



Denodo HDFS Custom Wrapper

Revision 20130703

NOTE

This document is confidential and proprietary of **Denodo Technologies**.
No part of this document may be reproduced in any form by any means without prior written authorization of **Denodo Technologies**.

Copyright © 2013
Denodo Technologies Proprietary and Confidential

CONTENTS

1 INTRODUCTION.....	4
2 HDFS.....	5
2.1 DELIMITED TEXT FILES.....	6
2.2 SEQUENCEFILES.....	6
2.3 MAPFILES.....	8
2.4 AVRO FILES.....	8
3 HDFS CUSTOM WRAPPER.....	10
3.1 HDFSDELIMITEDTEXTFILEWRAPPER.....	10
3.2 HDFSSEQUENCEFILEWRAPPER.....	11
3.3 HDFSMAPIFILEWRAPPER.....	12
3.4 HDFS AVROFILEWRAPPER.....	14
3.5 WEBHDFSFILEWRAPPER.....	16
4 HDFS COMPRESSED FILES.....	19
5 SOFTWARE REQUIREMENTS.....	20



1 INTRODUCTION

The `hdfs-customwrapper` distribution contains five Virtual DataPort custom wrappers capable of reading several file formats stored in **Hadoop Distributed File System** (HDFS).

Supported formats are:

1. Delimited text files (both directly from HDFS and also via HDFS over HTTP)
2. Sequence files
3. Map files
4. Avro files

2 HDFS

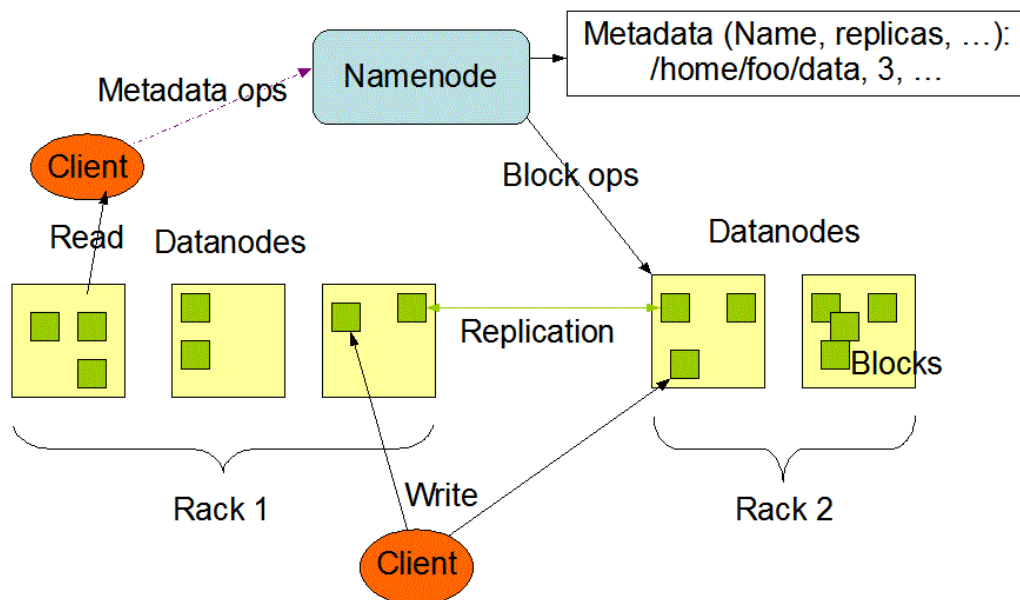
The Hadoop Distributed File System is a distributed, scalable, and portable file system used by the Hadoop platform.

In HDFS, data is divided into blocks and copies of these blocks are stored on other servers in the Hadoop cluster. That is, an individual file is actually stored as smaller blocks that are replicated across multiple servers in the entire cluster. This redundancy offers multiple benefits for Big Data processing:

1. Higher availability.
2. Better scalability: map and reduce functions can be executed on smaller subsets of large data sets.
3. Data locality: move the computation closer to the data to reduce latency.

HDFS has a master/slave architecture in which the **NameNode**, the master, manages the file system namespace and regulates clients access to files. And the **DataNodes**, the slaves, manage storage attached to the nodes that they run on.

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 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.



HDFS Architecture

2.1 DELIMITED TEXT FILES

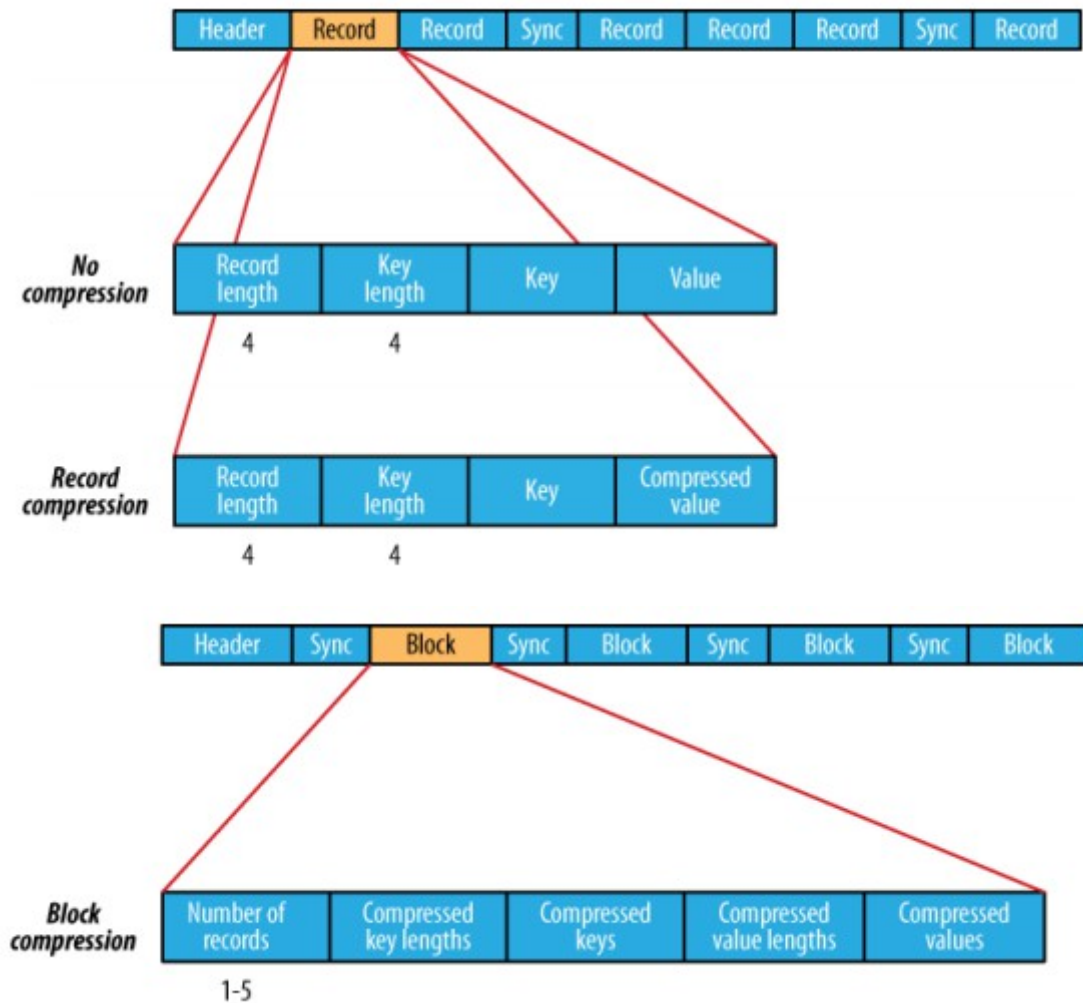
Delimited text files store plain text key/value pairs. The key/value pairs are delimited by a separator such as tab, space, comma, etc.

2.2 SEQUENCEFILES

Sequence files are binary record-oriented files, where each record has a serialized key and a serialized value.

The Hadoop framework supports compressing and decompressing sequence files transparently. Therefore, sequence files have three available formats:

1. No compression.
2. Record compression: only values are compressed.
3. Block compression: both keys and values are collected in 'blocks' separately and compressed.



Sequence file formats

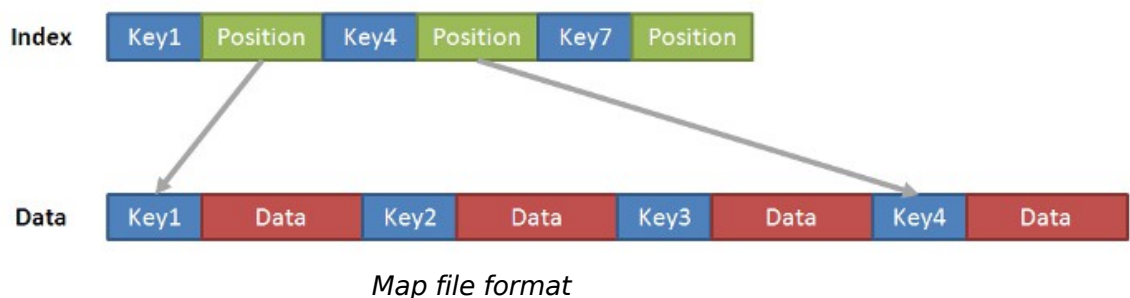
The three formats share a header that contains information which allows the reader to recognize their format:

1. Version: 3 bytes of magic header SEQ, followed by 1 byte of actual version number.
2. Key class name.
3. Value class name.
4. Compression: specifies if compression is turned on for keys/values.
5. Block compression: specifies if block-compression is turned on for keys/values.
6. Compression codec: CompressionCodec class which is used for compression of keys and/or values (if compression is enabled).
7. Metadata.
8. Sync: a sync marker to denote end of the header.

2.3 MAPFILES

A map is a directory containing two sequence files. The data file (/data) is identical to the sequence file and contains the data stored as binary key/value pairs. The index file (/index), which contains a key/value map with seek positions inside the data file to quickly access the data.

The index file is populated with the key and a LongWritable that contains the starting byte position of the record. Index does not contains all the keys but just a fraction of the keys. The index is read entirely into memory.



2.4 AVRO FILES

Avro is a data serialization format.

Avro data files are self-describing, containing the full schema for the data in the file. An Avro schema is defined using JSON. The schema allows you to define two types of data:

1. primitive data types: string, integer, long, float, double, byte, null and boolean.
2. complex type definitions: a record, an array, an enum, a map, a union or a fixed type.

```
{
  "namespace": "example.avro",
  "type": "record",
  "name": "User",
  "fields": [
    {
      "name": "name", "type": "string"
    },
    {
      "name": "favorite_number", "type": ["int", "null"]
    },
    {
      "name": "favorite_color", "type": ["string", "null"]
    }
  ]
}
```

Avro schema

Avro relies on schemas. When Avro data is read, the schema used when writing it is always present. This allows each datum to be written with no per-value overheads, making serialization both fast and small. This also facilitates use with dynamic, scripting languages, since data, together with its schema, is fully self-describing.

3 HDFS CUSTOM WRAPPER

hdfs-customwrapper library includes five custom wrappers:

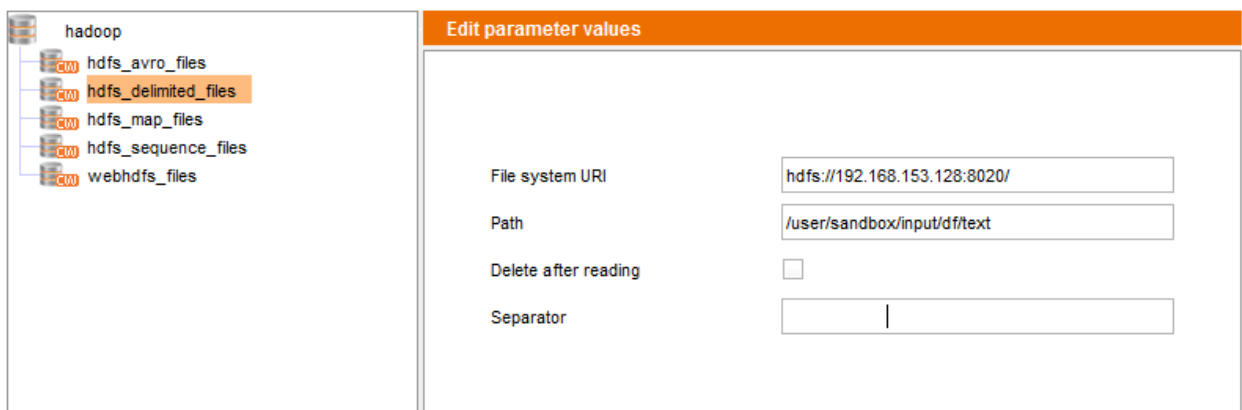
1. `com.denodo.connect.hadoop.hdfs.wrapper.HDFSDelimitedTextFileWrapper`
2. `com.denodo.connect.hadoop.hdfs.wrapper.HDFSSequenceFileWrapper`
3. `com.denodo.connect.hadoop.hdfs.wrapper.HDFSMapFileWrapper`
4. `com.denodo.connect.hadoop.hdfs.wrapper.HDFSAvroFileWrapper`
5. `com.denodo.connect.hadoop.hdfs.wrapper.WebHDFSFileWrapper`

3.1 HDFSDELIMITEDTEXTFILEWRAPPER

Custom wrapper for reading key/value delimited text files stored in HDFS.

The base views created from the `HDFSDelimitedTextFileWrapper` need the following **mandatory** parameters:

1. File system URI: A URI whose scheme and authority identify the file system. The scheme determines the file system implementation. The authority is used to determine the host, port, etc.
 1. For HDFS the URI has the form `hdfs://<ip>:<port>`.
 2. For Amazon S3 the URI has the form `s3n://<id>:<secret>@<bucket>` (Note that since the secret access key can contain slashes, each slash / should be replaced with the string `%2F`).
2. Path: input path for the delimited file or the directory containing the files.
3. Delete after reading: Requests that the file or directory denoted by the path be deleted when the wrapper terminates.
4. Separator: delimiter between the keys and values.



HDFSDelimitedTextFileWrapper base view edition

delimited_file_test	
key	text
value	text

View schema

The execution of the wrapper returns the key/value pairs contained in the file or group of files, if the Path input parameter denotes a directory.

Execute view delimited_file_test	
Total rows received: 5 (shown 5)	
key	value
1	One, two, buckle my shoe
2	Three, four, shut the door
3	Five, six, pick up sticks
4	Seven, eight, lay them straight
5	Nine, ten, a big fat hen

View results

3.2 HDFSSEQUENCEFILEWRAPPER

Custom wrapper for reading sequence files stored in HDFS.

The base views created from the HDFSSequenceFileWrapper need the following **mandatory** parameters:

- File system URI: A URI whose scheme and authority identify the file system. The scheme determines the file system implementation. The authority is used to determine the host, port, etc.
 - For HDFS the URI has the form `hdfs://<ip>:<port>`.
 - For Amazon S3 the URI has the form `s3n://<id>:<secret>@<bucket>` (Note that since the secret access key can contain slashes, each slash / should be replaced with the string %2F).
- Path: input path for the sequence file or the directory containing the files.
- Delete after reading: Requests that the file or directory denoted by the path be deleted when the wrapper terminates.
- Key class: key class name implementing `org.apache.hadoop.io.Writable` interface.
- Value class: value class name implementing `org.apache.hadoop.io.Writable` interface.

Edit parameter values	
File system URI	s3n://AKIAIF:uHLT7a%2Fujmwyq14@denodotest
Path	/sequencefile/sequence.seq
Delete after reading	<input type="checkbox"/>
Key class	org.apache.hadoop.io.IntWritable
Value class	org.apache.hadoop.io.Text

HDFSSequenceFileWrapper base view edition

sequence_files_test	
key	int
value	text

View schema

The execution of the wrapper returns the key/value pairs contained in the file or group of files, if the Path input parameter denotes a directory.

Execute view sequence_files_test	
Total rows received: 100 (shown 100)	
key	value
100	One, two, buckle my shoe
99	Three, four, shut the door
98	Five, six, pick up sticks
97	Seven, eight, lay them straight
96	Nine, ten, a big fat hen
95	One, two, buckle my shoe
94	Three, four, shut the door
93	Five, six, pick up sticks
92	Seven, eight, lay them straight
91	Nine, ten, a big fat hen
90	One, two, buckle my shoe

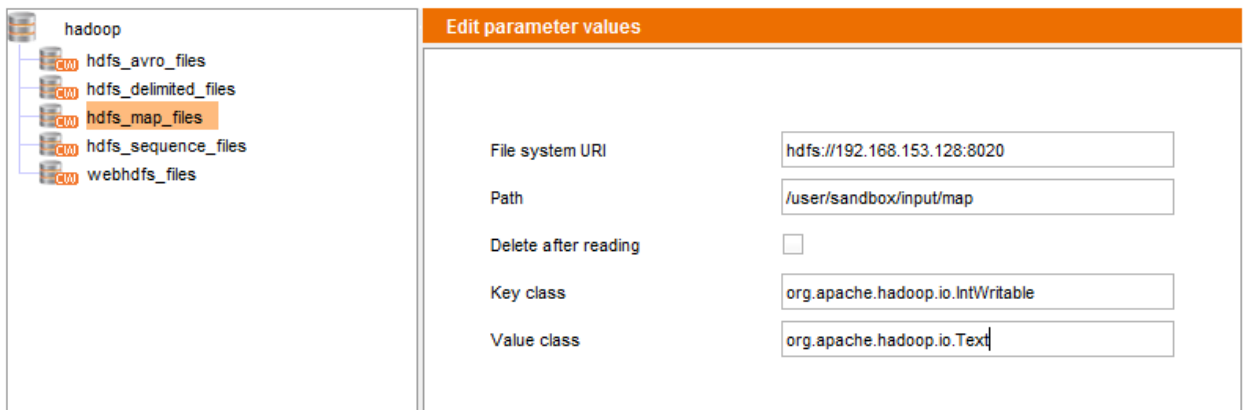
View results

3.3 HDFSMAPIFILEWRAPPER

Custom wrapper for reading map files stored in HDFS.

The base views created from the HDFSMapFileWrapper need the following mandatory parameters:

1. File system URI: A URI whose scheme and authority identify the file system. The scheme determines the file system implementation. The authority is used to determine the host, port, etc.
 1. For HDFS the URI has the form `hdfs://<ip>:<port>`.
 2. For Amazon S3 the URI has the form `s3n://<id>:<secret>@<bucket>` (Note that since the secret access key can contain slashes, each slash / should be replaced with the string %2F).
2. Path: input path for the directory containing the map file. Also the path to the index or data file could be specified. When using **Amazon S3**, a flat file system where there is no folder concept, the path to the index or data should be used.
3. Delete after reading: Requests that the file or directory denoted by the path be deleted when the wrapper terminates.
4. Key class: key class name implementing the `org.apache.hadoop.io.WritableComparable` interface. `WritableComparable` is used because records are sorted in **key order**.
5. Value class: value class name implementing the `org.apache.hadoop.io.Writable` interface.



HDFSMapFileWrapper base view edition

map_files_test	
key	int
value	text

View schema

The execution of the wrapper returns the key/value pairs contained in the file or group of files, if the Path input parameter denotes a directory.

Execute view map_files_test	
Total rows received: 1024 (shown 150)	
key	value
1	One, two, buckle my shoe
2	Three, four, shut the door
3	Five, six, pick up sticks
4	Seven, eight, lay them straight
5	Nine, ten, a big fat hen
6	One, two, buckle my shoe
7	Three, four, shut the door
8	Five, six, pick up sticks
9	Seven, eight, lay them straight
10	Nine, ten, a big fat hen
11	One, two, buckle my shoe

View results

3.4 HDFS AVRO FILE WRAPPER

Custom wrapper for reading Avro files stored in HDFS.

The base views created from the HDFS Avro File Wrapper need the following mandatory parameters:

1. File system URI: A URI whose scheme and authority identify the file system. The scheme determines the file system implementation. The authority is used to determine the host, port, etc.
 1. For HDFS the URI has the form `hdfs://<ip>:<port>`.
 2. For Amazon S3 the URI has the form `s3n://<id>:<secret>@<bucket>` (Note that since the secret access key can contain slashes, each slash / should be replaced with the string %2F).
2. Delete after reading: Requests that the file denoted by the path be deleted when the wrapper terminates.

There is also two parameters that are **mutually exclusive**:

1. Avro schema path: input path for the Avro schema file.
2. Avro schema JSON: JSON of the Avro schema.

hadoop

- hdfs_avro_files
- hdfs_delimited_files
- hdfs_map_files
- hdfs_sequence_files
- webhdfs_files

Edit parameter values

File system URI

Avro schema path

Avro schema JSON

Delete after reading
☐

HDFSavroFileWrapper base view edition

```
{
  "type": "map",
  "values": {
    "type": "record",
    "name": "ATM",
    "fields": [
      { "name": "serial_no", "type": "string" },
      { "name": "location", "type": { "type": "map", "values": { "type": "array", "items": "int" } } }
    ]
  }
}
```

Avro schema

avro_files_test	
avrofilepath	text
map	hdfs_avro_files_map
key	text
atm	hdfs_avro_files_map_map_record_atm
serial_no	text
location	hdfs_avro_files_map_map_record_atm_location
key	text
value	hdfs_avro_files_map_map_record_atm_location_location_record_value
int	int

View schema

The execution of the wrapper returns the values contained in the Avro file specified in the WHERE clause.

Execute view avro_files_test	
Total rows received: 1 (shown 1)	
avrofilepath	map
/user/sandbox/avro/MapComplex.avro	[Array]...

↓

RESULT -> map

key	atm
atm1	[Register]...
atm4	[Register]...
atm2	[Register]...

↓

RESULT -> map -> atm

serial_no	location
zxy555	[Array]...

↓

RESULT -> map -> atm -> location

key	value
central	[Array]...
downtown	[Array]...

↓

RESULT -> map -> atm -> location -> value

int
1
2
3

View results

3.5 WEBHDFSFILEWRAPPER

Custom wrapper for reading key/value delimited text files stored in HDFS using the **WebHDFS**.

3.5.1 About WebHDFS

WebHDFS provides HTTP REST access to HDFS. It supports all HDFS user operations including reading files, writing to files, making directories, changing permissions and renaming.

The advantage of WebHDFS are:

1. **Version-independent** REST-based protocol which means that can be read and written to/from Hadoop clusters no matter their version. This addresses the issue of using the Java API (RPC-based) that requires both the client and the Hadoop cluster to share the same version. Upgrading one without the other causes serialization errors meaning the client cannot interact with the cluster.
2. Read and write data in HDFS in a cluster behind a firewall. A proxy WebHDFS (for example: HttpFS) could be use, it acts as a gateway and is the only system that is allowed to send and receive data through the firewall. The only difference between using or not the proxy will be in the `host:port` pair where the HTTP requests are issued:
 1. Default port for WebHDFS is 50075.
 2. Default port for HttpFS is 14000.

3.5.2 Custom wrapper

The base views created from the `WebHDFSFileWrapper` need the following mandatory parameters:

1. Host IP: IP or `<bucket>.s3.amazonaws.com` for Amazon S3.
2. Host port: HTTP port. Default port for WebHDFS is 50075. For HttpFS is 14000. For Amazon S3 is 80.
3. Path: input path for the delimited file.
4. Separator: delimiter between the keys and values.
5. Delete after reading: Requests that the file or directory denoted by the path be deleted when the wrapper terminates.

Optional parameter:

1. User: The name of the the authenticated user when security is off. If is not set, the server may either set the authenticated user to a default web user, if there is any, or return an error response.
When using **Amazon S3** `<id>:<secret>` should be indicated.

Edit parameter values	
Host IP	<input type="text" value="192.168.153.128"/>
Host port	<input type="text" value="50075"/>
User	<input type="text"/>
Path	<input type="text" value="/user/sandbox/input/df/text"/>
Separator	<input type="text" value=" "/>
Delete after reading	<input type="checkbox"/>

WebHDFSFileWrapper base view edition

webhdfs_files_test	
key	text
value	text

View schema

The execution of the wrapper returns the key/value pairs contained in the file.

Execute view webhdfs_files_test	
Total rows received: 5 (shown 5)	
key	value
1	One, two, buckle my shoe
2	Three, four, shut the door
3	Five, six, pick up sticks
4	Seven, eight, lay them straight
5	Nine, ten, a big fat hen

View results

4 HDFS COMPRESSED FILES

Hadoop is intended for storing large data volumes, so compression becomes a mandatory requirement here. There are different compression formats available like DEFLATE, GZip, BZip2, Snappy and LZO.

Hadoop has native implementations of compression libraries for performance reasons and for non-availability of Java implementations:

Compression format	Java implementation	Native implementation
DEFLATE	Yes	Yes
gzip	Yes	Yes
bzip2	Yes	No
LZO	No	Yes
Snappy	No	Yes

Compression library implementations

For reading HDFS compressed files using the `hdfs-customwrapper` library there are two options:

1. Use the Java implementation. `hdfs-customwrapper` handles compressed files transparently.
2. Use the native implementation (for performance reasons or for non-availability of Java implementation). `hdfs-customwrapper` must have Hadoop native libraries in the `java.library.path`.

Hadoop comes with prebuilt native compression libraries for 32- and 64-bit Linux, which could be found in the `lib/native` directory. For other platforms, the libraries must be compiled, following the instructions on the Hadoop wiki at <http://wiki.apache.org/hadoop/NativeHadoop>.

In the **HortonWorks Sandbox** (see *Software requirements* section) native libraries could be found at `/usr/lib/hadoop/lib/native/{os.arch}`.

5 SOFTWARE REQUIREMENTS

hdfs-customwrapper has been tested in the HortonWorks Data Platform 1.2 Sandbox using Hadoop v1.1.2 and Avro v1.7.2.

HortonWorks Sandbox is a complete, self contained virtual machine with Apache Hadoop pre-configured. It can be downloaded from:

<http://hortonworks.com/products/hortonworks-sandbox/>

hdfs-customwrapper has also been tested in Cloudera QuickStart VM 4.2.0 using Hadoop v2.0-cdh4.3.0 and Avro v1.7.2.

Cloudera provides a virtual machine that gives a working Apache Hadoop environment out-of-the-box. It can be downloaded from:

<https://ccp.cloudera.com/display/SUPPORT/Cloudera+QuickStart+VM>

hdfs-customwrapper has been tested in Amazon S3 too, using Hadoop v1.1.2 and Avro v1.7.2. For more information see <http://aws.amazon.com/es/s3/>