CHAPTER 10

Caching and Memory Management

This chapter describes the cache pools in *Luna Server* shared memory and how you manage those cache pools for maximum performance. This chapter includes the following topics:

Luna object-cache overview

When an application requests a data object, *Luna Server* looks in shared memory for the requested object. The portion of memory that holds application data objects is called *cache*. When *Luna Server* can find a requested object in cache, it bypasses a fetch from disk, which saves time and improves performance.

Fetch requires the following tasks:

- Searching the database

  A fetch might impact performance due to network latency, disk latency, and read time.

- Mapping the database record or row to a server-application object

Figure 1 shows a graphical view of the fetch tasks.

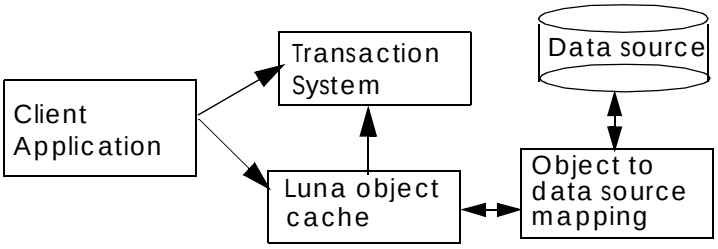

Figure 1. Fetching from data source disk

Types and number of objects in cache

Cached data objects fall into the following categories:

- An entity Bean that contains attribute data, as well as a primary key
- A collection

  A collection holds multiple entity Beans of a specific type, such as objects in a relationship or the result set from a query.

Multiple objects of a specific type might be in cache, each of which is identified by its key. One or more instances of a keyed object might exist in cache simultaneously. Some factors that require the need for cache to acquire additional instances include:

- Age, or *staleness,* of the cached instance
- Application isolation-level or data-locking requirements
- Changes to data values in a cached instance
- Commited changes to the back-end data source

Cache-dependent activities

This section discusses the following activities that involve cache:

- Creating new objects
- Finding or updating existing objects
- Finding or updating related objects
- Executing queries
- Beginning and ending a transaction

Naturally, creating a new object results in adding an instance to cache. However, the other actions might reuse an existing cached instance.

Creating new objects

The server application must locate the corresponding home-interface for the object, then call one of the home-interface methods that creates an instance of the object. For example, to create a new customer, the Bank of Luna application calls the `customerHome.newObject` method.

Luna Server checks the previously cached instances of the requested object type for the same key value that is passed to the `create` method. If a duplicate key exists in cache, the create fails.

If the new object has a unique key, the create succeeds and the new object instance is created in cache, where subsequent method calls can access it. When the client commits the transaction, the new cached instance is inserted into the database.

Finding existing objects

Entity Beans have an associated *`CacheTimeout`* property. The *`CacheTimeout`* value specifies how long a cached instance of that object type remains usable.

If cache does not contain a usable instance that satisfies the request, the server must read the current state of the requested data into cache from the data source. The server fetches data and creates an instance only if one of the following conditions is true:

- No instance for requested object exists in cache.
- The cached instance is stale.
- A client specifies a *`CacheTimeout`* period for the object that is shorter than the staleness of any cached instance.

  Typically, a client application uses an instance that is in cache during the *`CacheTimeout`* period that is set for the server-side object in the *Luna Administration Center Inspector* window. The application can override the *`CacheTimeout`* property for an instance of a particular object with the `com.luna.Runtime.setProperty` method.

If the application calls the `findByPrimaryKey` method, *Luna Server* caches the key value as a reference to the whole keyed object. When an application calls the `findAll` method, the *Luna Server* brings all the keys of the specified object type into cache. The objects themselves are brought into the cache only if needed.

Nonexistent keys

If the *Luna Server* cannot find the requested object in cache or its corresponding data source record on disk, the key for that object is marked in cache as *nonexistent.* The nonexistent indicator:

- enables an immediate fail of any additional requests for that same object, without a search through the data source.
- is removed from the cached key if an application requests a new object with that key. An instance is added to cache for the now existing key.

Prefetch

If the application calls the `loadAll` method instead of the `findAll` method to cache relationship data, all the data is loaded

into cache immediately, not just the primary keys. If the *Prefetch* property is set to `true` for any collection, a call to `findAll` is equivalent to a call to `loadAll`.

The `loadAll` method or *Prefetch* property involve a performance trade-off between disk-access time and memory-resource consumption. When cache holds the data for all objects of a particular type, the *Luna Server* does not need to wait for any one record to be read into cache. However, loading data for all the objects of a particular type diminishes the amount of free cache and can cause frequent paging to virtual memory or force other object types out of cache before they time out.

Collecting related objects

If the client requests a collection of related objects from a one-to-many relationship, the server must fetch all the related records. For example, the Bank of Luna application defines a one-to-many relationship between a customer and the trades in the customer portfolio. If a client needs the portfolio contents, it first requests the customer and then requests a collection of the equities that the customer owns.

The group of related records are collected in an array in cache. (Refer to Figure 6 on page 304.) In the Bank of Luna example, processing a customer portfolio creates an array in cache that is keyed on the customer primary key (account number) and contains, for each equity in the customer portfolio, the equity Bean primary key (trading symbol) and the quantity owned.

Generally, an application calls the relationship-lookup method to load the primary keys for related entities into cache then calls the `iterator` method to fetch the remaining data for the related entities, one entity at a time. For example, the following two calls load a Bank of Luna customer's trades into cache:

```
TradeCollection custTrades customer.trades();
Trade trade customer.trades().iterator();
```

The application might not load the trade instances. Instead, the application might iterate through the data on disk to summarize trade data, such as summing the total investment. Or the application might query the disk data to bring in only selected trades.

Executing a query

A call to the `query` method from the entity Bean home typically caches primary keys or entity Beans that *Luna Server* further processes according to a *query condition.*

Condition argument

The query condition:

- is a string that is passed as an argument to the `query` method.
- filters, or selects, from the data source only the records that meet the specified condition criteria.
- is similar to a `WHERE` clause in an SQL statement.

In the following example, the condition string selects from all trades in the data source, the trades that occurred on July 10, 1999:

```
tradeHome.query("tradeDate = '1999.July.10 '")
```

If no condition string exists, the `query` method acts like the `findAll` method. For example, the following unconditional query reads all trade Bean keys into cache:

```
tradeHome.query("")
```

Pushdown

The following *Luna Development Center* properties determine whether the data source or *Luna Server* process the condition:

Table 1. Pushdown properties

Object	Property	Enable pushdown	Disable pushdown
Query	*QueryPolicy*	`Attempt pushdown`	`No pushdown`
Relationship	*QueryPolicy*	`Attempt pushdown`	`No pushdown`
Data source	*SupportsQuery Pushdown*	`true`	`false`
Data source	*ConnectionProperty* `case_sensitive_sort`	`true`	`false`

By default, the *Luna Server* attempts to push as much of a query condition as possible to the data source for processing. If you set the appropriate properties to disable pushdown, *Luna Server* fetches all objects of a specific type or subtype into cache before applying the query condition, which uses as much cache as a `query` method call with no condition argument.

For more information about pushdown properties, refer to the online help for the *Development Center*.

Fetching actual records

As with relationships, the application typically uses the primary key references to fetch data source records that satisfy the query, loading each single record as a cache instance on an as-needed basis.

Note... If the prefetch property affects the query, the query brings records into cache immediately, rather than on an as-needed basis. For more information, refer to *"Prefetch"* on page 292.

Beginning and ending a transaction

When a transaction begins, *Luna Server* creates some *transient* objects in cache to assist in managing the transaction. Transient objects live only as long as they are needed and have no associated `CacheTimeout` period. When the transaction context with which these objects are associated is committed or rolls back, the transient objects are freed. For more information about transient objects, refer to *"Transient recyclers"* on page 304.

When an application updates a cached object, *Luna Server* creates a new instance, called a *copy on write* instance. If the transaction successfully commits, the copy on write version provides multiple applications with rapid access the latest version of the data that *Luna Server* wrote to disk.

Cache-pool architecture

Cache is divided into *cache pools*. Each *Luna Server* has one System cache pool and at least one user cache pool, named Default cache pool. You can create additional user cache pools as *"Configuring a cache pool"* on page 306 describes.

Each cache pool, in turn, is subdivided into *Recyclers*. A recycler holds objects of a specific type. For example, the Bank of Luna Default cache pool might contain a recycler that holds customerBean instances, a recycler that holds tradeBean instances, and so forth.

This section describes the various recyclers with the following topics:

- Cached object life-cycle
- Recycler types

Cache-pool life cycle

The *state* of each object instance in a recycler determines whether an application can use that instance. For most recyclers, state depends upon two factors:

- Transaction participation
- Age of the instance in cache

  Entities, queries, and relationships each have a `CacheTimeout` setting that specifies the maximum duration that an instance is usable since the instance is first read into cache.

A recycler might contain instances in any of the following states.

Instance state	Description
Locked	Instances in an active transaction that cannot change state or be discarded until the transaction commits or rolls back.
Unlocked	Instances that are not currently involved in an active transaction but have not timed out. *Luna Server* searches through unlocked instances when an application requests this specific object.
Free	Because each recycler contains instances of a specific object type, free slots of the appropriate size and type for that type remain in the recycler after their contents are purged. When *Luna Server* must read in a new instance (copy on read), or make a copy of a committed version (copy on write), it fills a free space in the appropriate recycler with the data for the new copy.

Figure 2 on page 298 illustrates the state transitions that object instances experience during their stay in a cache-pool recycler.

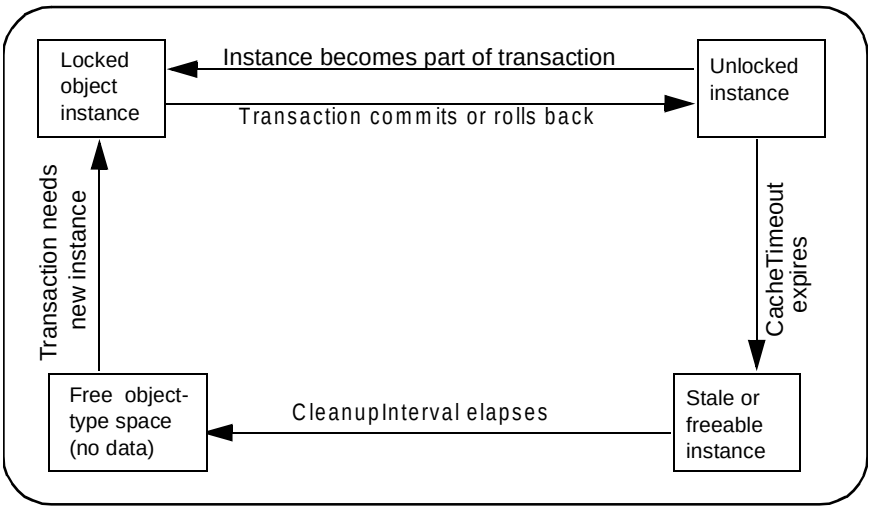

Figure 2. Lifecycle of a cached instance

Locked instances are not always returned to an unlocked state. For example, in Figure 3 on page 299 an application changes the data in a locked instance and then commits the changes. The state of the original instance changes to free because the committed version supersedes the original data. *Luna Server* creates an unlocked copy-on-write instance with an elapsed time of 0.

Note... In the example of Figure 3, no other application is using the pre-committed version of the locked instance.

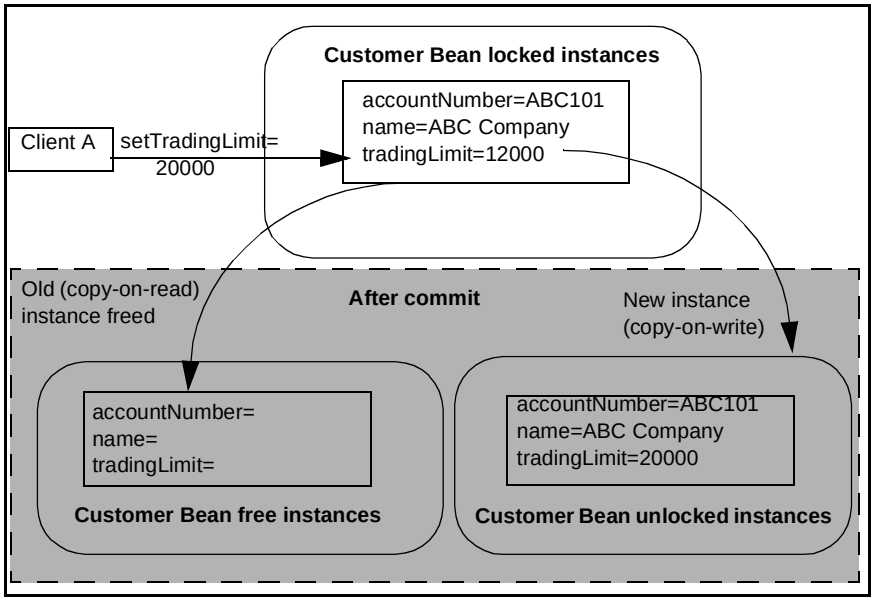

Figure 3. One client uses locked instance

Recycler Types

A user cache pool contains the following recycler types:

- Cacheable
- Aggregate
- Transient

You can distinguish the recyclers by type in the **Memory Recyclers (Preview)** cache-pool monitoring window of the *Luna Administration Center*.

Cacheable memory recyclers

A cacheable memory recycler holds entity-Bean or collection instances. A cacheable recycler can contain locked, unlocked, and free instances.

A logical *collection instance* actually contains multiple entity Beans in a single physical array. A single collection instance either represents a relationship or the result set from a query.

Multiple applications can reuse the same cacheable instance repeatedly until the instance `CacheTimeout` expires. After `cacheTimeout`, an unlocked instance can change state to free. A locked instance cannot change state to free until all the applications that are using it for active transactions commit or roll back. *"Reaper thread"* on page 305 describes the process that changes expired (stale) instances to free instances.

Aggregate memory recyclers

For a pictorial representation of this discussion about aggregate memory recyclers, refer to Figure 4 on page 301. An aggregate corresponds to a particular home object. Each aggregate in cache contains a hash list of primary keys for a particular object type. Each primary key in the hash list references a *proxy* for the corresponding keyed object. A proxy contains information from the home interface for the object, including a copy of the primary key and references to instances of the keyed object. Each instance that the proxy references is located in a cacheable recycler.

Entity-Bean aggregate recycler

Figure 4 shows the aggregate of cached primary keys and proxies that result from a call to `CustomerHome.findAll`. Figure 4 also shows how the aggregate relates the keys to corresponding instances in a cacheable recycler. Each instance is added to the cacheable recycler as an application needs a fresh set of data (other than the primary key) from the data source record with that key.

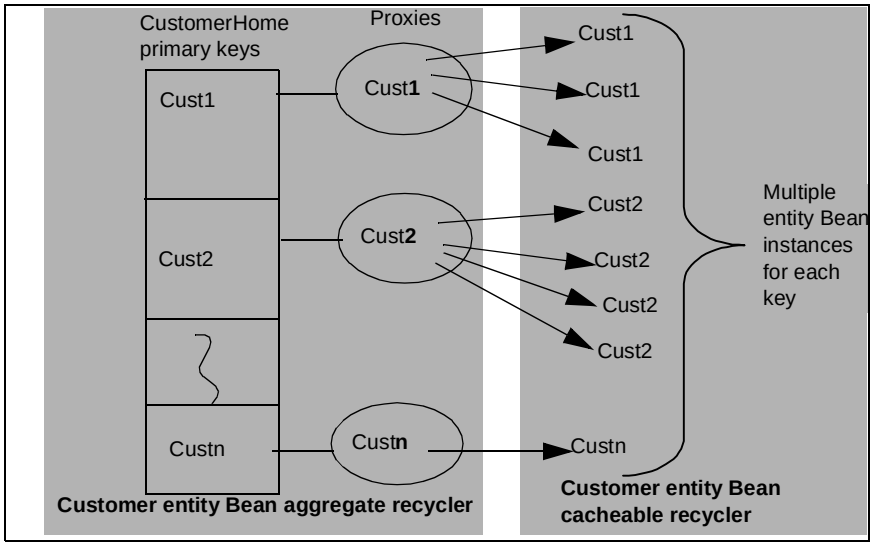

Figure 4. Customer home aggregate recycler

Query aggregate recycler

In a query aggregate, each primary key is the conditional string that is passed to the query method. In the following example, the query primary key is the string inside the parenthesis:

```
CustomerHome.query("accountNumber='Cust1'
                or accountNumber='Cust2'")
```

The query returns a collection (array) of primary keys and references. In this example, one array entry contains the following:

- `Customer.accountNumber`, which is the primary key
- Reference to the *proxy* for this `accountNumber` primary key

This example shows that a collection in a *cacheable* recycler contains references into an entity-Bean *aggregate* memory recycler. Figure 5 shows the relationship between a query aggregate recycler and an entity Bean aggregate recycler.

Note... A proxy contains a copy of the corresponding primary key in the hash table so that collections can reference the proxy.

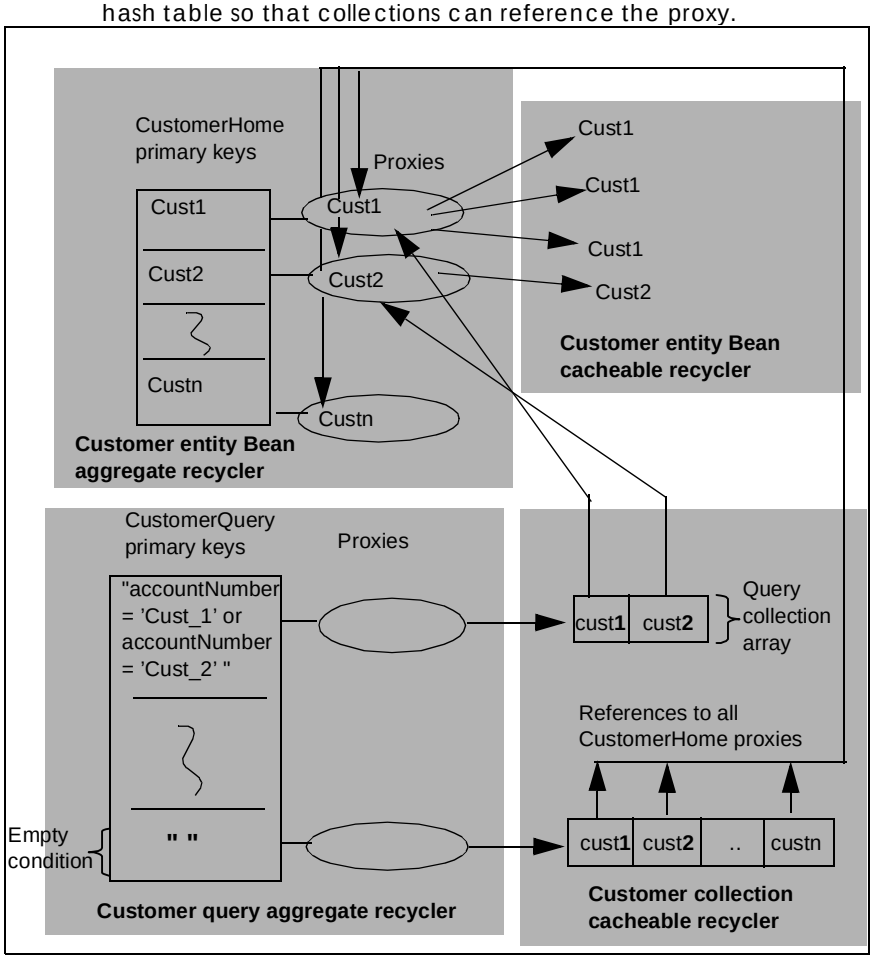

Figure 5. Query aggregate

The following query creates an aggregate entry with an empty string to represent the query primary key. The empty query condition returns a collection array that contains the primary keys of all customer keys in the data source:

```
CustomerHome.query("")
```

Aggregate recycler primary-key lists and proxies remain in the cache pool unless cache becomes 90 percent full. Even if all the cacheable instances for a particular proxy are freed so that the proxy no longer points to locked or unlocked instances, the proxy stays in memory, available for reuse.

Relationship aggregate recyclers

Relationship aggregate recyclers act similarly to query aggregate recyclers in that each relationship proxy points to a collection of primary keys for the related entities. Each key in the collection references the entity Bean instances with that key.

Figure 6 on page 304 shows how the various proxies, collections, and entity Beans reference each other. The primary key for a relationship is the primary key for the owning element. Thus, in Figure 6, the relationship aggregate for each collection of customer trades references the trade collection by the key of the customer that owns those trades.

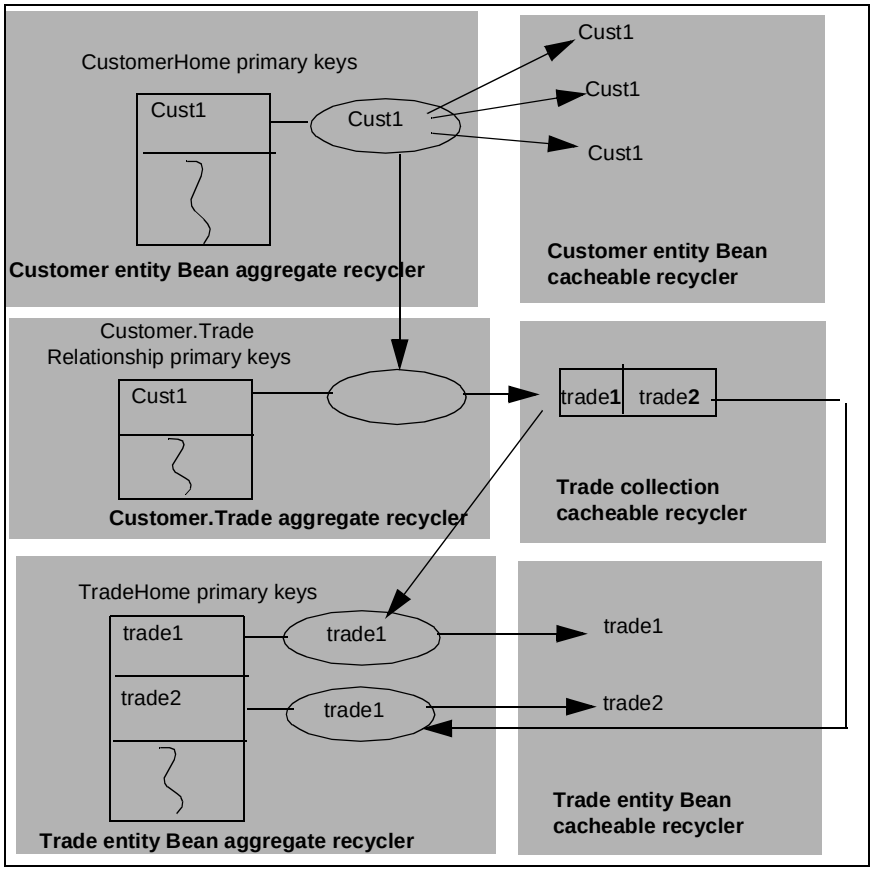

Figure 6. Relationship aggregate recyclers

Transient recyclers

Unlike other cached objects, transient objects cannot be shared. Because they remain in cache only as long as they are active, transient objects have only two states—locked and free.

Transient objects are not entity Beans or collections. They are Luna system objects that help the Luna object-cache subsystem

mark transactions, load Beans, and so forth. Each is an instance of a Luna system class.

The System cache pool contains only transient objects. The Default and other cache pools can also have transient objects.

Reaper thread

Each cache pool has a Reaper thread that periodically wakes up to purge objects from the cache pool. The `CleanupInterval` property for a cache pool determines the interval at which the Reaper thread wakes up.

The Reaper thread performs different stages of cleanup, depending upon how close to maximum memory capacity the filled portion of cache takes up:

- Turn stale instances into free spaces in cacheable recyclers
- Purge unlocked from cacheable recyclers
- Remove free space from all recyclers
- Purge unused proxies from aggregate recyclers

When the Reaper thread wakes up, it automatically changes stale objects to free object-size spaces, regardless of the memory usage. As the memory usage increases, the Reaper thread performs more aggressive cleanup tasks. Unlocked and free objects are purged when memory utilization reaches 80 percent.

Unused aggregate proxies are purged at a memory utilization of 90 percent. An unused proxy:

- references no instances.
- is not referenced by an instance.

  Figure 5 on page 302 shows how an aggregate proxy can remain in cache even after it no longer points to instances. In this example, the Reaper thread does not free the proxy for `custn` because the collection from the query for all customer beans still points to this query.

Cache-pool administration

This section is provided for the *Luna Server* administrator.

Luna provides memory-management tools, including tools for cache management. The *Luna Administration Center* and *Development Center* desktop applications and the `com.luna.support.CPManager` command-line utility enable an administrator to perform the following cache-pool tasks:

- View cache-pool statistics
- Add cache pools
- Assign specific Beans to named cache pools
- Specify the number and duration (or *`CacheTimeout`* period) of cached objects
- Enlarge the amount of memory available to a cache pool

This section introduces some of the tools that the desktop applications provide. For information about the command-line `com.luna.support.CPManager` utility, refer to the usage section of the utility. See *"Monitoring cache pools"* on page 426. To print the usage, run the utility as follows:

```
lunavbj com.luna.support.CPManager -?
```

Configuring a cache pool

Luna Server includes the Default user cache pool.

Configuring a new cache pool

As the *Luna Server* administrator, you might add user cache pools to a server to distribute application objects among several pools. You might dedicate a user cache pool to:

- Hold object types that you expect clients to need frequently.
- Hold collections that contain many objects.

- Improve response if cache-pool statistics indicate that the Default cache pool contains too many objects

Use the *Luna Administration Center* to create and configure cache pools. To create a new cache pool:

1. Right-click the **Cache pools** icon in the *Explorer* window of the *Administration Center* and select **New cache pool...** from the popup menu.

   The **New Cache Pool Wizard** dialog opens (Figure 7).

2. Refer to Table 2 on page 310 to find the corresponding cache-pool properties for the fields in the wizard dialog.
   - In the **New cache pool name** field, enter a unique name for the cache pool.
   - In the **Max size in bytes** field, enter a number of bytes to allocate to the cache poo or enter `-1` to allow the cache pool to dynamically consume available free memory space.
   - In the **Pool cleanup interval**, enter the number of seconds between reaper-thread activity.
   - Select `true` or `false` from the **Pool size may override limit** list box to determine if the cache pool can temporarily expand beyond the value in the **Max size in bytes** field in a low-memory situation.

3. Click the **Finish** button to close the wizard and complete the new cache-pool setup.

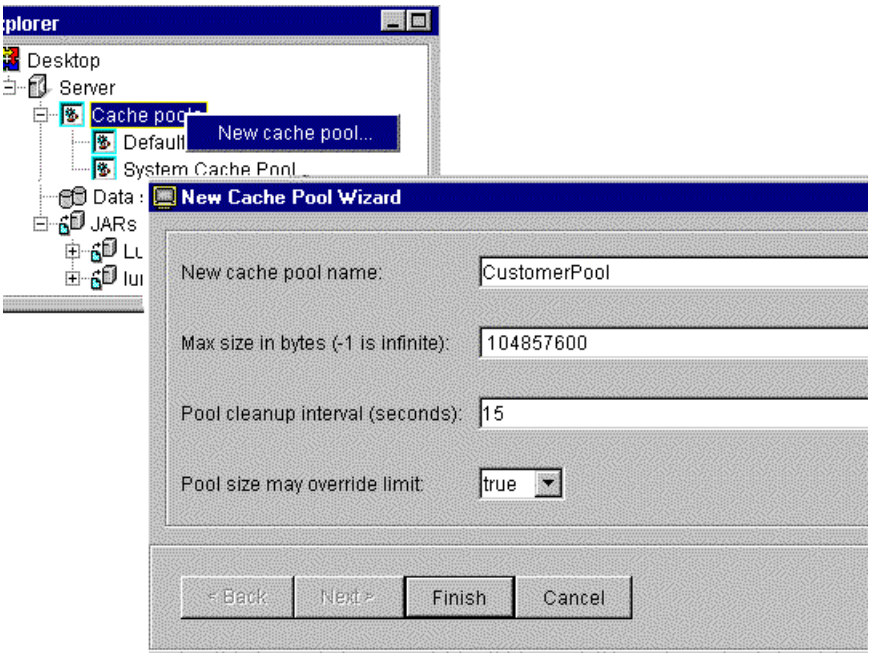

Figure 7. Adding a cache pool

Changing a cache-pool configuration

You can change any of the user cache-pool property values in the *Inspector* window of the *Administration Center* (Figure 8). For information about how to set cache-pool properties, refer to the *Administration Center* online help.

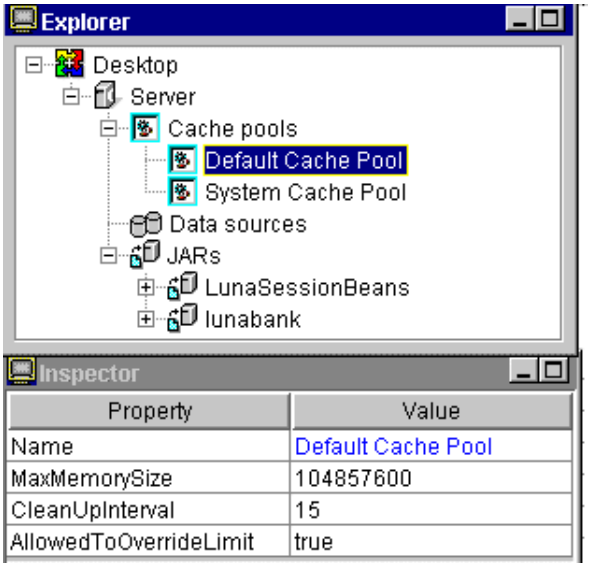

Figure 8. Default cache-pool properties

All user cache pools, including the Default cache pool, have the configuration properties that Table 2 on page 310 describes.

Table 2. Cache-pool configuration properties

Property	Description
`Name`	The name of the cache pool. [default: none]
`MaxMemorySize`	The amount of memory allocated to hold all instances of all cached object instances that are assigned to the cache pool. A value of `-1` indicates unlimited memory. Unlimited memory is the equivalent of the total memory available from the amount of memory allocated to this server at startup time. Express values in bytes. For example, the default value appears as 104857600 bytes. [default: 100MB]
`CleanUpInterval`	The interval after which the *reaper thread* releases memory from stale instances and, if necessary, other unlocked objects. For more information, see *"Reaper thread"* on page 305. [default: 15 seconds]
`AllowedToOverrideLimit`	An indicator to determine if the cache pool can exceed the `MaxMemorySize` if the reaper thread cannot recapture enough memory to complete a transaction. If set to `true`, an operation can continue even though the cache pool exceeds `MaxMemorySize`. If `false`, the application throws an exception when the cache pool exceeds `MaxMemorySize.` [default: true]

The *Administration Center* provides monitoring tools that you use to determine if you need to change any of the cache-pool properties. You can monitor the amount of memory that a cache pool uses, the number of objects in it, and the size of the cached

objects. For information about monitoring cache-pool usage, refer to *"Cache pool statistics"* on page 427.

Assigning entity Beans to a cache pool

Initially, entity Beans in an application are assigned to the Default cache pool. Once you add a new cache pool, you can assign particular Beans to it. You can also change entity Bean cache-pool properties that relate to the way an entity Bean behaves in cache.

Configuring entity Beans for cache

Refer to Figure 10 to reassign an entity bean and follow these general steps:

1. Select a Bean in the *Explorer* window of the *Administration Center*.

   All the properties and general information appear in the *Inspector* window.

2. Click the value for the `CachePool` property and select the desired cache pool for that Bean from the drop-down menu (Figure 9).

3. If appropriate, change other cache-pool properties for this Bean in the *Inspector* window (Figure 10).

   Table 3 on page 314 explains all the entity Bean properties that pertain to cache pools.

4. If desired, repeat Step 1 through Step 3 for other entity Beans.

5. Redeploy the application.

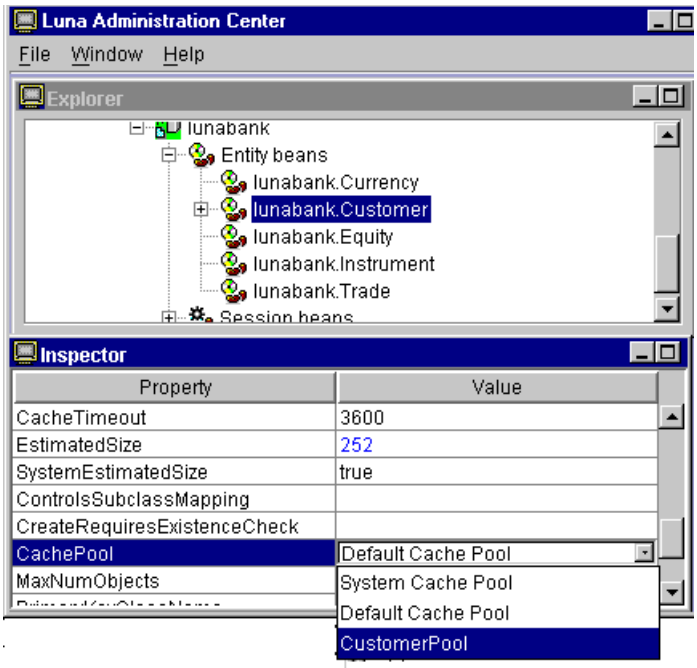

Figure 9. Moving an entity Bean to a new cache pool

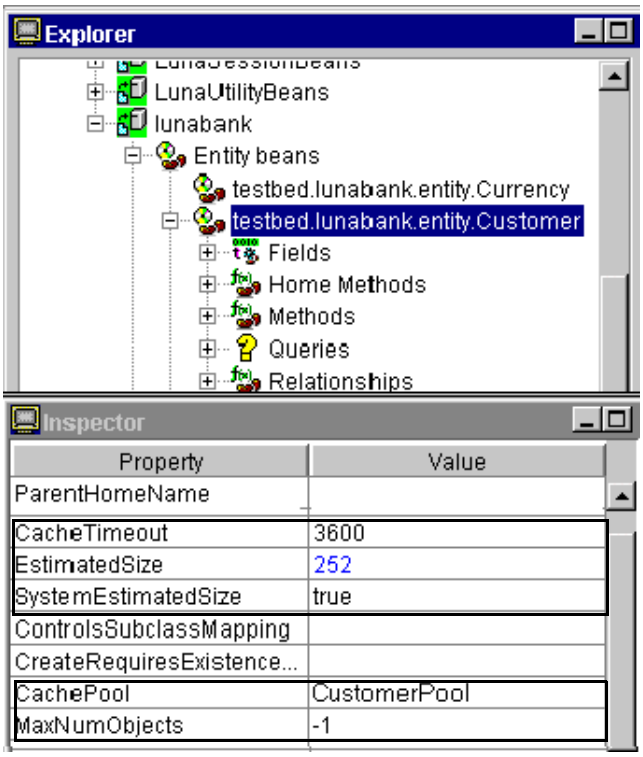

Figure 10. Adjusting cache-pool properties for a Bean

Among the several properties in the *Inspector* window for an entityBean, Table 3 describes those that pertain to how the entity Bean influences cache-pool size, instance duration, and other performance considerations for the cache pool that holds instances of this Bean type.

Table 3. Entity-Bean cache-pool properties

Property	Description
CachePool	The name of the cache pool that can contain instance of this entity type. [default: Default Cache Pool]
MaxNumObjects	The maximum number of instances of this entity Bean that its cache pool can contain concurrently. [default: -1 (unlimited)]
CacheTimeout	The shelf life of an entity in seconds. After *CacheTimeout* elapses for an instance of this entity in cache, the cached instance becomes unusable (stale). For the next request, the server performs a copy on read, which loads the instance from the data source. [default: 3600 seconds (1 hour)]
SystemEstimatedSize	If true, the Luna system fills in the *EstimatedSize* for this entity type and the user cannot change the estimate. If false, the administrator enters a value for *EstimatedSize*. [default: true]
EstimatedSize	If *System Estimated Size* is false, the user could change this value. The system calculates the size based on the type and number of fields. String fields are considered 50 characters long. [default: system-estimated size.]

Configuring queries and relationships for cache

You can set a separate *CacheTimeout* period for each query and relationship that is defined for the entity Bean. As with the entity

Bean cache-pool properties, you set properties for queries and relationships in the *Administration Center*.

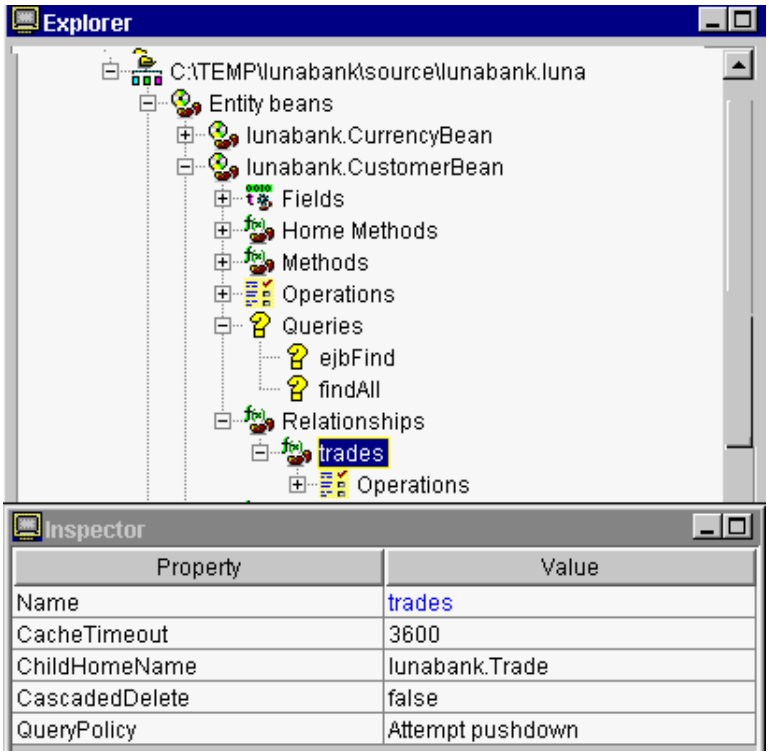

Figure 11. Relationship configuration properties

To set the `CacheTimeout` for an object that is subordinate to an entity Bean, refer to Figure 11 and follow these steps:

1. Expand the entity Bean in the *Administration Center Explorer* window.

2. Click the query method or relationship entity.

3. In the *Inspector* window, change the value for `CacheTimeout`.

Note... The *QueryPolicy* property affects the amount of cache that queries use. To ensure the fewest number of query results possible, set all the properties needed to enable pushdown that Table 1 on page 295 shows.

Checking results of object cache configuration

As with the cache-pool properties, use the monitoring facilities of the *Administration Center* to determine if further changes are needed to the entity Bean cache-pool properties. For information about using the monitoring tools and altering properties for improved performance, see *"Cache pool statistics"* on page 427.

Removing a cache pool

You may remove any cache pool except the Default and System cache pools. To remove a cache pool:

1. Right-click the cache pool name in the *Administration Center Explorer* window.

2. Select **Remove Pool** from the popup menu.

If you attempt to destroy a cache pool that has entity Beans assigned to it, the Luna object-cache system returns the Beans to the Default cache pool. You can reassign the Bean to a cache pool other than Default. To reassign an entity Bean to a named cache pool, refer to *"Assigning entity Beans to a cache pool"* on page 311.

Cache-pool properties

You can change cache-pool properties in the *Administration Center*. Refer to *"Changing a cache-pool configuration"* on page 308.

You might increase the maximum cache size if garbage collection occurs too frequently. The following situations can require an increase in memory size for the cache pool:

- A client application requests large volumes of data.
- The cache pool houses entity Beans with large `EstimatedSize` values.

  To check `EstimatedSize` for an entity Bean, look at the *Inspector* window properties for that Bean, as Figure 10 on page 313 shows.
- The application design includes one-to-many relationships.

  You can assess the frequency with which relationships are in cache by noting statistics in the **Memory Recycler (Preview)** list `LunaRelationship` recyclers.

  For more information about relationships, refer to *"Collecting related objects"* on page 293 and *"Aggregate memory recyclers"* on page 300.
- Queries involve complex joins or return all the keys for an entity type.

  You can assess the number of queries by noting statistics for `LunaQuery` recyclers in the **Memory Recycler (Preview)** list.

  For more information about queries, refer to *"Executing a query"* on page 294 and *"Aggregate memory recyclers"* on page 300.
- The application calls the `loadAll` method to load entire objects rather than keys for a collection, or the `Prefetch` property for the queried object is set to `true`.

Consider creating separate cache pools for large objects and collections, then specifying a larger `MaxMemorySize` for the cache pool that contains large objects.

You can also create a separate cache pool for small or infrequently changed objects that has a smaller `MaxMemorySize`. The smaller pool balances requirements of larger cache pools that contain active objects during the same application session.

Watch the number of locked instances in the *Administration Center* cache-pool monitor graphs and statistics. Consider how many instances your application typically requires to complete transaction. Allocate sufficient `MaxMemorySize` for the cache pool to accommodate peak periods of activity.

Entity-bean cache-pool properties

You might increase the `CacheTimeout` value to retain the following types of objects:

- Frequently used objects.
- Objects that are mapped to stable data. Stable data, such as customer or employee information, does not change frequently.

You can decrease the `CacheTimeout` to force a fetch of the most recent committed version of data from the back-end data source. For example, external applications can change the data in the data source causing Luna client application transactions to roll back because the concurrency check shows that the data has changed.

If necessary, set a separate `CacheTimeout` for queries and relationships than you do for the parent entity Bean. Refer to *"Configuring queries and relationships for cache"* on page 314.

Set the `MaxNumObjects` property to allow an adequate number of locked instances to complete transaction. Consider the affect of isolation levels on the number of instances with the same primary key that must remain in cache.

Leave the query pushdown properties set to allow the external data source to deselect as many records as possible before sending back query results. Refer to *"Pushdown"* on page 294.