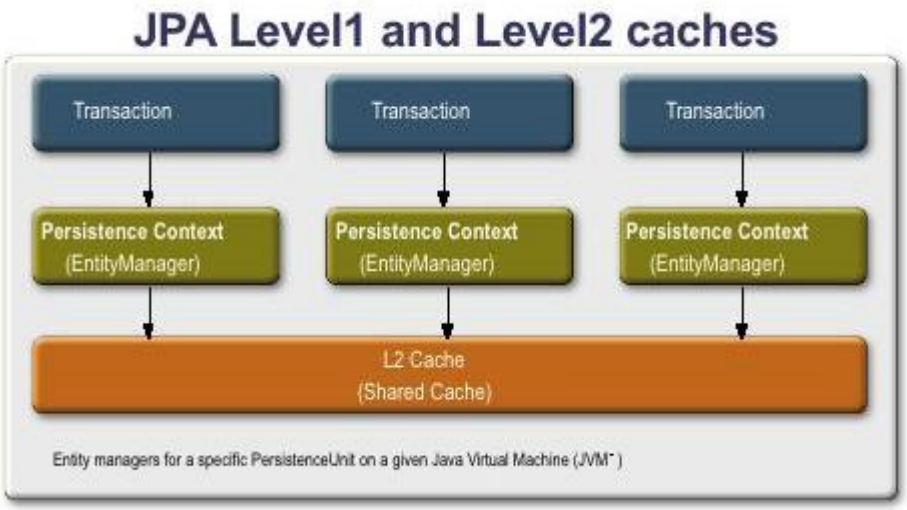
JPA Caching

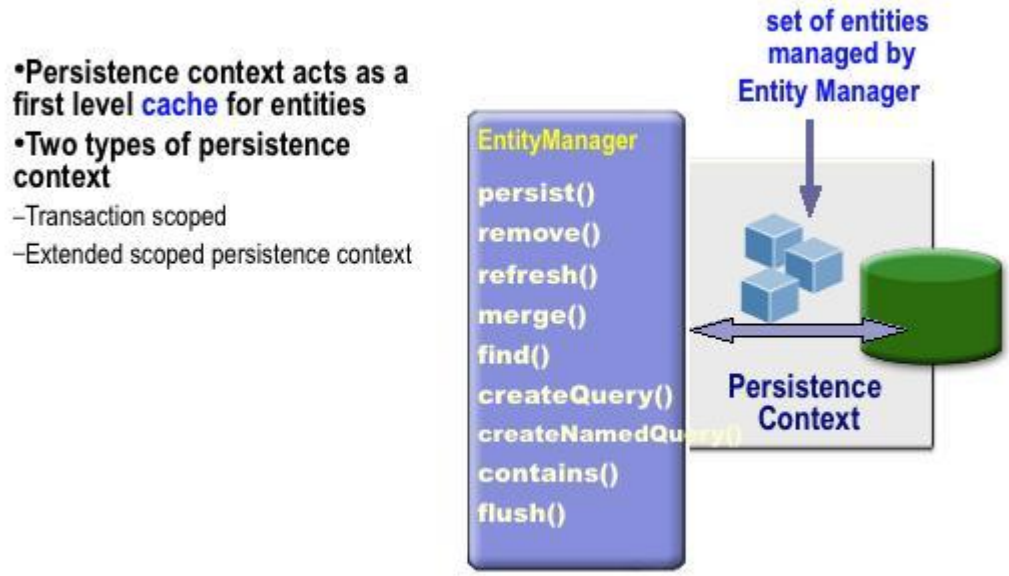
By caroljmcdonald on Aug 21, 2009

JPA Level 1 caching

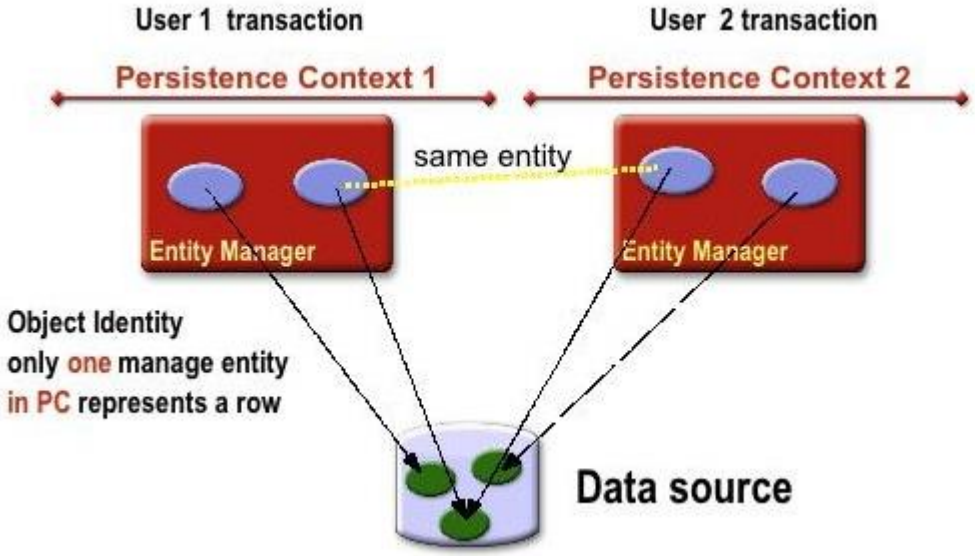
JPA has 2 levels of caching. The first level of caching is the persistence context.



The JPA Entity Manager maintains a set of Managed Entities in the Persistence Context.



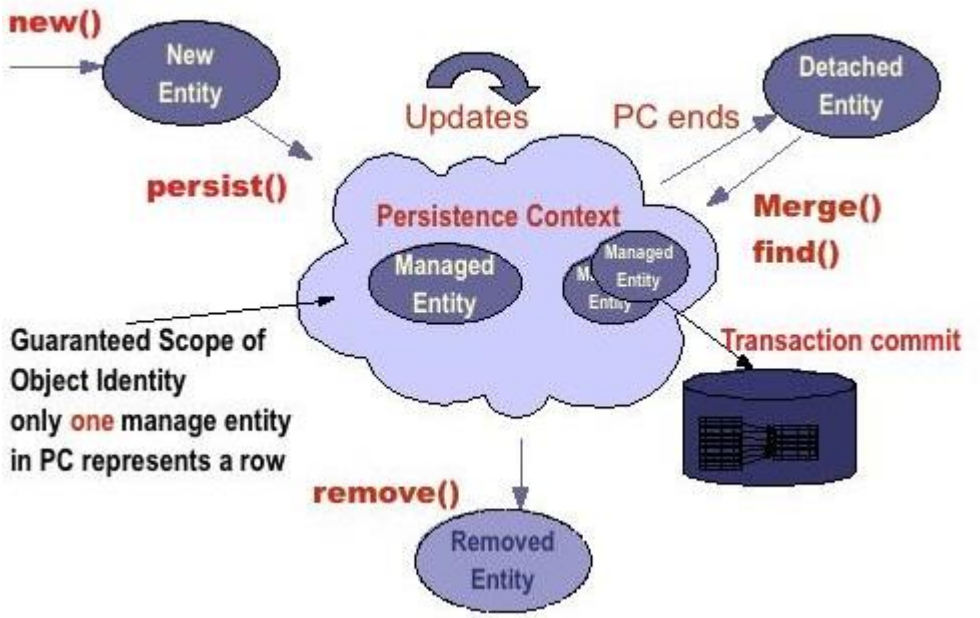
The Entity Manager guarantees that within a single Persistence Context, for any particular database row, there will be only one object instance. However the same entity could be managed in another User's transaction, so you should use either optimistic or pessimistic locking as explained in JPA 2.0 Concurrency and locking



The code below shows that a find on a managed entity with the same id and class as another in the same persistence context , will return the same instance.

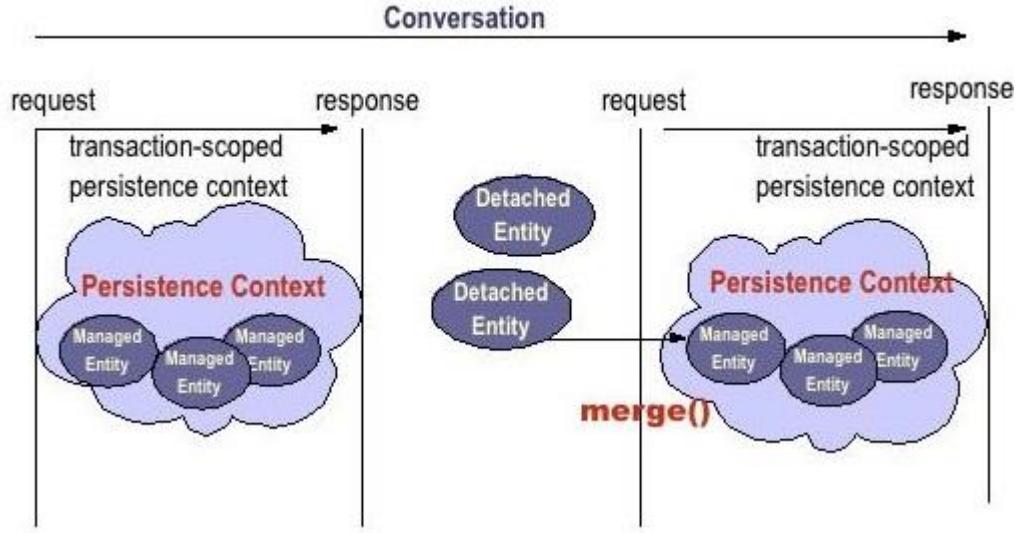
```
@Stateless public ShoppingCartBean implements ShoppingCart {

 @PersistenceContext EntityManager entityManager;

 public OrderLine createOrderLine(Product product,Order order) {
        OrderLine orderLine = new OrderLine(order, product);
        entityManager.persist(orderLine);    //Managed
        OrderLine orderLine2 =entityManager.find(OrderLine,
orderLine.getId()));
     (orderLine == orderLine2)    // TRUE
        return (orderLine);
    }

}
```

The diagram below shows the life cycle of an Entity in relation to the Persistent Context.



The code below illustrates the life cycle of an Entity. A reference to a container managed EntityManager is injected using the persistence context annotation. A new order entity is created and the entity has the state of new. Persist is called, making this a managed entity. because it is a stateless session bean it is by default using container managed transactions , when this transaction commits , the order is made persistent in the database. When the orderline entity is returned at the end of the transaction it is a detached entity.

The Persistence Context can be either Transaction Scoped-- the Persistence Context 'lives' for the length of the transaction, or Extended-- the Persistence Context spans multiple transactions. With a Transaction scoped Persistence Context, Entities are "Detached" at the end of a transaction.



As shown below, to persist the changes on a detached entity, you call the EntityManager's merge() operation, which returns an updated managed entity, the entity updates will be persisted to the database at the end of the transaction.

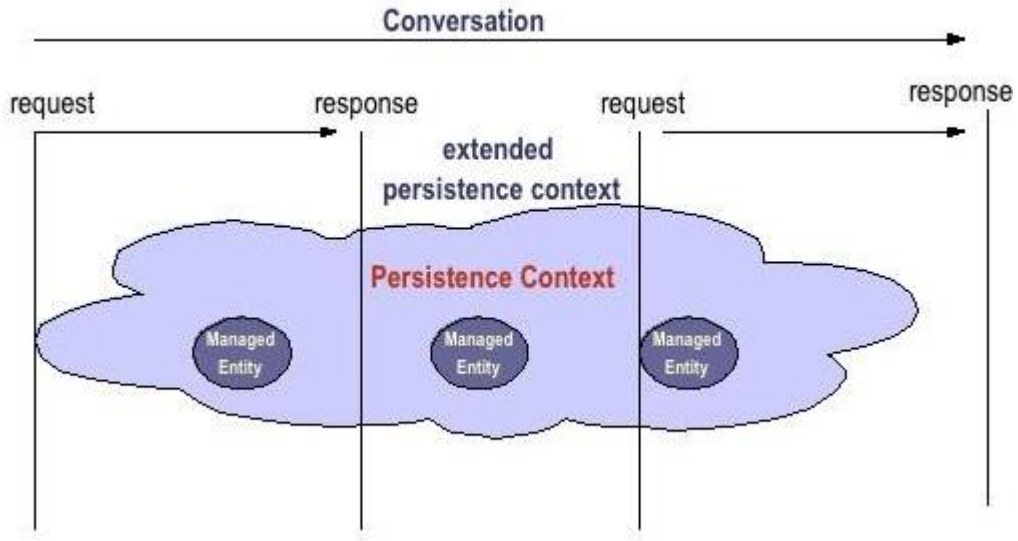
```
@Stateless public ShoppingCartBean implements ShoppingCart {
 @PersistenceContext EntityManager entityManager;

 public OrderLine createOrderLine(Product product,Order order) {
      OrderLine orderLine = new OrderLine(order, product);
      entityManager.persist(orderLine);
      return (orderLine);
   }

   public OrderLine updateOrderLine(OrderLine orderLine){
       OrderLine orderLine2 =entityManager.merge(orderLine));
       return orderLine2;
   }
}
```

Managed entity

Detached entity

Managed entity

An Extended Persistence Context spans multiple transactions, and the set of Entities in the Persistence Context stay Managed. This can be useful in a work flow scenario where a "conversation" with a user spans multiple requests.



The code below shows an example of a Stateful Session EJB with an Extended Persistence Context in a use case scenario to add line Items to an Order. After the Order is persisted in the createOrder method, it remains managed until the EJB remove method is called. In the addLineItem method , the Order Entity can be updated because it is managed, and the updates will be persisted at the end of the transaction.

```
@Stateful public class OrderMgr {

   //Specify that we want an EXTENDED
   @PersistenceContext(type=PersistenceContextType.EXTENDED)
         EntityManager em;

   //Cached order
   private Order order;

   //create and cache order
   public void createOrder(String itemId) {
      //order remains managed for the lifetime of the bean
      Order order = new Order(cust);
      em.persist(order);
   }

   public void addLineItem(OrderLineItem li){
       order.lineItems.add(li);
   }
```

Managed entity

Managed entity

The example below contrasts updating the Order using a transaction scoped Persistence Context verses an extended Persistence context. With the transaction scoped persistence context, an Entity Manager find must be done to look up the Order, this returns a Managed Entity which can be updated. With the Extended Persistence Context the find is not necessary. The performance advantage of not doing a database read to look up the Entity, must be weighed against the disadvantages of memory consumption for caching, and the risk of cached entities being updated by another transaction. Depending on the application and the risk of contention among concurrent

transactions this may or may not give better performance / scalability.

```
@Stateless
public class OrderMgr implements OrderService {

@PersistenceContext EntityManager em;
public void addLineItem(OrderLineItem li){
  // First, look up the order.
  Order order = em.find(Order.class, orderID);
  order.lineItems.add(li);
}
```

look up the order

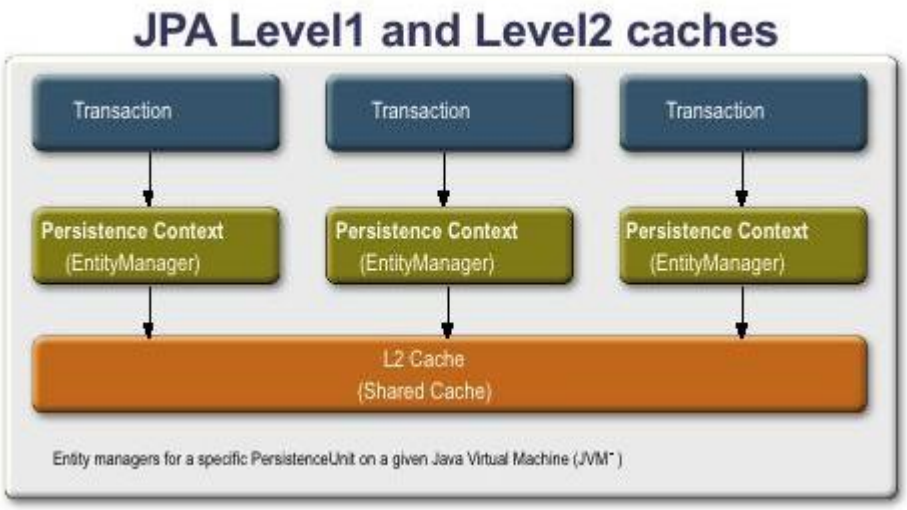
```
@Stateful
public class OrderMgr implements OrderService {
@PersistenceContext(type = PersistenceContextType.EXTENDED))
EntityManager em;
// Order is cached
Order order
public void addLineItem(OrderLineItem li){
    // No em.find invoked for the order object
    order.lineItems.add(li);
}
```

Managed entity

No em.find invoked

JPA second level (L2) caching

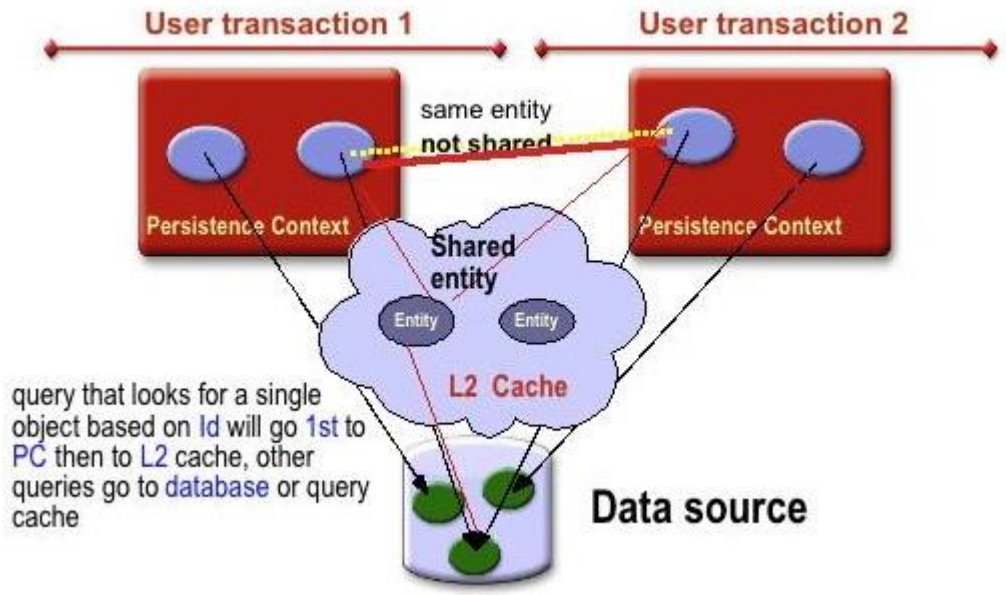
JPA second level (L2) caching shares entity state across various persistence contexts.



JPA 1.0 did not specify support of a second level cache, however, most of the persistence providers provided support for second level cache(s). JPA 2.0 specifies support for basic cache operations with the new Cache API, which is accessible from the EntityManagerFactory, shown below:

```
public class Cache {
//checks if object is in IdentityMap
public boolean contains(Class class, Object pk);
// invalidates object in the IdentityMap
public void evict(Class class, Object pk);
public void evict(Class class); // invalidates the class in the IdentityMap.
public void evictAll(); // Invalidates all classes in the IdentityMap
}
```

If L2 caching is enabled, entities not found in persistence context, will be loaded from L2 cache, if found.



The advantages of L2 caching are:

- avoids database access for already loaded entities
- faster for reading frequently accessed unmodified entities

The disadvantages of L2 caching are:

- memory consumption for large amount of objects
- Stale data for updated objects
- Concurrency for write (optimistic lock exception, or pessimistic lock)
- Bad scalability for frequent or concurrently updated entities

You should configure L2 caching for entities that are:

- read often
- modified infrequently
- Not critical if stale

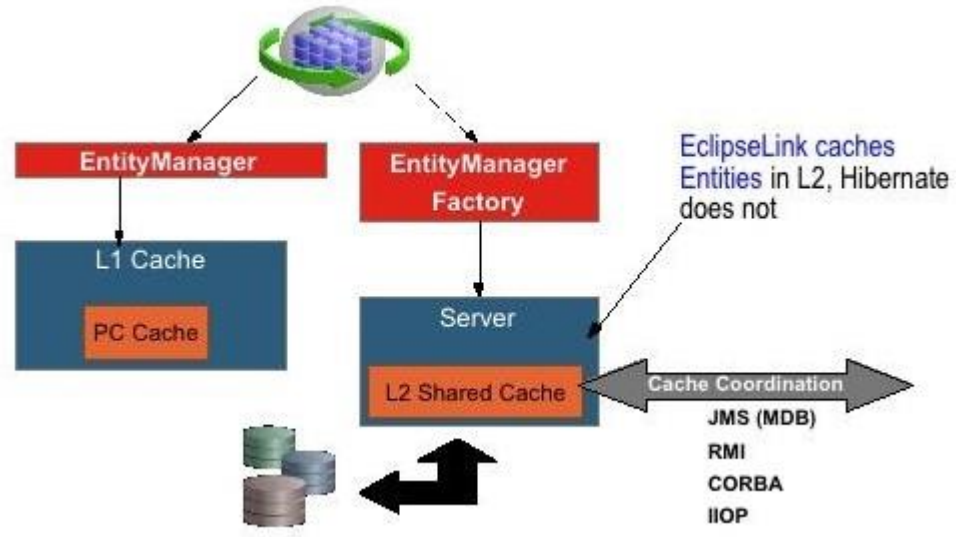
You should protect any data that can be concurrently modified with a locking strategy:

- Must handle optimistic lock failures on flush/commit
- configure expiration, refresh policy to minimize lock failures

The Query cache is useful for queries that are run frequently with the same parameters, for not modified tables.

The EclipseLink JPA persistence provider caching Architecture

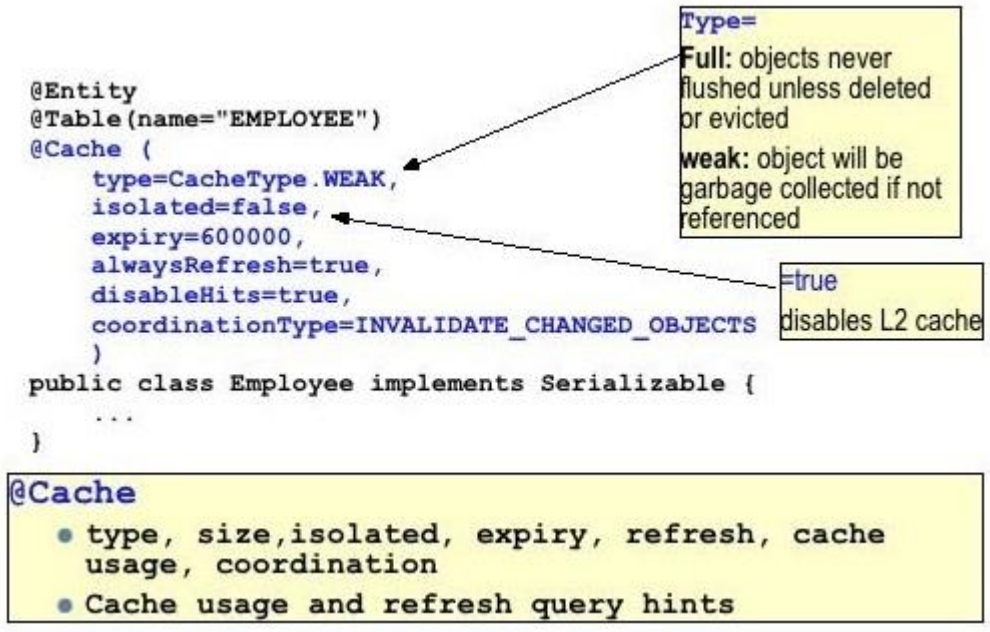
The EclipseLink caching Architecture is shown below.



Support for second level cache in EclipseLink is turned on by default, entities read are L2 cached. You can disable the L2 cache. EclipseLink caches entities in L2, Hibernate caches entity id and state in L2. You can configure caching by Entity type or Persistence Unit with the following configuration parameters:

- Cache isolation, type, size, expiration, coordination, invalidation,refreshing
- Coordination (cluster-messaging)
- Messaging: JMS, RMI, RMI-IIOP, …
- Mode: SYNC, SYNC+NEW, INVALIDATE, NONE

The example below shows configuring the L2 cache for an entity using the @Cache annotation



The Hibernate JPA persistence provider caching Architecture

The Hibernate JPA persistence provider caching architecture is different than EclipseLink: it is not configured by default, it does not cache enities just id and state, and you can plug in different L2 caches. The diagram below shows the different L2 cache types that you can plug into Hibernate.

-EHCache, OSCache, SwarmCacheProvider **(JVM)**
-JBoss TreeCache **Cluster**
-Can plug in others like Terracotta

		Cache Concurrency Strategy		
Cache	**Type**	**Read-only**	**Read-write**	**Transactional**
EHCache	memory, disk	Yes	Yes	
OSCache	memory, disk	Yes	Yes	
SwarmCache	clustered	Yes		
JBoss Cache	clustered	Yes		Yes

The configuration of the cache depends on the type of caching plugged in. The example below shows configuring the hibernate L2 cache for an entity using the @Cache annotation

not configured by default

```
<!-- optional configuration file parameter -->
net.sf.ehcache.configurationResourceName=/name_of_configuration_resource

@Entity
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
 public Class Country {
   private String name;
   ...
}
```

Cache Concurrency Strategy must be supported by cache provider

For More Information:

Introducing EclipseLink
EclipseLink JPA User Guide
Hibernate Second Level Cache
Speed Up Your Hibernate Applications with Second-Level Caching
Hibernate caching
Java Persistence API 2.0: What's New ?
Beginning Java™ EE 6 Platform with GlassFish™ 3
Pro EJB 3: Java Persistence API (JPA 1.0)