CQLinq Performance

Table of contents

1. Introduction

This document assumes that you are familiar with the C# LINQ syntax and have read the document about the CQLinq syntax. Also please have a look at the wikipedia definition for time complexity if you don't know what this notion means.

CQLinq is designed to run hundreds of queries per seconds against a large real-world code base. This means that most CQLinq queries should be executed in a few milliseconds in theory. In practices, this is true for most queries, but if you look at the set of default CQLinq queries and rules, you'll see that a few of them are executed in a few dozens of milliseconds on large code bases.

The default value for the time-out for CQLinq query execution duration is equals to two seconds, but this value is easily changeable in the Tools --> Options --> Code Query panel.

While writing the set of dozens of default CQLinq rules and queries, we have adapted the CQLinq design to make sure that it is always possible to run quickly even complex queries. The result of this work is shared in the present document.

Performance is an important topic for CQLinq, because the philosophy of the CppDepend tool is to provide useful feedbacks to the user as quickly as possible, in a few seconds.

Always strive for linear time complexity	**Use sequence usage operations if possible**	**Declare sub-sets before the main query loop**
Rely extensively on hashset	**Avoid many let clauses in the main query loop**	**Performance with many strings constants**

2. Always strive for linear time complexity

When writing a complex query that needs some sort of nested processing, often the most obvious approach is to nest a query inside another one. This is illustrated by the query below, where we are interested to match all methods that calls any method named Add:

```
from m in Methods
from users in Methods
where m.SimpleName == @"Add" && users.IsUsingMethod(m)
select users
```

The problem with this approach is that it leads to query that are executed in a slow polynomial time complexity (O (#Method^2) here).

Thanks to the CQLinq flexibility, in most cases it is possible to transform a slow polynomial time complexity, into a linear time complexity. For example our query can be rewritten:

```
let addMethods =
   from m in Methods
   where m.SimpleName == @"Add"
   select m

from m in addMethods
from user in m.MethodsCallingMe
select user
```

The query has now a linear time complexity O(#Methods) and concretely it gets executed in a few milliseconds, instead of several dozens of seconds!

Notice that here we rely on the fact that CQLinq allows a query to begin with a let clause (See the CQLinq Syntax for more details).

3. Use sequence usage operations if possible

Actually, the query obtained in the section above can be rewritten to be even faster and more concise thanks to the extension method UsingAny() defined in theExtensionMethodsSequenceUsage class.

```
Methods.UsingAny(Methods.WithSimpleName(@"Add")).Select(m => m)
```

Getting used to extension methods defined in the class ExtensionMethodsSequenceUsage is a good practice because they often leads to both faster and more concise CQLinq queries. Internally, these extension methods implementations have been optimized.

Let's take another example to match types that implement any interface defined in the namespace System. This can be written this way:

```
let interfaces = Namespaces.WithName("System").ChildTypes().Where(t =
> t.IsInterface)
from t in Application.Types
from i in interfaces
where t.Implement(i)
select t
```

But by using the extension method ThatImplementAny() tests shows that the rewritten version of query runs 10 times faster.

```
Types.ThatImplementAny(
  Namespaces.WithName("System").ChildTypes().Where(t => t.IsInterface
)
).Select(t => t)
```

The internal optimization of these extension methods is based on the fact that they actually replace a loop. Hence such implementation is free to rely on a smarter algorithm to filter the input sequence faster than with a loop. By implementing the same internal algorithm we can then rewrite our query to be as fast as the version calling ThatImplementAny().

This would look like:

```
let interfaces = Namespaces.WithName("System").ChildTypes().Where(t =
> t.IsInterface)
from i in interfaces
from t in i.TypesThatImplementMe
select t
```

4. Declare sub-sets before the main query loop

If you need to query over a sub-set of the code base, make sure to define this sub-set once for all, before the main query loop.

For example the following query…

```
from m in Application.Methods where
  m.IsUsing("System.GC.Collect()".AllowNoMatch()) ||
  m.IsUsing("System.GC.Collect(Int32)".AllowNoMatch()) ||
  m.IsUsing("System.GC.Collect(Int32,GCCollectionMode)".AllowNoMatch(
))
select m
```

…Can be rewritten this way, to be 5 to 10 times faster.

```
let gcCollectMethods = ThirdParty.Methods.WithFullNameIn(
   "System.GC.Collect()",
   "System.GC.Collect(Int32)",
   "System.GC.Collect(Int32,GCCollectionMode)")
from m in Application.Methods.UsingAny(gcCollectMethods)
select m
```

5. Rely extensively on hashset

The System.Collections.Generic.HashSet<T> class is essential to implement high performance algorithms. Indeed this class represents a collection on which theContains(T) method is executed in a constant time O(1) (i.e constant no matter the collection size!).

The ExtensionMethodsSet class offers several extension methods to work more effectively with the HashSet<T> class. The most important one is the extension method ToHashSet() that transforms any enumerable in a hashset. Let's precise that for objects instances of NDepend.API classes, an effective internal hash algorithm is provided and the user doesn't have to worry for that.

Concretely, when a query relies on set operations (union, intersection...) it is often performance wise to transform enumerables into hashsets. For example, by removing the call to the extension method ToHashSet(), the following queries is more than 200 times slower!

```
// <Name>Callers of refactored methods</Name>
let refactoredMethods = Application.Methods.Where(m => m.CodeWasChang
ed()).ToHashSet()
from caller in Application.Methods.UsingAny(refactoredMethods)
let refactoredMethodsCalled = caller.MethodsCalled.Intersect(refactor
edMethods)
where refactoredMethodsCalled.Count() > 0
select new { caller, refactoredMethodsCalled }
```

6. Avoid many let clauses in the main query loop

Defining a range variable through a let clause is a convenient syntax possibility offered by LINQ. The problem is that this syntax bonus can significantly slow down query execution because under the hood, each let clause forces to create a new object and copy all values already obtained before its declaration.

So we have here a trade-off here between performance and syntax elegance. The performance doesn't necessarily win, for example we decided to keep this default rule with 3 let clauses...

```
// <Name>CRAP methods</Name>
// Source: http://www.artima.com/weblogs/viewpost.jsp?thread=215899
from method in Application.Methods
where method.CyclomaticComplexity != null &&

method.PercentageCoverage != null
let CC = method.CyclomaticComplexity
let uncov = (100 - method.PercentageCoverage) / 100f
let CRAP = (CC * CC * uncov * uncov * uncov) + CC
where CRAP > 30
orderby CRAP descending, method.NbLinesOfCode descending
select new { method, CRAP, CC, uncov, method.PercentageCoverage, meth
od.NbLinesOfCode }
```

...That is around two times slower than this much less elegant version with a single let clause:

```
// <Name>CRAP methods</Name>
// Source: http://www.artima.com/weblogs/viewpost.jsp?thread=215899
from method in Application.Methods where method.CyclomaticComplexit
y != null && method.PercentageCoverage != null
let CRAP = (method.CyclomaticComplexity * method.CyclomaticComplexity
*             ((100 - method.PercentageCoverage) / 100f)*
             ((100 - method.PercentageCoverage) / 100f)*
             ((100 - method.PercentageCoverage) / 100f))

            + method.CyclomaticComplexity
where CRAP > 30
orderby CRAP descending, method.NbLinesOfCode descending
select new {method,
             CRAP, CC = method.CyclomaticComplexity ,
             uncov = ((100 - method.PercentageCoverage) / 100f),
             method.PercentageCoverage, method.NbLinesOfCode }
```

7. Performance with many string constants

It might happen that a query needs to enumerate a list of code elements names to match them. For example:

```
from t in Types where
t.Name == "Int32" || t.Name == "UInt32" || t.Name == "Int16" || t.Nam
e == "UInt16" ||
t.Name == "Int64" || t.Name == "UInt64" || t.Name == "Byte" || t.Nam
e == "SByte" ||
t.Name == "Single" || t.Name == "Double" || t.Name == "Decimal"
select t
```

On a very large code base with 50.000 types this query takes 25ms at best to run. A small optimization is possible to avoid calling again and again the property Nameon t by using an override of the method EqualsAny():

```
from t in Types where
t.Name.EqualsAny("Int32","UInt32", "Int16","UInt16",
                 "Int16","UInt16", "Byte","SByte",
                 "Single","Double", "Decimal")
select t
```

Now, this version of the query takes at best 20ms to run. The small performance gain is compensated by the fact that the 9 string parameters are passed again and again to the method EqualsAny().

An idea is to use an instance of HashSet<string> to get a string comparison in a constant time:

```
let hashset = new [] { "Int32","UInt32", "Int16","UInt16",
                       "Int16","UInt16", "Byte","SByte",
                       "Single","Double", "Decimal" }.ToHashSet()
from t in Types where
hashset.Contains(t.Name)
select t
```

Unfortunatly this version is much slower with a best run time equals to 150ms, because under the hood, the let clause provoques a performance hit for each loop. If we were facing dozens of string constants to compare with, this version with HashSet could end up being faster.

However the class ExtensionMethodsNaming presents the extension method WithNameIn() that can be used this way:

```
Types.WithNameIn("Int32","UInt32", "Int16","UInt16",
                 "Int16","UInt16", "Byte","SByte",
                 "Single","Double", "Decimal").Select(t => t)
```

This version is now much faster with a best run time of 12ms because it removes the need for a LINQ loop, and internally replaces it with a faster loop based on thefor syntax, coupled with the usage of a HashSet<string> without the let performance hit.