

Big Data for Engineers – Exercises

Spring 2020 – Week 3 – ETH Zurich

Introduction

This week we will cover mostly theoretical aspects of Hadoop and HDFS and we will discuss advantages and limitations of different storage models. The mandatory reading is [1] [Shvachko, K. et al. \(2010\). The Hadoop Distributed File System. In MSST.](#)

What is Hadoop?

"Hadoop provides a **distributed file system** and a **framework for the analysis and transformation** of very **large** data sets using the **MapReduce paradigm**." [1]

Several components are part of this framework. In this course you will study HDFS, MapReduce and HBase while this exercise focuses on HDFS and storage models.

Component	Description	First developer
HDFS	Distributed file system	Yahoo!
MapReduce	Distributed computation framework	Yahoo!
HBase	Column-oriented table service	Powerset (Microsoft)
Pig	Dataflow language and parallel execution framework	Yahoo!
Hive	Data warehouse infrastructure	Facebook
ZooKeeper	Distributed coordination service	Yahoo!
Chukwa	System for collecting management data	Yahoo!
Avro	Data serialization system	Yahoo! + Cloudera

1. The Hadoop Distributed File System

1.1 – State which of the following statements are true:

1. The HDFS namespace is a hierarchy of files and directories.
2. In HDFS, each block of the file is either 64 or 128 megabytes depending on the version and distribution of Hadoop in use, and this cannot be changed.
3. A client wanting to write a file into HDFS, first contacts the NameNode, then sends the data to it. The NameNode will write the data into multiple DataNodes in a pipelined fashion.
4. A DataNode may execute multiple application tasks for different clients concurrently.
5. The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster.
6. HDFS NameNodes keep the namespace in RAM.
7. The locations of block replicas are part of the persistent checkpoint that the NameNode stores in its native file system. ([more detail in \[1\]](#))
8. If the block size is set to 64 megabytes, storing a file of 80 megabytes will actually require 128 megabytes of physical memory (2 blocks of 64 megabytes each).
9. Every 6 hours the *NameNode* asks the *DataNode* for a complete report on the blocks stored on it.
10. HDFS is optimised for latency.

Solution

1. True, in contrast with the Object Storage logic model, HDFS is designed to handle a relatively small amount of huge files. A hierarchical file system can therefore be handled efficiently by a single NameNode.
2. False, the default size is either 64 or 128 megabytes but this can be easily changed in the configuration.
3. False, the client writes data to the DataNodes. No data goes through the NameNode.
4. True, each DataNode may execute multiple application tasks concurrently.
5. True, since each DataNode can execute multiple tasks concurrently, there may be more clients than DataNodes.
6. True, and an image of such namespace is also persisted in the NameNode file system.
7. False, the locations of block replicas may change over time and are not part of the persistent checkpoint.
8. False, the size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional file systems. Therefore 80 megabytes will be stored as a block of 64 megabytes + a block of

16 megabytes.

9. False, it is one of the DataNode's duties to automatically send a report to the NameNode.
10. False, HDFS is optimised for throughput.

1.2 – Where is the information stored? For each of the following, say if the information is stored in the disk of the NameNode, in the disk of a DataNode, in disks of both of them, or in none of them.

1. the list of blocks belonging to each file [NameNode | DataNode]
2. the files containing the actual blocks of data [NameNode | DataNode]
3. the block's metadata including checksum and generation stamp [NameNode | DataNode]
4. the location of block replicas [NameNode | DataNode] ([more detail in \[1\]](#))

Solution

1. NameNode
2. DataNode
3. DataNode
4. None. While this information is available in the NameNode memory, it is not persisted.

1.3 – A typical filesystem block size is 4096 bytes. How large is a block in HDFS? List at least two advantages of such choice.

Solution

Typical size for a block is either 64 or 128 megabytes. A large block size offers several important advantages.

1. it minimizes the cost of seeks. If the block is large enough, the time it takes to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. Thus, transferring a large file made of multiple blocks operates at the disk transfer rate.
2. it reduces clients' need to interact with the master because reads and writes on the same block require only one initial request to the master for block location information. The reduction is especially significant for our workloads because applications mostly read and write large files sequentially.
3. since on a large block, a client is more likely to perform many operations on a given block, it can reduce network overhead by keeping a persistent TCP connection to the datanode over an extended period of time.
4. it reduces the size of the metadata stored on the master. This allows us to keep the metadata in memory.

1.4 – How does the hardware cost grow as function of the amount of data we need to store in a Distributed File System such as HDFS? Why?

Solution

Linearly. HDFS is designed taking machine failure into account, and therefore DataNodes do not need to be (highly expensive) highly reliable machines. Instead standard commodity hardware can be used. Moreover the number of nodes can be increased as soon as it becomes necessary, avoiding wasting of resources when the amount of data is still limited. This is indeed the main advantage of scaling out compared to scaling up, which has exponential cost growth.

1.5 – Scalability, Durability and Performance on HDFS

Explain how HDFS accomplishes the following requirements:

1. Scalability
2. Durability
3. High sequential read/write performance

Solution

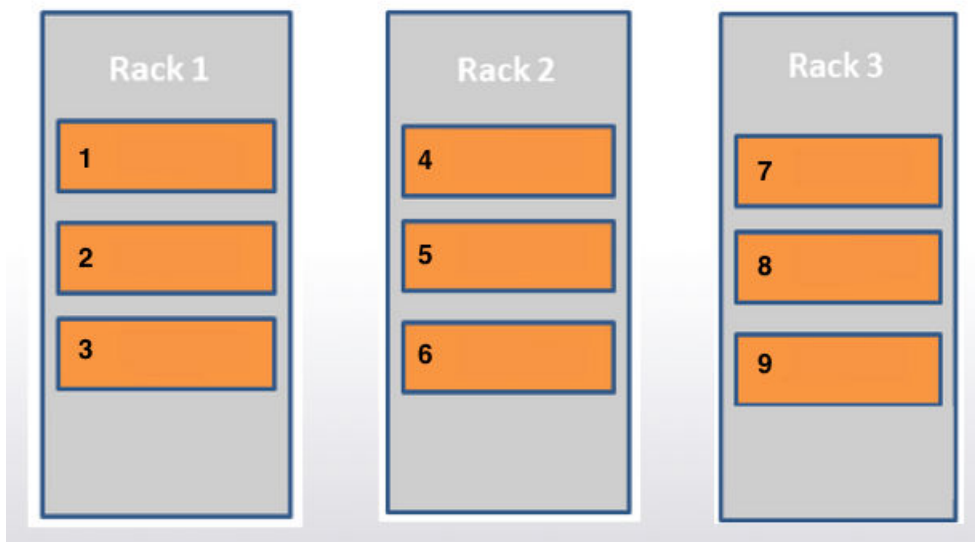
1. Scalability: by partitioning files into blocks and distributing them to many servers operating in parallel, HDFS can scale to potentially unlimited number of files of any size. By adding more DataNodes the storage capacity of the system can be increased arbitrarily. It has been demonstrated to scale beyond tens of petabytes (PB). More importantly, it does so with linear performance characteristics and cost.
2. Durability: HDFS creates multiple copies of each block (by default 3, on different racks) to minimize the probability of data loss.
3. High sequential read/write performance: by splitting huge files into blocks and spreading these into multiple machines, a file can be read in parallel by concurrently accessing multiple disks (on different nodes)

2. File I/O operations and replica management.

2.1 – Replication policy

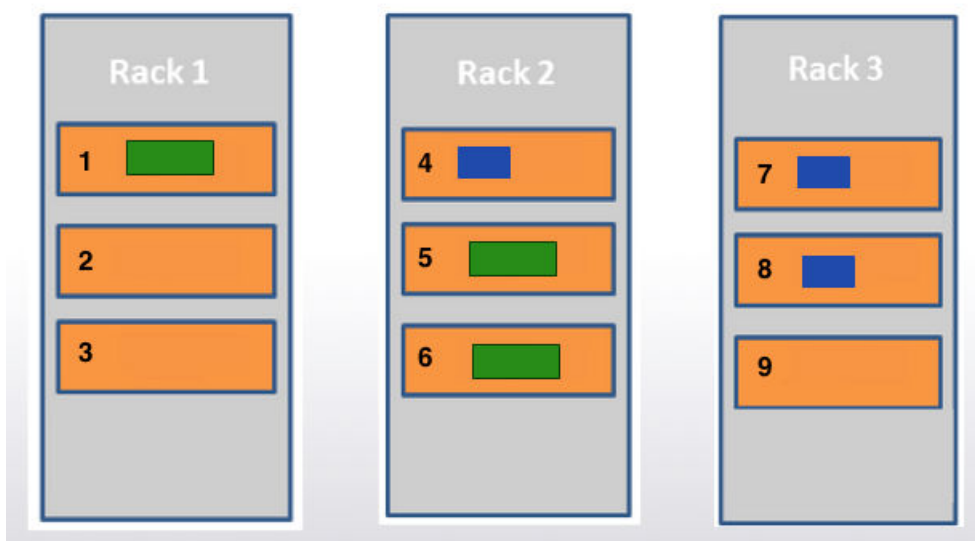
Assume your HDFS cluster is made of 3 racks, each containing 3 DataNodes. Assume also the HDFS is configured to use a block size of 100 megabytes and that a client is connecting from outside the datacenter (therefore no DataNode is privileged).

1. The client uploads a file of 150 megabytes. Draw in the picture below a possible blocks configuration according to the default HDFS replica policy. How many replicas are there for each block? Where are these replicas stored?
2. Can you find a with a different policy that, using the same number of replicas, improves the expected availability of a block? Does your solution show any drawbacks?
3. Referring to the picture below, assume a block is stored in Node 3, as well as in Node 4 and Node 5. If this block of data has to be processed by a task running on Node 6, which of the three replicas will be actually read by Node 6?



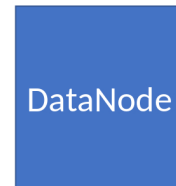
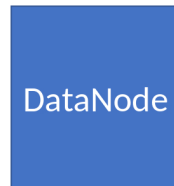
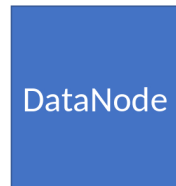
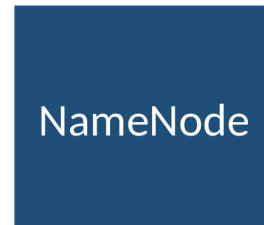
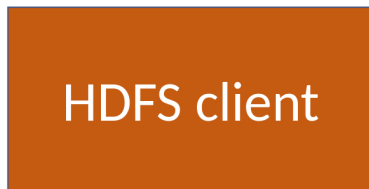
Solution

1. For each block independently, the HDFS's placement policy is to put one replica on a random node in a random rack, another on one node in a different rack, and the last on a different node in the same rack chosen for the second replica. A possible configuration is shown in the picture (but there are many more valid solutions).
2. One could decide to store the 3 replicas in 3 different racks, increasing the expected availability. However this would also slow down the writing process that would involve two inter-rack communications instead of one. Usually, the probability of failure of an entire rack is much smaller than the probability of failure of a node and therefore it is a good tradeoff to have 2/3 of the replicas in one rack.
3. Either the one stored in Node 4 or Node 5, assuming the intra-rack topology is such that the distance from these nodes to Node 6 is the same. In general, the reading priority is only based on the distance, not on which node was first selected in the writing process.



2.2 – File read and write data flow.

To get an idea of how data flows between the client interacting with HDFS, consider a diagram below which shows main components of HDFS.

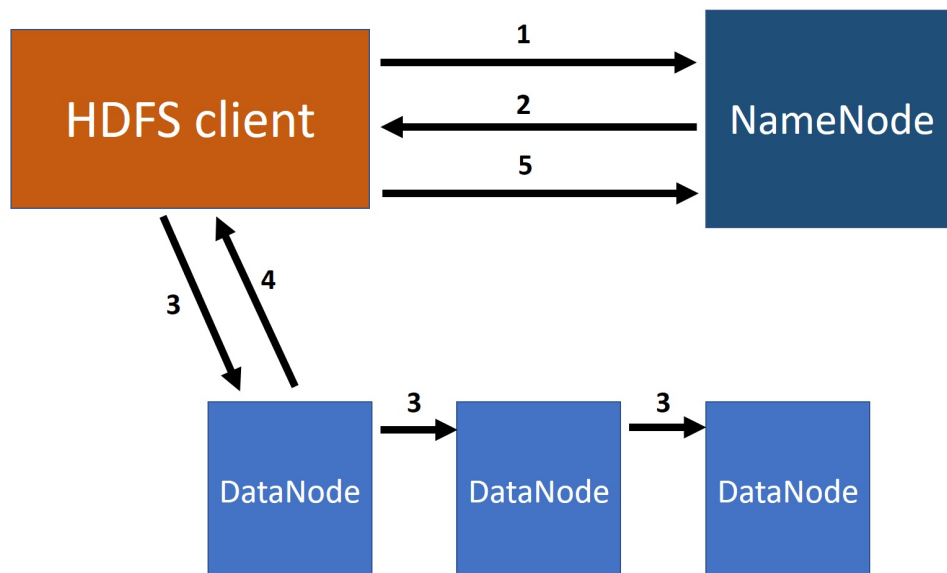


1. Draw the main sequence of events when a client copies a file to HDFS.
2. Draw the main sequence of events when a client reads a file from HDFS.
3. Why do you think a client writes data directly to datanodes instead of sending it through the namenode?

Solution

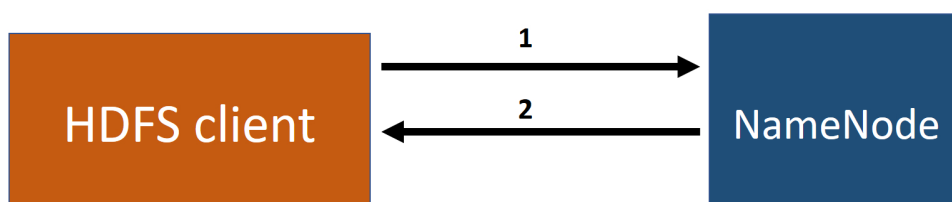
1. See diagram below; **steps 1-4** are applied for each block of the file.

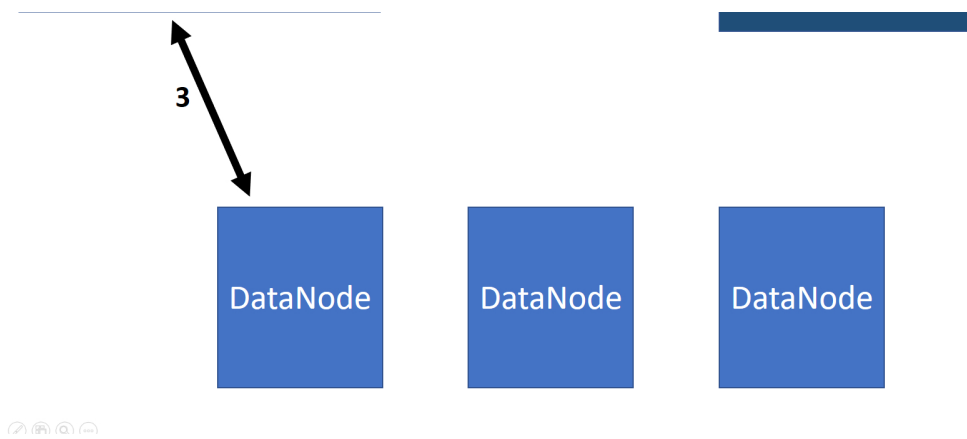
- Step 1 - HDFS client asks for DataNode to host replica of the i-th block of the file.
- Step 2 - NameNode replies with 3 (default replication factor) Datanodes' locations where to store the i-th block, ordered by increasing distance
- Step 3 - The client writes i-th block to DataNodes in pipeline fashion
- Step 4 - The first DataNode replies with acknowledgement
- Step 5 - The client sends to the NameNode a request to close the file



2. See diagram below

- Step 1 - HDFS client requests a file
- Step 2 - NameNode replies with a list of blocks and the locations of each replica, ordered by decreasing distance.
- Step 3 - The client reads each block from the closest datanode.





Solution

1. If the namenode was responsible for copying all files to datanodes then it would become a bottleneck.

3. Storage models

Last week we delve into the key-value model (incl. object storage, e.g., Amazon S3/Azure Blob). This week, we can contrast the object storage in the key-value model with block storage, which is used for distributed file systems.

3.1 – List two differences between Object Storage and Block Storage.

Solution

1. Object Storage treats the files as **black-box** objects, therefore offering a limited API (GET, PUT, DELETE), while Block Storage treats each file as a **sequence of blocks**, and hence offers a richer set of APIs to the user.
2. Pure Object Storage has a limit on object size, since the object cannot be partitioned across machines. Block Storage does not have this limitation and can split objects into blocks. Therefore, Block Storage can store PB files, whereas Object Storage is limited by the storage capacity of a single node. On the other hand, object storage can store more files than Block Storage.

3.2 – Compare Object Storage and Block Storage. For each of the following use cases, say which technology (object or block storage) better fits the requirements and briefly justify why.

1. Store Netflix movie files in such a way they are accessible from many client applications at the same time [Object storage | Block Storage]
2. Store experimental and simulation data from CERN [Object storage | Block Storage]
3. Store the auto-backups of iPhone/Android devices [Object storage | Block Storage]

Solution

1. Object Storage. Client applications do not need block-level access to stream the data, and an high level interface is best suited to this purpose.
2. Block Storage. Because it can handle large files and store more data than ordinary object storage.
3. Object Storage. Backups are usually written once and rarely read. When data is read, partial access to each file is not essential. The client devices do not need to know the block composition of the object being stored. In fact, Apple [publicly confirmed](#) that backups data for iOS is stored on Amazon S3 and Microsoft Azure.

3.3 – Cost of Object Storage (Optional)

Azure Object Storage offers different access tiers, which allow you to store blob object data in the most cost-effective manner.

Imagine you want to have a copy of your important documents, photos and videos to both ensure durability and to share them across your several devices. You need about 600GB of storage space. Two possibilities would be [hot and cool storage tiers on Azure Blob Storage](#).

1. Explain the difference between hot and cool storages.
2. Compare costs of cool and hot storages on [Azure Storage](#). Explain why prices are different. Which one would you choose? Why?
3. Another possible solution would be to pay for a [Dropbox Pro account](#). Compare costs of Dropbox and Azure storage. Which one is cheaper?
4. What about buying an external hard drive? List two advantages and two disadvantages of this solution.

Solution

1. Hot storage exploits replication for reliability, and cool storage erasure coding. Thus, Hot storage occupies more memory, but allows fast recovering. On the other hand, cool storage uses less memory (up to 50% less), but requires involving a lot of compute power to recover data.
2. Hot tier is cheaper for frequent data accesses, whereas cold tier is cheaper for long term storage with infrequent accesses.
3. Dropbox is cheaper.
4. A good external hard drive of 1TB is about 60 €, so one of the advantages is that this is the cheapest solution in the long run. Another advantage is the better latency and throughput that you could achieve. Drawbacks are the limited availability (assuming you don't carry the HD always with you), more limited durability (yes, hard drives [fail quite often](#)), and many other cost and time factors.

Explore on your own

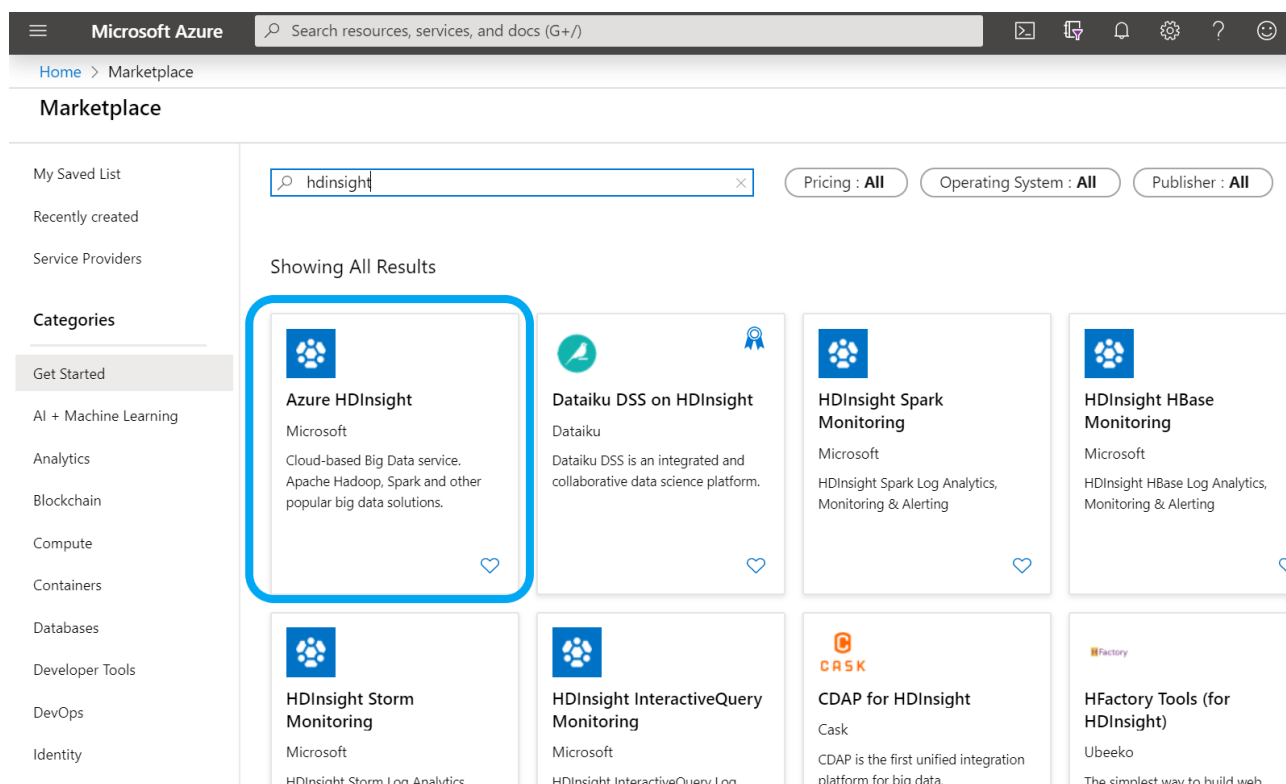
4. Run an HDFS cluster

It's time to work on a real environment! You will now create an HDFS cluster running on Azure and fill it with data. The purpose of this exercise is to familiarise you with shell commands and the operations to administer an HDFS deployment.

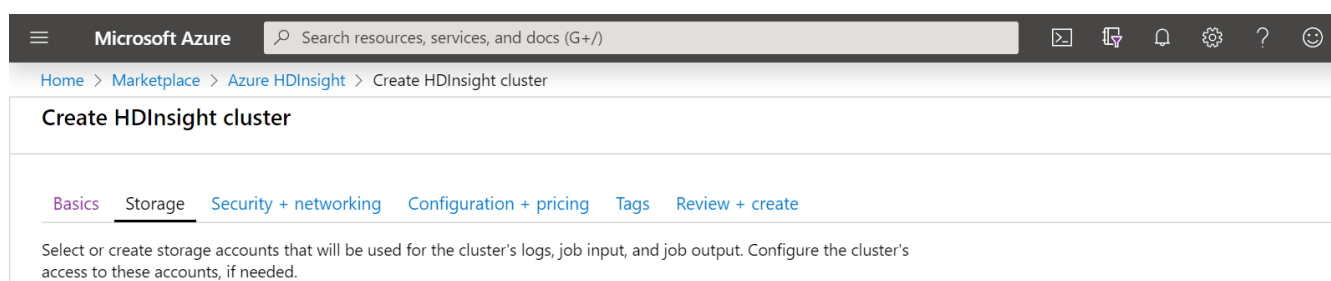
4.1 – Do the following to set up an HDFS cluster:

Important: we want you to use a small but real cluster for running HDFS rather than a single machine. But, these clusters burn Azure credit very quickly—the cheapest configuration consumes roughly **1 CHF per hour**, which is a lot relative to your overall credit—so it is very important for you to ****delete**** your cluster once you are done. Luckily, *it is possible to keep your data intact when you delete a cluster*, and see it again when you recreate it; we will touch upon this in the process. Now, let's start ...

1. Open the [Azure portal](#) and click on the "+ Create a Resource" button on the left. Type "hdinsight" in the search box, and select "HDInsight". HDInsight is Microsoft's cloud service which wraps Hadoop, HBase, Spark and other Big Data technologies; read more [here](#).



1. Click on Create, fill in the form with cluster name, user names and passwords, and select "Hadoop" as the cluster type. Create a new resource group, e.g., "exercise03". Click "Next".



Primary storage

Select or create a storage account that will be the default location for cluster logs and other output.

Primary storage type * Azure Storage

Selection method * ☒ Select from list ☐ Use access key

Primary storage account * Create new
The value must not be empty.

Container * clusterfinto-2020-03-01t16-24-39-079z

Data Lake Storage Gen1

Provide details for the cluster to access Data Lake Storage Gen1. The cluster will be able to access any Data Lake Storage Gen1

Review + create « Previous Next: Security + networking »

1. In the next we need to configure the location to store all the cluster data. The canonical storage layer for an HDFS cluster uses the Blob service of Windows Azure Storage and the primary advantage is that it allows you to delete your HDFS cluster without losing the stored data: you can recreate the cluster using the same Azure Storage Account and the same container and you will see the same data. This is useful, for example, if you don't have time to finish this exercise in one sitting: you can just delete your cluster, recreate it later, and continue your work. Azure storage is selected by default (see the screenshot). In "Select a Storage Account" click "Create new" and specify a name. **Important: if you are recreating your HDFS cluster and want to see the existing data, then choose "Select existing" and set the container name to the one that you see in the "Storage Accounts" tab of Azure—by default Azure generates a new container name every time you create a cluster, which then points to a different container.** Leave everything else as it is and click "Next".
2. In the "Security + networking" step do not choose anything and just click "Next".
3. Now we need to choose the configuration of the nodes in our HDFS cluster. Nodes in HDInsight have similar names yet types, compared with the concepts of HDFS you have learned from the lecture. In this exercise, you will see *head node (aka namenode)* and *worker node (aka datanode)*. To read more, this blog on "[Nodes in HDInsight](#)" is helpful. It will be enough to have only 2 **worker** nodes (see the screenshot). As for the node size, let us be wise and select an economical option: from the drop-down menu choose the option "**A4 - v2**"(see the screenshot); do the same for the **Head** nodes. Click "Next".

Microsoft Azure Search resources, services, and docs (G+)

Home > Marketplace > Azure HDInsight > Create HDInsight cluster

Create HDInsight cluster

Node configuration

Configure your cluster's size and performance, and view estimated cost information.

The cost estimate represented in the table does not include subscription discounts or costs related to storage, networking, or data transfer.

ⓘ This configuration will use 16 of 50 available cores in the East US region.
[View cores usage](#)

Node type	Node size	Number of ...	Estimated co
Head node	A4 v2 (4 Cores, 8 GB RAM), 0.22 EUR/hour	2	0.44 EUR
Worker node	A4 v2 (4 Cores, 8 GB RAM), 0.22 EUR/hour	2 ✓	0.44 EUR

☐ Enable autoscale (Preview) [Learn](#) [More](#)

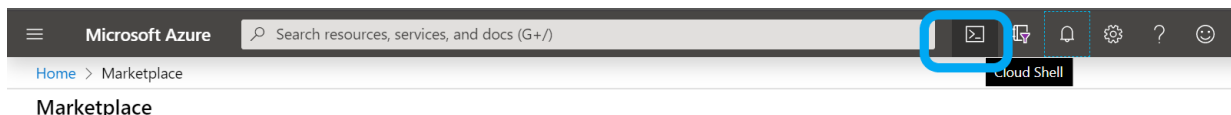
Review + create « Previous Next: Tags »

4. All the fields on the "Tags" step can be left as they are. Simply proceed by clicking "Next".
5. In the last step, "Review + create", check if the settings are as you intend. These clusters are expensive, so it is worth checking the price estimate at this step: for me it is 0.87 CHF/hour; if your price is larger than this, check your node sizes and counts. When done, initiate the cluster creation by clicking "Create". The process will take time, around 15—25 minutes; in my own case it took 20 minutes.

4.2 Accessing your cluster

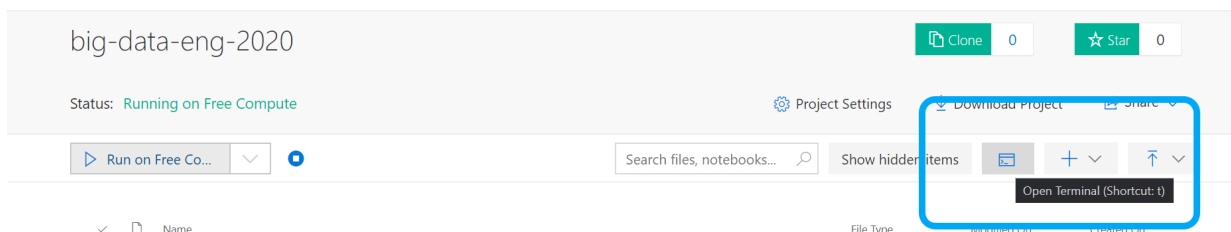
The standard way to interact with an HDFS cluster is via the Java API or a command-line interface. We will use the latter and for this, you will need to run the `ssh` program in a terminal in order to connect to your cluster. There are three options of how you can do this:

1. **On your own machine** you can just use a normal terminal if you have `ssh` installed. Linux usually has it, as does MacOS. Windows doesn't have it by default (powershell on Win10 does, though), but Windows users can use one of the browser-based options, which are described next, and the other option is to install [PuTTY](#).
2. **In your browser, two possibilities:**
 - A. Use the **Azure Cloud Shell**. Click on the Cloud Shell icon at the top of Azure Dashboard toolbar:



Choose "bash". It will request your approval for creating a Storage Account required for the shell; choose the corresponding subscription and you can also create a new storage account or use an old storage account.

- B. Use the built-in **terminal on Jupyter**. In your [notebooks.azure.com](#), find the terminal button on the right side, click on it.



In your terminal of choice, run the following (this command with everything filled-in is also available on the Azure page of your HDFS cluster, if you click "SSH + Cluster login" and select the host name):

```
ssh <ssh_user_name>@<cluster_name>-ssh.azurehdinsight.net
```

In this command, `<ssh_user_name>` is the "ssh username" that you have chosen in the first step of creating the HDFS cluster, and `<cluster_name>` also comes from that form. Note that the cluster name has to be suffixed with `-ssh`.

If after running the `ssh` command you see a message similar to this:

```
Welcome to HDInsight.
```

```
[...]
```

```
To run a command as administrator (user "root"), use "sudo <command>".  
See "man sudo_root" for details.
```

```
<ssh_user_name>@hn0-cluster:~$
```

then you have successfully connected to your HDFS cluster. Now proceed to the next task.

4.3 – Monitor the HDFS cluster

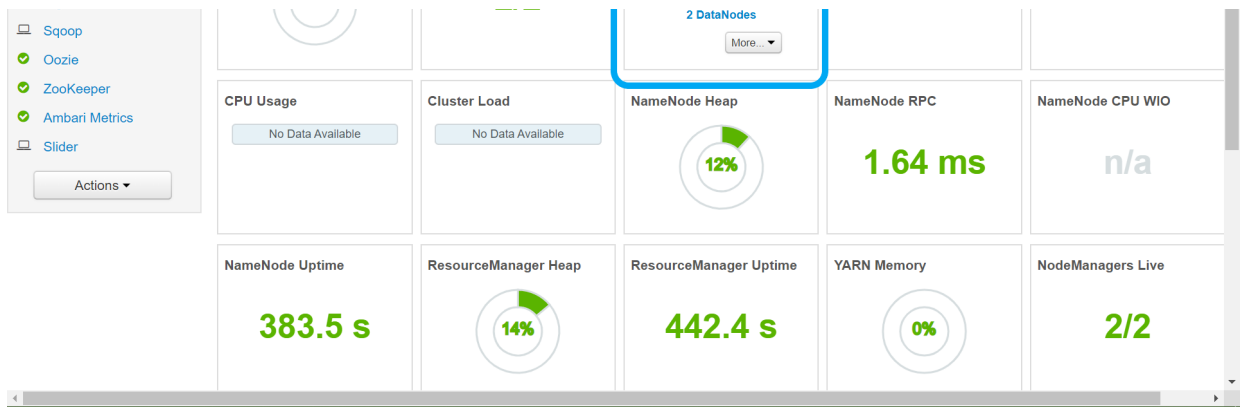
In the above screenshot, we highlight the **Cluster Dashboards** powered by Ambari. The Apache Ambari project is aimed at making Hadoop management simpler by developing software for provisioning, managing, and monitoring Apache Hadoop clusters. Ambari provides an intuitive, easy-to-use Hadoop management web UI backed by its RESTful APIs ([source](#)).

We utilize this monitoring dashboard to inspect the behaviors of our HDFS cluster. You will see prominent behavior changes (e.g., resource usage live for the nodes and network usage going up) esp. when uploading some new data.

Click into the Ambari home. You can explore the following:

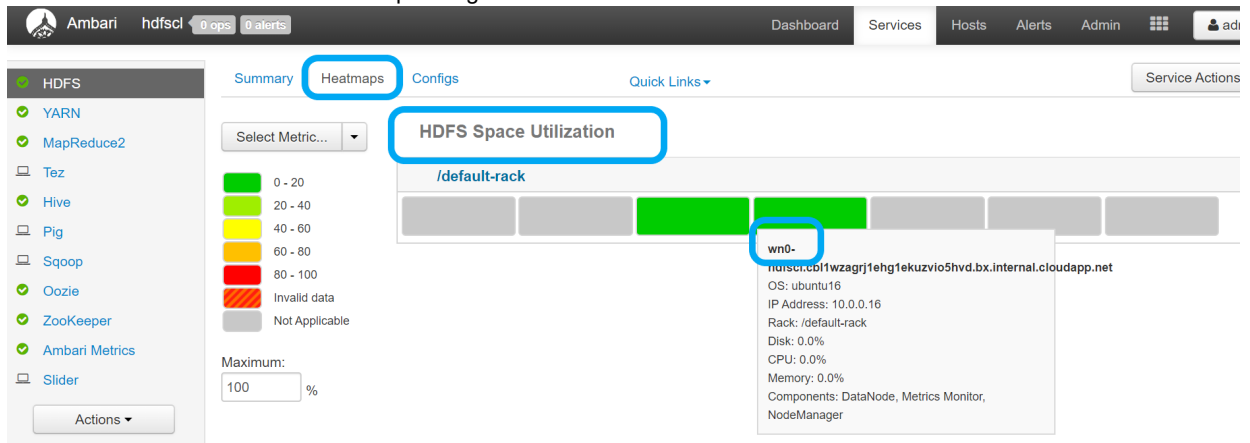
1. An overview of the cluster (namenode, datanode)
 - One namenode is active, the other one is standby





1. Space utilization on HDFS and the node details (2 namenodes, 2 datanodes, 3 zookeeper nodes)

- The "Zookeeper" nodes are selected by default (Zookeeper is a [distributed coordination service](#) used by HDFS). By default, HDInsight provides 3 Zookeeper nodes.
- You can click on each box to see the operating details of each node.



1. Under Hosts, you will find more details

Ambari

hdfscl

0 ops

0 alerts

Dashboard

Services

Hosts


Alerts

Admin

ad

Actions

Filter by host and component attributes or search by keyword ...

<input type="checkbox"/>	Name	IP Address	Rack	Cores	RAM	Disk Usage	Load Avg	Versions	Components
<input type="checkbox"/>	hn0-hdfscl.cbl1wzagrij1ehg1eku...	10.0.0.20	/default-rack	4 (4)	7.77GB	<div></div>	6.24	HDP-2.6.5.3015-8	21 Components
<input type="checkbox"/>	hn1-hdfscl.cbl1wzagrij1ehg1eku...	 10.0.0.19	/default-rack	4 (4)	7.77GB	<div></div>	1.17	HDP-2.6.5.3015-8	17 Components
<input type="checkbox"/>	wn0-hdfscl.cbl1wzagrij1ehg1ek...	10.0.0.16	/default-rack	4 (4)	7.77GB	<div></div>	0.48	HDP-2.6.5.3015-8	7 Components
<input type="checkbox"/>	wn2-hdfscl.cbl1wzagrij1ehg1ek...	10.0.0.17	/default-rack	4 (4)	7.77GB	<div></div>	0.28	HDP-2.6.5.3015-8	7 Components
<input type="checkbox"/>	zk0-hdfscl.cbl1wzagrij1ehg1eku...	10.0.0.13	/default-rack	2 (2)	3.83GB	<div></div>	0.63	HDP-2.6.5.3015-8	4 Components
<input type="checkbox"/>	zk1-hdfscl.cbl1wzagrij1ehg1eku...	10.0.0.11	/default-rack	2 (2)	3.83GB	<div></div>	1.81	HDP-2.6.5.3015-8	4 Components
<input type="checkbox"/>	zk2-hdfscl.cbl1wzagrij1ehg1eku...	10.0.0.12	/default-rack	2 (2)	3.83GB	<div></div>	0.49	HDP-2.6.5.3015-8	4 Components

Show: 10

1 - 7 of 7

4.4 – Upload a file into HDFS

Let us go over some common [commands](#):

```
ls, cat, mkdir, cp, rm, rmdir, copyFromLocal, copyToLocal
```

A quick note: we will use the `hadoop fs` keywords as a prefix to the commands, but we could have used `hdfs dfs` keywords as well. The only difference is that `hadoop fs` is a more “generic” command that allows you to interact with multiple file systems including Hadoop (but also your local file system), whereas `hdfs dfs` is specific to HDFS.

1. List the directories or files: `ls` (some of these commands could be slow)

- For a file `ls` returns statistics on the file: `$hadoop fs -ls /example/data/fruits.txt`
This gives you the following: `-rw-r--r-- 1 root supergroup 66 2020-03-01 17:42 /example/data/fruits.txt`

- For a directory it returns list of its direct children as in Unix: `$hadoop fs -ls /example`
This gives you the following:

```
Found 2 items
drwxr-xr-x - root supergroup 0 2020-03-01 17:43 /example/data
drwxr-xr-x - root supergroup 0 2020-03-01 17:43 /example/jars
```

- List files in a directory: `$hadoop fs -ls /example/data/*`
- List all the directories in your HDFS: `$hadoop fs -ls -R /`

2. Copies source paths to stdout: `cat`

- Inspect the file content (**DO NOT** do it for a large file):

```
$hadoop fs -cat /example/data/fruits.txt
```

3. **Copy files from source to destination:** `cp`

```
$hadoop fs -cp /example/data/fruits.txt /example/data/fruits-copy.txt
```

 This shouldn't return anything.

4. **Delete files:** `rm`

```
$hadoop fss -rm /example/data/fruits-copy.txt
```

This gives you: Deleted /example/data/fruits-copy.txt

- When you then check `$hadoop fs -ls /example/data/fruits-copy.txt`, you will see the file does not exist anymore by getting this message: `ls: '/example/data/fruits-copy.txt': No such file or directory`

5. Creates directories: `mkdir`

- We create a folder for our exercise by `$hadoop fs -mkdir /ex03`

6. Copy a file from the file system where your terminal locates to HDFS: `copyFromLocal`

- First we create a file on the file system where your terminal locates (can be local or on a cluster):

```
$ echo "we create a file on the cluster(hn0) file system" > cluster.txt
```

```
$ cat cluster.txt
```

- These two commands are equivalent:

```
$hadoop fs -copyFromLocal -f cluster.txt /ex03/
```

```
$hadoop fs -put -f cluster.txt /ex03/
```

- The `-f` option will overwrite the destination if it already exists.

7. The reverse operation of `copyFromLocal`, i.e., download a file from HDFS to the file system where your terminal locates: `copyToLocal`

- `$hadoop fs -copyToLocal /example/data/fruits.txt`
- If you do `$ ls` in your terminal, you should see the "fruits" file there.

8. Delete a directory: `rmdir`

- We cannot delete a non-empty folder directly, so we need to remove the files inside by `$hadoop fss -rm /ex03/*`
- We then remove the folder `$hadoop fs -rmdir /ex03`
- If you do `ls` again on the deleted folder, it should not exist any more. `$hadoop fs -ls /ex03`

Now we try to upload a big file (>1G) to our HDFS cluster and you can inspect the cluster with Ambari.

Download a compressed dump of all the articles on Wikipedia: `$ wget https://bigdataforeng.blob.core.windows.net/ex05/wiki.tar.gz`

Next, let's upload this file from the local file system to HDFS:

```
$hadoop fs -mkdir /bigdata
```

```
$hadoop fs -put ~/wikibig.tar.gz /bigdata
```

```
$hadoop fs -ls /bigdata
```

You should see a listing of the following form with details about the uploaded file (permissions, ownership, size and modification date):

```
Found 1 items
```

```
-rw-r--r-- 1 sshuser supergroup 1116569600 2020-03-01 17:45 /bigdata/wikibig.tar.gz
```

This was a quick overview of the basic shell commands and in a following exercise we will run a MapReduce computation over data stored in HDFS.

Feel free to try out the commands on your own. For instance, pick an image file in your computer (or you can also download a random one) and try to upload it to HDFS. You may need to create an empty directory before uploading.

Important: Don't forget to terminate and delete your cluster once you are done (!)