HDFS

Data and Analytics Training

Big Data – Self-Study Guide (201)

This study guide serves as a basis for your self-study on the basics of HDFS. The study guide gives you an overview on the main topics and presents a number of internal and external resources you should use to get a deeper understanding of HDFS, the related terms and technical details.

Please especially use the external resources provided. They are a main part of this study guide and are required to get a good understanding of the discussed topics.

Estimated Completion Time

4 hours (including time for external resources)

Prerequisites

We recommend completing the course "*Big Data – 101 – Getting started with Big Data*" to get a general understanding of the architecture and related technologies before starting with this course.

Objectives

After completing this study guide, you will be able to understand

- Working with remote desktop
- Browsing the HDFS file system
- Running a Hadoop map-reduce job
- HDFS blob storage
- Benefits of using Azure blob storage

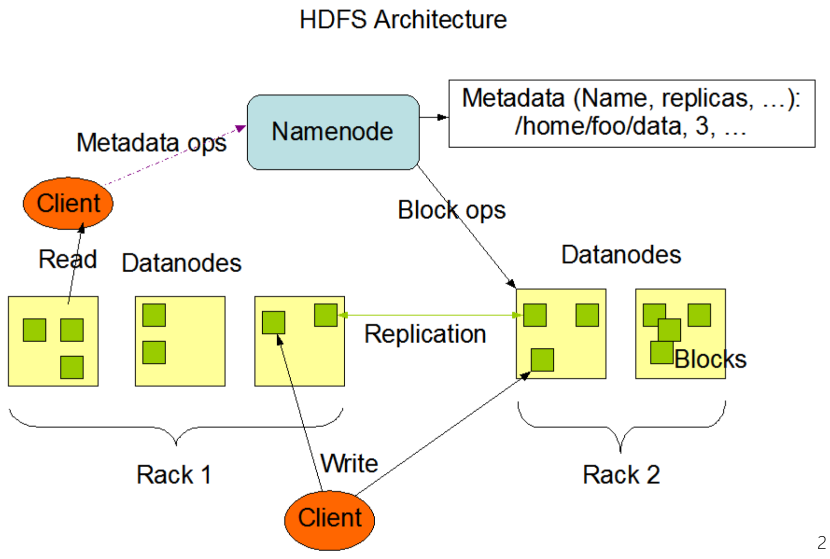
Contents

Introduction

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS was originally built as infrastructure for the Apache Nutch web search engine project. It is now an Apache Hadoop subproject.

HDFS has a master/slave architecture. A cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files.

Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes itself are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.[1]



[2]

In practice, you call for a file object by using the HDFS application programming interface (API), and the code locates the node where the data is located and returns the data to you. That's the short version, and, of course, it gets a little more complicated from there. HDFS can replicate the data to

[1] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Introduction

[2] http://hadoop.apache.org/docs/r1.2.1/images/hdfsarchitecture.gif

multiple nodes, and it uses a name node daemon to track where the data is and how it is (or isn't) replicated.

The NameNode and DataNode are pieces of software designed to run on commodity machines. These machines typically run a GNU/Linux operating system (OS). HDFS is built using the Java language; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines. A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The architecture does not preclude running multiple DataNodes on the same machine but in a real deployment that is rarely the case.
The existence of a single NameNode in a cluster greatly simplifies the architecture of the system. The NameNode is the arbitrator and repository for all HDFS metadata. The system is designed in such a way that user data never flows through the NameNode.[3]

Please watch this 33 minute video giving an introduction to the HDFS architecture: http://www.youtube.com/watch?v=ziqx2hJY8Hg

Data Replication

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.

The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.[4]

The replication factor is a property that can be set in the HDFS configuration file that will allow you to adjust the global replication factor for the entire cluster. For each block stored in HDFS, there will be n – 1 duplicated blocks distributed across the cluster. For example, if the replication factor was set to 3 (default value in HDFS) there would be one original block and two replicas.

[3] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Introduction

[4] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Data+Replication

To change the numbers of replications that should be kept, you can open the hdfs-site.xml file. This file is usually found in the conf/ folder of the installation directory. Change or add the following property to hdfs-site.xml:

```
<property>

<name>dfs.replication<name>

<value>3<value>

<description>Block Replication<description>

<property>
```

hdfs-site.xml is used to configure HDFS. Changing the dfs.replication property in hdfs-site.xml will change the default replication for all files placed in HDFS.

You can also change the replication factor on a per-file basis using the Hadoop FS shell.

```
[training@localhost ~]$ hadoop fs –setrep –w 3 /my/file
```

Alternatively, you can change the replication factor of all the files under a directory.

```
[training@localhost ~]$ hadoop fs –setrep –w 3 -R /my/dir
```

Data Blocks

HDFS is designed to support very large files. Applications that are compatible with HDFS are those that deal with large data sets. These applications write their data only once but they read it one or more times and require these reads to be satisfied at streaming speeds. HDFS supports write-once-read-many semantics on files. A typical block size used by HDFS is 64 MB. Thus, an HDFS file is chopped up into 64/128 MB chunks, and if possible, each chunk will reside on a different DataNodes.[5]

Each block is replicated some number of times—the default replication factor for HDFS is three. When addBlock() is invoked, space is allocated for each replica. Each replica is allocated on a different DataNode. The algorithm for performing this allocation attempts to balance performance and reliability. This is done by considering the following factors:

- The dynamic load on the set of DataNodes: Preference is given to more lightly loaded DataNodes.

[5] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Data+Replication

- The location of the DataNodes: Communication between two nodes in different racks has to go through switches. In most cases, network bandwidth between machines in the same rack is greater than network bandwidth between machines in different racks.

For the common case, when the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a node in a different (remote) rack, and the last on a different node in the same remote rack. This policy cuts the inter-rack write traffic which generally improves write performance. The chance of rack failure is far less than that of a node failure; therefore this co-location policy does not adversely impact data reliability and availability guarantees. However, it does reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three. With this policy, the replicas of a file do not evenly distribute across the racks. One third of replicas are on one node on some rack; the other two thirds of replicas are on distinct nodes one a different rack. This policy improves write performance without compromising data reliability or read performance.

Typically HDFS block sizes are significantly larger than the block sizes you would see for a traditional filesystem (for example, the filesystem on my laptop uses a block size of 4 KB). The block size setting is used by HDFS to divide files into blocks and then distribute those blocks across the cluster. For example, if a cluster is using a block size of 64 MB, and a 128-MB text file was put in to HDFS, HDFS would split the file into two blocks (128 MB/64 MB) and distribute the two chunks to the data nodes in the cluster.

Open the hdfs-site.xml file. This file is usually found in the conf/ folder of the Hadoop installation directory.Set the following property in hdfs-site.xml:

```
<property>

<name>dfs.block.size<name>

<value>134217728<value>

<description>Block size<description>

<property>
```

hdfs-site.xml is used to configure HDFS. Changing the dfs.block.size property in hdfs-site.xml will change the default block size for all the files placed into HDFS. In this case, we set the dfs.block.size to 128 MB. Changing this setting will not affect the block size of any files currently in HDFS. It will only affect the block size of files placed into HDFS after this setting has taken effect.[6]

[6] http://princetonits.com/technology/how-to-configure-replication-factor-and-block-size-for-hdfs/

There are a number of things that this impacts. Most obviously, a file will have fewer blocks if the block size is larger. This can potentially make it possible for client to read/write more data without interacting with the Namenode, and it also reduces the metadata size of the Namenode, reducing Namenode load (this can be an important consideration for extremely large file systems).

With fewer blocks, the file may potentially be stored on fewer nodes in total; this can reduce total throughput for parallel access, and make it more difficult for the MapReduce scheduler to schedule data-local tasks.

When using such a file as input for MapReduce (and not constraining the maximum split size to be smaller than the block size), it will reduce the number of tasks which can decrease overhead. But having fewer, longer tasks also means you may not gain maximum parallelism (if there are fewer tasks than your cluster can run simultaneously), increase the chance of stragglers, and if a task fails, more work needs to be redone. Increasing the amount of data processed per task can also cause additional read/write operations (for example, if a map task changes from having only one spill to having multiple and thus needing a merge at the end).[7]

Staging

A client request to create a file does not reach the NameNode immediately. In fact, initially the HDFS client caches the file data into a temporary local file. Application writes are transparently redirected to this temporary local file. When the local file accumulates data worth over one HDFS block size, the client contacts the NameNode. The NameNode inserts the file name into the file system hierarchy and allocates a data block for it. The NameNode responds to the client request with the identity of the DataNode and the destination data block. Then the client flushes the block of data from the local temporary file to the specified DataNode. When a file is closed, the remaining un-flushed data in the temporary local file is transferred to the DataNode. The client then tells the NameNode that the file is closed. At this point, the NameNode commits the file creation operation into a persistent store. If the NameNode dies before the file is closed, the file is lost.

The above approach has been adopted after careful consideration of target applications that run on HDFS. These applications need streaming writes to files. If a client writes to a remote file directly without any client side buffering, the network speed and the congestion in the network impacts throughput considerably. This approach is not without precedent. Earlier distributed file

[7] http://channel9.msdn.com/Forums/TechOff/Impact-of-changing-block-size-in-Hadoop-HDFS

systems, e.g. **AFS**, have used client side caching to improve performance. A POSIX requirement has been relaxed to achieve higher performance of data uploads.

Replication Pipelining

If an application adds data to HDFS by creating a new file and writing the data to it. After the file is closed, the bytes written cannot be altered or removed except that new data can be added to the file by reopening the file for append. HDFS implements a single-writer, multiple-reader model.

File lease

The HDFS client that opens a file for writing is granted a lease for the file; no other client can write to the file. The writing client periodically renews the lease by sending a heartbeat to the NameNode. When the file is closed, the lease is revoked. The lease duration is bound by a soft limit and a hard limit. Until the soft limit expires, the writer is certain of exclusive access to the file. If the soft limit expires and the client fails to close the file or renew the lease, another client can preempt the lease. If after the hard limit expires (one hour) and the client has failed to renew the lease, HDFS assumes that the client has quit and will automatically close the file on behalf of the writer, and recover the lease. The writer's lease does not prevent other clients from reading the file; a file may have many concurrent readers.[8]

Block replication

Block replicas are written to distinct data nodes (to cope with data node failure), or distinct racks (to cope with rack failure). Filesystem clients communicate with the name node to retrieve metadata (such as information about the directory hierarchy, or the mapping from a file to its blocks), and with the data nodes to retrieve data. It is important that clients retrieve the file data direct from the data nodes rather than having it routed via the name node, as the latter would create a bottleneck at the name node and severely limit the aggregate bandwidth available to clients.

When there is a need for a new block, the NameNode allocates a block with a unique block ID and determines a list of DataNodes to host replicas of the block. The DataNodes form a pipeline, the order of which minimizes the total network distance from the client to the last DataNode.[9] The rack placement policy is managed by the name node, and replicas are placed as follows[10]:

1. The first replica is placed on a random node in the cluster, unless the write originates from within the cluster, in which case it goes to the local node.

[8] http://www.aosabook.org/en/hdfs.html

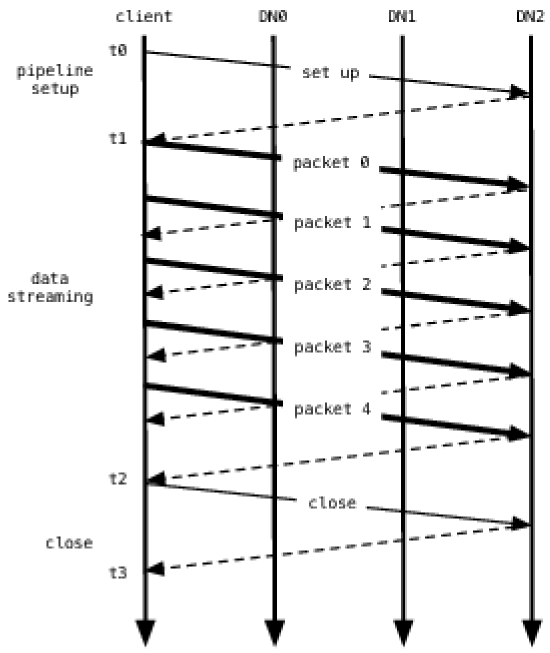
[9] http://www.aosabook.org/en/hdfs.html

[10] http://blog.cloudera.com/wp-content/uploads/2010/03/HDFS_Reliability.pdf

2. The second replica is written to a different rack from the first, chosen at random.
3. The third replica is written to the same rack as the second replica, but on a different node.
4. Fourth and subsequent replicas are placed on random nodes, although racks with many replicas are biased against, so replicas are spread out across the cluster.

The client then flushes the data block to the first DataNode. The first DataNode starts receiving the data in small portions (4 KB), writes each portion to its local repository and transfers that portion to the second DataNode in the list. The second DataNode, in turn starts receiving each portion of the data block, writes that portion to its repository and then flushes that portion to the third DataNode. Finally, the third DataNode writes the data to its local repository. Thus, a DataNode can be receiving data from the previous one in the pipeline and at the same time forwarding data to the next one in the pipeline. Thus, the data is pipelined from one DataNode to the next.

After data are written to an HDFS file, HDFS does not provide any guarantee that data are visible to a new reader until the file is closed. If a user application needs the visibility guarantee, it can explicitly call the hflush operation. Then the current packet is immediately pushed to the pipeline, and the hflush operation will wait until all DataNodes in the pipeline acknowledge the successful transmission of the packet. All data written before the hflush operation are then certain to be visible to readers.



[11]

[11] http://www.aosabook.org/en/hdfs.html

If no error occurs, block construction goes through three stages as shown in the above figure illustrating a pipeline of three DataNodes (DN) and a block of five packets. In the picture, bold lines represent data packets, dashed lines represent acknowledgment messages, and thin lines represent control messages to setup and close the pipeline. Vertical lines represent activity at the client and the three DataNodes where time proceeds from top to bottom. From `t0` to `t1` is the pipeline setup stage. The interval `t1` to `t2` is the data streaming stage, where `t1` is the time when the first data packet gets sent and `t2` is the time that the acknowledgment to the last packet gets received. Here an hflush operation transmits `packet 2`. The hflush indication travels with the packet data and is not a separate operation. The final interval `t2` to `t3` is the pipeline close stage for this block.

In a cluster of thousands of nodes, failures of a node (most commonly storage faults) are daily occurrences. A replica stored on a DataNode may become corrupted because of faults in memory, disk, or network. HDFS generates and stores checksums for each data block of an HDFS file. Checksums are verified by the HDFS client while reading to help detect any corruption caused either by client, DataNodes, or network. When a client creates an HDFS file, it computes the checksum sequence for each block and sends it to a DataNode along with the data. A DataNode stores checksums in a metadata file separate from the block's data file. When HDFS reads a file, each block's data and checksums are shipped to the client. The client computes the checksum for the received data and verifies that the newly computed checksums matches the checksums it received. If not, the client notifies the NameNode of the corrupt replica and then fetches a different replica of the block from another DataNode.

When a client opens a file to read, it fetches the list of blocks and the locations of each block replica from the NameNode. The locations of each block are ordered by their distance from the reader. When reading the content of a block, the client tries the closest replica first. If the read attempt fails, the client tries the next replica in sequence. A read may fail if the target DataNode is unavailable, the node no longer hosts a replica of the block, or the replica is found to be corrupt when checksums are tested.

HDFS permits a client to read a file that is open for writing. When reading a file open for writing, the length of the last block still being written is unknown to the NameNode. In this case, the client asks one of the replicas for the latest length before starting to read its content.

The design of HDFS I/O is particularly optimized for batch processing systems, like MapReduce, which require high throughput for sequential reads and writes. Ongoing efforts will improve read/write response time for applications that require real-time data streaming or random access.[12]

[12] http://www.aosabook.org/en/hdfs.html

Secondary Name Node

The role of the secondary name node has caused considerable confusion for Hadoop users, since it has a misleading name. It is not a backup name node in the sense that it can take over the primary name node's function, but rather a checkpointing mechanism. During operation the name node maintains two on-disk data structures to represent the filesystem state: an image file and an edit log. The image file is a checkpoint of the filesystem metadata at a point in time, and the edit log is a transactional redo log of every filesystem metadata mutation since the image file was created. Incoming changes to the filesystem metadata (such as creating a new file) are written to the edit log2. When the name node starts, it reconstructs the current state by replaying the edit log. To ensure that the log doesn't grow without bound, at periodic intervals the edit log is rolled, and a new checkpoint is created by applying the old edit log to the image. This process is performed by the secondary name node daemon, often on a separate machine to the primary since creating a checkpoint has similar memory requirements to the name node itself. A side effect of the checkpointing mechanism is that the secondary holds an out-of-date copy of the primary's persistent state, which, in extremis, can be used to recover the filesystem's state.[13]

To understand the Safemode in HDFS, please read through this small section:

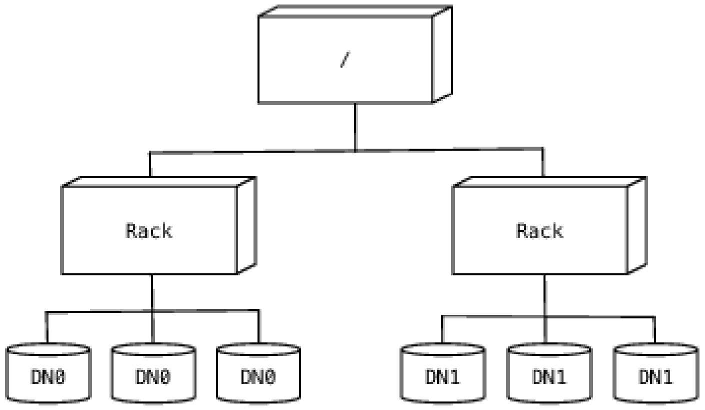
http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Safemode

Racks[14]

For a large cluster, it may not be practical to connect all nodes in a flat topology. A common practice is to spread the nodes across multiple racks. Nodes of a rack share a switch, and rack switches are connected by one or more core switches. Communication between two nodes in different racks has to go through multiple switches. In most cases, network bandwidth between nodes in the same rack is greater than network bandwidth between nodes in different racks. The below figure describes a cluster with two racks, each of which contains three nodes.

[13] http://blog.cloudera.com/wp-content/uploads/2010/03/HDFS_Reliability.pdf

[14] http://www.aosabook.org/en/hdfs.html



HDFS estimates the network bandwidth between two nodes by their distance. The distance from a node to its parent node is assumed to be one. A distance between two nodes can be calculated by summing the distances to their closest common ancestor. A shorter distance between two nodes means greater bandwidth they can use to transfer data.

HDFS allows an administrator to configure a script that returns a node's rack identification given a node's address. The NameNode is the central place that resolves the rack location of each DataNode. When a DataNode registers with the NameNode, the NameNode runs the configured script to decide which rack the node belongs to. If no such a script is configured, the NameNode assumes that all the nodes belong to a default single rack.

The placement of replicas is critical to HDFS data reliability and read/write performance. A good replica placement policy should improve data reliability, availability, and network bandwidth utilization. Currently HDFS provides a configurable block placement policy interface so that the users and researchers can experiment and test alternate policies that are optimal for their applications.

After all target nodes are selected for a write process, nodes are organized as a pipeline in the order of their proximity to the first replica. Data are pushed to nodes in this order. For reading, the NameNode first checks if the client's host is located in the cluster. If yes, block locations are returned to the client in the order of its closeness to the reader. The block is read from DataNodes in this preference order.

This policy reduces the inter-rack and inter-node write traffic and generally improves write performance. Because the chance of a rack failure is far less than that of a node failure, this policy does not impact data reliability and availability guarantees. In the usual case of three replicas, it can reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three.

Container for Blob storage

The Azure blob storage is the implementation for HDFS in the Azure environment. To use blobs, you first create an Azure storage account. As part of this, you specify an Azure data center that will store the objects you create using this account. Both the cluster and the storage account must be hosted in the same data center (Hive metastore SQL database and Oozie metastore SQL database must also be located in the same data center). Wherever it lives, each blob you create belongs to some container in your storage account. This container may be an existing Blob storage container created outside of HDInsight, or it may be a container that is created for an HDInsight cluster.[15]

You can follow this guide to create your Azure storage account:

http://azure.microsoft.com/en-us/documentation/articles/storage-create-storage-account/

When provisioning an HDInsight cluster from Azure Management Portal, there are two options: *quick create* and *custom create*. The quick create option requires the Azure Storage account created beforehand.

http://azure.microsoft.com/en-us/documentation/articles/hdinsight-administer-use-management-portal/#create

Using the quick create option, you can choose an existing storage account. The provision process creates a new container with the same name as the HDInsight cluster name. If a container with the same name already exists, - will be used. For example, myHDIcluster-1. This container is used as the default file system.

After following the steps in the guide linked above, you should have a running HDInsight cluster with an underlying HDFS file system.

You can find more details on the cluster provisioning in the course "*Big Data – 202 – Provisioning HDInsight*". For now it's important that you create a simple cluster for being able to do the first steps in HDFS.

Working with Remote Desktop

Once we've got the cluster, we may presumably want to do some big data processing. Here we'll use the example data available in the cluster. Later we'll use some of our own data.

[15] http://azure.microsoft.com/en-us/documentation/articles/hdinsight-use-blob-storage/#benefits

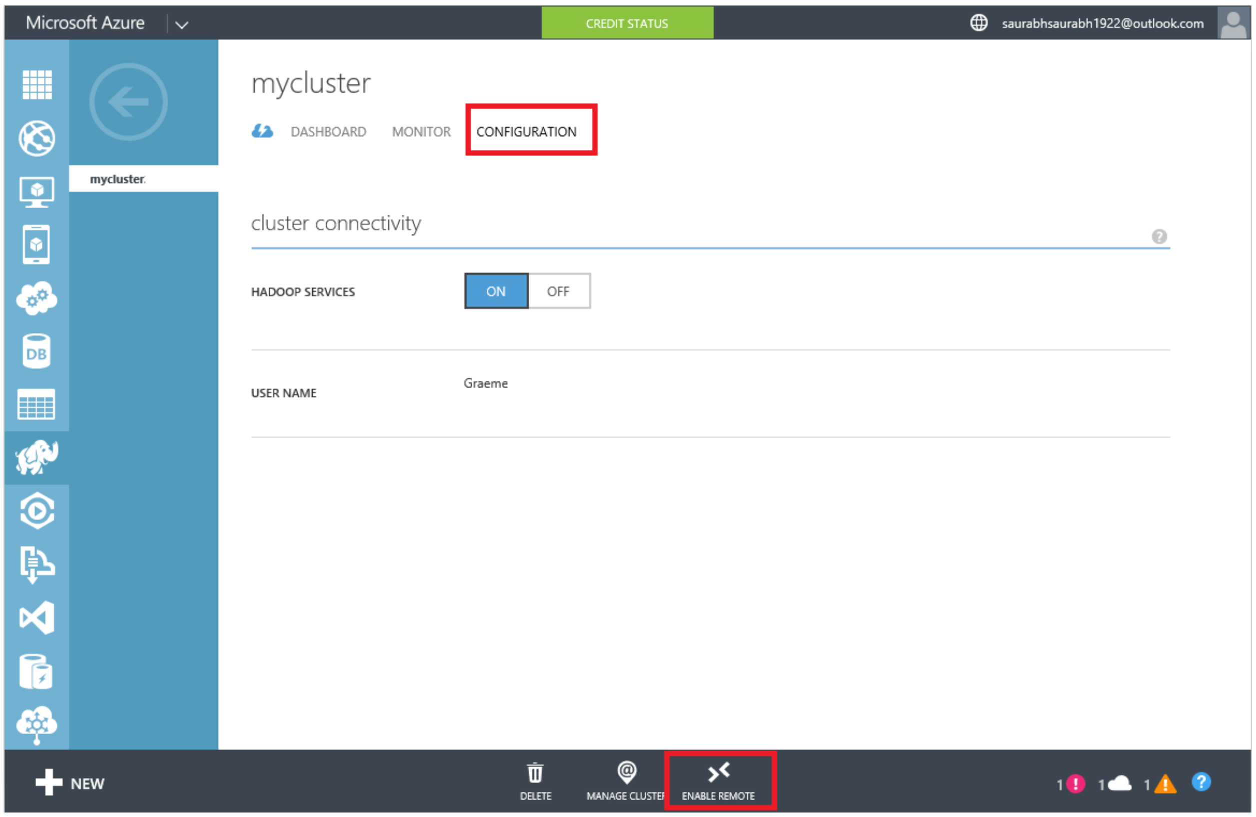
So, the first thing is actually to interoperate with that environment. We want to do something with the cluster. How do we connect to it? What do we use to go and initiate some work to navigate that file system? There are certain things we can do in the management portal using windows Azure.

But we probably want to do is actually connect to it and do something on it. The simplest thing that we can do is, because it is effectively a virtual machine running in Windows Azure, we can remote desktop to it. So we can go and open remote desktop and work with it. Now to do that, for security reasons, remote desktop access isn't enabled by default. And so what we have to do is enable that. Configure a user with permission to initiate a remote desktop connection. And actually, what we'll do is we specify an expiration date. So automatically, remote desktop would be disabled after a period of time. So this is all about securing access to HDInsight cluster.

So, let's have a look at how we do that.

Enable Remote

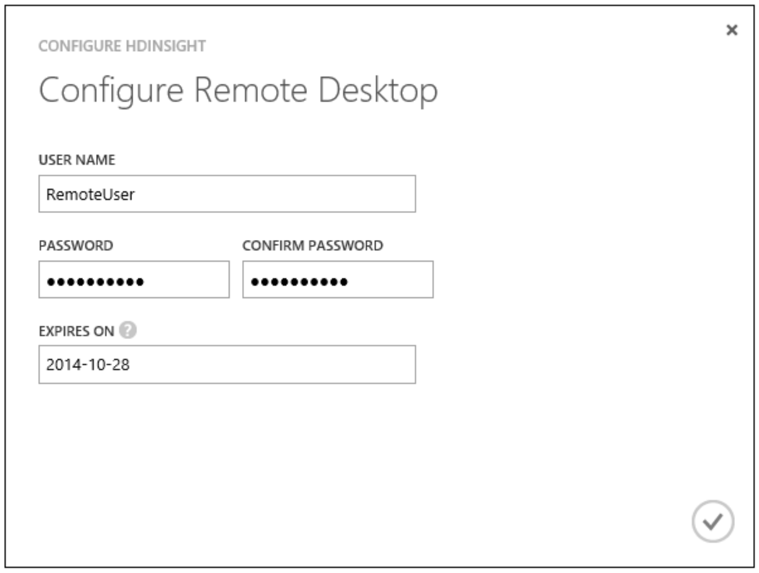
1. Under management portal, click on HDInsight in the left pane
2. Drill down into the cluster you want to enable remote desktop into
3. Click on CONFIGURATION on the top menu
4. Click on ENABLE REMOTE at the bottom



HADOOP SERVICES [✓ ON] OFF

USER NAME Graeme

5. Fill in the user name that would be used to access this remote connection
6. Provide an expiry date when the remote access would be disabled automatically.


CONFIGURE HDINSIGHT

Configure Remote Desktop

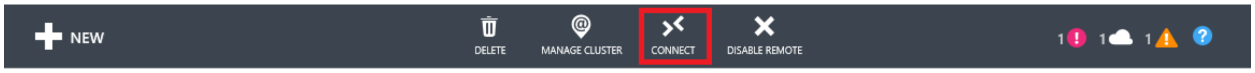
USER NAME: RemoteUser

PASSWORD: ●●●●●●●●●●

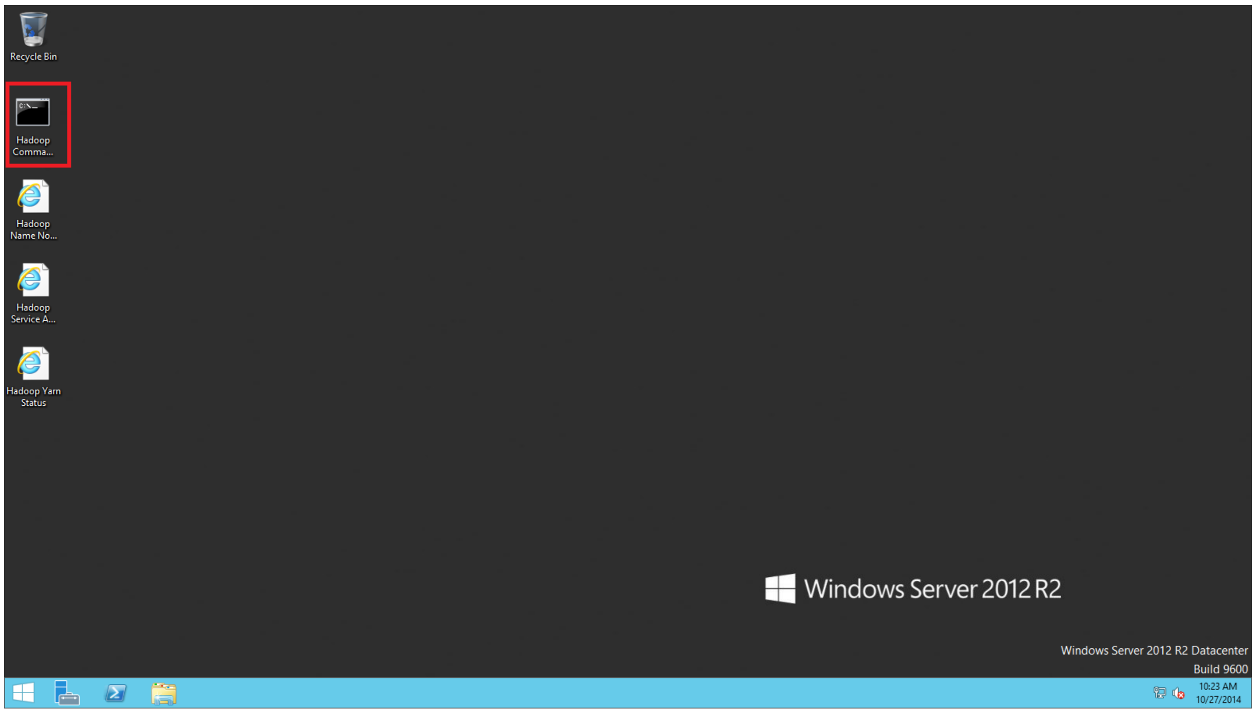
CONFIRM PASSWORD: ●●●●●●●●●●

EXPIRES ON: 2014-10-28

7. Click on CONNECT at the bottom



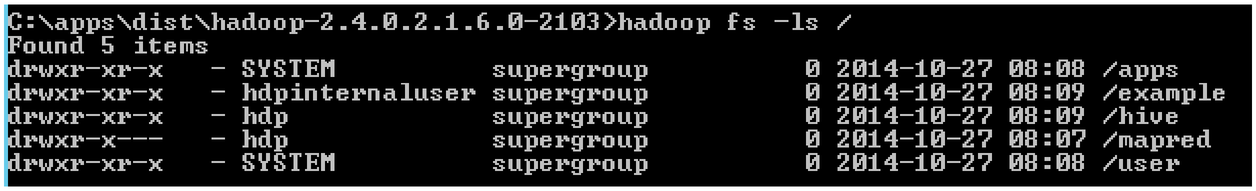
8. You'll get a prompt in the browser to open or save the .rdp file
9. Connect to the remote desktop using this .rdp file



10. Enter the credentials that you set up while enabling remote connection

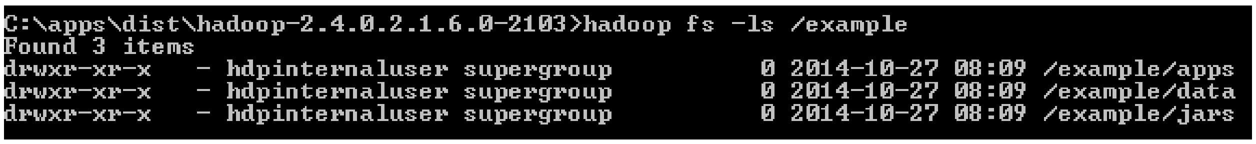
Browsing the HDFS file system

1. Launch the Hadoop Command Line from the remote desktop
2. Run the command to list the items in the file system:



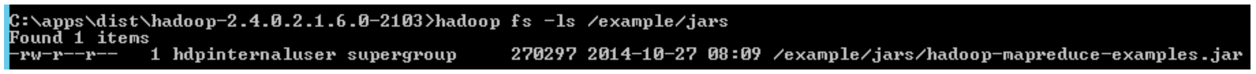
```
C:\apps\dist\hadoop-2.4.0.2.1.6.0-2103>hadoop fs -ls /
Found 5 items
drwxr-xr-x   - SYSTEM           supergroup          0 2014-10-27 08:08 /apps
drwxr-xr-x   - hdpinternaluser supergroup          0 2014-10-27 08:09 /example
drwxr-xr-x   - hdp              supergroup          0 2014-10-27 08:09 /hive
drwxr-x---   - hdp              supergroup          0 2014-10-27 08:07 /mapred
drwxr-xr-x   - SYSTEM           supergroup          0 2014-10-27 08:08 /user
```

3. Explore the example folder:



```
C:\apps\dist\hadoop-2.4.0.2.1.6.0-2103>hadoop fs -ls /example
Found 3 items
drwxr-xr-x   - hdpinternaluser supergroup          0 2014-10-27 08:09 /example/apps
drwxr-xr-x   - hdpinternaluser supergroup          0 2014-10-27 08:09 /example/data
drwxr-xr-x   - hdpinternaluser supergroup          0 2014-10-27 08:09 /example/jars
```

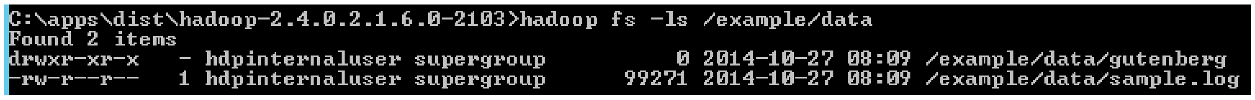
4. Let’s look at jars, that sounds interesting. And we’ll talk about what jars are in just a minute.



```
C:\apps\dist\hadoop-2.4.0.2.1.6.0-2103>hadoop fs -ls /example/jars
Found 1 items
-rw-r--r--   1 hdpinternaluser supergroup     270297 2014-10-27 08:09 /example/jars/hadoop-mapreduce-examples.jar
```

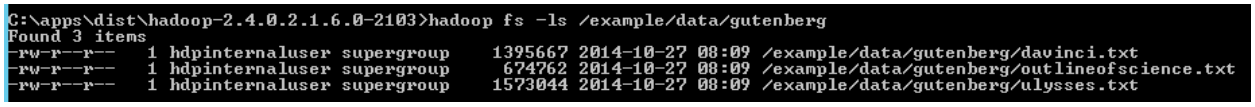
It is a jar for Java (Java is based on coffee, it is a coffee jar). It is a jar where you can put your Java code into. So, if you are a Java programmer, you know what it is, this is old hat to you. And if you are from the .NET world, and you are not involved in Java programming, it might surprise you to learn that when you compile a Java component, you can file it into a jar. So, this jar is up here and it contains some Java classes, and those classes contains methods, and those methods in this case are going to be map methods and reduce methods. So, what we've got here are a bunch of examples compiled to a jar, where we have classes that do map-reduce jobs.

5. Let's look at some data that we are going to work on:



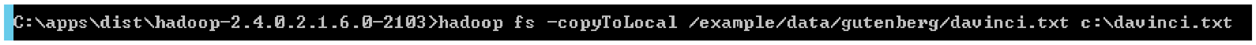
```
C:\apps\dist\hadoop-2.4.0.2.1.6.0-2103>hadoop fs -ls /example/data
Found 2 items
drwxr-xr-x   - hdpinternaluser supergroup          0 2014-10-27 08:09 /example/data/gutenberg
-rw-r--r--   1 hdpinternaluser supergroup      99271 2014-10-27 08:09 /example/data/sample.log
```

6. Some big data is available inside this gutenberg folder



```
C:\apps\dist\hadoop-2.4.0.2.1.6.0-2103>hadoop fs -ls /example/data/gutenberg
Found 3 items
-rw-r--r--   1 hdpinternaluser supergroup    1395667 2014-10-27 08:09 /example/data/gutenberg/davinci.txt
-rw-r--r--   1 hdpinternaluser supergroup     674762 2014-10-27 08:09 /example/data/gutenberg/outlineofscience.txt
-rw-r--r--   1 hdpinternaluser supergroup    1573044 2014-10-27 08:09 /example/data/gutenberg/ulysses.txt
```

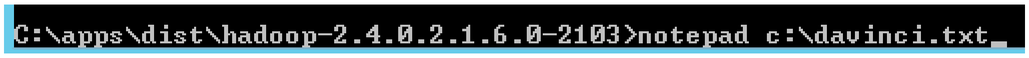
7. Let's view what's there in one of these files.



```
C:\apps\dist\hadoop-2.4.0.2.1.6.0-2103>hadoop fs -copyToLocal /example/data/gutenberg/davinci.txt c:\davinci.txt
```

The reason we're doing this, is partly because the file is too big to view online using the cat command, which is the possible UNIX command for doing this type of thing, and partly because we want to emphasise the point that what we are looking at is not the file system on the cluster service itself. It has its own local file system, just as in any windows server. What we are looking here is the HDFS file system. It is stored in the windows Azure blob store. And it is shared across all of the nodes in the cluster. So, that's a different thing in the local file system.

8. Take a look at the file now



```
C:\apps\dist\hadoop-2.4.0.2.1.6.0-2103>notepad c:\davinci.txt_
```

This is effectively just free text. There is no structure to this. It is just, text that's been taken from a book and dropped unto here. So this is, a better example is, tweet on twitter, social media comment, that type of thing isn't something we can very easily divide it into rows and columns. Or any sort of structured data here.

So, that's what we have in the HDFS file system that we've got to work with.

You can find a detailed introduction in the different HDFS shell commands here:

http://hadoop.apache.org/docs/r0.18.3/hdfs_shell.html

Some short examples in the most important shell commands can be found below[16]:

Create directories

```
hadoop fs -mkdir <paths>

hadoop fs -mkdir /user/subfolder/dir1 /user/ subfolder /dir2
```

List the directory content

```
hadoop fs -ls <args>

hadoop fs -ls /user/subfolder
```

Upload a file in HDFS

```
hadoop fs -put <localsrc> ... <HDFS_dest_Path>

hadoop fs -put /home/subfolder/Samplefile.txt  /user/subfolder/dir3/
```

Download a file from HDFS

```
hadoop fs -get <hdfs_src> <localdst>

hadoop fs -get /user/subfolder/dir3/Samplefile.txt /home/
```

See contents of a file

```
hadoop fs -cat <path[filename]>

hadoop fs -cat /user/subfolder/dir1/abc.txt
```

Copy a file from source to destination

```
hadoop fs -cp <source> <dest>
```

[16] http://java.dzone.com/articles/top-10-hadoop-shell-commands

```
hadoop fs -cp /user/subfolder/dir1/abc.txt /user/subfolder/dir2
```

Copy a file from/To Local file system to HDFS

```
hadoop fs -copyFromLocal <localsrc> URI

hadoop fs -copyFromLocal /home/subfolder/abc.txt  /user/subfolder/abc.txt
```

copyToLocal

```
hadoop fs -copyToLocal [-ignorecrc] [-crc] URI <localdst>
```

Move file from source to destination.

```
hadoop fs -mv <src> <dest>

hadoop fs -mv /user/subfolder/dir1/abc.txt /user/subfolder/dir2
```

Remove a file or directory in HDFS

```
hadoop fs -rm <arg>

hadoop fs -rm /user/subfolder/dir1/abc.txt
```

Recursive version of delete

```
hadoop fs -rmr <arg>

hadoop fs -rmr /user/subfolder/
```

HDFS architecture

As described above HDFS is stored in a blob container in Windows Azure Storage. We used the kind of classic UNIX paths to navigate in the previous chapter. So, we started with the root with just a "/" and then folder and slashes. We are used to using a windows environment, that "/" is equivalent to "c:\" as the root of the c drive.

The "/" is root of the entire storage system. So, we were able to navigate that using various UNIX commands. You saw –ls. There is –lsr, which is recursive, so you go and look what is in the subfolders and so on. And there is –cp and –copyToLocal, we saw an example of that. And the usual things you would expect. There is much more than what we saw. There is huge wealth of these commands being used.

The one thing I do want to point out is that we did everything from the command prompt that is specific for working with Hadoop. We would be using effectively native UNIX type commands.

If you are working with your HDFS container (your Hadoop distributed file system) remotely, then what you are actually connecting to is your windows azure blob storage account. And the syntax for doing that is to use that wasb:// prefix. So, wasb stands for windows azure storage blob.

```
wasb://data@myaccount.blob.core.windows.net/logs/file.txt
```

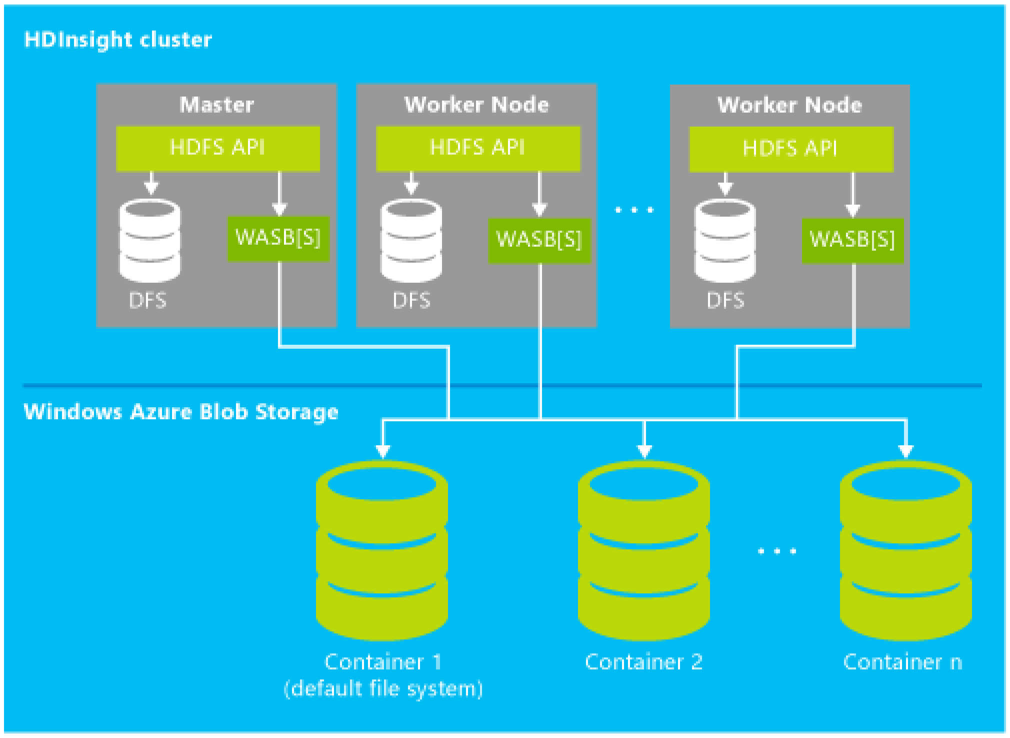
And then we've got the "data" in the sense that it is the name of the container that we're connecting to. And then we've got "myaccount" name, "blob.core.windows.net" that is common across all windows azure blob storage, and then we've got the path that we would be navigating through. So, that slash we've got after that is how we are navigating through that file system. So that's the wasb://<container>@<account>/<navigation path>

Another important takeaway here is that these HDFS blob containers can be retained even when the HDInsight cluster is deleted. This saves cost as we are paying for only what we use.

Azure blob storage

Azure HDInsight supports both Hadoop Distributed Files System (HDFS) and Azure Blob storage for storing data. Blob storage is a robust, general purpose Azure storage solution. Blob storage provides a full-featured HDFS interface for a seamless experience by enabling the full set of components in the Hadoop ecosystem to operate (by default) directly on the data. Blob storage is not just a low-cost solution; storing data in Blob storage enables the HDInsight clusters used for computation to be safely deleted without losing user data.

HDInsight storage architecture



HDInsight storage architecture

The diagram provides an abstract view of the HDInsight storage architecture:

HDInsight provides access to the distributed file system that is locally attached to the compute nodes. This file system can be accessed using the fully qualified URI. For example:

hdfs://<namenodehost>/<path>

In addition, HDInsight provides the ability to access data stored in Blob storage. The syntax to access Blob storage is:

wasb[s]://<containername>@<accountname>.blob.core.windows.net/<path>

Hadoop supports a notion of default file system. The default file system implies a default scheme and authority; it can also be used to resolve relative paths. During the HDInsight provision process, an Azure Storage account and a specific Blob storage container from that account is designated as the default file system.

In addition to this storage account, you can add additional storage accounts from either the same Azure subscription or different Azure subscriptions during the provision process.

- **Containers in the storage accounts that are connected to a cluster:** Because the account name and key are stored in the *core-site.xml*, you have full access to the blobs in those containers.
- **Public containers or public blobs in the storage accounts that are NOT connected to a cluster:** You have read-only permission to the blobs in the containers.
    - Public container allows you to get a list of all blobs available in that container and get container metadata. Public blob allows you to access the blobs only if you know the exact URL.
- **Private containers in the storage accounts that are NOT connected to a cluster:** You cannot access the blobs in the containers unless you define the storage account when you submit the WebHCat jobs.

The storage accounts defined in the provision process and their keys are stored in %HADOOP_HOME%/conf/core-site.xml. The default behavior of HDInsight is to use the storage accounts defined in the core-site.xml file. It is not recommended to edit the core-site.xml file because the cluster headnode(master) may be re-imaged or migrated at any time, and any changes to those files will be lost.

Multiple WebHCat jobs, including Hive, MapReduce, Hadoop streaming and Pig, can carry a description of storage accounts and metadata with them (It currently works for Pig with storage accounts but not for metadata.)

Blob storage containers store data as key/value pairs, and there is no directory hierarchy. However the "/" character can be used within the key name to make it appear as if a file is stored within a directory structure. For example, a blob's key may be *input/log1.txt*. No actual *input* directory exists, but due to the presence of the "/" character in the key name, it has the appearance of a file path.

Benefits of Azure Blob storage

The implied performance cost of not having compute and storage co-located is mitigated by the way the compute clusters are provisioned close to the storage account resources inside the Azure data center, where the high speed network makes it very efficient for the compute nodes to access the data inside Blob storage.

There are several benefits associated with storing the data in Blob storage instead of HDFS:

- **Data reuse and sharing:** The data in HDFS is located inside the compute cluster. Only the applications that have access to the compute cluster can use the data using HDFS API. The data in Blob storage can be accessed either through the HDFS APIs or through the Blob Storage REST APIs. Thus, a larger set of applications (including other HDInsight clusters) and tools can be used to produce and consume the data.

- Data archiving: Storing data in Blob storage enables the HDInsight clusters used for computation to be safely deleted without losing user data.
- Data storage cost: Storing data in DFS for the long term is more costly than storing the data in Blob storage, since the cost of a compute cluster is higher than the cost of a Blob storage container. In addition, because the data does not have to be reloaded for every compute cluster generation, you are saving data loading costs as well.
- Elastic scale-out: While HDFS provides you with a scaled-out file system, the scale is determined by the number of nodes that you provision for your cluster. Changing the scale can become a more complicated process than relying on the Blob storage's elastic scaling capabilities that you get automatically.
- Geo-replication: Your Blob storage containers can be geo-replicated through the Azure Portal. While this gives you geographic recovery and data redundancy, a fail-over to the geo-replicated location will severely impact your performance and may incur additional costs. So Microsoft’s recommendation is to choose the geo-replication wisely and only if the value of the data is worth the additional cost.

Certain MapReduce jobs and packages may create intermediate results that you don't really want to store in the Blob storage container. In that case, you can still elect to store the data in the local HDFS. In fact, HDInsight uses DFS for several of these intermediate results in Hive jobs and other processes.