HDFS

Jim Harner

1/12/2021

4.3 HDFS

When data sets exceed the storage capacity of a single physical machine, we can spread them across a number of machines. Filesystems that manage the storage across a network of machines are called *distributed filesystems*. Since they are network based, the complications of network programming arise, e.g., experiencing node failures without data loss. Thus, distributed filesystems are more complex than regular disk filesystems.

HDFS is Hadoop's main filesystem, but Hadoop has a general-purpose filesystem abstraction. Thus, Hadoop integrates with other storage systems, e.g., the local filesystem and Amazon S3.

4.3.1 HDFS Design

HDFS is a filesystem designed for storing very large files with streaming data access, running on clusters of commodity hardware.

- Very large files: Many data lakes are in the gigabytes, terabytes, or even petabytes in size.

- Streaming data access: HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, and then various analyses are performed on that dataset over time.

- Commodity hardware: Hadoop doesn't require expensive, highly reliable hardware. It's designed to run on clusters of commodity hardware for which the chance of node failure across the cluster is high, at least for large clusters.

Files in HDFS are broken into block-sized chunks (128 MB by default), which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage. HDFS blocks are large to minimize the cost of seeks.

Having a block abstraction for a distributed filesystem brings several benefits over a file abstraction:

- a file can be larger than any single disk in the network;
- the storage is easy to manage, e.g., relating to disk failure;

- replication allows fault tolerance and availability.

HDFS's `fsck` (file system check) command understands blocks:

```
hdfs fsck / -files -blocks
```

You can run this in `bash`, but the output is large.

An HDFS cluster has two types of nodes operating in a master-worker pattern:
* a namenode (the master);
* a number of datanodes (workers).

The namenode manages the filesystem namespace. Datanodes store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that

they are storing. A client accesses the filesystem on behalf of the user by communicating with the namenode and datanodes.

4.3.2 HDFS Input/Output

As a user you must interact with both the *local filesystem* and *hdfs*. Locally, you use standard Linux commands, e.g., `cd`, `rm`, etc. Many regular filesystems commands have similar functionality in hdfs, but the syntax is different and certain limitations exist since hdfs is *stateless*, e.g., there is no `cd` command.

You can experiment in the shell with the following scripts to show the `hdfs` commands. (Note: the `hdfs` commands executed in the section are run for illustrative purposes, i.e., they are run in the `rstudio` container—not the `hadoop` container.)

```
# show the hdfs commands
hdfs | head
```

```
## Usage: hdfs [--config confdir] [--loglevel loglevel] COMMAND
##        where COMMAND is one of:
##   dfs                  run a filesystem command on the file systems supported in Hadoop.
##   classpath            prints the classpath
##   namenode -format     format the DFS filesystem
##   secondarynamenode    run the DFS secondary namenode
##   namenode             run the DFS namenode
##   journalnode          run the DFS journalnode
##   zkfc                 run the ZK Failover Controller daemon
##   datanode             run a DFS datanode
```

The `hdfs` command of interest is `dfs` which runs a filesystem command on the distributed file systems supported in Hadoop.

`hdfs dfs` shows the `dfs` commands and their options. Alternately, you can use `hadoop fs` to issue commands.

```
# show the options for the dfs command
hdfs dfs | head
# hadoop fs
# echo $? show the exit code in the last command, which should be 0
echo $?
```

```
## Usage: hadoop fs [generic options]
##  [-appendToFile <localsrc> ... <dst>]
##  [-cat [-ignoreCrc] <src> ...]
##  [-checksum <src> ...]
##  [-chgrp [-R] GROUP PATH...]
##  [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
##  [-chown [-R] [OWNER][:[GROUP]] PATH...]
##  [-copyFromLocal [-f] [-p] [-l] [-d] <localsrc> ... <dst>]
##  [-copyToLocal [-f] [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
##  [-count [-q] [-h] [-v] [-t [<storage type>]] [-u] [-x] <path> ...]
##  [-cp [-f] [-p | -p[topax]] [-d] <src> ... <dst>]
##  [-createSnapshot <snapshotDir> [<snapshotName>]]
##  [-deleteSnapshot <snapshotDir> <snapshotName>]
##  [-df [-h] [<path> ...]]
##  [-du [-s] [-h] [-x] <path> ...]
##  [-expunge]
##  [-find <path> ... <expression> ...]
##  [-get [-f] [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
##  [-getfacl [-R] <path>]
```

```
##  [-getfattr [-R] {-n name | -d} [-e en] <path>]
##  [-getmerge [-nl] [-skip-empty-file] <src> <localdst>]
##  [-help [cmd ...]]
##  [-ls [-C] [-d] [-h] [-q] [-R] [-t] [-S] [-r] [-u] [<path> ...]]
##  [-mkdir [-p] <path> ...]
##  [-moveFromLocal <localsrc> ... <dst>]
##  [-moveToLocal <src> <localdst>]
##  [-mv <src> ... <dst>]
##  [-put [-f] [-p] [-l] [-d] <localsrc> ... <dst>]
##  [-renameSnapshot <snapshotDir> <oldName> <newName>]
##  [-rm [-f] [-r|-R] [-skipTrash] [-safely] <src> ...]
##  [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
##  [-setfacl [-R] [{-b|-k} {-m|-x <acl_spec>} <path>]|[--set <acl_spec> <path>]]
##  [-setfattr {-n name [-v value] | -x name} <path>]
##  [-setrep [-R] [-w] <rep> <path> ...]
##  [-stat [format] <path> ...]
##  [-tail [-f] <file>]
##  [-test -[defsz] <path>]
##  [-text [-ignoreCrc] <src> ...]
##  [-touchz <path> ...]
##  [-truncate [-w] <length> <path> ...]
##  [-usage [cmd ...]]
##
## Generic options supported are:
## -conf <configuration file>        specify an application configuration file
## -D <property=value>               define a value for a given property
## -fs <file:///|hdfs://namenode:port> specify default filesystem URL to use, overrides 'fs.defaultFS' p
## -jt <local|resourcemanager:port>  specify a ResourceManager
## -files <file1,...>                specify a comma-separated list of files to be copied to the map re
## -libjars <jar1,...>               specify a comma-separated list of jar files to be included in the
## -archives <archive1,...>          specify a comma-separated list of archives to be unarchived on the
##
## The general command line syntax is:
## command [genericOptions] [commandOptions]
##
## 0
```

Note that the format of the commands involving the distributed file system is `hdfs dfs <options>`.

We can list the files and directories with a `-ls` option, make a directory with the `-mkdir` option, etc. These can be run as bash commands.

```
hdfs dfs -mkdir temp
hdfs dfs -ls
```

```
## Found 1 items
## drwxr-xr-x   - rstudio rstudio          0 2021-01-12 21:51 temp
```

At this point, the directory has no content. Copy `cdat.csv` form the local filesystem into hdfs and list.

```
# set the local working directory to s2_hdfs
cd /home/rstudio/rspark-tutorial/m4_hadoop/s3_hdfs
hdfs dfs -copyFromLocal cdat.csv temp/
hdfs dfs -ls temp/
```

```
## Found 1 items
## -rw-r--r--   3 rstudio rstudio      34809 2021-01-12 21:51 temp/cdat.csv
```

Disk usage by itself can be found using the `du` option.

```
hdfs dfs -du
```

```
## 34809  temp
```

Remove the file and directory

```
hdfs dfs -rm -f temp/cdat.csv
hdfs dfs -rmdir temp
```

```
## Deleted temp/cdat.csv
```

These commands illustrate how data can be loaded from the local filesystem into hdfs. Below we illustrate how to load data into hdfs from within R. However, in a production system data would typically be loaded into hdfs using Sqoop as an interface to a database or Flume for real-time data. Kafka provides a more modern interface to both static and real-time data.

HDFS has a permissions model for files and directories that is much like the POSIX model.

The `rhdfs` package in RHadoop provides basic connectivity to the Hadoop Distributed File System. R programmers can browse, read, write, and modify files stored in HDFS from within R. The results are best seen by stepping through the code line-by-line using: `Run -> Run Selected Line(s)`.

These R statements are essentially identical to the bash commands above. First, load the `rhdfs` library and initialize hdfs.

```
library(rhdfs)
```

```
## Loading required package: rJava
```

```
##
## HADOOP_CMD=/opt/hadoop/bin/hadoop
```

```
##
## Be sure to run hdfs.init()
```

```
hdfs.init()
```

Make a new directory and list the files for the user `rstudio`.

```
hdfs.mkdir("temp")
```

```
## [1] TRUE
```

```
hdfs.ls("/user/rstudio")
```

```
##   permission   owner   group size          modtime               file
## 1 drwxr-xr-x rstudio rstudio    0 2021-01-12 21:51 /user/rstudio/temp
```

Import a file and list:

```
hdfs.put("cdat.csv", "temp")
```

```
## [1] TRUE
```

```
hdfs.ls("temp")
```

```
##   permission   owner   group  size          modtime                         file
## 1 -rw-r--r-- rstudio rstudio 34809 2021-01-12 21:51 /user/rstudio/temp/cdat.csv
```

Note: this does not work in interactive mode. Notice `hdfs.ls` has relative addressing.

Remove the file and directory:

```
hdfs.rm("temp/cdat.csv")
```

```
## [1] TRUE
```

```
hdfs.rm("temp")
```

```
## [1] TRUE
```

4.3.3 YARN

YARN (Yet Another Resource Negotiator) is Hadoop's cluster resource management system. YARN provides APIs for requesting and working with cluster resources.

YARN provides its core services via two types of long-running daemons:

- a resource manager (one per cluster) to manage the use of resources across the cluster, and
- node managers running on all the nodes in the cluster to launch and monitor containers.

See the batch diagram in Section 4.1. A container executes an application-specific process with a constrained set of resources (memory, CPU, etc.).

YARN has a flexible model for making *resource requests*. A request for a set of containers can express the amount of computer resources required for each container (memory and CPU), as well as *locality constraints* for the containers in that request. *Locality* is critical in ensuring that distributed data processing algorithms use the cluster bandwidth efficiently. The idea is to bring the algorithm to the data rather than moving data.

YARN allocates resources to applications according to some defined policy. YARN has three schedulers:

- FIFO: the order of submission (first in, first out);
- Capacity; a separate dedicated queue allows the small job to start as soon as it is submitted;
- Fair Scheduler: dynamically balance resources among all running jobs.