Article
Alexander Koblov · Jul 29, 2016 10m read

On $Sequence function

In this article we are going to compare $Increment and $Sequence functions.

First of all, a note for readers who have never heard of $Increment. $Increment is a Caché ObjectScript function which performs an atomic operation to increment its argument by 1 and return the resulting value. You can only pass a global or local variable node as a parameter to $Increment, not an arbitrary expression. $Increment is heavily used when assigning sequential IDs. In such cases parameter of $Increment is usually a global node. $Increment guarantees that each process using it gets a unique ID.

```
for i=1:1:10000 {
     set Id = $Increment(^Person) ; new Id
     set surname = ##class(%PopulateUtils).LastName() ; random last name
     set name = ##class(%PopulateUtils).FirstName()  ; random first name
     set ^Person(Id) = $ListBuild(surname, name)
}
```

The problem with $Increment is that if many processes are adding rows in parallel, these processes might spend time waiting for their turn to atomically change value of global node that holds the ID — ^Person in the sample above

$Sequence is a new function that was designed to handle this problem. $Sequence is available since Caché 2015.1. Just like $Increment, $Sequence atomically increments value of its parameter. Unlike $Increment, $Sequence will reserve some subsequent counter values for current process and during the next call in the same process it will simply return the next value from the reserved range. $Sequence automatically calculates how many values to reserve. More often process calls $Sequence, more values $Sequence reserves:

```
USER>kill ^myseq

USER>for i=1:1:15 {write "increment:",$Seq(^myseq)," allocated:",^myseq,! }
increment:1 allocated:1
increment:2 allocated:2
increment:3 allocated:4
increment:4 allocated:4
increment:5 allocated:8
increment:6 allocated:8
increment:7 allocated:8
increment:8 allocated:8
increment:9 allocated:16
increment:10 allocated:16
increment:11 allocated:16
increment:12 allocated:16
increment:13 allocated:16
increment:14 allocated:16
increment:15 allocated:16
```

When $Sequence(^myseq) returned 9, next 8 values (up to 16) were already reserved for current process. If other

process calls $Sequence, it would get value of 17, not 10.

$Sequence is designed for processes that simultaneously increment some global node. Since $Sequence reserve values, IDs might have gaps if process does not use all of the values that were reserved. The main usage of $Sequence is generation of sequential IDs. Compared with $Sequence, $Increment is more generic function.

Let ' s compare performance of $Increment and $Sequence:

```
Class DC.IncSeq.Test
{

ClassMethod filling()
{
    lock +^P:"S"
    set job = $job
     for i=1:1:200000 {
         set Id = $Increment(^Person)
         set surname = ##class(%PopulateUtils).LastName()
         set name = ##class(%PopulateUtils).FirstName()
         set ^Person(Id) = $ListBuild(job, surname, name)
     }
     lock -^P:"S"
}

ClassMethod run()
{
    kill ^Person
    set z1 = $zhorolog
    for i=1:1:10 {
        job ..filling()
     }
     lock ^P
     set z2 = $zhorolog - z1
     lock
     write "done:",z2,!
}

}
```

Method run jobs off 10 processes, each inserting 200 ' 000 records into ^Person global. To wait until child processes will finish. method run tries to get exclusive lock on ^P. When child processes finish their job and release shared lock on ^P, run will acquire an exclusive lock on ^P and continue execution. Right after this we record time from $zhorolog system variable and calculate how much time it took to insert these records. My multi-core notebook with slow HDD took 40 seconds (for science, I ran it several times before, so this was 5th run):

```
USER>do ##class(DC.IncSeq.Test).run()
done:39.198488
```

It ' s interesting to drill down into these 40 seconds. By running %SYS.MONLBL we can see that total 100 seconds were spent getting ID. 100 seconds / 10 processes = each process spent 10 seconds to acquire new ID, 1.7 second to get first name and last name, and 28.5 seconds to write data to data global.

First column in %SYS.MONLBL report below is line number, second is how many times this line was executed, and third is how many seconds it took to execute this line.

```
 ; ** Source for Method 'filling' **
1            10    .001143    lock +^P:"S"
2            10    .000055    set job = $JOB
3            10    .000118     for i=1:1:200000 {
4       1998499 100.356554         set Id = $Increment(^Person)
5       1993866  10.409804         set surname = ##class(%PopulateUtils).LastName()
6       1990461   6.347832         set name = ##class(%PopulateUtils).FirstName()
7       1999762  285.54603         set ^Person(Id) = $ListBuild(job, surname, name)
8       1999825   3.393706     }
9            10    .000259     lock -^P:"S"
 ; ** End of source for Method 'filling' **
 ;
 ; ** Source for Method 'run' **
1             1    .005503    kill ^Person
2             1    .000002    set z1 = $zhorolog
3             1    .000002    for i=1:1:10 {
4            10    .201327        job ..filling()
5             0          0     }
6             1  43.472692     lock ^P
7             1     .00003     set z2 = $zhorolog - z1
8             1     .00001     lock
9             1    .000053     write "done:",z2,!
 ; ** End of source for Method 'run' **
```

Total time (43.47 seconds) is 4 seconds more than during previous run because of profiling.

Let ' s replace one thing in our test code, in filling method. We will change $Increment(^Person) to $Sequence(^Person) and run test again:

```
USER>do ##class(DC.IncSeq.Test).run()
done:5.135189
```

This result is surprising. Ok, $Sequence decreased time to get ID, but where did 28.5 seconds to store data in global go? Let ' s check ^%SYS.MONLBL:

```
 ; ** Source for Method 'filling' **
1            10    .001181    lock +^P:"S"
2            10    .000026    set job = $JOB
3            10    .000087     for i=1:1:200000 {
4       1802473   1.996279         set Id = $Sequence(^Person)
5       1784910   4.429576         set surname = ##class(%PopulateUtils).LastName()
6       1853508   3.829051         set name = ##class(%PopulateUtils).FirstName()
7       1838752  32.281624         set ^Person(Id) = $ListBuild(job, surname, name)
8       1951569     1.0243     }
9            10    .000219     lock -^P:"S"
 ; ** End of source for Method 'filling' **
 ;
 ; ** Source for Method 'run' **
1             1    .006514    kill ^Person
2             1    .000002    set z1 = $zhorolog
3             1    .000002    for i=1:1:10 {
4            10    .385055        job ..filling()
5             0          0     }
6             1   6.558119     lock ^P
7             1    .000011     set z2 = $zhorolog - z1
```

```
8             1    .000008     lock
9             1    .000025     write "done:",z2,!
 ; ** End of source for Method 'run' **
```

Now, each process spends 0.2 seconds instead of 10 seconds for ID acquisition. What is not clear is why storing data takes only 3.23 seconds per process? The reason is globals nodes are stored in data blocks, and usually each block has size of 8192 bytes. Before changing global node value (like set ^Person(Id) = ⋯), process locks the whole block. If several processes are trying to change data inside of the same block at the same time, only one process will be allowed to change the block and others will have to wait for it to finish.

Let ' s look at global created using $Increment to generate new IDs. Sequential records would almost never have the same process ID (remember -- we stored process ID as the first element of data list):

```
1:    ^Person(100000)    =    $lb("12950","Kelvin","Lydia")
2:     ^Person(100001)    =    $lb("12943","Umansky","Agnes")
3:     ^Person(100002)    =    $lb("12945","Frost","Natasha")
4:     ^Person(100003)    =    $lb("12942","Loveluck","Terry")
5:     ^Person(100004)    =    $lb("12951","Russell","Debra")
6:     ^Person(100005)    =    $lb("12947","Wells","Chad")
7:     ^Person(100006)    =    $lb("12946","Geoffrion","Susan")
8:     ^Person(100007)    =    $lb("12945","Lennon","Roberta")
9:     ^Person(100008)    =    $lb("12944","Beatty","Mark")
10:     ^Person(100009)    =    $lb("12946","Kovalev","Nataliya")
11:     ^Person(100010)    =    $lb("12947","Klingman","Olga")
12:     ^Person(100011)    =    $lb("12942","Schultz","Alice")
13:     ^Person(100012)    =    $lb("12949","Young","Filomena")
14:     ^Person(100013)    =    $lb("12947","Klausner","James")
15:     ^Person(100014)    =    $lb("12945","Ximines","Christine")
16:     ^Person(100015)    =    $lb("12948","Quine","Mary")
17:     ^Person(100016)    =    $lb("12948","Rogers","Sally")
18:     ^Person(100017)    =    $lb("12950","Ueckert","Thelma")
19:     ^Person(100018)    =    $lb("12944","Xander","Kim")
20:     ^Person(100019)    =    $lb("12948","Ubertini","Juanita")
```

Concurrent processes were trying to write data into the same block and were spending more time waiting than actually changing data. Using $Sequence, IDs are generated in chunks, so different processes would most likely use different blocks:

```
1:     ^Person(100000)    =    $lb("12963","Yezek","Amanda")
// 351 records with process number 12963
353:     ^Person(100352)    =    $lb("12963","Young","Lola")
354:     ^Person(100353)    =    $lb("12967","Roentgen","Barb")
```

If this sample looks like something you are doing in your projects, consider using $Sequence instead of $Increment. Of course, consult with documentation before replacing every occurrence of $Increment with $Sequence.

And sure, don ' t believe tests provided here -- double check this yourself.

Starting with Caché 2015.2, you can configure tables to use $Sequence instead of $Increment. There is a system function $system.Sequence.SetDDLUseSequence for that, and the same option is available from SQL Settings in Management Portal.

Also, there is new storage parameter in class definition -- IDFunction, which is set to " increment " by default, that

means that $Increment is used for Id generation. You can change it to “ sequence ” (Inspector > Storage > Default > IDFunction).

Bonus

Another quick test I conducted on my notebook: it ’ s a small ECP configuration with DB Server located on host operation system and Application Server on guest VM on the same notebook. I mapped ^Person to remote database. It is a basic test, so I don ’ t want to make generalizations based on it. There are things to consider when using $Increment and ECP. Having said that, here are the results:

With $Increment:

```
USER>do ##class(DC.IncSeq.Test).run()
done:163.781288
```

^%SYS.MONLBL:

```
 ; ** Source for Method 'filling' **
1            10    .000503         --     lock +^P:"S"
2            10    .000016    set job = $job
3            10    .000044    for i=1:1:200000 {
4       1843745 1546.57015        set Id = $Increment(^Person)
5       1880231   6.818051        set surname = ##class(%PopulateUtils).LastName()
6       1944594   3.520858        set name = ##class(%PopulateUtils).FirstName()
7       1816896  16.576452        set ^Person(Id) = $ListBuild(job, surname, name)
8       1933736    .895912    }
9            10    .000279    lock -^P:"S"
 ; ** End of source for Method 'filling' **
 ;
 ; ** Source for Method 'run' **
1             1    .000045    kill ^Person
2             1    .000001    set z1 = $zhorolog
3             1    .000007    for i=1:1:10 {
4            10    .059868        job ..filling()
5             0          0    }
6             1 170.342459    lock ^P
7             1    .000005    set z2 = $zhorolog - z1
8             1    .000013    lock
9             1    .000018    write "done:",z2,!
 ; ** End of source for Method 'run' **
```

$Sequence:

```
USER>do ##class(DC.IncSeq.Test).run()
done:13.826716
```

^%SYS.MONLBL

```
 ; ** Source for Method 'filling' **
1            10    .000434     lock +^P:"S"
2            10    .000014    set job = $job
3            10    .000033    for i=1:1:200000 {
```

```
4       1838247  98.491738        set Id = $Sequence(^Person)
5       1712000   3.979588        set surname = ##class(%PopulateUtils).LastName()
6       1809643   3.522974        set name = ##class(%PopulateUtils).FirstName()
7       1787612  16.157567        set ^Person(Id) = $ListBuild(job, surname, name)
8       1862728    .825769    }
9            10    .000255    lock -^P:"S"
 ; ** End of source for Method 'filling' **
 ;
 ; ** Source for Method 'run' **
1             1    .000046    kill ^Person
2             1    .000002    set z1 = $zhorolog
3             1    .000004    for i=1:1:10 {
4            10    .037271        job ..filling()
5             0          0    }
6             1  14.620781    lock ^P
7             1    .000005    set z2 = $zhorolog - z1
8             1    .000013    lock
9             1    .000016    write "done:",z2,!
 ; ** End of source for Method 'run' **
```

#Best Practices #ObjectScript #Caché

Source URL:https://community.intersystems.com/post/sequence-function