hdfs3 Documentation

Release 0.1.2

Continuum Analytics

February 03, 2017

Contents

Use HDFS natively from Python.

The Hadoop File System (HDFS) is a widely deployed, distributed, data-local file system written in Java. This file system backs most clusters running Hadoop and Spark.

Pivotal produced libhdfs3, an alternative native C/C++ HDFS client that interacts with HDFS without the JVM, exposing first class support to non-JVM languages like Python.

This library, `hdfs3`, is a lightweight Python wrapper around the C/C++ `libhdfs3` library. It provides both direct access to libhdfs3 from Python as well as a typical Pythonic interface.

```
>>> from hdfs3 import HDFileSystem
>>> hdfs = HDFileSystem(host='localhost', port=8020)
>>> hdfs.ls('/user/data')
>>> hdfs.put('local-file.txt', '/user/data/remote-file.txt')
>>> hdfs.cp('/user/data/file.txt', '/user2/data')
```

HDFS3 files comply with the Python File interface. This enables interactions with the broader ecosystem of PyData projects.

```
>>> with hdfs.open('/user/data/file.txt') as f:
...     data = f.read(1000000)

>>> with hdfs.open('/user/data/file.csv.gz') as f:
...     df = pandas.read_csv(f, compression='gzip', nrows=1000)
```

CHAPTER 1

Motivation

We choose to use an alternative C/C++/Python HDFS client rather than the default JVM client for the following reasons:

- **Convenience:** Interactions between Java libraries and Native (C/C++/Python) libraries can be cumbersome. Using a native library from Python smoothes over the experience in development, maintenance, and debugging.
- **Performance:** Native libraries like `libhdfs3` do not suffer the long JVM startup times, improving interaction.

CHAPTER 2

Related Work

- libhdfs3: The underlying C++ library
- snakebite: Another Python HDFS library using Protobufs
- Dask: Parent project, a parallel computing library in Python
- Dask.distributed: Distributed computing in Python

2.1 Installation

2.1.1 Conda

Both the `hdfs3` Python library and the compiled `libhdfs3` library (and its dependencies) are available from the conda-forge repository using conda:

```
$ conda install hdfs3 -c conda-forge
```

Note that conda packages are only available for the `linux-64` platform.

2.1.2 PyPI and apt-get

Alternatively you can install the `libhdfs3.so` library using a system installer like `apt-get`:

```
echo "deb https://dl.bintray.com/wangzw/deb trusty contrib" | sudo tee /etc/apt/sources.list.d/bintra
sudo apt-get install -y apt-transport-https
sudo apt-get update
sudo apt-get install libhdfs3 libhdfs3-dev
```

And install the Python wrapper library, `hdfs3`, with `pip`:

```
pip install hdfs3
```

2.1.3 Build from Source

See the libhdfs3 installation instructions to install the compiled library.

You can download the `hdfs3` Python wrapper library from github and install normally:

```
git clone git@github.com:dask/hdfs3
cd hdfs3
python setup.py install
```

2.2 Quickstart

Import `hdfs3` and connect to an HDFS cluster:

```
>>> from hdfs3 import HDFileSystem
>>> hdfs = HDFileSystem(host='localhost', port=8020)
```

Write data to file:

```
>>> with hdfs.open('/tmp/myfile.txt', 'wb') as f:
...     f.write(b'Hello, world!')
```

Read data back from file:

```
>>> with hdfs.open('/tmp/myfile.txt') as f:
...     print(f.read())
```

Interact with files on HDFS:

```
>>> hdfs.ls('/tmp')

>>> hdfs.put('local-file.txt', '/tmp/remote-file.txt')

>>> hdfs.cp('/tmp/remote-file.txt', '/tmp/copied-file.txt')
```

2.3 Examples

2.3.1 Word count

Setup

In this example, we'll use the `hdfs3` library to count the number of words in text files (Enron email dataset, 6.4 GB) stored in HDFS.

Copy the text data from Amazon S3 into HDFS on the cluster:

```
$ hadoop distcp s3n://AWS_SECRET_ID:AWS_SECRET_KEY@blaze-data/enron-email hdfs:///tmp/enron
```

where `AWS_SECRET_ID` and `AWS_SECRET_KEY` are valid AWS credentials.

Code example

Import `hdfs3` and other standard libraries used in this example:

```
>>> import hdfs3
>>> from collections import defaultdict, Counter
```

Initalize a connection to HDFS, replacing `NAMENODE_HOSTNAME` and `NAMENODE_PORT` with the hostname and port (default: 8020) of the HDFS namenode.

```
>>> hdfs = hdfs3.HDFileSystem('NAMENODE_HOSTNAME', port=NAMENODE_PORT)
```

Generate a list of filenames from the text data in HDFS:

```
>>> filenames = hdfs.glob('/tmp/enron/*/*')
>>> print(filenames[:5])

['/tmp/enron/edrm-enron-v2_nemec-g_xml.zip/merged.txt',
'/tmp/enron/edrm-enron-v2_ring-r_xml.zip/merged.txt',
'/tmp/enron/edrm-enron-v2_bailey-s_xml.zip/merged.txt',
'/tmp/enron/edrm-enron-v2_fischer-m_xml.zip/merged.txt',
'/tmp/enron/edrm-enron-v2_geaccone-t_xml.zip/merged.txt']
```

Print the first 1024 bytes of the first text file:

```
>>> print(hdfs.head(filenames[0]))

b'Date: Wed, 29 Nov 2000 09:33:00 -0800 (PST)\r\nFrom: Xochitl-Alexis Velasc
o\r\nTo: Mark Knippa, Mike D Smith, Gerald Nemec, Dave S Laipple, Bo Barnwel
l\r\nCc: Melissa Jones, Iris Waser, Pat Radford, Bonnie Shumaker\r\nSubject:
 Finalize ECS/EES Master Agreement\r\nX-SDOC: 161476\r\nX-ZLID: zl-edrm-enro
n-v2-nemec-g-2802.eml\r\n\r\nPlease plan to attend a meeting to finalize the
 ECS/EES  Master Agreement \r\ntomorrow 11/30/00 at 1:30 pm CST.\r\n\r\nI wi
ll email everyone tomorrow with location.\r\n\r\nDave-I will also email you
the call in number tomorrow.\r\n\r\nThanks\r\nXochitl\r\n\r\n**********\r\n
EDRM Enron Email Data Set has been produced in EML, PST and NSF format by ZL
 Technologies, Inc. This Data Set is licensed under a Creative Commons Attri
bution 3.0 United States License <http://creativecommons.org/licenses/by/3.0
/us/> . To provide attribution, please cite to "ZL Technologies, Inc. (http:
//www.zlti.com)."\r\n**********\r\nDate: Wed, 29 Nov 2000 09:40:00 -0800 (P
ST)\r\nFrom: Jill T Zivley\r\nTo: Robert Cook, Robert Crockett, John Handley
, Shawna'
```

Create a function to count words in each file:

```
>>> def count_words(file):
...     word_counts = defaultdict(int)
...     for line in file:
...         for word in line.split():
...             word_counts[word] += 1
...     return word_counts

>>> print(count_words(['apple banana apple', 'apple orange']))

defaultdict(int, {'apple': 3, 'banana': 1, 'orange': 1})
```

Count the number of words in the first text file:

```
>>> with hdfs.open(filenames[0]) as f:
...     counts = count_words(f)

>>> print(sorted(counts.items(), key=lambda k_v: k_v[1], reverse=True)[:10])

[(b'the', 1065320),
 (b'of', 657220),
 (b'to', 569076),
 (b'and', 545821),
 (b'or', 375132),
 (b'in', 306271),
```

```
 (b'shall', 255680),
 (b'be', 210976),
 (b'any', 206962),
 (b'by', 194780)]
```

Count the number of words in all of the text files. This operation required about 10 minutes to run on a single machine with 4 cores and 16 GB RAM:

```
>>> all_counts = Counter()
>>> for fn in filenames:
...     with hdfs.open(fn) as f:
...         counts = count_words(f)
...         all_counts.update(counts)
```

Print the total number of words and the words with the highest frequency from all of the text files:

```
>>> print(len(all_counts))

8797842

>>> print(sorted(all_counts.items(), key=lambda k_v: k_v[1], reverse=True)[:10])

[(b'0', 67218380),
 (b'the', 19586868),
 (b'-', 14123768),
 (b'to', 11893464),
 (b'N/A', 11814665),
 (b'of', 11724827),
 (b'and', 10253753),
 (b'in', 6684937),
 (b'a', 5470371),
 (b'or', 5227805)]
```

The complete Python script for this example is shown below:

```
# word-count.py

import hdfs3
from collections import defaultdict, Counter

hdfs = hdfs3.HDFileSystem('NAMENODE_HOSTNAME', port=NAMENODE_PORT)

filenames = hdfs.glob('/tmp/enron/*/*')
print(filenames[:5])
print(hdfs.head(filenames[0]))


def count_words(file):
    word_counts = defaultdict(int)
    for line in file:
        for word in line.split():
            word_counts[word] += 1
    return word_counts

print(count_words(['apple banana apple', 'apple orange']))

with hdfs.open(filenames[0]) as f:
    counts = count_words(f)
```

```
print(sorted(counts.items(), key=lambda k_v: k_v[1], reverse=True)[:10])

all_counts = Counter()

for fn in filenames:
    with hdfs.open(fn) as f:
        counts = count_words(f)
        all_counts.update(counts)

print(len(all_counts))
print(sorted(all_counts.items(), key=lambda k_v: k_v[1], reverse=True)[:10])
```

2.4 HDFS Configuration

2.4.1 Defaults

This library tries to find `core-site.xml` and `hdfs-site.xml` in typical locations and reads default configuration parameters from there. They may also be specified manually when constructing the `HDFileSystem` object.

2.4.2 Short-circuit reads in HDFS

Typically in HDFS, all data reads go through the datanode. Alternatively, a process that runs on the same node as the data can bypass or *short-circuit* the communication path through the datanode and instead read directly from a file.

HDFS and `hdfs3` can be configured for short-circuit reads using the following two steps:

- Set the `LIBHDFS3_CONF` environment variable to the location of the `hdfs-site.xml` configuration file (e.g., `export LIBHDFS3_CONF=/etc/hadoop/conf/hdfs-site.xml`).
- Configure the appropriate settings in `hdfs-site.xml` on all of the HDFS nodes:

```
<configuration>
  <property>
    <name>dfs.client.read.shortcircuit</name>
    <value>true</value>
  </property>

  <property>
    <name>dfs.domain.socket.path</name>
    <value>/var/lib/hadoop-hdfs/dn_socket</value>
  </property>
</configuration>
```

The above configuration changes should allow for short-circuit reads. If you continue to receive warnings to `retry the same node but disable read shortcircuit feature`, check the above settings. Note that the HDFS reads should still function despite the warning, but performance might be impacted.

For more information about configuring short-circuit reads, refer to the HDFS Short-Circuit Local Reads documentation.

2.4.3 High-availability mode

Although HDFS is resilient to failure of data-nodes, the name-node is a single repository of metadata for the system, and so a single point of failure. High-availability (HA) involves configuring fall-back name-nodes which can take over

in the event of failure. A good description han be found **'here'**_.

In the case of libhdfs3, the library used by hdfs3, the configuration required for HA can be passed to the client directly in python code, or included in configuration files. The parameters required are detailed in the **'libhdfs3 documentation'**_. The environment variable `LIBHDFS3_CONF` can be used to point the client to the appropriate preference file.

In python code, this could look like the following:

```
host = "nameservice1"
conf = {"dfs.nameservices": "nameservice1",
        "dfs.ha.namenodes.nameservice1": "namenode113,namenode188",
        "dfs.namenode.rpc-address.nameservice1.namenode113": "hostname_of_server1:8020",
        "dfs.namenode.rpc-address.nameservice1.namenode188": "hostname_of_server2:8020",
        "dfs.namenode.http-address.nameservice1.namenode188": "hostname_of_server1:50070",
        "dfs.namenode.http-address.nameservice1.namenode188": "hostname_of_server2:50070",
        "hadoop.security.authentication": "kerberos"
}
fs = HDFileSystem(host=host, pars=conf)
```

Note that no `port` is specified (requires hdfs version 0.1.3)

2.5 API

HDFileSystem([host, port, user, ...])	Connection to an HDFS namenode
HDFileSystem.cat(path)	Return contents of file
HDFileSystem.chmod(path, mode)	Change access control of given path
HDFileSystem.chown(path, owner, group)	Change owner/group
HDFileSystem.df()	Used/free disc space on the HDFS system
HDFileSystem.du(path[, total, deep])	Returns file sizes on a path.
HDFileSystem.exists(path)	Is there an entry at path?
HDFileSystem.get(hdfs_path, local_path[, ...])	Copy HDFS file to local
HDFileSystem.getmerge(path, filename[, ...])	Concat all files in path (a directory) to output file
HDFileSystem.get_block_locations(path[, ...])	Fetch physical locations of blocks
HDFileSystem.glob(path)	Get list of paths mathing glob-like pattern (i.e., with "*"s).
HDFileSystem.info(path)	File information (as a dict)
HDFileSystem.ls(path[, detail])	List files at path
HDFileSystem.mkdir(path)	Make directory at path
HDFileSystem.mv(path1, path2)	Move file at path1 to path2
HDFileSystem.open(path[, mode, replication, ...])	Open a file for reading or writing
HDFileSystem.put(filename, path[, chunk])	Copy local file to path in HDFS
HDFileSystem.read_block(fn, offset, length)	Read a block of bytes from an HDFS file
HDFileSystem.rm(path[, recursive])	Use recursive for *rm -r*, i.e., delete directory and contents
HDFileSystem.set_replication(path, replication)	Instruct HDFS to set the replication for the given file.
HDFileSystem.tail(path[, size])	Return last bytes of file
HDFileSystem.touch(path)	Create zero-length file

HDFile(fs, path, mode[, replication, buff, ...])	File on HDFS
HDFile.close()	Flush and close file, ensuring the data is readable
HDFile.flush()	Send buffer to the data-node; actual write to disc may happen later
HDFile.info()	Filesystem metadata about this file

Continued on next page

Table 2.2 – continued from previous page

HDFile.read([length])	Read bytes from open file
HDFile.readlines()	Return all lines in a file as a list
HDFile.seek(offset[, from_what])	Set file read position.
HDFile.tell()	Get current byte location in a file
HDFile.write(data)	Write bytes to open file (which must be in w or a mode)

HDFSMap(hdfs, root[, check])	Wrap a HDFileSystem as a mutable mapping.

class hdfs3.core.**HDFileSystem**(*host=None*, *port=None*, *user=None*, *ticket_cache=None*, *token=None*, *pars=None*, *connect=True*)

Connection to an HDFS namenode

```
>>> hdfs = HDFileSystem(host='127.0.0.1', port=8020)
```

cat(*path*)

Return contents of file

chmod(*path*, *mode*)

Change access control of given path

Exactly what permissions the file will get depends on HDFS configurations.

Parameters **path** : string

file/directory to change

mode : integer

As with the POSIX standard, each octal digit refers to user-group-all, in that order, with read-write-execute as the bits of each group.

Examples

```
>>> hdfs.chmod('/path/to/file', 0o777)  # make read/writeable to all
>>> hdfs.chmod('/path/to/file', 0o700)  # make read/writeable only to user
>>> hdfs.chmod('/path/to/file', 0o100)  # make read-only to user
```

chown(*path*, *owner*, *group*)

Change owner/group

connect()

Connect to the name node

This happens automatically at startup

df()

Used/free disc space on the HDFS system

disconnect()

Disconnect from name node

du(*path*, *total=False*, *deep=False*)

Returns file sizes on a path.

Parameters **path** : string

where to look

total : bool (False)

to add up the sizes to a grand total

deep : bool (False)

whether to recurse into subdirectories

exists(*path*)
Is there an entry at path?

get(*hdfs_path*, *local_path*, *blocksize=65536*)
Copy HDFS file to local

get_block_locations(*path*, *start=0*, *length=0*)
Fetch physical locations of blocks

getmerge(*path*, *filename*, *blocksize=65536*)
Concat all files in path (a directory) to output file

glob(*path*)
Get list of paths mathing glob-like pattern (i.e., with "*"s).

If passed a directory, gets all contained files; if passed path to a file, without any "*", returns one-element list containing that filename. Does not support python3.5's "**" notation.

head(*path*, *size=1024*)
Return first bytes of file

info(*path*)
File information (as a dict)

ls(*path*, *detail=True*)
List files at path

Parameters path : string/bytes

location at which to list files

detail : bool (=True)

if True, each list item is a dict of file properties; otherwise, returns list of filenames

mkdir(*path*)
Make directory at path

mv(*path1*, *path2*)
Move file at path1 to path2

open(*path*, *mode='rb'*, *replication=0*, *buff=0*, *block_size=0*)
Open a file for reading or writing

Parameters path: string

Path of file on HDFS

mode: string

One of 'rb', 'wb', or 'ab'

replication: int

Replication factor; if zero, use system default (only on write)

block_size: int

Size of data-node blocks if writing

put (*filename*, *path*, *chunk=65536*)
Copy local file to path in HDFS

read_block (*fn*, *offset*, *length*, *delimiter=None*)
Read a block of bytes from an HDFS file

Starting at `offset` of the file, read `length` bytes. If `delimiter` is set then we ensure that the read starts and stops at delimiter boundaries that follow the locations `offset` and `offset + length`. If `offset` is zero then we start at zero. The bytestring returned will not include the surrounding delimiter strings.

If offset+length is beyond the eof, reads to eof.

Parameters **fn: string**

Path to filename on HDFS

offset: int

Byte offset to start read

length: int

Number of bytes to read

delimiter: bytes (optional)

Ensure reading starts and stops at delimiter bytestring

See also:

`hdfs3.utils.read_block`

Examples

```
>>> hdfs.read_block('/data/file.csv', 0, 13)
b'Alice, 100\nBo'
>>> hdfs.read_block('/data/file.csv', 0, 13, delimiter=b'\n')
b'Alice, 100\nBob, 200'
```

rm (*path*, *recursive=True*)
Use recursive for *rm -r*, i.e., delete directory and contents

set_replication (*path*, *replication*)
Instruct HDFS to set the replication for the given file.

If successful, the head-node's table is updated immediately, but actual copying will be queued for later. It is acceptable to set a replication that cannot be supported (e.g., higher than the number of data-nodes).

tail (*path*, *size=1024*)
Return last bytes of file

touch (*path*)
Create zero-length file

walk (*path*)
Get all file entries below given path

class `hdfs3.core.`**HDFile** (*fs*, *path*, *mode*, *replication=0*, *buff=0*, *block_size=0*)
File on HDFS

Matches the standard Python file interface.

Examples

```
>>> with hdfs.open('/path/to/hdfs/file.txt') as f:
...     bytes = f.read(1000)
>>> with hdfs.open('/path/to/hdfs/file.csv') as f:
...     df = pd.read_csv(f, nrows=1000)
```

close()
Flush and close file, ensuring the data is readable

flush()
Send buffer to the data-node; actual write to disc may happen later

info()
Filesystem metadata about this file

read(*length=None*)
Read bytes from open file

readline(*chunksize=65536*, *lineterminator='\n'*)
Return a line using buffered reading.

Reads and caches chunksize bytes of data, and caches lines locally. Subsequent readline calls deplete those lines until empty, when a new chunk will be read. A read and readline are not therefore generally pointing to the same location in the file; *seek()* and *tell()* will give the true location in the file, which will be one chunk in even after calling *readline* once.

Line iteration uses this method internally.

readlines()
Return all lines in a file as a list

seek(*offset*, *from_what=0*)
Set file read position. Read mode only.

Attempt to move out of file bounds raises an exception. Note that, by the convention in python file seek, offset should be <=0 if from_what is 2.

Parameters **offset** : int

byte location in the file.

from_what : int 0, 1, 2

if 0 (befault), relative to file start; if 1, relative to current location; if 2, relative to file end.

Returns new position

tell()
Get current byte location in a file

write(*data*)
Write bytes to open file (which must be in w or a mode)

class hdfs3.mapping.**HDFSMap**(*hdfs*, *root*, *check=False*)
Wrap a HDFileSystem as a mutable mapping.

The keys of the mapping become files under the given root, and the values (which must be bytes) the contents of those files.

Parameters **hdfs** : HDFileSystem

root : string

path to contain the stored files (directory will be created if it doesn't exist)

check : bool (=True)

performs a touch at the location, to check writeability.

Examples

```
>>> hdfs = hdfs3.HDFileSystem()
>>> mw = HDFSMap(hdfs, '/writable/path/')
>>> mw['loc1'] = b'Hello World'
>>> list(mw.keys())
['loc1']
>>> mw['loc1']
b'Hello World'
```

2.6 Known Limitations

2.6.1 Forked processes

The `libhdfs3` library is not fork-safe. If you start using `hdfs3` in a parent process and then fork a child process, using the library from the child process may produce random errors because of system resources that will not be available (such as background threads). Common solutions include the following:

- Use threads instead of processes
- Use Python 3 and a multiprocessing context using either the "spawn" or "forkserver" method (see multiprocessing docs)
- Only instantiate `HDFileSystem` within the forked processes, do not ever start an `HDFileSystem` within the parent processes