Article
Timur Safin · Aug 26, 2016 8m read

Caché MapReduce - Basic interfaces for MapReduce implementation (part II)

In the prior part of this series we have provided introduction to Google MapReduce approach, but still not covered their possible ObjectScript implementation. Which we will start to explain today.

We still plan to model Caché ObjectScript MapReduce implementation quite close to the original Google C++ example given before. We start from Mapper and Reducer abstract interfaces which will be implemented elsewhere later:

```
/// Base class for MapReduce Mapper.
Class MR.Base.Mapper
{
Method Map(MapInput As MR.Base.Iterator, MapOutput As MR.Base.Emitter) [ Abstract ] {
 }
}
/// Base class for MapReduce Reducer.
Class MR.Base.Reducer
{
Method Reduce(ReduceInput As MR.Base.Iterator, ReduceOutput As MR.Base.Emitter) [ Abs
tract ] { }
}
```

Originally, we've started with separate implementations of MapInput and ReduceInput interface classes (as in C++), but then we've realized that they are about the same – iteration over the stream, until there is end reached, so they got unified to the common MR.Base.Iterator:

```
Class MR.Base.Iterator
{
Method GetNext() As %String [ Abstract ] { }
Method IsAtEnd() As %Boolean [ Abstract ] { }
}
```

Using globals as a channel

Original Google MapReduce implementation used Google GFS cluster file-system as a transport media between nodes for intermediate and final storage. We have different media to establish interaction between cluster Caché nodes and it's called –Enterprise Cache Protocol (ECP). Right now it's used by application servers for operating with remote data mounted from data server, but there is nothing stopping us from using ECP as a simple "control bus" interface, where we will put <key, value> pairs or similar operating data from the sender side and retrieve

those commands on the receiver side. If implementation actors (i.e. reduce and combine workers) will be running on the same machine, then they could even use ultra-fast, scratch data space on the CACHETEMP
If there would be inheritance implemented between parent and child processes for process-private globals opened by parent ^||PPG, or may be some magical locals be automatically marshalled between processes, then PPG or inherited local arrays would be much more convenient to use here, for emitter operations. Dreams, dreams

Either way, be they local for the particular node, or, may be, mapped remotely via ECP, *globals* are very convenient, and well established mechanism, which is very fitting as transport for sending intermediate data between MapReduce stages. Let try to use it this way.

Emitters and some dirty magic

You remember that between moment when data is emitted from mapper, and when element of a collection retrieved by reducer there is some heavy shuffle algorithm used on master node. Actually, in the MUMPS/ObjectScript environment, we could entirely avoid this overhead if we will use some built-in functions, which will handle aggregation for us. Luckily, all this aggregation can be done without any extra traffic between involved parties (master, mapper or reducer). [*More details about this handy optimization will be given later*]

Here is basic emitter interface we introduce:

```
Class MR.Base.Emitter Extends MR.Base.Iterator
{
/// emit $listbuild(key,value(s))
Method Emit(EmitList... As %String) [ Abstract ] { }
}
```

Semantically, emitter is very similar to input iterator (thus we see it's ancestor of MR.Base.Iterator), but in addition to iterator interface we will introduce some extra functions which will send data to the intermediate collection (e.g. Emit function).
At the beginning of the MapReduce project our Emit function was very close to classical one, it was accepting 2 arguments for simple case of <key, value> pair. But then we've discovered some necessity to extend it to something more multi-dimensional (i.e. handling generic tuples), thus right now it's accepting variable number of arguments (though, most of the time there are either single or 2 arguments passed).

This is still abstract interface, and more "meat" will be added quite soon:

If we need to keep the order of emitted data then we could use implementation similar to MR.Emitter.Ordered:

```
/// Emitter which maintains the order of (key,value(s))
Class MR.Emitter.Ordered Extends (%RegisteredObject, MR.Base.Emitter)
{
/// global name serving as data channel
Property GlobalName As %String;
Method %OnNew(initval As %String) As %Status
{
    $$$ThrowOnError($length(initval)>0)
    set ..GlobalName = initval
    quit $$$OK
}
Parameter AUTOCLEANUP = 1;
```

```
Method %OnClose() As %Status
{
    if ..#AUTOCLEANUP {
        if $data(@i%GlobalName) {
            kill @i%GlobalName
        }
    }
    Quit $$$OK
}
…
```

Globals are global, and they not automatically cleaned up, once process, which has created them, will be terminated. But in our case we use globals as rather intermediate, not as a persistent storage, so sometimes we do need to proceed automatic cleanup. That's why we have introduced #AUTOCLEANUP class parameter (which is enabled by default, so the global those name is stored to GlobalName property, will be killed at the moment emitter is deleted (see %OnClose callback). Please pay attention that there is 1 required parameter we enforce in the %New() constructor of this class – it expects to receive global name with whom we will operate.

```
…
Method IsAtEnd() As %Boolean
{
    quit ($data(@i%GlobalName)\10)=0
}
/// emit $listbuild(key,value)
Method Emit(EmitList... As %String)
{
    #dim list As %String = ""
    for i=1:1:$get(EmitList) {
        set $li(list,i) = $get(EmitList(i))
    }
    #dim name As %String = ..GlobalName
    set @name@($seq(@name)) = list
}
/// returns emitted $lb(key,value)
Method GetNext() As %String
{
    #dim value As %String
    #dim index As %String = $order(@i%GlobalName@(""),1,value)
    if index'="" {
        kill @i%GlobalName@(index)
        quit value
    } else {
        kill @i%GlobalName
        quit ""
    }
}
Method Dump()
{
    zwrite @i%GlobalName
}
}
```

Do you still remember that Emitter is ancestor of Iterator, so it supposed to implement not only Emit function but IsAtEnd and GetNext as well?

- IsAtEnd is simple – if there are data in the global (i.e. $data returns 1x) then we assume collection is not empty.
- Emit creates new node at the end of global, keeping passed argument list in $listbuild(list) format. As you already aware, $sequential function could be used (almost) at the same cases when $increment is used, but it has one big advantage for us – their performance observes less contention over global mode used if it's working over ECP (see "On $Sequence function" by Alexander Koblov)
- On the other hand GetNext() retrieves head of a collection ($order(@i%GlobalName@(“”)) and then delete the head. To pass <key,value> pair we use $lb<> data structure. Lists created by $listbuild() function allow to keep not only simple (key, value) pair, but tuple of any desired size. We will use this ability later.

Data structure as implemented by MR.Emitter.Ordered is a classical FIFO (“First in – First Out”) collection: we push data at the end, and pop from the head.

Autoaggregating emitter

If you look into the sorted collection implementation, and compare to what we send in word-count example, then you will immediately recognize that:

- Actually, we do not care about order we send keys to emitter. After we write them unordered as keys (subscript values) of some particular global node they to be automatically sorted by the underlining btree* database storage;
- And in majority of cases when we emit <word, 1> in mapper then we assume to proceed in the reducer the simple aggregation of those 1s, essentially, it will work as simple $increment counter.

So why bother and produce unnecessary traffic if we could use magical $increment from ObjectScript?

```
/// Emitter which sorts by keys all emitted pairs or tuples (key, value(s))
Class MR.Emitter.Sorted Extends MR.Emitter.Ordered
{
Property AutoIncrement As %Boolean [ InitialExpression = 1 ];
/// emit $listbuild(key,value)
Method Emit(EmitList... As %String)
{
    #dim name As %String = ..GlobalName
    #dim key As %String
    #dim value As %String
    if $get(EmitList)=1 {
        // special case - only key name given, no value
        set key = $get(EmitList(1))
        quit:key=""
        if ..AutoIncrement {
            #dim dummyX As %Integer = $increment(@name@(key)) ; $seq is non-
deterministic
        } else {
            set @name@(key) = 1
        }
    } else {
        set value = $get(EmitList(EmitList))
        set EmitList = EmitList - 1
        for i=1:1:$get(EmitList) {
            #dim index As %String = $get(EmitList(i))
            quit:index=""
            set name = $name(@name@(index))
        }
```

```
        if ..AutoIncrement {
            #dim dummyY As %Integer = $increment(@name,value)
        } else {
            set @name = value
        }
    }
}
…
```

There is simplest case, when there is 1 argument passed to Emit function (the key only). So if we are acting in AutoIncrement mode then we immediately count the sum of all key with the same name passed. If it's not in autoincrementing mode, then we simply (re)define this key in global, holding 1 as the value.

For 2 argument mode (key, value) and in autoincrementing mode, we accumulate in the corresponding node of a global the sum of all values passed so far.

But there is more – this function is able to handle any size of a tuple <key, value(s)>, even more than 2. When there is no autoincrement enabled it's still easy – emitter will just define an appropriate node in a global where all arguments passed to the functions become subscripts of a global node. On the other hand, for autoincrement mode it acts a little differently, and should consider last argument as value for multi-dimensional node, so all arguments before the last one will be subscript names.

This unusual extension of a key-value nature data use in algorithm is not typical to our knowledge, we are unaware whether there is something similar used by other MapReduce implementations (probably, because of the fact they operate with strict key-value or bigtable stores, and not with hierarchical arrays like in our case, where efficient $increment facilities available). We'll show later some practical example of such extension.

IsAtEnd is still not redefined by this class, and is inherited from parent MR.Emitter.Ordered (i.e. it returns non-null value of there is some data under GloRef name used for intermediate store).

But GetNext should be a little bit more complex than before.

```
…
/// returns emitted $lb(key,value)
Method GetNext() As %String
{
    #dim name As %String = ..GlobalName
    #dim value As %String
    #dim ref As %String = $query(@name,1,value)
    if ref'="" {
        zkill @ref
        #dim i As %Integer
        #dim refLen As %Integer = $qlength(ref)
        #dim baseLen As %Integer = $qlength(name)
        #dim listbuild = ""
        for i=baseLen+1:1:refLen {
            set $li(listbuild,i-baseLen)=$qs(ref,i)
        }
        set $li(listbuild,*+1)=value
        quit listbuild
    }
    quit ""
}
…
```

We expect $LB<> data returned from the function, but Emit stores tuple data into global as subscripts, so we need to reassemble $listbuild from the subscripts and there stored node value. The returned node is immediately killed in the global so it will be removed from "collection" and not available for other workers.

In this 2nd part of series we have introduced basic interfaces used by our ObjectScript MapReduce implementation. In a future parts we will put all things together, and will provide more concrete examples.

#C++ #Data Model #Globals #Key Value #Caché

Source URL: https://community.intersystems.com/post/cach%C3%A9-mapreduce-basic-interfaces-mapreduce-implementation-part-ii