Jumbo

Quick Start guide

# Introduction

Jumbo is a distributed data processing system for Microsoft .Net Core, in a similar style as Apache Hadoop. Jumbo allows the user to create a pipeline of processing steps (stages), where each stage is broken into tasks that are executed in parallel on a cluster of machines. Jumbo supports MapReduce, but has more flexibility and also allows for alternate processing pipelines.

Jumbo consists of a distributed file system in the spirit of Google File System and the Hadoop DFS, and a processing engine called Jumbo Jet.

The Jumbo DFS uses a single NameServer (analogous to the Hadoop Namenode) that stores file system metadata, and numerous DataServers (Hadoop Datanodes) that store file data. Jumbo Jet uses a central JobServer responsible for fault tolerance and scheduling (the Jobtracker in Hadoop 1.0) and numerous TaskServers responsible for task execution (Hadoop Tasktrackers).

The purpose of this release is for distributed systems enthusiasts who would like to experiment with a data processing system besides Hadoop. Check the source to see what design alternatives I used. Or maybe you’d like to see how .Net’s capabilities over Java (such as proper generics or LINQ) influenced the code.

**Jumbo is not production quality code!** Jumbo was created as a case study on the design of MapReduce-style systems. It is not a full-fledged alternative for Hadoop and lacks many of the features that would be required in a real environment. Features were added as I needed them or if I felt like adding them. Some features were added but rarely used, so may not perform well or have bugs.

Jumbo’s primary purpose was to help me understand how Hadoop works. It is loosely based on Hadoop, and blatantly borrows a number of design elements from Hadoop, while differing in some other areas. It was not designed to compete with Hadoop, and not originally intended to be released. I am releasing it now because I think some people may find it interesting, but it should not be used for production systems.

Jumbo works pretty well. It can be significantly faster than Hadoop for certain workloads, especially on small clusters (Jumbo has been tested up to approximately 50 nodes). I’ve also tried to make creating data processing applications for Jumbo easy and a pleasant experience. By all means, use it to run processing jobs to see how it compares to Hadoop; you can even use it for some non-critical real workloads if you really like. But please don’t entrust Jumbo to hold your mission-critical company data. It’ll probably work fine 99% of the time, but I won’t be responsible if it breaks and you lose your stuff.

If you have any questions, please contact me through <http://www.ookii.org>. I will try to help you if I can, but please understand that I’m not officially supporting this release.

This section helps you to get Jumbo running on your system. This document assumes you have some knowledge about systems like Hadoop; it is not a full introduction to distributed data processing in general.

# Prerequisites

Jumbo is designed to run on Windows and Linux using Microsoft .Net Core. The following are the minimum requirements:

* Microsoft .Net 6.0
* Microsoft PowerShell Core 7.2.2

Jumbo was tested using these versions; newer versions are expected to work but have not been tested.

This is a port of Jumbo to .Net Core; Jumbo was originally written to use both Microsoft .Net and Mono. Most of the scale and performance testing was done on Linux using Mono; Windows support was used mainly for debugging. No large-scale performance testing using .Net Core has been done.

Jumbo includes two administration websites (DfsWeb and JetWeb). These will run using the Kestrel web server included with .Net Core.

The provided scripts used to start Jumbo on a cluster use [PowerShell remoting over SSH](https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/ssh-remoting-in-powershell-core?view=powershell-6), which must be correctly configured on all nodes of the cluster. If you are running in a single-node environment, no configuration is necessary as the scripts will directly invoke commands on the local host. On Windows, you can also configure Jumbo to use WinRM remoting by modifying Get-JumboConfig.ps1.

You can also start Jumbo manually, in which case PowerShell is not required, but this is not recommended.

# Building Jumbo

Jumbo can be built on Linux and Windows by simply running dotnet build from the solution’s root directory. You can run the unit tests by running dotnet test (note: this will take several minutes).

You can create a distribution of Jumbo, with all the required binaries and scripts to run it, by running ./Publish-Release.ps1 <path>, e.g. ./Publish-Release.ps1 /jumbodist

# Configuration

## Quick configuration for one node

Want to simply try Jumbo on one node? Here’s what you do:

1. In bin/dfs.config, set the NameServer image directory and the DataServer block directory to local directories (not the same directory, and they should be empty before you first use Jumbo).
2. In bin/jet.config, set the JobServer archive directory and TaskServer task directory.

That’s it, you can now run Jumbo on one node (see “Running Jumbo” below).

## Basic configuration and deployment

In order to run Jumbo, you must first configure some values in Get-JumboConfig.ps1. You must set the $JUMBO\_HOME directory to the directory where Jumbo is installed (this path must be the same on all nodes). If you change the $JUMBO\_LOG directory, also make the corresponding modifications in bin/common.config. Check the comments in the file for further available configuration options.

Using bin/common.config, you can modify the log directory and configure rack-awareness (used by the task scheduler to get data locality).

In order to start Jumbo using the provided scripts, you must also specify which nodes it will run on. Jumbo uses a two-level configuration system for this: the deploy/groups file specifies groups of nodes, and files matching the group names contain the actual node names. By default, there are two groups defined: deploy/masters, and deploy/nodes.

Groups are used only when Jumbo is deployed to your cluster using the provided Deploy-Jumbo.ps1 script. Dividing nodes into multiple groups allows you specify different configuration settings for each group. When running Jumbo, it simply starts the servers on all nodes in all groups, without distinguishing the groups.

The masters group is special and is only used during deployment. You only need to modify the masters file if you intend to use the deployment script (see below). This group should contain the node(s) that run the NameServer and JobServer. The contents of the masters file are skipped when running Jumbo; the Get-JumboConfig.ps1 file specifies which node should run the NameServer, and which node the JobServer.

Specify all nodes that should run DataServers and TaskServers in the nodes file. Alternatively, you can define your own groups. If you only wish to evaluate Jumbo on a single node, you can leave the content of these files at “localhost”.

You can deploy Jumbo to multiple nodes using the Deploy-Jumbo.ps1 script, which will copy all Jumbo files including the configuration to $JUMBO\_HOME on all the nodes. This way, $JUMBO\_HOME does not need to be a network path available on all the nodes.

When deploying, you can use different configuration files for each group (this is the purpose of groups). To do this, create files names common.groupname.config, dfs.groupname.config and jet.groupname.config (replace groupname with the name of the group) in the same directory as Deploy-Jumbo.ps1. If any of those files does not exist for a group, that group uses the default configuration.

## Configuring the Jumbo DFS

### Running without the DFS

It is not required to use the Jumbo DFS to run data processing jobs. You can also use the local file system. To do this, modify the bin/dfs.config file and set the file system URL to file:///root/path, where /root/path is the root of the file system as visible by Jumbo. If this path is a file share accessible on every node, you can even use this when running Jumbo on multiple nodes.

Certain features are not available when the Jumbo DFS is not used. It’s generally recommended to use the Jumbo DFS instead of the local file system except for debugging purposes.

### Configuring the DFS

There are four values that you must specify in bin/dfs.config to use the DFS:

1. Set the file system URL to the host name and port of the NameServer.
2. Set the NameServer image directory to a local directory where the NameServer’s metadata will be stored.
3. Set the replicationFactor to an appropriate value. It’s recommended to use 3 replicas unless you have fewer than 3 data servers (in which case set it to the number of data servers).
4. Set the DataServer block directory to a local directory where the file data for each node will be stored.

See the configuration file documentation for information on the other options that are available.

The below is an example of a typical bin/dfs.config:

$lang: XML$  
<?xml version="1.0" encoding="utf-8"?>  
<ookii.jumbo.dfs>  
 <fileSystem url="jdfs://somenode:9000"/>  
 <nameServer blockSize="128MB"  
 replicationFactor="3"  
 imageDirectory="/jumbo/nameserver" />  
 <dataServer port="9001"  
 blockStoragePath="/jumbo/dataserver"/>  
</ookii.jumbo.dfs>

## Configuring Jumbo Jet

There are two values that you must specify in bin/jet.config:

1. Set the JobServer host name and port.
2. Set the TaskServer task directory to a local directory where configuration, task log and intermediate data files will be stored on each node.

See the configuration file documentation for information on the other options that are available.

The below is an example of a typical bin/jet.config:

$lang: XML$  
<?xml version="1.0" encoding="utf-8"?>  
<ookii.jumbo.jet>  
 <jobServer hostName="somenode"  
 port="9500"  
 archiveDirectory="/jumbo/jobarchive"  
 broadcastAddress="192.168.0.255"  
 broadcastPort="9550" />  
 <taskServer port="9501"  
 taskDirectory="/jumbo/taskserver"  
 taskSlots="2"  
 fileServerPort="9502"  
 fileServerMaxConnections="4"  
 immediateCompletedTaskNotification="true"/>  
 <fileChannel memoryStorageSize="2GB"  
 spillBufferSize="100MB"/>  
 <mergeRecordReader maxFileInputs="10"/>  
</ookii.jumbo.jet>

# Running Jumbo

Before you can run Jumbo, you must format the file system (you can skip this step if you’re not using the DFS). To do this, run dotnet bin/NameServer.dll -format on the node that will run the NameServer.

Once Jumbo is configured, run it by running ./Start-Dfs.ps1 and ./Start-Jet.ps1.

If all went well, Jumbo should be running now. If not, check the log directory you specified and check the log files to see what went wrong.

Open your browser to <http://localhost:35000> to see the DFS administration page, and <http://localhost:36000> for the Jet administration page. If you changed the configuration, are not hosting the administration sites using the default method, or are accessing the pages from a computer other than the one running the NameServer and JobServer respectively you may need to use different URLs.

# Running your first data processing job

To help you get started, this demonstrates how to upload a text file to the DFS and run WordCount on the job.

To run this job, you need some utf-8 plain text files to use as input. You can generate some random text using another Jumbo sample job (which we’ll do below), but if you want some non-random text, why not use [Project Gutenberg](http://www.gutenberg.org/)? For example, you could use [Moby Dick](http://www.gutenberg.org/cache/epub/2701/pg2701.txt).

First, you must upload a text file. Any plain text file stored as utf-8 will do. If your file is named mobydick.txt and is stored in the current directory, use the following:

> ./DfsShell.ps1 put mobydick.txt /

This will store the file somefile.txt in the root of the DFS. Verify it by running ./DfsShell.ps1 ls. The output should look something like this:

> ./DfsShell.ps1 ls  
Directory listing for /  
  
2013-02-16 20:53 1,257,260 mobydick.txt

DfsShell is your key to interacting with the DFS. Use it to upload and download files, manipulate the namespace, view status, and more. You can also use the “Browse file system namespace” option in the DFS administration website to view the contents of the file system.

Note you can also upload a bunch of files to a directory and use the whole directory as input for the job (don’t use the root in this case; use ./DfsShell.ps1 mkdir to create a directory).

Now we’ll run the job:

> ./JetShell.ps1 job bin/Ookii.Jumbo.Jet.Samples.dll wordcount /mobydick.txt /wcoutput  
236 [1] INFO Ookii.Jumbo.Jet.Jobs.JobRunnerInfo (null) - Created job runner for job WordCount, InputPath = /mobydick.txt, OutputPath = /wcoutput  
427 [1] INFO Ookii.Jumbo.Jet.JetClient (null) - Saving job configuration to DFS file /JumboJet/job\_{56f57c95-f6e4-445a-b87d-8fd1ce408db5}/job.xml.  
977 [1] INFO Ookii.Jumbo.Jet.JetClient (null) - Uploading local file /home/sgroot/jumbo/build/bin/Ookii.Jumbo.Jet.Samples.dll to DFS directory /JumboJet/job\_{56f57c95-f6e4-445a-b87d-8fd1ce408db5}.  
1051 [1] INFO Ookii.Jumbo.Jet.JetClient (null) - Running job 56f57c95-f6e4-445a-b87d-8fd1ce408db5.  
0.0 %; finished: 0/1 tasks; WordCount: 0.0 %  
100.0 %; finished: 1/1 tasks; WordCount: 100.0 %  
  
Job completed.  
Start time: 2013-02-16 20:56:42.397  
End time: 2013-02-16 20:56:44.016  
Duration: 00:00:01.6193970 (1.619397s)

Let’s examine what we did here. The JetShell script is used to launch jobs, which we’re doing here by specifying the job command. The next argument specifies the assembly file containing the job (in this case, the included assembly with sample jobs), followed by the name of the job. The remaining arguments are specific to the job; for WordCount here we’re specifying the input and output path.

Tip: want to see what jobs are available in an assembly? Simply omit all the arguments after the assembly name. Similarly, you can typically omit all the arguments after the job name to see the arguments for the job.

Let’s see what that did to the file system:

> ./DfsShell.ps1 ls  
Directory listing for /  
  
2012-07-11 17:15 <DIR> JumboJet  
2013-02-16 20:53 1,257,260 mobydick.txt  
2013-02-16 20:56 <DIR> wcoutput

You can see there are two new directories. JumboJet is a working directory for the Jet execution engine; it’s not important for the user. The wcoutput directory contains the output. Let’s check it out:

> ./DfsShell.ps1 ls /wcoutput  
Directory listing for /wcoutput  
  
2013-02-16 20:56 468,008 WordCountAggregation-00001

As you can see, there is one file. The files are named after the tasks that produced them, and in this sample there was only one task because the input file was quite small. You can view the results using DfsShell as well:

> ./DfsShell.ps1 cat /wcoutput/WordCountAggregation-00001  
[The, 549]  
[Project, 79]  
[Gutenberg, 20]  
[EBook, 1]  
[of, 6587]  
[Moby, 79]  
[Dick;, 9]  
[or, 758]  
[Whale,, 39]  
[by, 1113]  
[Herman, 4]  
[Melville, 4]  
[This, 102]  
[eBook, 5]  
[is, 1586]  
...

That probably kept going for a while, depending on the size of the file. That tells you exactly how often each word occurred in your text file. Note: the WordCount sample just splits the text on spaces, so if you uploaded something that isn’t just plain text (like an HTML file), the results might be a bit weird. Even in this case you’ll notice that some of the “words” include punctuation marks, and that different capitalizations of the same word are counted separately. That’s a limitation of the sample, not of Jumbo. The user guide will introduce a more advanced version of WordCount that overcomes some of these limitations.

Don’t forget to check out your job in the Jet administration website. You can see lots of cool statistics there.

And that’s your very first job! But wait a second? Isn’t Jumbo for distributed processing? But unless you uploaded a very large text file (larger than the block size for the file system), the job probably only had one task.

We can make this more interesting by using another sample job included with Jumbo, GenerateText, to generate some larger input for the WordCount job:

> ./JetShell.ps1 job bin/Ookii.Jumbo.Jet.Samples.dll generatetext /bigtext 64 256MB  
256 [1] INFO Ookii.Jumbo.Jet.Jobs.JobRunnerInfo (null) - Created job runner for job GenerateText, OutputPath = /bigtext, SizePerTask = 256MB, TaskCount = 64  
411 [1] INFO Ookii.Jumbo.Jet.JetClient (null) - Saving job configuration to DFS file /JumboJet/job\_{b4c04385-df32-458b-8b74-f41e0364e05e}/job.xml.  
518 [1] INFO Ookii.Jumbo.Jet.JetClient (null) - Uploading local file /home/sgroot/jumbo/build/bin/Ookii.Jumbo.Jet.Samples.dll to DFS directory /JumboJet/job\_{b4c04385-df32-458b-8b74-f41e0364e05e}.  
596 [1] INFO Ookii.Jumbo.Jet.JetClient (null) - Uploading local file /tmp/Ookii.Jumbo.Jet.Generated.9ee75f0181f24f6691303f8106e79503.dll to DFS directory /JumboJet/job\_{b4c04385-df32-458b-8b74-f41e0364e05e}.  
647 [1] INFO Ookii.Jumbo.Jet.JetClient (null) - Running job b4c04385-df32-458b-8b74-f41e0364e05e.  
0.0 %; finished: 0/64 tasks; GenerateTaskStage: 0.0 %  
3.1 %; finished: 2/64 tasks; GenerateTaskStage: 3.1 %  
24.8 %; finished: 15/64 tasks; GenerateTaskStage: 24.8 %  
94.5 %; finished: 58/64 tasks; GenerateTaskStage: 94.5 %  
100.0 %; finished: 64/64 tasks; GenerateTaskStage: 100.0 %  
  
Job completed.  
Start time: 2013-02-16 21:13:10.875  
End time: 2013-02-16 21:13:20.624  
Duration: 00:00:09.7494140 (9.749414s)

The parameters for this job indicate the output path, the number of tasks, and the size to generate per task. So we generated 16GB of random text, using 64 tasks each generating 256MB. I’m running this example on 32 nodes; if you’re using a smaller cluster, you may want to scale down the size accordingly. Just make sure you use more than one generator task or a total size that’s larger than the DFS block size. You can see the files it created by running ./DfsShell ls /bigtext.

Then, we can simply run WordCount as before:

> ./JetShell.ps1 job bin/Ookii.Jumbo.Jet.Samples.dll wordcount /bigtext /wcoutput  
236 [1] INFO Ookii.Jumbo.Jet.Jobs.JobRunnerInfo (null) - Created job runner for job WordCount, InputPath = /bigtext, OutputPath = /wcoutput  
496 [1] INFO Ookii.Jumbo.Jet.JetClient (null) - Saving job configuration to DFS file /JumboJet/job\_{db8b43d7-6446-4d88-b2a9-6647031d98a9}/job.xml.  
665 [1] INFO Ookii.Jumbo.Jet.JetClient (null) - Uploading local file /home/sgroot/jumbo/build/bin/Ookii.Jumbo.Jet.Samples.dll to DFS directory /JumboJet/job\_{db8b43d7-6446-4d88-b2a9-6647031d98a9}.  
710 [1] INFO Ookii.Jumbo.Jet.JetClient (null) - Running job db8b43d7-6446-4d88-b2a9-6647031d98a9.  
0.0 %; finished: 0/320 tasks; WordCount: 0.0 %; WordCountAggregation: 0.0 %  
3.8 %; finished: 12/320 tasks; WordCount: 4.7 %; WordCountAggregation: 0.0 %  
15.9 %; finished: 51/320 tasks; WordCount: 19.9 %; WordCountAggregation: 0.0 %  
20.0 %; finished: 64/320 tasks; WordCount: 25.0 %; WordCountAggregation: 0.0 %  
28.8 %; finished: 92/320 tasks; WordCount: 35.9 %; WordCountAggregation: 0.0 %  
38.8 %; finished: 124/320 tasks; WordCount: 48.4 %; WordCountAggregation: 0.0 %  
40.0 %; finished: 128/320 tasks; WordCount: 50.0 %; WordCountAggregation: 0.0 %  
41.9 %; finished: 134/320 tasks; WordCount: 52.3 %; WordCountAggregation: 0.0 %  
52.2 %; finished: 167/320 tasks; WordCount: 65.2 %; WordCountAggregation: 0.0 %  
59.7 %; finished: 191/320 tasks; WordCount: 74.6 %; WordCountAggregation: 0.0 %  
60.0 %; finished: 192/320 tasks; WordCount: 75.0 %; WordCountAggregation: 0.0 %  
67.8 %; finished: 217/320 tasks; WordCount: 84.8 %; WordCountAggregation: 0.0 %  
76.3 %; finished: 244/320 tasks; WordCount: 95.3 %; WordCountAggregation: 0.0 %  
80.0 %; finished: 256/320 tasks; WordCount: 100.0 %; WordCountAggregation: 0.0 %  
85.3 %; finished: 272/320 tasks; WordCount: 100.0 %; WordCountAggregation: 26.7 %  
94.1 %; finished: 301/320 tasks; WordCount: 100.0 %; WordCountAggregation: 70.4 %  
100.0 %; finished: 320/320 tasks; WordCount: 100.0 %; WordCountAggregation: 100.0 %  
  
Job completed.  
Start time: 2013-02-16 21:18:57.619  
End time: 2013-02-16 21:19:18.695  
Duration: 00:00:21.0754110 (21.075411s)

Now we had quite a few more tasks, with two stages: WordCount, which reads a piece of the input and counts the words locally, and WordCountAggregation, which aggregates all the pieces of the first stage. You could compare the WordCount stage with a map stage, and the WordCountAggregation stage with a reduce stage (except that this version of WordCount actually uses hash table aggregation, which is not possible with the current version of Hadoop).

Depending on the cluster configuration, you’ll probably also find that there’s more than one output file this time, because each task in the WordCountAggregation stage creates its own output file.

Want to know more about how this example works and how to create your own processing jobs? Move on to the user guide!