Contents

This page describes a number of ways to **improve or optimize your query performance** within a DevForce application. It has been our experience however, that much, if not most of the time, performance bottlenecks are not where a developer thinks they are. Premature optimization often results in little or no performance gains for a substantial expense in time and effort. We strongly recommend that benchmarking one of the more important tools you use in attempting to optimize the performance of your application. In other words, benchmark before optimizing.

Logging

Among the most useful tools for determining if a query operation is taking too long is the *DebugLog*. DevForce provides the *DebugFns* and *TraceFns* classes in *IdeaBlade.Core* assembly that can be used to write out timings for any query.

It is often important to distinguish the amount of time spent actually executing the query on the server from the amount of time spent transporting the results to the client and merging them into an EntityManager's cache.

On the server, query interception and the *EntityServerQueryInterceptor* can be used as interception points where both a query and the amount of time it takes to execute, on the server, can be logged.

On the client, the *EntityManager.Querying* and *EntityManager.Queried* (see query-client-lifecycle-events) events are good places to time the 'total cost' of a query.

Look at the SQL being executed

This can be accomplish either by turning on the *ShouldLogSqlQueries* option in the *IdeaBladeConfig.LoggingElement* (see App.config in detail) or using a SQL profiler. Microsoft's SQL profiler is an excellent tool for this task.

Profile the Entity Framework's performance

There are several tools available that profile Entity Framework performance. We recommend EFProf.

Entity Framework performance tips

Since most queries within DevForce get translated into calls to the Entity Framework, improving the performance of the Entity Framework query is an obvious first step. Begin by reading the Microsoft Entity Framework Performance Considerations topic.

Several techniques are worth exploring:

1. Entity Framework **"View Generation"**, a step that precedes the processing of the first query, can be time consuming for large models. Consider using Microsoft's EDMGen tool to "*pre-generate*" EF's "*Views*" of your model; Code First developers should refer to the Microsoft "**Entity Framework Power Tools**" (located here and described here) instead. We'll show you how to do it in our precompiled views topic.

   Use of EDMGen pre-generated views was made obsolete in EF 6. The newer approach, used by the EF Power Tools, requires that a DbContext is used; because of this, pre-generated views cannot be used with a DevForce "database first" model.
2. Entity Framework Compiled Queries may improve performance of frequently used expensive queries.

3. Add Includes -(see Using Includes). Use of the *Include* method can have an enormous impact on performance. In many n-tier applications the pipe between the client and the BOS is among the slowest parts of the system and multiple round trips to a server can impose a substantial performance burden. Judicious use of Includes can reduce the number of round trips substantially.

4. Remove Includes -(see Using Includes). As useful as are and despite the possibility of using them to improve performance; we have seen far too many examples of excessive use of **Includes**. The problem is that every *Include* causes a minimum of one additional query to be performed on the server. If this data is likely to be needed on the client within a relatively short period of time then it often makes sense to reduce roundtrips by 'prefetching' this data. However, if the data happens to not be needed on the server or is only needed in a small number of use cases, then allowing this data to be fetched as needed ('lazy queries') is often the much better choice.

5. Consider querying data asynchronously. Asynchronous queries don't execute any faster, but they won't block your UI and will make an application appear more responsive.

6. Consider using Stored Procedure Queries or E-SQL queries instead of LINQ for queries in some situations.

7. Consider using database views and mapping some of your entities to these views.

8. After *ViewGeneration*, metadata lookup is the next most time consuming part of the Entity Framework. However, because metadata lookup is globally cached per Application Domain, it only occurs once when the first query is passed to a DevForce Entity Server. Obviously, this means that the cost of spinning up an EntityServer (which should occur rarely) can be expensive. This is not usually a problem with distributed n-tier (and Silverlight) applications because this effect is only felt once by the first user with his/her first query. However, when testing 2-tier, the EntityServer runs as part of the client process and is spun up each time an application restarts.

9. Consider breaking up your model into smaller submodels. Model's with more than 500 entity types can be very cumbersome and both *ViewGeneration* and metadata lookup can be expensive.

10. Favor TPH (Table per hierarchy) over TPT (Table per type) inheritance. (See the article, 'Performance Considerations when using TPT (Table per Type) Inheritance in the Entity Framework'.)

11. Consider using **paging** when returning large result sets. In some cases the time spent moving large amounts of data over the wire can be drastically reduced by filtering the data returned with the Take and Skip operators. (See Query Paging).