Article
Timothy Leavitt · May 12, 2016 6m read

Writing better-performing loops in Caché ObjectScript

The topic of for/while loop performance in Caché ObjectScript came up in discussion recently, and I'd like to share some thoughts/best practices with the rest of the community. While this is a basic topic in itself, it's easy to overlook the performance implications of otherwise-reasonable approaches. In short, loops iterating over $ListBuild lists with $ListNext or over a local array with $Order are the fastest options.

As a motivating example, we will consider looping over the pieces of a comma-delimited string.

A natural way to write such a loop, in minimal code, is:

```
For i=1:1:$Length(string,",") {
 Set piece = $Piece(string,",",i)
 //Do something with piece...
}
```

This is straightforward, although many coding style guidelines might suggest:

```
Set n = $Length(string,",")
For i=1:1:n {
 Set piece = $Piece(string,",",i)
 //Do something with piece...
}
```

There isn't actually a performance difference between the two, since the end condition is not evaluated for each iteration. (I originally had this wrong - thanks to Mark for pointing out that it's just a style concern and not a performance issue.)

For the case of a delimited string, we can achieve better performance. As the string gets longer, $Piece(string,",",i) becomes more expensive: it must walk the string up to the end of the ith piece. This can be improved by using $ListBuild lists. For example, using $ListFromString, $ListLength, and $List:

```
Set list = $ListFromString(string,",")
Set n = $ListLength(list)
For i=1:1:n {
 Set piece = $List(list,i)
 //Do something with piece...
}
```

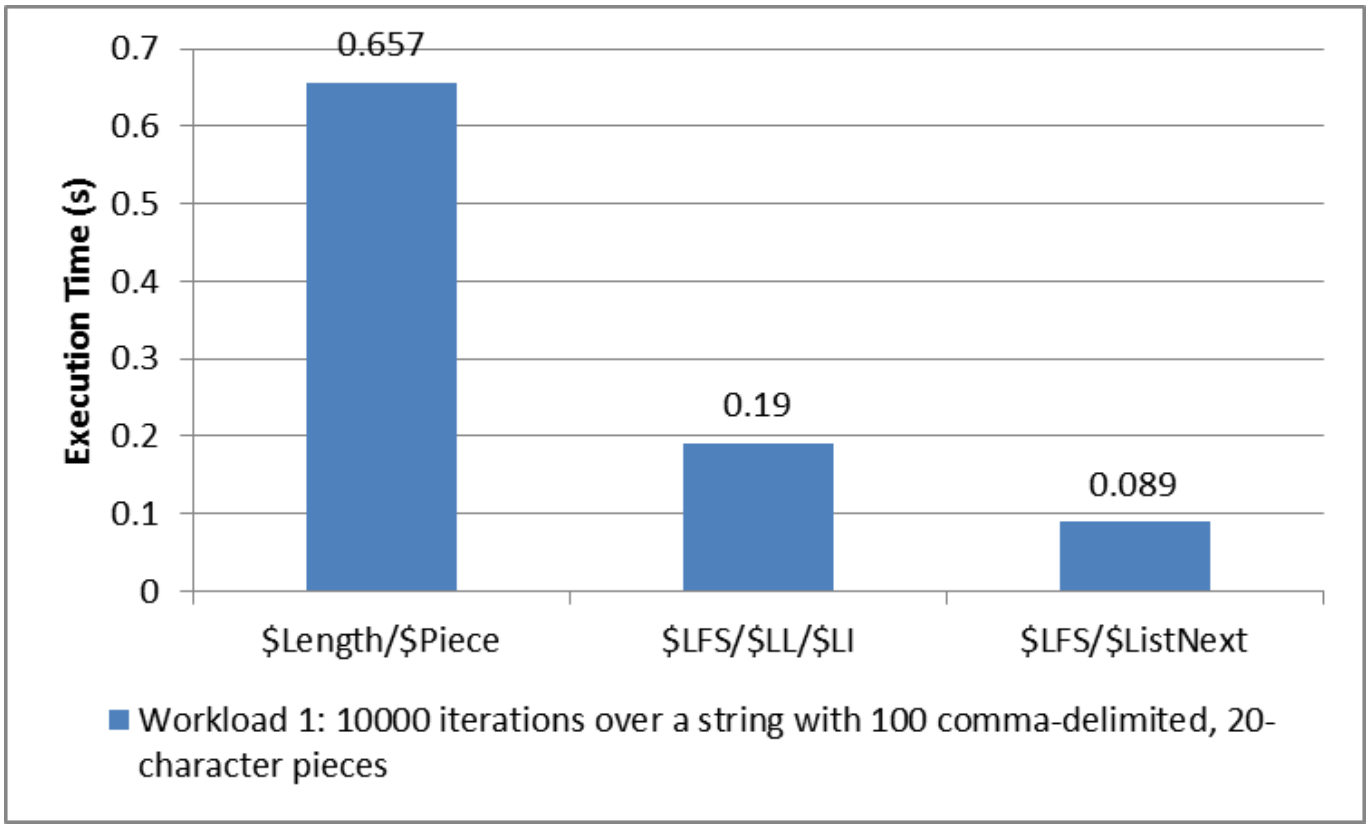
This will typically perform better than $Length/$Piece, particularly when the pieces are longer. In the $Length/$Piece approach, each of the n iterations scans the characters in the first i pieces. In the $ListFromString/$ListLength/$List approach, it now follows i pointers in the $ListBuild structure for each of the n iterations. We would expect this to perform better, and it does, but the execution time is still $O(n^2)$. Assuming that the loop will not change the list, we can do better - O(n) - by using $ListNext:

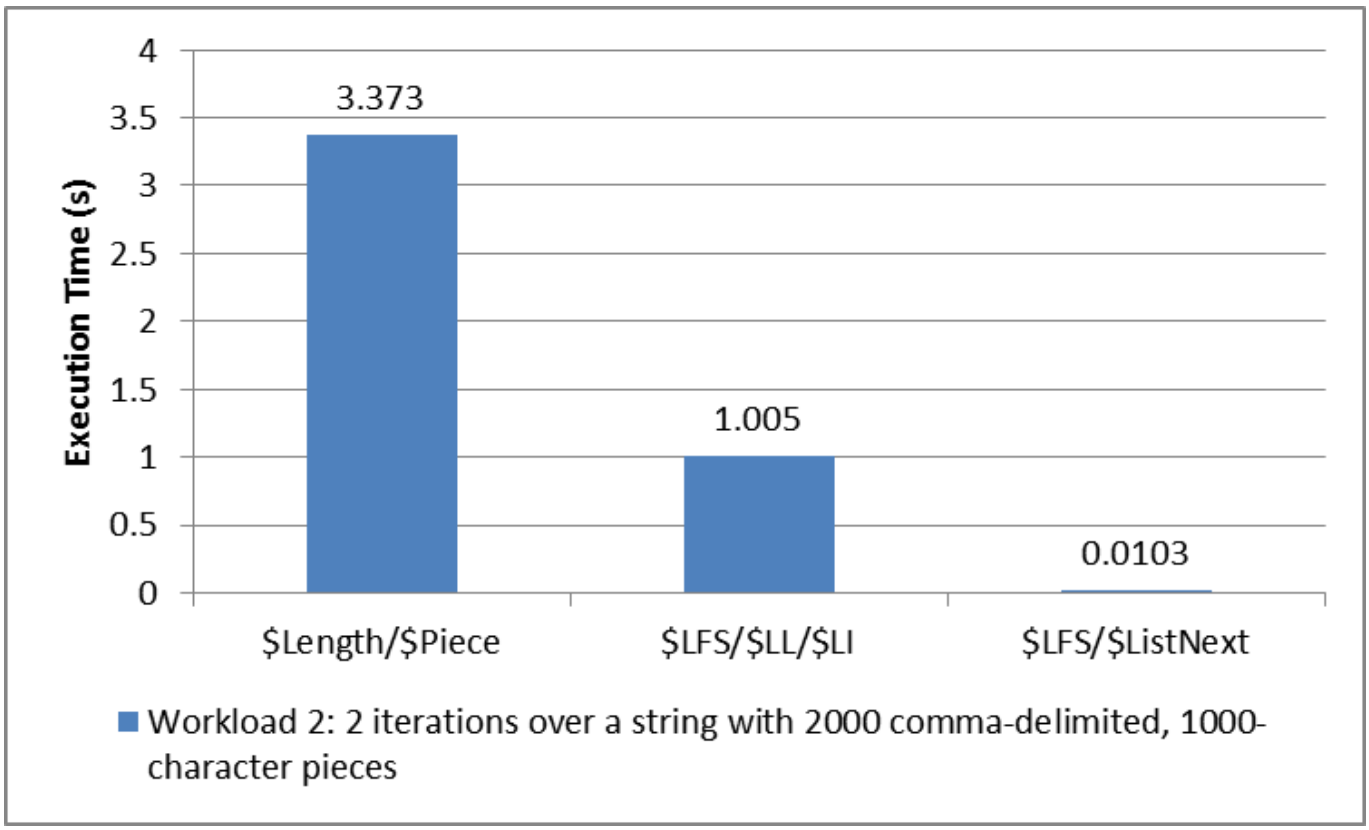
```
Set list = $ListFromString(string,",")
Set pointer = 0
While $ListNext(list,pointer,piece) {
 //Do something with piece...
}
```

Rather than following pointers through the list from the beginning to the ith piece each time like $List does, the variable " pointer " keeps track of the current position in the list. Therefore, instead of n(n+1)/2 total " follow the pointer " operations (i in each of the n iterations for $List), there are now just n of them (one in each iteration for $ListNext).

Finally, converting the string's pieces to array with integer subscripts may be a good option; in general, iterating over a local array with $Order may be slightly to significantly faster than $ListNext (depending on the length of the list elements). Of course, if you're starting out with a delimited string, it'll take a bit of work to convert it into an array. If you're iterating repeatedly, need to modify the parts of the list, or need to iterate backwards, this additional work would likely be worth it.

Here are some sample execution times (including all required conversions) for some different input sizes:





These numbers came from:

```
USER>d ##class(DC.LoopPerformance).Run(10000,20,100)
Iterating 10000 times over all the pieces of a string with 100 ,-delimited pieces of
length 20:
Using $Length/$Piece (hardcoded delimiter): .657383 seconds
Using $Length/$Piece: 1.083932 seconds
Using $ListFromString/$ListLength/$List (hardcoded delimiter): .189867 seconds
Using $ListFromString/$ListLength/$List: .189938 seconds
Using $ListFromString/$ListNext (hardcoded delimiter): .089618 seconds
Using $ListFromString/$ListNext: .089242 seconds
Using $Order over an equivalent local array with integer subscripts: .072485 seconds
****************************************************
Using $ListFromString/$ListNext (not including conversion to $ListBuild list): .05832
9 seconds
Using one-argument $Order over an equivalent local array with integer subscripts: .06
0327 seconds
Using three-argument $Order over an equivalent local array with integer subscripts: .
069508 seconds

USER>d ##class(DC.LoopPerformance).Run(2,1000,2000)
Iterating 2 times over all the pieces of a string with 2000 ,-delimited pieces of len
gth 1000:
Using $Length/$Piece (hardcoded delimiter): 3.372927 seconds
Using $Length/$Piece: 11.739316 seconds
Using $ListFromString/$ListLength/$List (hardcoded delimiter): 1.004757 seconds
Using $ListFromString/$ListLength/$List: .997821 seconds
Using $ListFromString/$ListNext (hardcoded delimiter): .010489 seconds
Using $ListFromString/$ListNext: .010268 seconds
Using $Order over an equivalent local array with integer subscripts: .000839 seconds
****************************************************
Using $ListFromString/$ListNext (not including conversion to $ListBuild list): .00305
```

```
3 seconds
Using one-argument $Order over an equivalent local array with integer subscripts: .00
0938 seconds
Using three-argument $Order over an equivalent local array with integer subscripts: .
000677 seconds
```

View the code (DC.LoopPerformance) on Gist

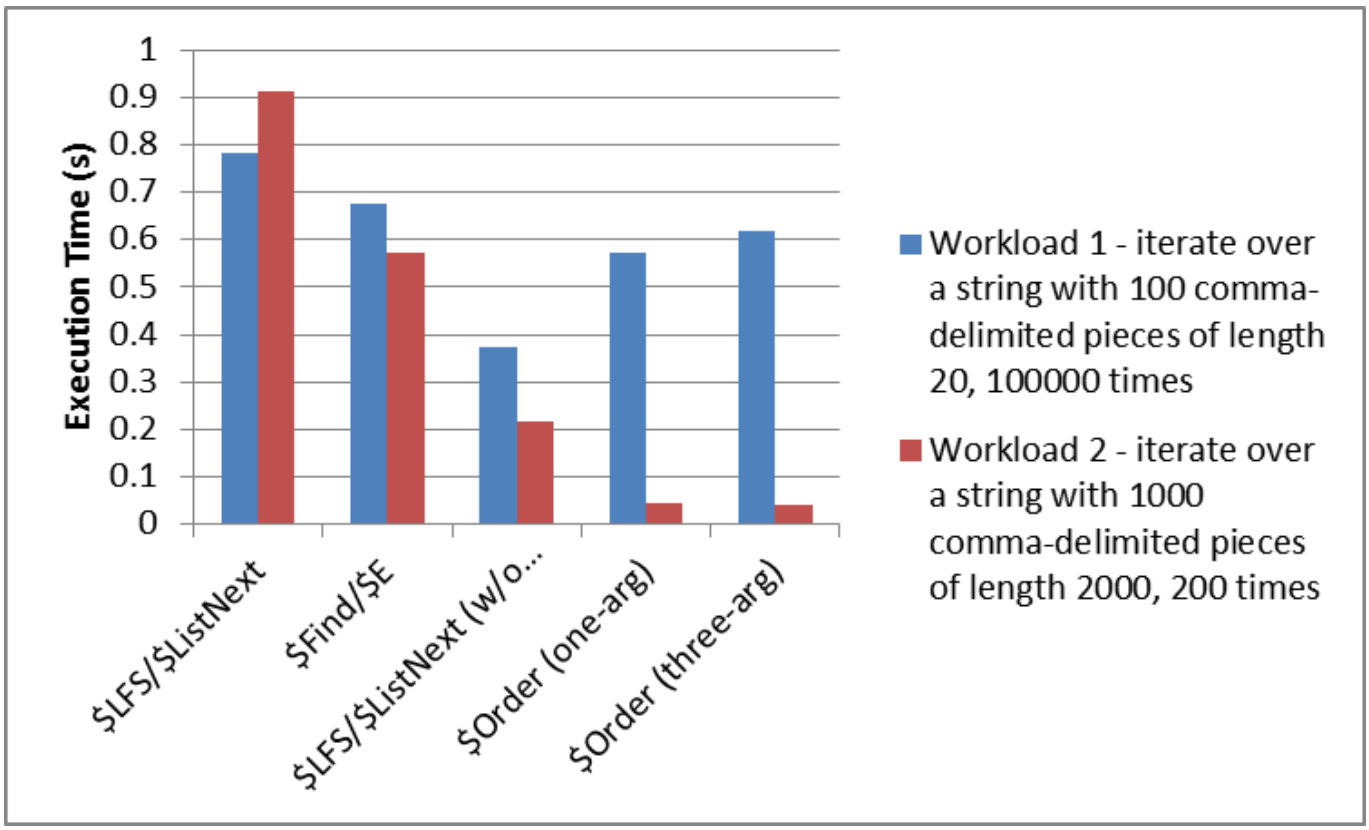
Update:

The discussion has turned up a few other interesting variations with good performance, which are worth comparing to each other. See the RunLinearOnly method and the different implementations tested in the updated Gist.

```
USER>d ##class(DC.LoopPerformance).RunLinearOnly(100000,20,100)
Iterating 100000 times over all the pieces of a string with 100 ,-delimited pieces of
 length 20:
Using $ListFromString/$ListNext (While): .781055 seconds
Using $ListFromString/$ListNext (For/Quit): .8438 seconds
Using $ListFromString/$ListNext (While, not including conversion to $ListBuild list):
 .37448 seconds
Using $Find/$Extract (Do...While): .675877 seconds
Using $Find/$Extract (For/Quit): .746064 seconds
Using one-argument $Order (For): .589697 seconds
Using one-argument $Order (While): .570996 seconds
Using three-argument $Order (For): .688088 seconds
Using three-argument $Order (While): .617205 seconds

USER>d ##class(DC.LoopPerformance).RunLinearOnly(200,2000,1000)
Iterating 200 times over all the pieces of a string with 1000 ,-delimited pieces of l
ength 2000:
Using $ListFromString/$ListNext (While): .913844 seconds
Using $ListFromString/$ListNext (For/Quit): .925076 seconds
Using $ListFromString/$ListNext (While, not including conversion to $ListBuild list):
 .21842 seconds
Using $Find/$Extract (Do...While): .572115 seconds
Using $Find/$Extract (For/Quit): .610531 seconds
Using one-argument $Order (For): .044251 seconds
Using one-argument $Order (While): .04467 seconds
Using three-argument $Order (For): .043631 seconds
Using three-argument $Order (While): .042568 seconds
```

The following chart compares the While/Do...While versions of these methods. Particularly, see the relative performance of $ListFromString/$ListNext compared to $Extract/$Find, and of $ListNext (without conversion from string to $ListBuild list) compared to $Order over a local array with integer subscripts.



In summary:

- If you're starting with a delimited string, $Find/$Extract is the best-performing option, although the code is slightly less intuitive than $ListFromString/$ListNext.
- Given a choice of data structures, traversal of $ListBuild lists seems to have a slight performance edge over an equivalent local array for small inputs, while the local array offers significantly better performance for larger inputs. I'm not sure why there's such a huge difference in performance. (Relatedly, it would be worth comparing the cost of random inserts and removals in a $ListBuild list vs. a local array with integer subscripts; I suspect that set $list would be much faster than shifting the values in such an array.)
- A While or Do...While loop performs *slightly* better than an equivalent argumentless For loop.

#Best Practices #Code Snippet #Coding Guidelines #ObjectScript #Tips & Tricks #Caché