process that manages the Coherence Extend*TCP connections coming in from the clients.

How a Single Coherence Client Process Connects to a Single Coherence Server

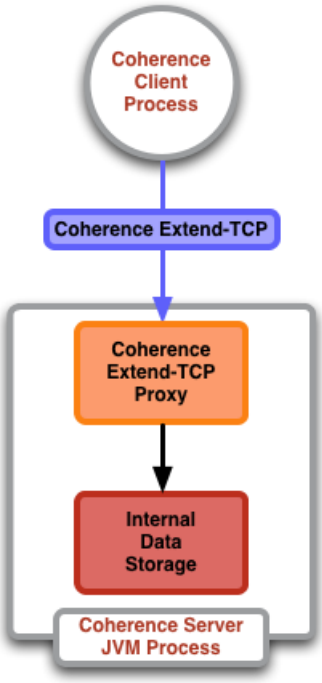
---

Note: Coherence provides cross-platform client support by providing **native** clients for Java, .NET (C#), and C++. This allows different platforms to access, modify, query, (and so on), data between programming languages by connecting to the Coherence data grid. For more information on a data grid, see Chapter 1, "Defining a Data Grid".

Figure 20–2 illustrates how a request is passed from a Coherence client process to internal data storage:

1. A Coherence client process uses Extend*TCP to initiate and maintain a connection to the Coherence (server-side) Data Grid.
2. Extend*TCP connects to the Coherence Server JVM process, specifically the Extend*Proxy service that is running within the process space.
3. The Extend*Proxy service routes the client requests to the internal data storage.

Figure 20–2 Single Client, Single Server

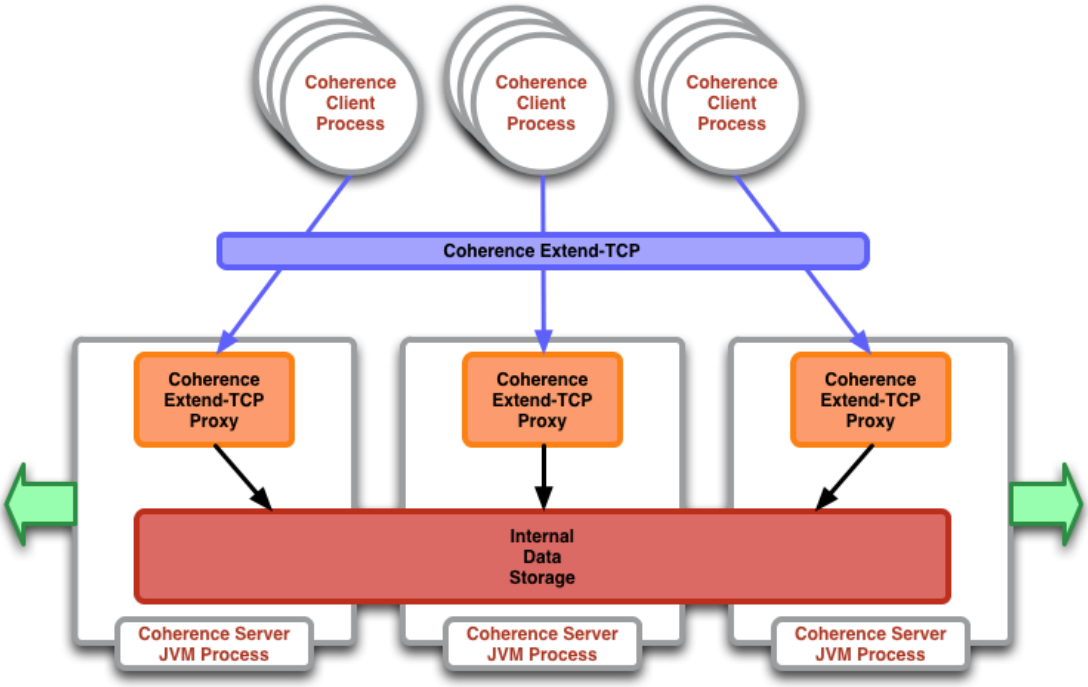


Considering Multiple Clients and Servers

This section assumes that the components involved remain the same as in the previous example. Figure 20–3 illustrates how requests are passed from the client processes to internal data storage where there are multiple Coherence server JVM processes:

1. The Coherence client **processes** use Extend*TCP to initiate and maintain a connection to the Coherence (server-side) Data Grid.
2. Extend*TCP connects to **a single** Coherence server JVM process, specifically the Extend*Proxy service that is running within the process space. In the event of failure the client process will connect to another Coherence server JVM process that is running the Extend*Proxy service.
3. The Extend*Proxy service then routes the client requests to the correct Coherence server JVM process (that is, the process or processes that hold the data relevant to the client request) represented by the "internal data storage" in the diagram.

Figure 20–3 Multiple Clients and Servers



Glossary

Coherence Clustering and Federation

For most Coherence deployments, there will be a single cluster in each data center (through TCMP), and federation between each data center (through Coherence*Extend). While TCMP scalability is dependent on many variables, a good rule of thumb is that with solid networking infrastructure, a cluster of 100 server JVMs is readily supported, and a cluster of 1000 server JVMs is possible but requires far more care to achieve.

Oracle Coherence supports both homogeneous server clusters and the federated server model. Any application or server process that is running the Coherence software is called a cluster node. All cluster nodes on the same network will automatically cluster together. Cluster nodes use a peer-to-peer protocol, which means that any cluster node can talk directly to any other cluster node.

Coherence is logically sub-divided into clusters, services and caches. A Coherence cluster is a group of cluster nodes that share a group address, which allows the cluster nodes to communicate. Generally, a cluster node will only join one cluster, but it is possible for a cluster node to join (be a member of) several different clusters, by using a different group address for each cluster.

Within a cluster, there exists any number of named services. A cluster node can participate in (join) any number of these services; when a cluster node joins a service, it automatically has all of the information from that service available to it; for example, if the service is a replicated cache service, then joining the service includes replicating the data of all the caches in the service. These services are all peer-to-peer, which means that a cluster node typically plays both the client and the server role through the service; furthermore, all of these services will failover in the event of cluster node failure without any data loss.

Failback

Failback is an extension to failover that allows a server to reclaim its responsibilities when it restarts. For example, "When the server came back up, the processes that it was running previously were failed back to it."

Failover

Failover refers to the ability of a server to assume the responsibilities of a failed server. For example, "When the server died, its processes failed over to the backup server."

JCache

JCache (also known as JSR-107), is a caching API specification that is currently in progress. While the final version of this API has not been released yet, Oracle and

other companies with caching products have been tracking the current status of the API. The API has been largely solidified at this point. Few significant changes are expected going forward.

The .Net and C++ platforms do not have a corresponding multi-vendor standard for data caching.

Load Balancer

A load balancer is a hardware device or software program that delegates network requests to several servers, such as in a server farm or server cluster. Load balancers typically can detect server failure and optionally retry operations that were being processed by that server at the time of its failure. Load balancers typically attempt to keep the servers to which they delegate equally busy, hence the use of the term "balancer". Load balancer devices often have a high-availability option that uses a second load balancer, allowing one of the load balancer devices to die without affecting availability.

Server Cluster

A server cluster is composed of multiple servers that are mutually aware of each other. Because of this, the servers can communicate directly with each other, safely share responsibilities, and are able to assume the responsibilities failed servers. This simplifies development because there is no longer any need to use asynchronous messaging (which may require idempotent and/or compensating transactions) or synchronous two-phase commits (which may block indefinitely and reduce system availability).

Due to the need for global coordination, clustering scales to a lesser degree than federation, but generally with much stronger data integrity guarantees.

Server Farm

The loosest form of coupling, a server farm uses multiple servers to handle increased load and provide increased availability. It is common for a load-balancer to be used to assign work to the various servers in the server farm, and server farms often share back-end resources, such as database servers, but each server is typically unaware of other servers in the farm, and usually the load-balancer is responsible for failover.

Farms are commonly used for stateless services such as delivering static web content or performing high volumes of compute-intensive operations for high-performance computing (HPC).

Server Federation

A federated server model is similar to a server farm, but allows multiple servers to work together even if they were not originally intended to do so. Federation may operate synchronously through technologies such as distributed XA transactions, or asynchronously through messaging solutions such as JMS. Federation may be used to scale out (for example, with federated databases, data can be "partitioned" or "shared" across multiple database instances). Federation may also be used to integrate between heterogeneous systems (for example, sharing data between applications within a web portal).

In the context of scale-out, federated systems are primarily used when data integrity is important but not absolutely crucial, such as for many stateful web applications or transactional compute grids. In the former case, HTTP sessions are generally require only a "best effort" guarantee. In the latter case, there is usually an external system of record that ensures data integrity (by rolling back any invalid transactions that the compute grid submits).

Federation supports enormous scale at the cost of data integrity guarantees (though such guarantees can be reinstated with the elimination of transparent failover/failback and dynamic partitioning, which is the case for most WAN-style deployments).