Getting Data into Your H2O Cluster

The first step toward building and scoring your models is getting your data into the H2O cluster/Java process that's running on your local or remote machine. Whether you're importing data, uploading data, or retrieving data from HDFS or S3, be sure that your data is compatible with H2O.

Supported File Formats

H2O currently supports the following file types:

- CSV (delimited, UTF-8 only) files (including GZipped CSV)
- ORC
- SVMLight
- ARFF
- XLS (BIFF 8 only)
- XLSX (BIFF 8 only)
- Avro version 1.8.0 (without multifile parsing or column type modification)
- Parquet

Notes:

- H2O supports UTF-8 encodings for CSV files. Please convert UTF-16 encodings to UTF-8 encoding before parsing CSV files into H2O.
- ORC is available only if H2O is running as a Hadoop job.
- Users can also import Hive files that are saved in ORC format (experimental).
- If you encounter issues importing XLS or XLSX files, you may be using an unsupported version. In this case, re-save the file in BIFF 8 format. Also note that XLS and XLSX support will eventually be deprecated.
- When doing a parallel data import into a cluster:
  - If the data is an unzipped csv file, H2O can do offset reads, so each node in your cluster can be directly reading its part of the csv file in parallel.
  - If the data is zipped, H2O will have to read the whole file and unzip it before doing the parallel read.

  So, if you have very large data files reading from HDFS, it is best to use unzipped csv. But if the data is further away than the LAN, then it is best to use zipped csv.

Data Sources

H2O supports data ingest from various data sources. Natively, a local file system, remote file systems, HDFS, S3, and some relational databases are supported. Additional data sources can be accessed through a generic HDFS API, such as Alluxio or OpenStack Swift.

Default Data Sources

- Local File System
- Remote File
- S3
- HDFS
- JDBC
- Hive

Local File System

Data from a local machine can be uploaded to H2O via a push from the client. For more information, refer to Uploading a File.

Remote File

Data that is hosted on the Internet can be imported into H2O by specifying the URL. For more information, refer to Importing a File.

HDFS-like Data Sources

Various data sources can be accessed through an HDFS API. In this case, a library providing access to a data source needs to be passed on a command line when H2O is launched. (Reminder: Each node in the cluster must be launched in the same way.) The library must be compatible with the HDFS API in order to be registered as a correct HDFS `FileSystem`.

Alluxio FS

Required Library

To access Alluxio data source, an Alluxio client library that is part of Alluxio distribution is required. For example, `alluxio-1.3.0/core/client/target/alluxio-core-client-1.3.0-jar-with-dependencies.jar`.

H2O Command Line

```
java -cp alluxio-core-client-1.3.0-jar-with-dependencies.jar:build/h2o.jar water.H2OApp
```

URI Scheme

An Alluxio data source is referenced using `alluxio://` schema and location of Alluxio master. For example,

```
alluxio://localhost:19998/iris.csv
```

core-site.xml Configuration

Not supported.

IBM Swift Object Storage

Required Library

To access IBM Object Store (which can be exposed via Bluemix or Softlayer), IBM's HDFS driver `hadoop-openstack.jar` is required. The driver can be obtained, for example, by running BigInsight instances at location `/usr/iop/4.2.0.0/hadoop-mapreduce/hadoop-openstack.jar`.

Note: The jar available at Maven central is not compatible with IBM Swift Object Storage.

H2O Command Line

```
java -cp hadoop-openstack.jar:h2o.jar water.H2OApp
```

URI Scheme

Data source is available under the regular Swift URI structure: `swift://<CONTAINER>.<SERVICE>/path/to/file` For example,

```
swift://smalldata.h2o/iris.csv
```

core-site.xml Configuration

The core-site.xml needs to be configured with Swift Object Store parameters. These are available in the Bluemix/Softlayer management console.

```
<configuration>
  <property>
    <name>fs.swift.service.SERVICE.auth.url</name>
    <value>https://identity.open.softlayer.com/v3/auth/tokens</value>
  </property>
  <property>
    <name>fs.swift.service.SERVICE.project.id</name>
    <value>...</value>
  </property>
  <property>
    <name>fs.swift.service.SERVICE.user.id</name>
    <value>...</value>
  </property>
  <property>
    <name>fs.swift.service.SERVICE.password</name>
    <value>...</value>
  </property>
  <property>
    <name>fs.swift.service.SERVICE.region</name>
    <value>dallas</value>
  </property>
  <property>
    <name>fs.swift.service.SERVICE.public</name>
    <value>false</value>
  </property>
</configuration>
```

Google Cloud Storage Connector for Hadoop & Spark

Required Library

To access the Google Cloud Store Object Store, Google's cloud storage connector, `gcs-connector-latest-hadoop2.jar` is required. The official documentation and driver can be found here.

H2O Command Line

```
H2O on Hadoop:
hadoop jar h2o-driver.jar -libjars /path/to/gcs-connector-latest-hadoop2.jar

Sparkling Water
export SPARK_CLASSPATH=/home/nick/spark-2.0.2-bin-hadoop2.6/lib_managed/jar/gcs-connector-
latest-hadoop2.jar
sparkling-water-2.0.5/bin/sparkling-shell --conf "spark.executor.memory=10g"
```

URI Scheme

Data source is available under the regular Google Storage URI structure: `gs://<BUCKETNAME>/path/to/file` For example,

```
gs://mybucket/iris.csv
```

core-site.xml Configuration

core-site.xml must be configured for at least the following properties (class, project-id, bucketname) as shown in the example below. A full list of configuration options is found here.

```
<configuration>
    <property>
            <name>fs.gs.impl</name>
            <value>com.google.cloud.hadoop.fs.gcs.GoogleHadoopFileSystem</value>
    </property>
    <property>
            <name>fs.gs.project.id</name>
            <value>my-google-project-id</value>
    </property>
    <property>
            <name>fs.gs.system.bucket</name>
            <value>mybucket</value>
    </property>
</configuration>
```

Direct Hive Import

H2O supports direct ingestion of data managed by Hive in Hadoop. This feature is available only when H2O is running as a Hadoop job. Internally H2O uses metadata in Hive Metastore database to determine the location and format of given Hive table. H2O then imports data directly from HDFS so limitations of supported formats mentioned above apply. Data from hive can pulled into H2O using `import_hive_table` function. H2O can read Hive table metadata two ways - either via direct Metastore access or via JDBC.

Note: When ingesting data from Hive in Hadoop, direct Hive import is preferred over Using the Hive 2 JDBC Driver.

Requirements

- The user running H2O must have read access to Hive and the files it manages.
- For Direct Metastore access, the Hive jars and configuration must be present on H2O job classpath - either by adding it to yarn.application.classpath (or similar property for your resource manger of choice) or by adding Hive jars and configuration to libjars.
- For JDBC metadata access, the Hive JDBC Driver must be on H2O job classpath.

Limitations

- The imported table must be stored in a format supported by H2O.
- CSV: The Hive table property `skip.header.line.count` is currently not supported. CSV files with header rows will be imported with the header row as data.
- Partitioned tables with different storage formats. H2O supports importing partitioned tables that use different storage formats for different partitions; however in some cases (for example large number of small partitions), H2O may run out of memory while importing, even though the final data would easily fit into the memory allocated to the H2O cluster.

Importing Examples

Example 1: Access Metadata via Metastore

This example shows how to access metadata via the metastore.

1. Start the H2O jar in the terminal with your downloaded Hive JDBC driver in the classpath

```
# start the h2o.jar
hadoop jar h2odriver.jar -libjars hive-jdbc-standalone.jar -nodes 3 -mapperXmx 6g
```

2. Import data in R or Python.

R Python

```
# basic import
basic_import <- h2o.import_hive_table("default", "table_name")

# multi-format import
multi_format_enabled <- h2o.import_hive_table("default",
                                              "table_name",
                                              allow_multi_format=True)

# import with partition filter
with_partition_filter <- h2o.import_hive_table("default",
                                               "table_name",
                                               [["2017", "02"]])
```

Example 2: Access Metadata via JDBC

This example shows how to access metadata via JDBC.

1. Start the H2O jar in the terminal with your downloaded Hive JDBC driver in the classpath

```
# start the h2o.jar
hadoop jar h2odriver.jar -libjars hive-jdbc-standalone.jar -nodes 3 -mapperXmx 6g
```

2. Import data in R or Python.

R Python

```
# basic import of metadata via JDBC
basic_import <- h2o.import_hive_table("jdbc:hive2://hive-server:10000/default",
"table_name")
```

JDBC Databases

Relational databases that include a JDBC (Java database connectivity) driver can be used as the source of data for machine learning in H2O. Currently supported SQL databases are MySQL, PostgreSQL, MariaDB, Netezza, Amazon Redshift, Teradata, and Hive. (Refer to Using the Hive 2 JDBC Driver for more information.) Data from these SQL databases can be pulled into H2O using the `import_sql_table` and `import_sql_select` functions.

Refer to the following articles for examples about using JDBC data sources with H2O.

- Setup postgresql database on OSX
- Restoring DVD rental database into postgresql
- Building H2O GLM model using Postgresql database and JDBC driver

Note: The handling of categorical values is different between file ingest and JDBC ingests. Te JDBC treats categorical values as Strings. Strings are not compressed in any way in H2O memory, and using the JDBC interface might need more memory and additional data post-processing (converting to categoricals explicitly).

`import_sql_table`

This function imports a SQL table to H2OFrame in memory. This function assumes that the SQL table is not being updated and is stable. Users can run multiple SELECT SQL queries concurrently for parallel ingestion.

Note: Be sure to start the h2o.jar in the terminal with your downloaded JDBC driver in the classpath:

```
java -cp <path_to_h2o_jar>:<path_to_jdbc_driver_jar> water.H2OApp
```

The `import_sql_table` function accepts the following parameters:

- `connection_url`: The URL of the SQL database connection as specified by the Java Database Connectivity (JDBC) Driver. For example, "jdbc:mysql://localhost:3306/menagerie?&useSSL=false"
- `table`: The name of the SQL table
- `columns`: A list of column names to import from SQL table. Default is to import all columns.
- `username`: The username for SQL server
- `password`: The password for SQL server
- `optimize`: Specifies to optimize the import of SQL table for faster imports. Note that this option is experimental.
- `fetch_mode`: Set to DISTRIBUTED to enable distributed import. Set to SINGLE to force a sequential read by a single node from the database.
- `num_chunks_hint`: Optionally specify the number of chunks for the target frame.

R Python

```
connection_url <- "jdbc:mysql://172.16.2.178:3306/ingestSQL?&useSSL=false"
table <- "citibike20k"
username <- "root"
password <- "abc123"
my_citibike_data <- h2o.import_sql_table(connection_url, table, username, password)
```

`import_sql_select`

This function imports the SQL table that is the result of the specified SQL query to H2OFrame in memory. It creates a temporary SQL table from the specified sql_query. Users can run multiple SELECT SQL queries on the temporary table concurrently for parallel ingestion, and then drop the table.

Note: Be sure to start the h2o.jar in the terminal with your downloaded JDBC driver in the classpath:

```
java -cp <path_to_h2o_jar>:<path_to_jdbc_driver_jar> water.H2OApp
```

The `import_sql_select` function accepts the following parameters:

- `connection_url`: URL of the SQL database connection as specified by the Java Database Connectivity (JDBC) Driver. For example, “jdbc:mysql://localhost:3306/menagerie?&useSSL=false”
- `select_query`: SQL query starting with *SELECT* that returns rows from one or more database tables.
- `username`: The username for the SQL server
- `password`: The password for the SQL server
- `optimize`: Specifies to optimize import of SQL table for faster imports. Note that this option is experimental.
- `use_temp_table`: Specifies whether a temporary table should be created by `select_query`.
- `temp_table_name`: The name of the temporary table to be created by `select_query`.
- `fetch_mode`: Set to DISTRIBUTED to enable distributed import. Set to SINGLE to force a sequential read by a single node from the database.

R Python

```
connection_url <- "jdbc:mysql://172.16.2.178:3306/ingestSQL?&useSSL=false"
select_query <-  "SELECT  bikeid  from  citibike20k"
username <- "root"
password <- "abc123"
my_citibike_data <- h2o.import_sql_select(connection_url, select_query, username, password)
```

Using the Hive 2 JDBC Driver

H2O can ingest data from Hive through the Hive v2 JDBC driver by providing H2O with the JDBC driver for your Hive version. A demo showing how to ingest data from Hive through the Hive v2 JDBC driver is available here. The basic steps are described below.

Notes:

- Direct Hive Import is preferred over using the Hive 2 JDBC driver.
- H2O can only load data from Hive version 2.2.0 or greater due to a limited implementation of the JDBC interface by Hive in earlier versions.

1. Set up a table with data.

a. Retrieve the AirlinesTest dataset from S3.
b. Run the CLI client for Hive:

```
beeline -u jdbc:hive2://hive-host:10000/db-name
```

c. Create the DB table:

```
CREATE EXTERNAL TABLE IF IT DOES NOT EXIST AirlinesTest(
  fYear STRING ,
  fMonth STRING ,
  fDayofMonth STRING ,
  fDayOfWeek STRING ,
  DepTime INT ,
  ArrTime INT ,
  UniqueCarrier STRING ,
  Origin STRING ,
  Dest STRING ,
  Distance INT ,
  IsDepDelayed STRING ,
  IsDepDelayed_REC INT
)
    COMMENT 'test table'
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    LOCATION '/tmp';
```

d. Import the data from the dataset. Note that the file must be present on HDFS in /tmp.

```
LOAD DATA INPATH '/tmp/AirlinesTest.csv' OVERWRITE INTO TABLE AirlinesTest
```

2. Retrieve the Hive JDBC client jar.

- For Hortonworks, Hive JDBC client jars can be found on one of the edge nodes after you have installed HDP: `/usr/hdp/current/hive-client/lib/hive-jdbc-<version>-standalone.jar`. More information is available here.
- For Cloudera, install the JDBC package for your operating system, and then add `/usr/lib/hive/lib/hive-jdbc-<version>-standalone.jar` to your classpath. More information is available here:.
- You can also retrieve this from Maven for the desire version using `mvn dependency:get -Dartifact=groupId:artifactId:version`.

3. Add the Hive JDBC driver to H2O's classpath.

```
# Add the Hive JDBC driver to H2O's classpath
java -cp hive-jdbc.jar:<path_to_h2o_jar> water.H2OApp
```

4. Initialize H2O in either R or Python and import data.

R Python

```
# initialize h2o in R
library(h2o)
h2o.init(extra_classpath=["hive-jdbc-standalone.jar"])
```

5. After the jar file with JDBC driver is added, then data from the Hive databases can be pulled into H2O using the aforementioned `import_sql_table` and `import_sql_select` functions.

R Python

```
connection_url <- "jdbc:hive2://localhost:10000/default"
select_query <- "SELECT * FROM AirlinesTest;"
username <- "username"
password <- "changeit"

airlines_dataset <- h2o.import_sql_select(connection_url,
                                          select_query,
                                          username,
                                          password)
```

Connecting to Hive in a Kerberized Hadoop Cluster

When importing data from Kerberized Hive on Hadoop, it is necessary to configure the h2odriver to authenticate with the Hive instance via a delegation token. Since Hadoop does not generate delegation tokens for Hive automatically, it is necessary to provide the h2odriver with additional configurations.

H2O is able to generate Hive delegation tokens in three modes:

- On the driver side, a token can be generated on H2O cluster start.

- On the mapper side, a token refresh thread is started, periodically re-generating the token.
- A combination of both of the above.

H2O arguments used to configure the JDBC URL for Hive delegation token generation:

- `hiveHost` - The full address of HiveServer2, for example `hostname:10000`
- `hivePrincipal` - Hiveserver2 Kerberos principal, for example `hive/hostname@DOMAIN.COM`
- `hiveJdbcUrlPattern` - (optional) Can be used to further customize the way the driver constructs the Hive JDBC URL. The default pattern used is `jdbc:hive2://{{host}}/;{{auth}}` where `{{auth}}` is replaced by `principal={{hivePrincipal}}` or `auth=delegationToken` based on context

Note on libjars:

In the examples below, we are omitting the `-libjars` option of the `hadoop.jar` command because it is not necessary for token generation. You may need to add it to be able to import data from Hive via JDBC.

Generating the Token in the Driver

The advantage of this approach is that the keytab does not need to be distributed into the Hadoop cluster.

Requirements:

- The Hive JDBC driver is on h2odriver classpath via the HADOOP_CLASSPATH environment variable. (Only used to acquire Hive delegation token.)
- The `hiveHost`, `hivePrincipal` and optionally `hiveJdbcUrlPattern` arguments are present. (See above for details.)

Example command:

```
export HADOOP_CLASSPATH=/path/to/hive-jdbc-standalone.jar
hadoop jar h2odriver.jar \
    -nodes 1 -mapperXmx 4G \
    -hiveHost hostname:10000 -hivePrincipal hive/hostname@EXAMPLE.COM \
    -hiveJdbcUrlPattern "jdbc:hive2://{{host}}/;
{{auth}};ssl=true;sslTrustStore=/path/to/keystore.jks"
```

Generating the Token in the Mapper and Token Refresh

This approach generates a Hive delegation token after the H2O cluster is fully started up and then periodically refreshes the token. Delegation tokens usually have a limited life span, and for long-running H2O clusters, they need to be refreshed. For this to work, the user's keytab and principal need to available to the H2O Cluster Leader node.

Requirements:

- The Hive JDBC driver is on the h2o mapper classpath (either via libjars or YARN configuration).
- The `hiveHost`, `hivePrincipal` and optionally `hiveJdbcUrlPattern` arguments are present. (See above for details.)
- The `principal` argument is set with the value of the users's Kerberos principal.
- The `keytab` argument set pointing to the file with the user's Kerberos keytab file.
- The `refreshTokens` argument is present.

Example command:

```
hadoop jar h2odriver.jar [-libjars /path/to/hive-jdbc-standalone.jar] \
    -nodes 1 -mapperXmx 4G \
    -hiveHost hostname:10000 -hivePrincipal hive/hostname@EXAMPLE.COM \
    -pricipal user/host@DOMAIN.COM -keytab path/to/user.keytab \
    -refreshTokens
```

Note on refreshTokens: The provided keytab will be copied over to the machine running the H2O Cluster leader node. For this reason, we strongly recommended that both YARN and HDFS be secured with encryption.

Generating the Token in the Driver with Refresh in the Mapper

This approach is a combination of the two previous scenarios. Hive delegation token is first generated by the h2odriver and then periodically refreshed by the H2O Cluster leader node.

This is the best-of-both-worlds approach. The token is generated first in the driver and is available immediately on cluster start. It is then periodically refreshed and never expires.

Requirements:

- The Hive JDBC driver is on the h2o driver and mapper classpaths.
- The `hiveHost`, `hivePrincipal` and optionally `hiveJdbcUrlPattern` arguments are present. (See above for details.)
- The `refreshTokens` argument is present.

Example command:

```
export HADOOP_CLASSPATH=/path/to/hive-jdbc-standalone.jar
hadoop jar h2odriver.jar [-libjars /path/to/hive-jdbc-standalone.jar] \
    -nodes 1 -mapperXmx 4G \
    -hiveHost hostname:10000 -hivePrincipal hive/hostname@EXAMPLE.COM \
    -refreshTokens
```

Using a Delegation Token when Connecting to Hive via JDBC

When running the actual data-load, specify the JDBC URL with the delegation token parameter:

R Python

```
my_citibike_data <- h2o.import_sql_table(
    "jdbc:hive2://hostname:10000/default;auth=delegationToken",
    "citibike20k", "", ""
)
```