Coherence Developer's Guide

13 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* (../../coh.360/e15830/toc.htm), *Toplink* Essentials (../../coh.360/e15830/toc.htm), *JPA* (../../coh.360/e15830/toc.htm), 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.

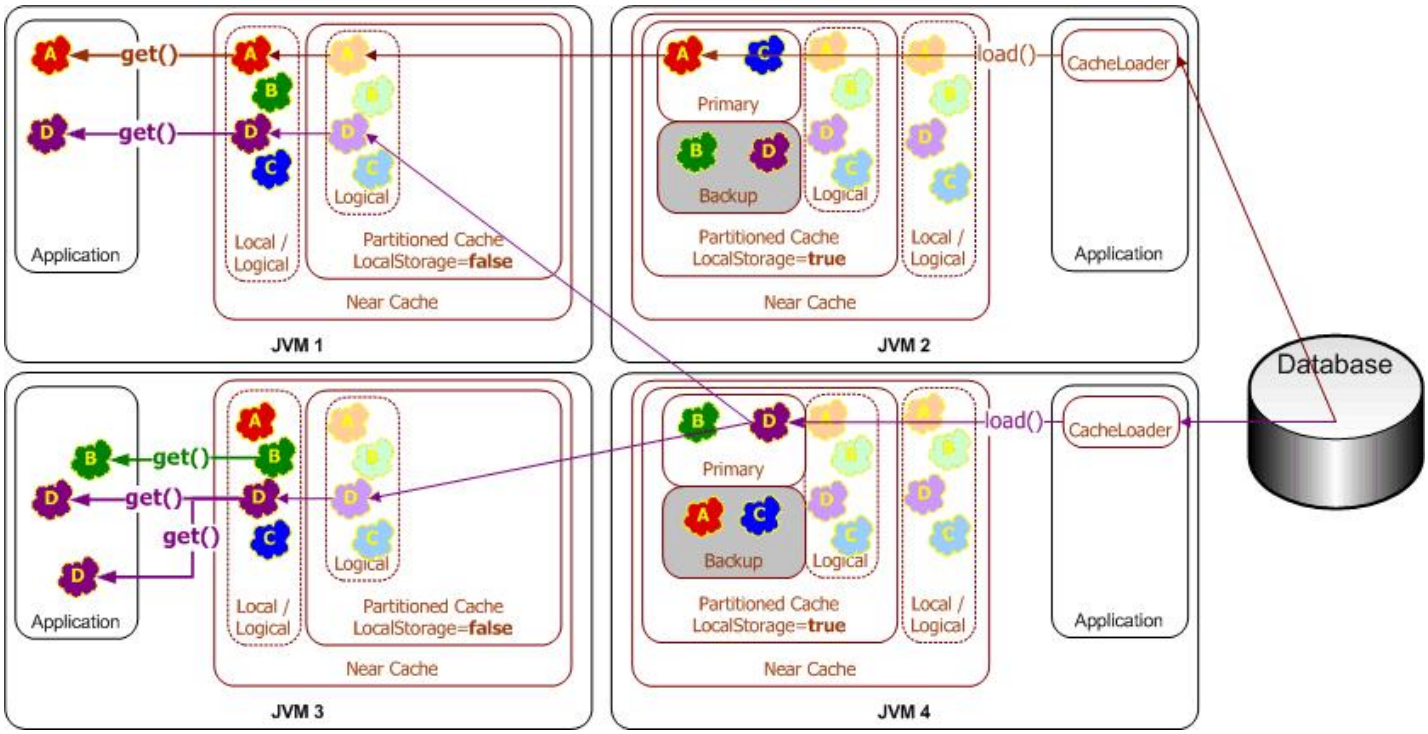
0

0

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.

0 0 ***Figure 13-1 Read-Through Caching***



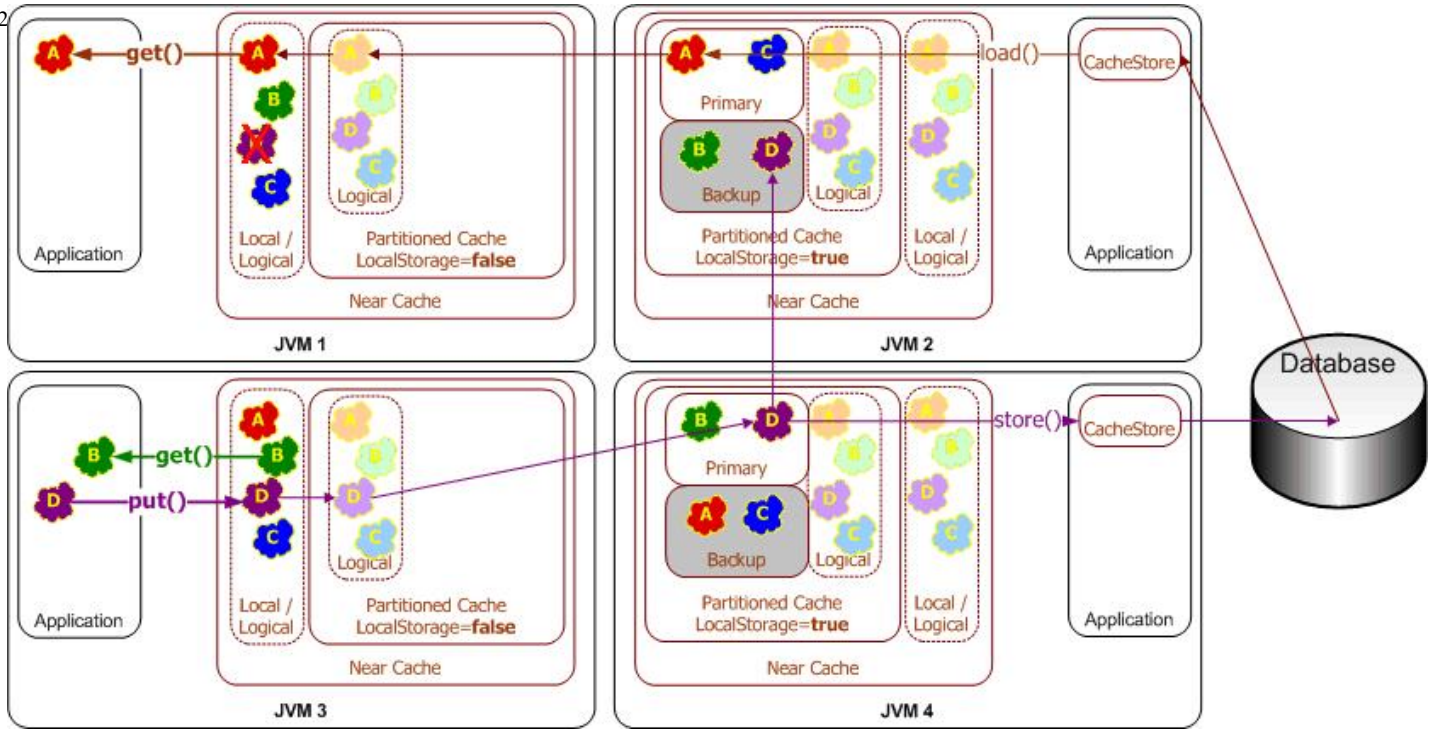
0

0

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.

0 0 ***Figure 13-2 Write-Through Caching***



0 0

Write-Behind Caching

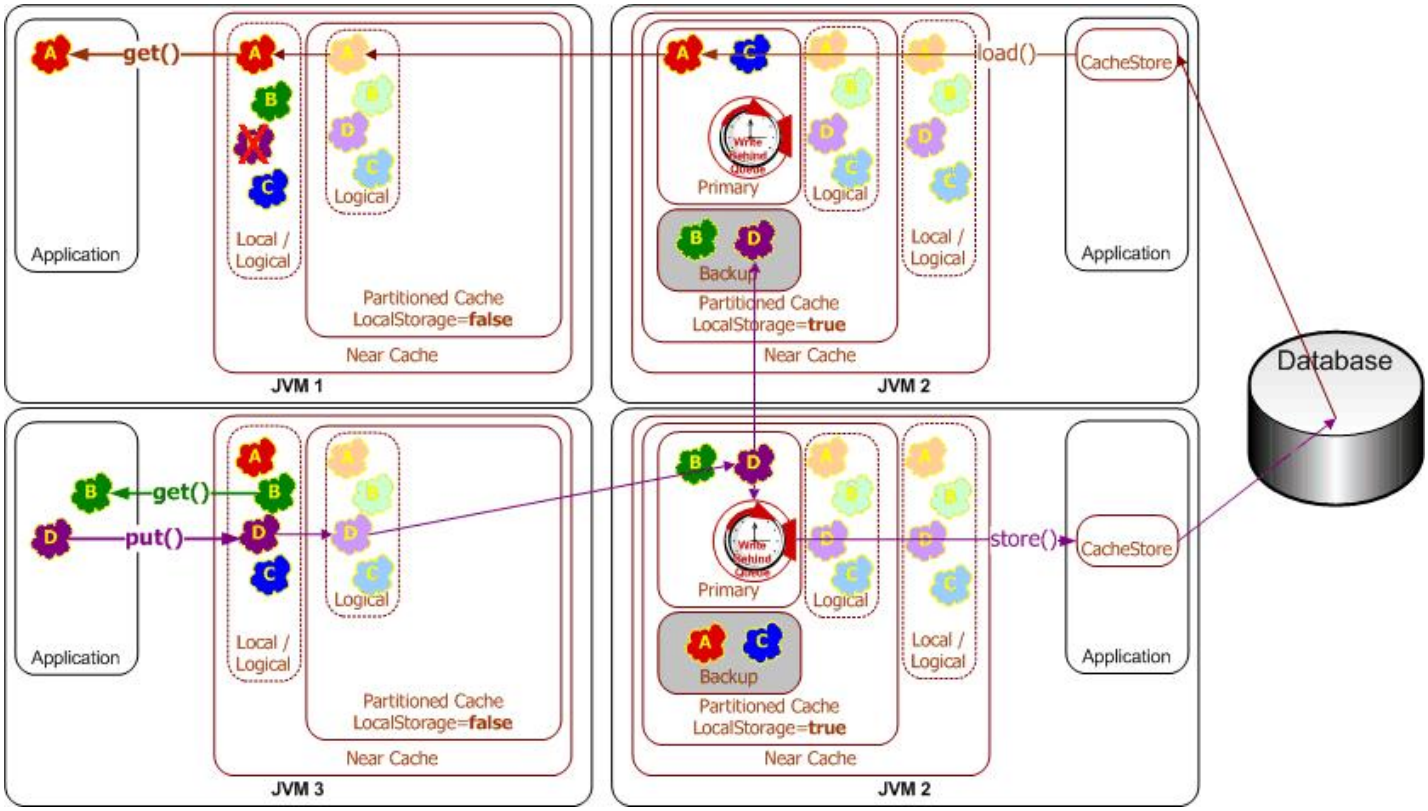
In the **Write-Behind** scenario, modified cache entries are asynchronously written to the data source after a configured delay, whether after 10 seconds, 20 minutes, a day, a week or even longer. Note that this only applies to cache inserts and updates - cache entries are removed synchronously from the data source. 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()`** (../../coh.360/e18814/toc.htm) 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 13-3 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`** (../../coh.360/e18814/toc.htm). 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`**> (../../coh.360/e15723/appendix_cacheconfig.htm#COHDG367) element in the **`coherence-cache-config.xml`** file. Refresh-ahead assumes that you have also set an expiration time (**`<expiry-delay>`**) for entries in the cache.

The XML code fragment in Example 13-1 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 13-1 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-
            </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

0 0

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.

0 0

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 inaccurate prediction, 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.

0 0

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).

0 0

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`** (../../coh.360/e18814/toc.htm) for read-only caches
- **`CacheStore`** (../../coh.360/e18814/toc.htm) 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)** (../../coh.360/e18814/toc.htm) and **loadAll(Collection keys)** (../../coh.360/e18814/toc.htm), and the **CacheStore** interface adds the methods **store(Object key, Object value)** (../../coh.360/e18814/toc.htm), **storeAll(Map mapEntries)** (../../coh.360/e18814/toc.htm), **erase(Object key)** (../../coh.360/e18814/toc.htm), and **eraseAll(Collection colKeys)** (../../coh.360/e18814/toc.htm).

See "Sample CacheStore" and "Sample Controllable CacheStore" for example **CacheStore** implementations.

()
()

Plugging in a CacheStore Implementation

To plug in a **CacheStore** module, specify the **CacheStore** implementation class name within the **distributed-scheme** (../../coh.360/e15723/appendix_cacheconfig.htm#COHDG345), **backing-map-scheme** (../../coh.360/e15723/appendix_cacheconfig.htm#COHDG3451), **cachestore-scheme** (../../coh.360/e15723/appendix_cacheconfig.htm#COHDG342), or **read-write-backing-map-scheme** (../../coh.360/e15723/appendix_cacheconfig.htm#COHDG367), cache configuration element.

The **read-write-backing-map-scheme** (../../coh.360/e15723/appendix_cacheconfig.htm#COHDG367) configures a **com.tangosol.net.cache.ReadWriteBackingMap** (../../coh.360/e18814/toc.htm). 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** (../../coh.360/e15723/appendix_cacheconfig.htm#COHDG342)).

Example 13-2 illustrates a cache configuration that specifies a **CacheStore** module. The **<init-params>** (../../coh.360/e15723/manage_modreportbatch.htm#COHDG291) 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 "Using Parameter Macros" (cache_config.htm#BABHCCHI).

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 13-2 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 (../../coh.360/e15723/appendix_operational.htm#COHDG4441) 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).

Sample CacheStore

This section provides a very basic implementation of the **`com.tangosol.net.cache.CacheStore`** (../../coh.360/e18814/toc.htm) interface. The implementation in Example 13-3 uses a single database connection by using JDBC, and does not use bulk operations. A complete implementation would use a connection pool, and, if write-behind is used, implement **`CacheStore.storeAll()`** (../../coh.360/e18814/toc.htm) for bulk JDBC inserts and updates. "Cache of a Database" (cache_examples.htm#BACGHIFE) provides an example of a database cache configuration.

> **Tip:**
>
> Save processing effort by bulk loading the cache. The following example use the `put` method to write values to the cache store. Often, performing bulk loads with the `putAll` method will result in a savings in processing effort and network traffic. For more information on bulk loading, see Chapter 18, "Pre-Loading the Cache." (api_preloadcache.htm#CACCFCFJ)

Example 13-3 Implementation of the CacheStore Interface

```
package com.tangosol.examples.coherence;


import com.tangosol.net.cache.CacheStore;
import com.tangosol.util.Base;

import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import java.util.Collection;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;


/**
* An example implementation of CacheStore
* interface.
*
* @author erm 2003.05.01
*/
public class DBCacheStore
        extends Base
        implements CacheStore
    {
    // ----- constructors ---------------------------------------------------
    /**
    * Constructs DBCacheStore for a given database table.
    *
    * @param sTableName the db table name
    */
    public DBCacheStore(String sTableName)
        {
        m_sTableName = sTableName;
        configureConnection();
        }

        /**
        * Set up the DB connection.
        */
        protected void configureConnection()
                {
                try
                        {
                        Class.forName("org.gjt.mm.mysql.Driver");
                        m_con = DriverManager.getConnection(DB_URL, DB_USERNAME, DB_PASSWO
```

```
                    m_con.setAutoCommit(true);
                    }
            catch (Exception e)
                    {
                    throw ensureRuntimeException(e, "Connection failed");
                    }
            }


    // ---- accessors -------------------------------------------------------

/**
* Obtain the name of the table this CacheStore is persisting to.
*
* @return the name of the table this CacheStore is persisting to
*/
public String getTableName()
    {
    return m_sTableName;
    }

/**
* Obtain the connection being used to connect to the database.
*
* @return the connection used to connect to the database
*/
public Connection getConnection()
    {
    return m_con;
    }


// ----- CacheStore Interface --------------------------------------------

/**
* Return the value associated with the specified key, or null if the
* key does not have an associated value in the underlying store.
*
* @param oKey  key whose associated value is to be returned
*
* @return the value associated with the specified key, or
*         <tt>null</tt> if no value is available for that key
*/
public Object load(Object oKey)
    {
    Object     oValue = null;
    Connection con    = getConnection();
    String     sSQL   = "SELECT id, value FROM " + getTableName()
                      + " WHERE id = ?";
    try
        {
```

```
            PreparedStatement stmt = con.prepareStatement(sSQL);

            stmt.setString(1, String.valueOf(oKey));

            ResultSet rslt = stmt.executeQuery();
            if (rslt.next())
                {
                oValue = rslt.getString(2);
                if (rslt.next())
                    {
                    throw new SQLException("Not a unique key: " + oKey);
                    }
                }
            stmt.close();
            }
        catch (SQLException e)
            {
            throw ensureRuntimeException(e, "Load failed: key=" + oKey);
            }
        return oValue;
        }

    /**
    * Store the specified value under the specific key in the underlying
    * store. This method is intended to support both key/value creation
    * and value update for a specific key.
    *
    * @param oKey    key to store the value under
    * @param oValue  value to be stored
    *
    * @throws UnsupportedOperationException  if this implementation or the
    *         underlying store is read-only
    */
    public void store(Object oKey, Object oValue)
        {
        Connection con     = getConnection();
        String     sTable  = getTableName();
        String     sSQL;

        // the following is very inefficient; it is recommended to use DB
        // specific functionality that is, REPLACE for MySQL or MERGE for Oracle
if (load(oKey) != null)
                {
                // key exists - update
         sSQL = "UPDATE " + sTable + " SET value = ? where id = ?";
                }
        else
                {
                // new key - insert
         sSQL = "INSERT INTO " + sTable + " (value, id) VALUES (?,?)";
                }
```

```
    try
            {
            PreparedStatement stmt = con.prepareStatement(sSQL);
            int i = 0;
            stmt.setString(++i, String.valueOf(oValue));
            stmt.setString(++i, String.valueOf(oKey));
            stmt.executeUpdate();
            stmt.close();
            }
    catch (SQLException e)
            {
            throw ensureRuntimeException(e, "Store failed: key=" + oKey);
            }
    }

/**
* Remove the specified key from the underlying store if present.
*
* @param oKey key whose mapping is to be removed from the map
*
* @throws UnsupportedOperationException  if this implementation or the
*         underlying store is read-only
*/
public void erase(Object oKey)
    {
    Connection con  = getConnection();
    String     sSQL = "DELETE FROM " + getTableName() + " WHERE id=?";
    try
        {
        PreparedStatement stmt = con.prepareStatement(sSQL);

        stmt.setString(1, String.valueOf(oKey));
        stmt.executeUpdate();
        stmt.close();
        }
    catch (SQLException e)
        {
        throw ensureRuntimeException(e, "Erase failed: key=" + oKey);
        }
    }

    /**
    * Remove the specified keys from the underlying store if present.
    *
    * @param colKeys  keys whose mappings are being removed from the cache
    *
    * @throws UnsupportedOperationException  if this implementation or the
    *         underlying store is read-only
    */
    public void eraseAll(Collection colKeys)
            {
```

```
            throw new UnsupportedOperationException();
            }

    /**
    * Return the values associated with each the specified keys in the
    * passed collection. If a key does not have an associated value in
    * the underlying store, then the return map will not have an entry
    * for that key.
    *
    * @param colKeys  a collection of keys to load
    *
    * @return a Map of keys to associated values for the specified keys
    */
    public Map loadAll(Collection colKeys)
            {
            throw new UnsupportedOperationException();
            }

    /**
    * Store the specified values under the specified keys in the underlying
    * store. This method is intended to support both key/value creation
    * and value update for the specified keys.
    *
    * @param mapEntries   a Map of any number of keys and values to store
    *
    * @throws UnsupportedOperationException  if this implementation or the
    *         underlying store is read-only
    */
    public void storeAll(Map mapEntries)
            {
            throw new UnsupportedOperationException();
            }

/**
* Iterate all keys in the underlying store.
*
* @return a read-only iterator of the keys in the underlying store
*/
public Iterator keys()
    {
    Connection con  = getConnection();
    String     sSQL = "SELECT id FROM " + getTableName();
    List       list = new LinkedList();

    try
        {
        PreparedStatement stmt = con.prepareStatement(sSQL);
        ResultSet         rslt = stmt.executeQuery();
        while (rslt.next())
            {
            Object oKey = rslt.getString(1);
```

```
                list.add(oKey);
                }
            stmt.close();
            }
        catch (SQLException e)
            {
            throw ensureRuntimeException(e, "Iterator failed");
            }

        return list.iterator();
        }


    // ----- data members ---------------------------------------------------

    /**
    * The connection.
    */
    protected Connection m_con;

    /**
    * The db table name.
    */
    protected String m_sTableName;

        /**
        * Driver class name.
        */
        private static final String DB_DRIVER   = "org.gjt.mm.mysql.Driver";

        /**
        * Connection URL.
        */
        private static final String DB_URL      = "jdbc:mysql://localhost:3306/CacheStore"

        /**
        * User name.
        */
        private static final String DB_USERNAME = "root";

        /**
        * Password.
        */
        private static final String DB_PASSWORD = null;
    }
```

0 0

Sample Controllable CacheStore

This section illustrates the implementation of a controllable cache store. In this scenario, the application can control when updated values in the cache are written to the data store. The most common use case for this scenario is during the initial population of the cache from the data store at startup. At startup, there is no need to write values in the cache back to the data store. Any attempt to do so would be a waste of resources.

The **`Main.java`** file in Example 13-4 illustrates two different approaches to interacting with a controllable cache store:

- Use a controllable cache (note that it must be on a different service) to enable or disable the cache store. This is illustrated by the **`ControllableCacheStore1`** class.
- Use the **`CacheStoreAware`** interface to indicate that objects added to the cache do not need to be stored. This is illustrated by the **`ControllableCacheStore2`** class.

Both **`ControllableCacheStore1`** and **`ControllableCacheStore2`** extend the **`com.tangosol.net.cache.AbstractCacheStore`** (../../coh.360/e18814/toc.htm) class. This helper class provides unoptimized implementations of the **`storeAll`** and **`eraseAll`** operations.

The **`CacheStoreAware.java`** file is an interface which can be used to indicate that an object added to the cache should not be stored in the database.

See "Cache of a Database" (cache_examples.htm#BACGHIFE) for a sample cache configurations.

Example 13-4 provides a listing of the **`Main.java`** interface.

Example 13-4 Main.java - Interacting with a Controllable CacheStore

```
import com.tangosol.net.CacheFactory;
import com.tangosol.net.NamedCache;
import com.tangosol.net.cache.AbstractCacheStore;
import com.tangosol.util.Base;

import java.io.Serializable;
import java.util.Date;

public class Main extends Base
    {

    /**
     * CacheStore implementation which is controlled by a control cache
     */
    public static class ControllableCacheStore1 extends AbstractCacheStore
        {
        public static final String CONTROL_CACHE = "cachestorecontrol";

        String m_sName;

        public static void enable(String sName)
            {
            CacheFactory.getCache(CONTROL_CACHE).put(sName, Boolean.TRUE);
            }

        public static void disable(String sName)
            {
            CacheFactory.getCache(CONTROL_CACHE).put(sName, Boolean.FALSE);
            }

        public void store(Object oKey, Object oValue)
            {
            Boolean isEnabled = (Boolean) CacheFactory.getCache(CONTROL_CACHE).get(m_sName
            if (isEnabled != null && isEnabled.booleanValue())
                {
                log("controllablecachestore1: enabled " + oKey + " = " + oValue);
                }
            else
                {
                log("controllablecachestore1: disabled " + oKey + " = " + oValue);
                }
            }

        public Object load(Object oKey)
            {
            log("controllablecachestore1: load:" + oKey);
            return new MyValue1(oKey);
            }

        public ControllableCacheStore1(String sName)
```

```
        {
        m_sName = sName;
        }

    }

/**
 * CacheStore implementation which is controlled by values
 * implementing the CacheStoreAware interface
 */
public static class ControllableCacheStore2 extends AbstractCacheStore
    {

    public void store(Object oKey, Object oValue)
        {
        boolean isEnabled = oValue instanceof CacheStoreAware ? !((CacheStoreAware) oV
        if (isEnabled)
            {
            log("controllablecachestore2: enabled " + oKey + " = " + oValue);
            }
        else
            {
            log("controllablecachestore2: disabled " + oKey + " = " + oValue);
            }
        }

    public Object load(Object oKey)
        {
        log("controllablecachestore2: load:" + oKey);
        return new MyValue2(oKey);
        }

    }

public static class MyValue1 implements Serializable
    {
    String m_sValue;

    public String getValue()
        {
        return m_sValue;
        }

    public String toString()
        {
        return "MyValue1[" + getValue() + "]";
        }

    public MyValue1(Object obj)
        {
        m_sValue = "value:" + obj;
```

```
        }
    }

public static class MyValue2 extends MyValue1 implements CacheStoreAware
    {
    boolean m_isSkipStore = false;

    public boolean isSkipStore()
        {
        return m_isSkipStore;
        }

    public void skipStore()
        {
        m_isSkipStore = true;
        }

    public String toString()
        {
        return "MyValue2[" + getValue() + "]";
        }

    public MyValue2(Object obj)
        {
        super(obj);
        }

    }

public static void main(String[] args)
    {
    try
        {

        // example 1

        NamedCache cache1 = CacheFactory.getCache("cache1");

        // disable cachestore
        ControllableCacheStore1.disable("cache1");
        for(int i = 0; i < 5; i++)
            {
            cache1.put(new Integer(i), new MyValue1(new Date()));
            }

        // enable cachestore
        ControllableCacheStore1.enable("cache1");
        for(int i = 0; i < 5; i++)
            {
            cache1.put(new Integer(i), new MyValue1(new Date()));
            }
```

```
            // example 2

            NamedCache cache2 = CacheFactory.getCache("cache2");

            // add some values with cachestore disabled
            for(int i = 0; i < 5; i++)
                {
                MyValue2 value = new MyValue2(new Date());
                value.skipStore();
                cache2.put(new Integer(i), value);
                }

            // add some values with cachestore enabled
            for(int i = 0; i < 5; i++)
                {
                cache2.put(new Integer(i), new MyValue2(new Date()));
                }


            }
        catch(Throwable oops)
            {
            err(oops);
            }
        finally
            {
            CacheFactory.shutdown();
            }
        }

    }
```

Example 13-5 provides a listing of the **CacheStoreAware.java** interface.

0 0 ***Example 13-5 CacheStoreAware.java interface***

```
public interface CacheStoreAware
    {
    public boolean isSkipStore();
    }
```

0
0

Implementation Considerations

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

0

()

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 (../../coh.360/e15723/api_querycache.htm#COHDG136) 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 data set directly from the database into the cache (batching blocks of data into the cache by using **`NamedCache.putAll()`** (../../coh.360/e18814/toc.htm). The loader process will need to use a "Controllable Cachestore (../../coh.360/e15723/cache_rtwtwbra.htm#COHDG486)" 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.

()
()

Re-entrant Calls

The **`CacheStore`** implementation must not call back into the hosting cache service. This includes ORM 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 thread pool 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`** (../../coh.360/e18814/toc.htm) 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`** (../../coh.360/e18814/toc.htm) 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 third 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).