Contents

The DevForce *EntityManager* maintains two caches that are used to determine caching behavior:

- An entity cache that contains every entity that gets added to an *EntityManager*.
- A **query cache** that contains every query that has been processed against a backend datastore that satisfies a specific set of criteria (described later in this section).

The rules by which a query's results are processed into the *EntityManager's* cache is determined by a QueryStrategy. When following the default, “normal” *QueryStrategy*, the *EntityManager* tries first to satisfy a query from data in its cache; if it cannot be satisfied by the cache, it then reaches out to the data source.

Query Cache

When a *EntityManager* begins to process a normal query, it checks its query cache to see if it has processed this exact query before. If the *EntityManager* finds the query in the query cache, it assumes that the entities which are needed to satisfy the query are in the entity cache; accordingly, it satisfies the query entirely from the entity cache without consulting the data source.

A one-to-many entity navigation, such as from an Employee to the Employee’s Orders, is translated implicitly to an *IEntityQuery* instance that also enters the query cache. The next time the application navigates from that same Employee to its Orders, the *EntityManager* will recognize that it has performed the query before and look only in the cache for those Orders.

If you use an EntityKeyQuery, DevForce performs a safe optimization and can check the cache first even if the query has not been cached. The *EntityKeyQuery* also occurs when performing scalar navigation, such as from an OrderDetail to its parent Order.

The query cache grows during the course of a session. Certain operations clear it in order to maintain cache coherency; removing an entity from the cache is one such operation. The developer can also clear the query cache explicitly, or add queries to the cache that he or she knows can be satisfied from the cache.

DevForce caches queries to improve performance. This analysis applies to both entity queries and entity navigation. Both use the *Optimized* fetch strategy by default. Consider a query for employees with **FirstName** = "Nancy". The *QueryStrategy* is *Normal* which means the fetch strategy is *Optimized*, which means that the retrieval will be from cache when possible.

When we execute this query in an empty *EntityManager*, there will be a trip across the network to fetch the entities from the data source. We get back "Nancy Davolio" and "Nancy Sinatra". If we execute the query again, the *EntityManager* satisfies the query from the entity cache and returns the same result; it does not seek data from the data source.

During the first run, the *EntityManager* stored the query in its Query Cache. The *EntityManager* stores the query in the query cache when (a) the query is successful, (b) it searched the data source (not just the cache), and c) the query is invertible. The second time it found the query in the Query Cache and thus knew it could apply the cache to the query instead.

If we change "Nancy Davolio" to "Sue" and run the query again, we get back just "Nancy Sinatra". If we change "Sally Wilson" to "Nancy Wilson" and run it again, we’ll get the principals of a strange duet. So far, everything is working fine.

Meanwhile, another user saves "Nancy Ajram" to the data source. We run our query again and … we still have just a duet. The *EntityManager* didn’t go to the data source so it doesn’t find the Lebanese pop star.

Such behavior may be just fine for this application. If it is not, the developer has choices. She can:

- Use a *QueryStrategy* with a different *FetchStrategy* that looks at the database first. This is typically the best practice.
- Clear the query cache explicitly by calling *EntityManager.QueryCache Clear* method.
- Clear the query cache implicitly by removing any entity from the entity cache.

You can also find other methods to manipulate the query cache by using the *EntityManager's QueryCache* property.

Removing an entity clears the query cache

When we remove an entity from an *EntityManager*’s entity cache (via the *RemoveEntity* or *RemoveEntities* methods), DevForce automatically clears the entire query cache for that EntityManager.

Note that we are NOT talking about deleting an entity here. Deletion does not cause a clearing of the query cache.

Suppose we frequently query for employees hired this year. If we issue this query twice, the first query fetches the employees from the database; the second retrieves them from the cache. The second query is almost instantaneous.

Then we remove an unrelated entity such as a **Customer** or an **Address**. We query again. Instead of reading from the cache as it did before, the *EntityManager* goes back to the database for these employees.

Seems unfair, doesn't it? But it's the safe thing to do.

If we issue the same query multiple times, we expect the same results every time. We expect a different result only if data relevant to our query have changed.

The *EntityManager* will search the local cache instead of the database only if it "believes" that all essential information necessary to perform the query are resident in the cache. If it "thinks" that the cache has been compromised, it should go back to the data source to satisfy the query.

Removing an entity compromises the cache. For sure it invalidates at least one query – the query that fetched it in the first place. But is that the only invalidated query? The *EntityManager* does not know. So it does the safe thing and forgets all queries.

You and I know (or we think we know) that removing a **Customer** or **Address** has no bearing on employees hired this year. The *EntityManager* cannot be so sure.

There are circumstances when (a) we have to remove an entity and (b) we are certain that no queries will be adversely affected. For example, our query may return entities which we've marked as inactive. We never want inactive entities in our cache but, for reasons we need not explain here, we have inactive entities in the cache.

We want to remove those entities. Being inactive they cannot possibly contribute to a correct query result.

Unfortunately, removing those entities clears the entire query cache. The *EntityManager* will satisfy future queries from the database until it has rebuilt its query cache.

This is not a problem if we rarely have to purge inactive entities. But what if we have to purge them after almost every query? (This is not a common scenario.) We will never have a query cache and we will always search the database. The performance of our application will degrade.

Fortunately, there is a *RemoveEntities* signature that can remove entities without clearing the query cache. In the full knowledge of the risk involved, we can call

EntityManager.RemoveEntities(entitiesToRemove, false)

The *false* parameter tells the *EntityManager* that is should not clear the query cache.

Remember: removing an entity and deleting it are different operations. Removing it from the cache erases it from client memory; it says nothing about whether or not the entity should be deleted from its permanent home in remote storage. "Delete", on the other hand, is a command to expunge the entity from permanent storage. The "deleted" entity stays in cache until the program can erase it from permanent storage.

Cache use when disconnected

When the *EntityManager* is in the disconnected state, it will satisfy a navigation, or a query submitted with the *Normal* QueryStrategy, from the entity cache alone; it will not attempt to search the data source. The *EntityManager* raises an exception if it discovers during query processing that it can't reach the data source and a query strategy has been specified that requires that the data source be accessed. See Take offline to learn more about disconnected operations.

Modifications

Each business object carries a read-only *EntityState* property that indicates if the object is new, modified, marked for deletion, or unchanged since it was last retrieved.

It bears repeating that our local modifications affect only the cached copy of a business object, not its version in the data source. The data source version won't be updated until the application tells the *EntityManager* to save the changed object.

It follows that the data source version can differ from our cached copy either because we modified the cached copy or because another user saved a different version to the data source after we retrieved our copy.

It would be annoying at best if the *EntityManager* overwrote our local changes each time it queried the data source. Fortunately, in a *normal* query, the *EntityManager* will only replace an *unmodified* version of an object already in the cache; our modified objects are preserved until we save or undo them.

Stale entity data

All of this is convenient. But what if another user has made changes to a cached entity? The local application is referencing the cached version and is unaware of the revisions. For the remainder of the user session, the application will be using out-of-date data.

The developer must choose how to cope with this possibility. Delayed recognition of non-local changes is often acceptable. A list of U.S. States or zip codes is unlikely to change during a user session. Employee name changes may be too infrequent and generally harmless to worry about. In such circumstances the default caching and query behavior is fine.

> If concurrency checking is enabled and the user tries to save a changed object to the data source, DevForce will detect the collision with the previously modified version in the data source. The update will fail and DevForce will report this failure to the application which can take steps to resolve it.

Some objects are so volatile and critical that the application must be alert to external changes. The developer can implement alternative approaches to maintaining entity currency by invoking optional DevForce facilities for managing cached objects and forcing queries that go to the data source and merge the results back into the cache.

The facilities for this are detailed in the Query Strategy section.