Caché MapReduce - putting it all together – WordCount example (part III)

Article
Timur Safin · Sep 2, 2016 11m read

Caché MapReduce - putting it all together – WordCount example (part III)

In part I of this series we have introduced MapReduce as a generic concept, and in part II we started to approach Caché ObjectScript implementation via introducing abstract interfaces. Now we will try to provide more concrete examples of applications using MapReduce.

WordCount – Simple sequential implementation

MapReduce is all about parallelization and scalability. But let admit– it is very hard to understand and debug applications if they are parallel from the beginning. For simplicity sake we rather start from sequential version of wordcount algorithm, and then will add some parallelism.

Sequential mapper is quite simple and as expected:

```
Class MR.Sample.WordCount.Mapper Extends (%RegisteredObject, MR.Base.Mapper)
{
/// read strings from MR.Base.Iterator and count words
Method Map(MapInput As MR.Base.Iterator, MapOutput As MR.Base.Emitter)
{
    while 'MapInput.IsAtEnd() {
        #dim line As %String = MapInput.GetNext()
        //#dim pattern As %Regex.Matcher = ##class(%Regex.Matcher).%New("[\w]+")
        #dim pattern As %Regex.Matcher = ##class(%Regex.Matcher).%New("[^\s]+")
        set pattern.Text = line
        while pattern.Locate() {
            #dim word As %String = pattern.Group
            do MapOutput.Emit(word)
        }
    }
}}
```

It receives input stream via MapInput, and emits data to the MapOutput. The algorithm is obvious –if there is some input data in the stream (i.e. *Not Map.Input.IsAtEnd()*) – it will read a next line using MapInput.GetNext(), will split it to words via %Regex.Matcher (see Using Regular Expression in Caché), and each found word will be sent to output emitter.

Reducer is even simpler:

```
Class MR.Sample.WordCount.Adder Extends (%RegisteredObject, MR.Base.Reducer)
```

```
{
Method Reduce(ReduceInput As MR.Base.Iterator, ReduceOutput As MR.Base.Emitter)
{
    #dim result As %Numeric = 0
    while 'ReduceInput.IsAtEnd() {
        #dim value As %String = ReduceInput.GetNext() ; get <key,value> in $listbuild
 format
        #dim word As %String = $li(value,1)
        #dim count As %Integer = +$li(value,2)
        set result = result + count
    }
    do ReduceOutput.Emit("Count", result)
}
}
```

While there is no end of stream (‘*ReduceInput.IsAtEnd()*) it continue to consume input ReduceInput stream, and at each iteration it receives pair in listbuild format (i.e. *$lb(word,count)*). This function aggregates total word count in “result” variable, and emits final result to the next stage of MapReduce algorithm via ReduceOutput “pipe”.

So, now we have shown mapper and reducer, but how main part of applications connects them together? It might get complex soon, but for the beginning we will start from *sequential* implementation of a wordcount algorithm which still will be using MapReduce idiom (yes, I know this makes no much sense if work sequentially, but this simplification step is necessary before making it work in parallel or even on remote nodes).

```
/// Very simple, single-threaded "map-reduce" example.
/Class MR.Sample.WordCount.App Extends %RegisteredObject
{
ClassMethod MapReduce() [ ProcedureBlock = 0 ]
{
    new
    //kill ^mtemp.Map,^mtemp.Reduce
    #dim infraPipe As MR.Sample.GlobalPipe = ##class(MR.Sample.GlobalPipe).%New($name
(^mtemp.Map($J)))
    for i=1:1 {
        #dim fileName As %String = $piece($Text(DATA+i),";",3)
        quit:fileName=""
        // map
        #dim inputFile As MR.Input.FileLines = ##class(MR.Input.FileLines).%New(FileN
ame)
        #dim mapper As MR.Sample.WordCount.Mapper = ##class(MR.Sample.WordCount.Mappe
r).%New()
        do mapper.Map(inputFile, infraPipe)
        // reduce
        #dim outPipe As MR.Base.Emitter = ##class(MR.Emitter.Sorted).%New($name(^mtem
p.Reduce($J)))
        #dim reducer As MR.Sample.WordCount.Adder = ##class(MR.Sample.WordCount.Adder
).%New()
        while 'infraPipe.IsAtEnd() {
            do reducer.Reduce(infraPipe, outPipe)
        }
        do outPipe.Dump()
    }
    quit
DATA
    ;;C:\Users\Timur\Documents\mapreduce\data\war_and_peace_vol1.txt
    ;;C:\Users\Timur\Documents\mapreduce\data\war_and_peace_vol2.txt
```

```
    ;;C:\Users\Timur\Documents\mapreduce\data\war_and_peace_vol3.txt
    ;;C:\Users\Timur\Documents\mapreduce\data\war_and_peace_vol4.txt
    ;;
}
}
```

Let me try explain this code line by line:

- We need to disable procedure block symbol allocation semantics [ProcedureBlock = 0] because we will use array of literal constants embedded to the code under DATA tag and used via $TEXT function. There we store the text (filenames) we plan to work with. For this particular example we use texts of 4 volumes of “War and Peace” by Leo Tolstoy;
- As an intermediate global storage we will use ^mtemp.Map($J) and ^mtemp.Reduce($J) globals. They are automatically mapped to CACHETEMP, thus will not be journaled inside of transactions, and will be not evicted to persistent store as long as it’s possible. Consider them as “kind of in-memory” global;
- Intermediate pipe infraPipe will be instance of MR.Sample.GlobalPipe class, which is alias to MR.EmitterSorted class (and, if you remember from Part II, which is automatically cleaned up at the end of this program);

Class MR.Sample.GlobalPipe Extends (%RegisteredObject, MR.Emitter.Sorted) { }

- We loop over $text(DATA+i) lines, and retrieve the 3rd argument of string, delimited by “;” character. If result is not empty – we use the retrieved value as a file name to input text.
- Input iterator for the mapper will be instance of MR.Input.FileLines, which we did not show yet. The class is rather simple:

```
Class MR.Input.FileLines Extends (%RegisteredObject, MR.Base.Iterator)
{
Property File As %Stream.FileCharacter;
Method %OnNew(FileName As %String) As %Status
{
    set ..File = ##class(%Stream.FileCharacter).%New()
    #dim sc As %Status = ..File.LinkToFile(FileName)
    quit sc
}
Method GetNext() As %String
{
    if $isobject(..File) && '..File.AtEnd {
        quit ..File.ReadLine()
    }
    quit ""
}
Method IsAtEnd() As %Boolean
{
    quit '$isobject(..File) || ..File.AtEnd
}
}
```

So, returning back to the MR.Sample.WordCount.App application we write today:

- Mapper object will be instance of the already known MR.Sample.WordCount.Mapper (see above). Instance objects will be created separately for each individual file;
- In the loop we sequentially invoke Map function in the mapper object created, passing input iterator opened for a file. In this case map stage is actually linearized in the sequential loop, and this is not typical for MapReduce, but this is good for education purposes;
- At the reduce stage we have: output emitter pipe (outPipe) is the instance of MR.Emitter.Sorted, pointing to ^mtemp.Reduce($J). Do you still remember what is the specific of the MR.Emitter.Sorted class (why it's *sorted*)? Because it relies on btree* nature of global storage, and key-values pairs stored there become naturally sorted in the underlying persistent store. Furthermore we could immediately proceed "auto-increment" for values sent to the pipe.
- Reducer object is an instance of MR.Sample.WordCount.Adder described <<above>>.
- For each opened file (at the same iteration of a larger loop) we call reducer.Reduce function, passing there as arguments the intermediate pipe created (infraPipe object modified at the Map step), and output pipe as 2nd argument;

Enough said, let see how it works?

```
DEVLATEST:MAPREDUCE:23:53:27:.000203>do ##class(MR.Sample.WordCount.App).MapReduce()
^mtemp.Reduce(3276,"Count")=114830
^mtemp.Reduce(3276,"Count")=123232
^mtemp.Reduce(3276,"Count")=130276
^mtemp.Reduce(3276,"Count")=109344
```

For each book opened, we have calculated the corresponding number of words in this volume, which then sequentially displayed at the end of iteration. We still have 2 questions unanswered:

- What is the final, aggregated number of words in all volumes?
- And, what is actually more important at the moment, do we actually sure that these numbers were correct? [How to validate them?]

We will start from the 2nd question – verification is easy, given standard Linux/Unix/Cygwin GNU *wc* utility.

```
Timur@TimurYoga2P /cygdrive/c/Users/Timur/Documents/mapreduce/data
$ wc -w war*.txt
 114830 war_and_peace_vol1.txt
 123232 war_and_peace_vol2.txt
 130276 war_and_peace_vol3.txt
 109344 war_and_peace_vol4.txt
 477682 total
```

So wordcounts calculated for each separate volume were correct, so we could proceed and implement aggregation.

Modified Reducer – now with summary counted

To implement aggregation we need to introduce 2 simple changes to the code we have created:

- We will use "extract to function" refactoring for part of Mapper code, because at a later steps we need to

have it as *classmethod*, and not have it embedded to the main code. This will significantly simplify future parallelization efforts and will even make possible to invoke it via remote execution (hopefully);
- And secondly, we will move out instantiation of a reducer and Reduce function invocation from the loop. The idea is to not kill resultant pipe at end of each iteration, but rather accumulate whole data (automatically counting aggregated data for all 4 volumes to be processed);

In all other aspects these 2 samples are equal – they both open ^mtemp.Map($J) and ^mptemp.Reduce($J) as map and reduce step intermediate and final data storage.

```
Class MR.Sample.WordCount.AppSum Extends %RegisteredObject
{
ClassMethod Map(FileName As %String, infraPipe As MR.Sample.GlobalPipe)
{
    #dim inputFile As MR.Input.FileLines      = ##class(MR.Input.FileLines).%New(File
Name)
    #dim mapper As MR.Sample.WordCount.Mapper = ##class(MR.Sample.WordCount.Mapper).%
New()
    do mapper.Map(inputFile, infraPipe)
}
ClassMethod MapReduce() [ ProcedureBlock = 0 ]
{
    new
    #dim infraPipe As MR.Sample.GlobalPipe    = ##class(MR.Sample.GlobalPipe).%New($n
ame(^mtemp.Map($J)))
    #dim outPipe As MR.Base.Emitter           = ##class(MR.Emitter.Sorted).%New($name
(^mtemp.Reduce($J)))
    #dim reducer As MR.Sample.WordCount.Adder = ##class(MR.Sample.WordCount.Adder).%N
ew()
    for i=1:1 {
        #dim fileName As %String = $piece($Text(DATA+i),";",3)
        quit:fileName=""
        do ..Map(fileName, infraPipe)
        //do infraPipe.Dump()
    }
    while 'infraPipe.IsAtEnd() {
        do reducer.Reduce(infraPipe, outPipe)
    }
    do outPipe.Dump()
    quit
DATA
    ;;C:\Users\Timur\Documents\mapreduce\data\war_an_peace_vol1.txt
    ;;C:\Users\Timur\Documents\mapreduce\data\war_an_peace_vol2.txt
    ;;C:\Users\Timur\Documents\mapreduce\data\war_an_peace_vol3.txt
    ;;C:\Users\Timur\Documents\mapreduce\data\war_an_peace_vol4.txt
    ;;
}
}
```

Parallel implementation

Let admit it - changing the simple word-count algorithm, working over several files, to use MapReduce paradigm was not very convenient and very obvious, so this is not the 1st thing you might try to do in real life. But potential gains well worth the pain introduced: reasonable parallelism may allow to achieve the time not possible in sequential algorithm (in my case, for example, sequential algorithm time was ~4.5 seconds, but parallel version completed in 2.6 seconds. Not *that* much different, but still is respected improvement (taking into account the small

input volume set, and limitations of my Haswell 2 low-power core laptop).

Quite recently we have extracted Map stage into the separate class-method, providing it 2 arguments, input file name and output global name. We have done it on purpose - this separate function now is easy to parallelize, if we use standard Caché ObjectScript worker services ($system.WorkMgr). This version below is further modification of a sequential version we have created recently, but with few more workers added to the formula.

```
/// Version #2 More advanced, multiple-workers "map-reduce" example.
Class MR.Sample.WordCount.AppWorkers Extends %RegisteredObject
{
ClassMethod Map(FileName As %String, InfraPipeName As %String) As %Status
{
    #dim inputFile As MR.Input.FileLines        = ##class(MR.Input.FileLines).%New(Fi
leName)
    #dim mapper As MR.Sample.WordCount.Mapper   = ##class(MR.Sample.WordCount.Mapper)
.%New()
    #dim infraPipe As MR.Sample.GlobalPipeClone = ##class(MR.Sample.GlobalPipeClone).
%New(InfraPipeName)
    do mapper.Map(inputFile, infraPipe)
    quit $$$OK
}
ClassMethod MapReduce() [ ProcedureBlock = 0 ]
{
    new
    #dim infraPipe As MR.Sample.GlobalPipe = ##class(MR.Sample.GlobalPipe).%New($name
(^mtemp.Map($J)))
    #dim outPipe As MR.Base.Emitter = ##class(MR.Emitter.Sorted).%New($name(^mtemp.Re
duce($J)))
    #dim reducer As MR.Sample.WordCount.Adder = ##class(MR.Sample.WordCount.Adder).%N
ew()
    #dim sc As %Status = $$$OK
    // do $system.WorkMgr.StopWorkers()
    #dim queue As %SYSTEM.WorkMgr = $system.WorkMgr.Initialize("/multicompile=1", .sc
)
    quit:$$$ISERR(sc)
    for i=1:1 {
        #dim fileName As %String = $piece($Text(DATA+i),";",3)
        quit:fileName=""
        //do ..Map(fileName, infraPipe)
        set sc = queue.Queue("##class(MR.Sample.WordCount.AppWorkers).Map", fileName,
 infraPipe.GlobalName)
        quit:$$$ISERR(sc)
    }
    set sc = queue.WaitForComplete() quit:$$$ISERR(sc)
    while 'infraPipe.IsAtEnd() {
        do reducer.Reduce(infraPipe, outPipe)
    }
    do outPipe.Dump()
    quit
DATA
    ;;C:\Users\Timur\Documents\mapreduce\data\war_an_peace_vol1.txt
    ;;C:\Users\Timur\Documents\mapreduce\data\war_an_peace_vol2.txt
    ;;C:\Users\Timur\Documents\mapreduce\data\war_an_peace_vol3.txt
    ;;C:\Users\Timur\Documents\mapreduce\data\war_an_peace_vol4.txt
    ;;
}
```

The difference between prior AppSum and current AppWorkers is subtle, but important – instead of directly calling Map function in the class, we call this function via $system.WorkMgr.Queue API. This API allows to call either bare routine, or class-method, like the one created in our case. But such inter-processes communications, while providing extra functionality, came with extra limitations – we may not pass anything beyond simple scalar values (i.e. numbers and strings).

In the MR.Sample.WordCount.AppSum::Map case the 2nd argument was an object of MR.Sample.GlobalPipe class. We can not pass object instances to workers, and, in general, when we need to pass object between processes (parent and worker in our case) we need to invent some “serializing/deserializing” schema (hopefully simple). For GlobalPipe “simple serialization” is truely simple – we just pass a name of a global used. That’s why the 2nd argument in our MR.SampleWordCount.AppWorkers::Map function becomes the string of global name, not the object.

Please see workers documentation here, but, in general, if we want to rely on workers heuristics (to use as much workers as we have available [licensed] hardware cores) then we need to initialize workers with oddly named “/multicompile=1” modifier. [This modifier initially used for the parallel compilation in the ObjectScript class compiler, thus this strange name inherited]. Once we queued class-method calls and their arguments via $system.WorkMgr.Queue call, we need to invoke them all and wait completion of all via $system.WorkMgr.WaitForComplete.

All concurrent mappers will use the same temporary global (infraPipe) to output intermediate results, but we will not observe any visible collision effect due to appropriate database engine support (Caché database engine is inherently multi-process, by design). If there will be a chance we will return back to the topic of locking and lockless concurrent algorithms and data-structures.

On the other hand, reducer here will be invoked from the single thread (from parent, master process, in our case), because we need to calculate the aggregated sum for all intermediate data. That is why reducer is invoked outside of worker loop.

In this series we have introduced MapReduce algorithm in general, covered basic infrastructure necessary for MapReduce in Caché ObjectScript *and have created 1st rather simple MapReduce example counting words in the input stream(s). The next step we will continue cover MapReduce idioms using classic example – now it will AgeAverage example from WikiPedia. Stay tuned.*

#Data Model #Distributed Data Management #Object Data Model #Caché
50 4 1 3 820

Source URL: https://community.intersystems.com/post/cach%C3%A9-mapreduce-putting-it-all-together-%E2%80%93-wordcount-example-part-iii