Managing Data Relationships

in

Distributed Cache

June 17, 2009

© Copyright 2009 by Alachisoft. All rights reserved

Introduction

A distributed cache lets you greatly improve application performance and scalability. Application performance is improved because an in-memory cache is a lot faster for data access than a database. And, scalability is achieved by growing the cache to multiple servers as a distributed cache and gaining not only more storage capacity but also more transactions per second throughput.

Despite such powerful benefits, there is one issue facing many in-memory caches. And that has to do with the fact that most data is relational whereas a cache is usually a simple hash-table with a key-value pair concept. Each item is stored in the cache independently without any knowledge of any other related items. And, this makes it difficult for applications to keep track of relationships among different cached items both for fetching them and also for data integrity in case one item is updated or removed and its related items are also updated or removed in the database. When this happens, the cache does not know about it and cannot handle it.

A typical real-life application deals with relational data that has one-to-one, many-to-one, one-to-many, and many-to-many relationships with other data elements in the database. This requires referential integrity to be maintained across different related data elements. Therefore, in order to preserve data integrity in the cache, the cache must understand these relationships and maintain the same referential integrity.

To handle these situations, Cache Dependency was introduced by Microsoft in ASP.NET Cache. Cache Dependency allows you to relate various cached elements and then whenever you update or remove any cached item, the cache automatically removes all its related cached items so as to ensure data integrity. Then, when your application does not find these related items in the cache the next time it needs them, the application goes to the database and fetches the latest copy of these items and then caches them again with correct referential integrity maintained.

This is a great feature in ASP.NET Cache but ASP.NET Cache is by design a stand-alone cache that is good only for single-server in-process environments. But, for scalability, you must use a distributed cache that can live outside of your application process and can scale to multiple cache servers. NCache is such a cache and fortunately provides the same Cache Dependency feature in a distributed environment. You can have cached items in one physical cache server depend on cached items in another physical cache server as long as they're both part of the same logical clustered cache. And, NCache takes care of all data integrity issues mentioned above.

This article explains how you can use Cache Dependency to handle one-to-one, one-to-many, and many-to-many relationships in the cache. It uses NCache as an example but the same concepts apply to ASP.NET Cache.

Although, NCache provides various types of dependencies including Cache Dependency, File Dependency, Sql Dependency, and Custom Dependency, this article only discusses the Cache Dependency for handling relationships among cached items.

What is Cache Dependency?

Cache Dependency is a feature that lets you specify that one cached item depends on another cached item. Then if the second cached item is ever updated or removed, the first item that was depending on it is also removed from the cache. Cache Dependency lets you specify multi-level dependencies where A depends on B which then depends on C. Then, if C is updated or removed, both A and B are removed from the cache.

Below is a brief example of how to use Cache Dependency to specify multi-level dependency.

```
public void CreateDependencies()
{
      Cache _cache = NCache.InitializeCache("myReplicatedCache");

      string keyC = "objectC-1000";
      ObjectC objC = new ObjectC();

      string keyB = "objectB-1000";
      ObjectB objB = new ObjectB();

      string keyA = "objectA-1000";
      ObjectA objA = new ObjectA();

      _cache.Add(keyC, objC, null,
                  Cache.NoAbsoluteExpiration,
                  Cache.NoSlidingExpiration,
                  CacheItemPriority.Default, null, null);

      string[] BDependsOn = { keyC };
      _cache.Add(keyB, objB, new CacheDependency(null, BDependsOn),
                  Cache.NoAbsoluteExpiration,
                  Cache.NoSlidingExpiration,
                  CacheItemPriority.Default, null, null);

      string[] ADependsOn = { keyB };
      _cache.Add(keyA, objA, new CacheDependency(null, ADependsOn),
                  Cache.NoAbsoluteExpiration,
                  Cache.NoSlidingExpiration,
                  CacheItemPriority.Default, null, null);

      // please note that by removing "C", you will also be
      // removing "A" & "B"
      _cache.Remove(keyC);
      _cache.Dispose();
}
```

Figure 1: Multi-layer Cache Dependency

Data Relationships

The following example is used in this article to demonstrate how various types of relationships are handled in the cache.

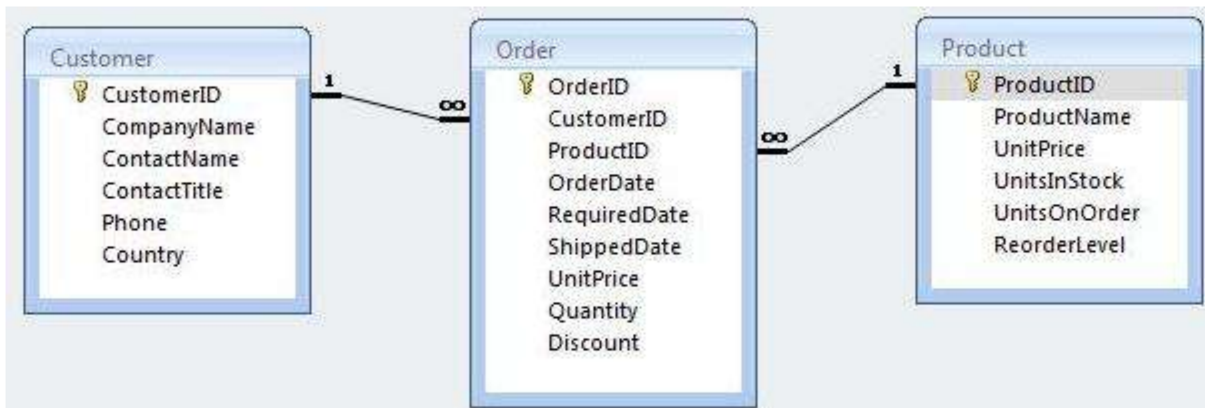

Figure 2: Relationships in the database

In the above diagram, the following relationships are shown:

1. **One to Many:** There are two such relationships and they are:
    a. Customer to Order
    b. Product to Order
2. **Many to One:** There are two such relationships and they are:
    a. Order to Customer
    b. Order to Product
3. **Many to Many:** There is one such relationship and that is:
    a. Customer to Product (via Order)

For the above relationships, the following domain objects are designed.

```
class Customer
{
    public string CustomerID;
    public string CompanyName;
    public string ContactName;
    public string ContactTitle;
    public string Phone;
    public string Country;

    public IList<Order> _OrderList;
}

class Product
{
    public int ProductID;
    public string ProductName;
    public Decimal UnitPrice;
    public int UnitsInStock;
    public int UnitsOnOrder;
    public int ReorderLevel;
```

```
        public IList<Order> _OrderList;
    }

    class Order
    {
        public int OrderId;
        public string CustomerID;
        public DateTime OrderDate;
        public DateTime RequiredDate;
        public DateTime ShippedDate;

        public int ProductID;
        public Decimal UnitPrice;
        public int Quantity;
        public Single Discount;

        public Customer _Customer;
        public Product _Product;
    }
```

As you can see, the Customer and Product classes contain an _OrderList to contain a list of all Order objects that are related to this customer. Similarly, the Order class contains _Customer and _Product data members to point to the related Customer or Product object. Now, it is the job of the persistence code that is loading these objects from the database to ensure that whenever a Customer is loaded, all its Order objects are also loaded.

Below, I'll show how each of these relationships is handled in the cache.

Handling One-to-One/Many-to-One Relationships

Whenever you have fetched an object from the cache that also has a one-to-one or many-to-one relationship with another object, your persistence code might have also loaded the related object. However, it is not always required to load the related object because the application may not need it at that time. If your persistence code has loaded the related object then you need to handle it.

There are two ways you can handle this. I will call one optimistic and one pessimistic way and will explain each of them below:

1. **Optimistic handling of relationships:** In this, we assume that even though there are relationships, nobody else is going to modify the related object separately. Whoever wants to modify the related objects will fetch it through the primary object in the cache and will therefore be in a position to modify both primary and related objects. In this case, we do not have to store both of these objects separately in the cache. Therefore, the primary object contains the related object and both of them are stored as one cached item in the cache. And, no Cache Dependency is created between them.
2. **Pessimistic handling of relationships:** In this case, you assume that the related object can be independently fetched and updated by another user and therefore the related object must be stored as a separate cached item. Then, if anybody updates or removes the related object, you want your primary object to also be removed

from the cache. In this case, you'll create a Cache Dependency between the two objects.

Below is the source code for handling the optimistic situation. Please note that both the primary object and both of its related objects are cached as one item because the serialization of the primary object would also include the related objects.

```
static void Main(string[] args)
{
    Cache _cache = NCache.InitializeCache("myReplicatedCache");

    OrderFactory oFactory = new OrderFactory();
    Order order = new Order();

    order.OrderID = 1000;
    oFactory.LoadFromDb(order);

    Customer cust = order._Customer;
    Product prod = order._Product;

    // please note that Order object serialization will
    // also include Customer and Product objects
    _cache.Add(orderKey, order, null,
                Cache.NoAbsoluteExpiration,
                Cache.NoSlidingExpiration,
                CacheItemPriority.Default);

    NCache.Cache.Dispose();
}
```

Figure 3: Optimistic handling of many-to-one relationship

Below is the source code for handling the pessimistic situation since the optimistic scenario does not require any use of Cache Dependency.

```
static void Main(string[] args)
{
    Cache _cache = NCache.InitializeCache("myReplicatedCache");

    OrderFactory oFactory = new OrderFactory();
    Order order = new Order();

    order.OrderID = 1000;
    oFactory.LoadFromDb(order);

    Customer cust = order._Customer;
    Product prod = order._Product;

    string custKey = "Customer:CustomerID:" + cust.CustomerID;
    _cache.Insert(custKey, Customer);

    string prodKey = "Product:ProductID:" + prod.ProductID;
    _cache.Insert(prodKey, prod);

    string[] depKeys = { prodKey, custKey };
    string orderKey = "Order:OrderID:" + order.OrderID;

    // We are setting _Customer and _Product to null so they
    // don't get serialized with Order object
    order._Customer = null;
    order._Product = null;
    _cache.Add(orderKey, order, new CacheDependency(null, depKeys),
               Cache.NoAbsoluteExpiration,
               Cache.NoSlidingExpiration,
               CacheItemPriority.Default);

    NCache.Cache.Dispose();
}
```

Figure 4: Pessimistic handling of many-to-one relationships

The above code loads an Order object from the database and both Customer and Product objects are automatically loaded with it because the Order object has a many-to-one relationship with them. The application then adds Customer and Product objects to the cache and then adds the Order object to the cache but with a dependency on both Customer and Product objects. This way, if any of these Customer or Product objects are updated or removed in the cache, the Order object is automatically removed from the cache to preserve data integrity. The application does not have to keep track of this relationship.

Handling One-to-Many Relationships

Whenever you have fetched an object from the cache that also has a one-to-many one relationship with another object, your persistence code may load both the primary object and a collection of all its one-to-many related objects. However, it is not always required to load the related objects because the application may not need them at this time. If your persistence code has loaded the related objects then you need to handle them in the cache.

Please note that the related objects are all kept in one collection and this introduces issues of its own that are discussed below.

There are three ways you can handle this. I will call one optimistic, one mildly pessimistic, and one really pessimistic way and will explain each of them below:

1. **Optimistic handling of relationships:** In this, we assume that even though there are relationships, nobody else is going to modify the related objects separately. Whoever wants to modify the related objects will fetch them through the primary object in the cache and will therefore be in a position to modify both primary and related objects. In this case, we do not have to store both of these objects separately in the cache. Therefore, the primary object contains the related object and both of them are stored as one cached item in the cache. And, no Cache Dependency is created between them.
2. **Mildly pessimistic handling of relationships:** In this case, you assume that the related objects can be independently fetched but only as the entire collection and never as individual objects. Therefore, you store the collection as one cached item and create a dependency from the collection to the primary object. Then, if anybody updates or removes the primary object, you want your collection to also be removed from the cache.
3. **Really pessimistic handling of relationships:** In this case, you assume that all objects in the related collection can also be individually fetched by the application and modified. Therefore, you must not only store the collection but also all their individual objects in the cache separately. Please note however that this would likely cause performance issues because you're making multiple trips to the cache which may be residing across the network on a cache server. I will discuss this in the next section that deals with "Handling Collections in Cache".

Below is an example of how you can handle one-to-many relationships optimistically. Please note that the collection containing the related objects is serialized as part of the primary object when being put in the cache.

```
static void Main3(string[] args)
{
    Cache _cache = NCache.InitializeCache("myReplicatedCache");

    CustomerFactory cFactory = new CustomerFactory();
    Customer cust = new Customer();

    cust.CustomerID = "ALFKI";
    cFactory.LoadFromDb(cust);

    // please note that _OrderList will automatically get
    // serialized along with the Customer object
    string custKey = "Customer:CustomerID:" + cust.CustomerID;
    _cache.Add(custKey, cust, null,
                Cache.NoAbsoluteExpiration,
                Cache.NoSlidingExpiration,
                CacheItemPriority.Default);

    NCache.Cache.Dispose();
}
```

Figure 5: Handling one-to-many relationship optimistically

Below is an example of how to handle one-to-many relationship mildly pessimistically.

```
static void Main(string[] args)
{
    Cache _cache = NCache.InitializeCache("myReplicatedCache");

    CustomerFactory cFactory = new CustomerFactory();
    Customer cust = new Customer();

    cust.CustomerID = "ALFKI";
    cFactory.LoadFromDb(cust);

    IList<Order> orderList = cust._OrderList;

    // please note that _OrderList will not be get
    // serialized along with the Customer object
    cust._OrderList = null;
    string custKey = "Customer:CustomerID:" + cust.CustomerID;
    _cache.Add(custKey, cust, null,
                Cache.NoAbsoluteExpiration,
                Cache.NoSlidingExpiration,
                CacheItemPriority.Default);
    // let's reset the _OrderList back
    cust._OrderList = orderList;

    string[] depKeys = { custKey };
    string orderListKey = "Customer:OrderList:CustomerId" + cust.CustomerID;
    _cache.Add(orderListKey, orderList,
                new CacheDependency(null, depKeys),
                Cache.NoAbsoluteExpiration,
                Cache.NoSlidingExpiration,
                CacheItemPriority.Default);

    NCache.Cache.Dispose();
}
```

Figure 6: Handling one-to-many relationship mildly pessimistically

In the above example, the list of Order objects that are related to this Customer are cached separately. The entire collection is cached as one item because we are assuming that nobody will direclty modify individual Order objects separately. The application will always fetch it through this Customer and modify and re-cache the entire collection again.

Another case is the pessimistic handling of one-to-many relationships which is similar to how we handle collections in the cache. That topic is discussed in the next section.

Handling Collections in the Cache

There are many situations where you fetch a collection of objects from the database. This could be due to a query you ran or it could be a one-to-many relationship returning a

collection of related objects on the "many" side. Either way, what you get is a collection of objects that must be handled in the cache appropriately.

There are two ways to handle collections as explained below:

1. **Optimistic handling of collections:** In this, we assume that the entire collection should be cached as one item because nobody will individually fetch and modify the objects kept inside the collection. The collection might be cached for a brief period of time and this assumption may be very much valid.
2. **Pessimistic handling of collections:** In this case, we assume that individual objects inside the collection can be fetched separately and modified. Therefore, we cache the entire collection but then also cache each individual object and create a dependency from the collection to the individual objects.

Below is an example of how to handle collections optimistically.

```
static void Main(string[] args)
{
    Cache _cache = NCache.InitializeCache("myReplicatedCache");

    CustomerFactory cFactory = new CustomerFactory();
    Customer cust = new Customer();

    IList<Customer> custList = cFactory.LoadByCountry("United States");

    string custListKey = "CustomerList:LoadByCountry:Country:United States";

    // please note that all Customer objects kept in custList
    // will be serialized along with the custList
    _cache.Add(custListKey, custList, null,
                Cache.NoAbsoluteExpiration,
                Cache.NoSlidingExpiration,
                CacheItemPriority.Default);

    NCache.Cache.Dispose();
}
```

Figure 7: Handling collections optimistically

In the above example, the entire collection is cached as one item and all the Customer objects kept inside the collection are automatically serialized along with the collection and cache. Therefore, there is no need to create any Cache Dependency here.

Below is an example of how to handle collections pessimistically.

```
static void Main(string[] args)
{
    Cache _cache = NCache.InitializeCache("myReplicatedCache");

    CustomerFactory cFactory = new CustomerFactory();
    Customer cust = new Customer();

    IList<Customer> custList = cFactory.LoadByCountry("United States");
    ArrayList custKeys = new ArrayList();

    // Let's cache individual Customer objects and also build
    // an array of keys to be used later in CacheDependency
    foreach (Customer c in custList)
    {
        string custKey = "Customer:CustomerID:" + c.CustomerID;
        custKeys.Add(custKey);
        _cache.Insert(custKey, c);
    }

    string custListKey = "CustomerList:LoadByCountry:Country:United States";

    // please note that this collection has a dependency on all
    // objects in it separately. So, if any of them are updated or
    // removed, this collection will also be removed from cache
    _cache.Add(custListKey, custList,
                new CacheDependency(null, custKeys.ToArray()),
                Cache.NoAbsoluteExpiration,
                Cache.NoSlidingExpiration,
                CacheItemPriority.Default);

    NCache.Cache.Dispose();
}
```
Figure 8: Handling collections pessimistically

In the above example, each object in the collection is cached as a separate item and then the entire collection is cache as well as one item. The collection has a Cache Dependency on all its objects that are cached separately. This way, if any of these objects are updated or remove, the collection is also removed from the cache.