Machine Learning with Sparkling Water: H2O + Spark

Michal Malohlava Jakub Hava Nidhi Mehta

Edited by: Vinod Iyengar & Angela Bartz

---

http://h2o.ai/resources

March 2018: Third Edition

Machine Learning with Sparkling Water: H2O + Spark
by Michal Malohlava, Jakub Hava, & Nidhi Mehta
Edited by: Vinod Iyengar & Angela Bartz

Published by H2O.ai, Inc.
2307 Leghorn St.
Mountain View, CA 94043

March 2018: Third Edition

Photos by ©H2O.ai, Inc.

Printed in the United States of America.

Contents

What is H2O?

H2O.ai is focused on bringing AI to businesses through software. Its flagship product is H2O, the leading open source platform that makes it easy for financial services, insurance companies, and healthcare companies to deploy AI and deep learning to solve complex problems. More than 9,000 organizations and 80,000+ data scientists depend on H2O for critical applications like predictive maintenance and operational intelligence. The company – which was recently named to the CB Insights AI 100 – is used by 169 Fortune 500 enterprises, including 8 of the world's 10 largest banks, 7 of the 10 largest insurance companies, and 4 of the top 10 healthcare companies. Notable customers include Capital One, Progressive Insurance, Transamerica, Comcast, Nielsen Catalina Solutions, Macy's, Walgreens, and Kaiser Permanente.

Using in-memory compression, H2O handles billions of data rows in-memory, even with a small cluster. To make it easier for non-engineers to create complete analytic workflows, H2O's platform includes interfaces for R, Python, Scala, Java, JSON, and CoffeeScript/JavaScript, as well as a built-in web interface, Flow. H2O is designed to run in standalone mode, on Hadoop, or within a Spark Cluster, and typically deploys within minutes.

H2O includes many common machine learning algorithms, such as generalized linear modeling (linear regression, logistic regression, etc.), Naïve Bayes, principal components analysis, k-means clustering, and word2vec. H2O implements best-in-class algorithms at scale, such as distributed random forest, gradient boosting, and deep learning. H2O also includes a Stacked Ensembles method, which finds the optimal combination of a collection of prediction algorithms using a process known as "stacking." With H2O, customers can build thousands of models and compare the results to get the best predictions.

H2O is nurturing a grassroots movement of physicists, mathematicians, and computer scientists to herald the new wave of discovery with data science by collaborating closely with academic researchers and industrial data scientists. Stanford university giants Stephen Boyd, Trevor Hastie, and Rob Tibshirani advise the H2O team on building scalable machine learning algorithms. And with hundreds of meetups over the past several years, H2O continues to remain a word-of-mouth phenomenon.

Try it out

- Download H2O directly at `http://h2o.ai/download`.
- Install H2O's R package from CRAN at `https://cran.r-project.org/web/packages/h2o/`.

- Install the Python package from PyPI at `https://pypi.python.org/pypi/h2o/`.

Join the community

- To learn about our training sessions, hackathons, and product updates, visit `http://h2o.ai`.
- To learn about our meetups, visit `https://www.meetup.com/topics/h2o/all/`.
- Have questions? Post them on Stack Overflow using the **h2o** tag at `http://stackoverflow.com/questions/tagged/h2o`.
- Have a Google account (such as Gmail or Google+)? Join the open source community forum at `https://groups.google.com/d/forum/h2ostream`.
- Join the chat at `https://gitter.im/h2oai/h2o-3`.

Sparkling Water Introduction

Sparkling Water allows users to combine the fast, scalable machine learning algorithms of H2O with the capabilities of Spark. With Sparkling Water, users can drive computation from Scala, R, or Python and use the H2O Flow UI, providing an ideal machine learning platform for application developers.

Spark is an elegant and powerful general-purpose, open-source, in-memory platform with tremendous momentum. H2O is an in-memory application for machine learning that is reshaping how people apply math and predictive analytics to their business problems.

Integrating these two open-source environments provides a seamless experience for users who want to make a query using Spark SQL, feed the results into H2O to build a model and make predictions, and then use the results again in Spark. For any given problem, better interoperability between tools provides a better experience.

For additional examples, please visit the Sparkling Water GitHub repository at `https://github.com/h2oai/sparkling-water/tree/master/examples`.

Have Questions about Sparkling Water?

- Post them on Stack Overflow using the **sparkling-water** tag at `http://stackoverflow.com/questions/tagged/sparkling-water`.
- Join the chat at `https://gitter.im/h2oai/sparkling-water`.

Typical Use Cases

Sparkling Water excels in leveraging existing Spark-based workflows needed to call advanced machine learning algorithms. We identified three the most common use-cases which are described below.

Model Building

A typical example involves multiple data transformations with help of Spark API, where a final form of data is transformed into H2O frame and passed to an H2O algorithm. The constructed model estimates different metrics based on the testing data or gives a prediction that can be used in the rest of the data pipeline (see Figure 1).

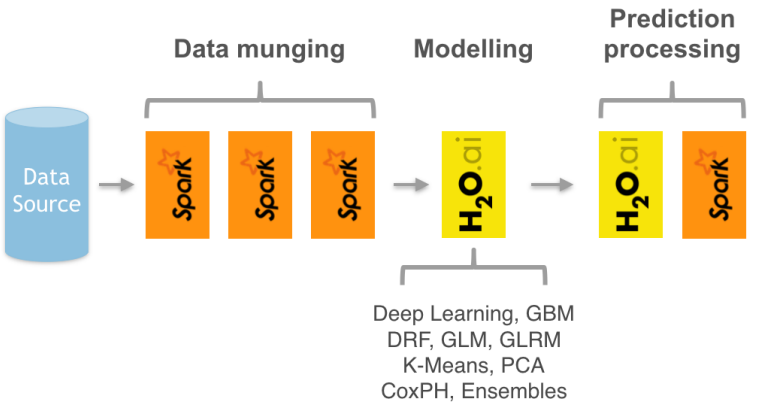


Figure 1: Sparkling Water extends existing Spark data pipeline with advanced machine learning algorithms.

Data Munging

Another use-case includes Sparkling Water as a provider of ad-hoc data transformations. Figure 2 shows a data pipeline benefiting from H2O's parallel data load and parse capabilities, while Spark API is used as another provider of data transformations. Furthermore, H2O can be used as in-place data transformer.

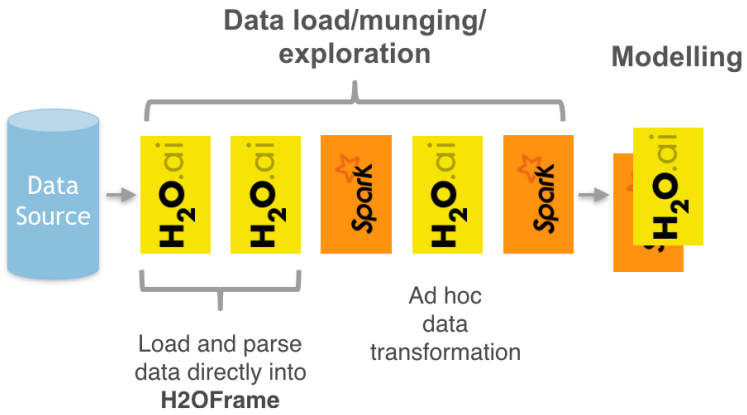


Figure 2: Sparkling Water introduces H2O parallel load and parse into Spark pipelines.

Stream Processing

The last use-case depicted on Figure 3 introduces two data pipelines. The first one, called an off-line training pipeline, is invoked regularly (e.g., every hour or every day), utilizes Spark as well as H2O API and provides an H2O model as output. The H2O API allows the model to be exported in a form independent on H2O run-time. The second one processes streaming data (with help of Spark Streaming or Storm) and utilizes the model trained in the first pipeline to score the incoming data. Since the model is exported with no run-time dependency to H2O, the streaming pipeline can be lightweight and independent on H2O or Sparkling Water infrastructure.

Features

Sparkling Water provides transparent integration for the H2O engine and its machine learning algorithms into the Spark platform, enabling:

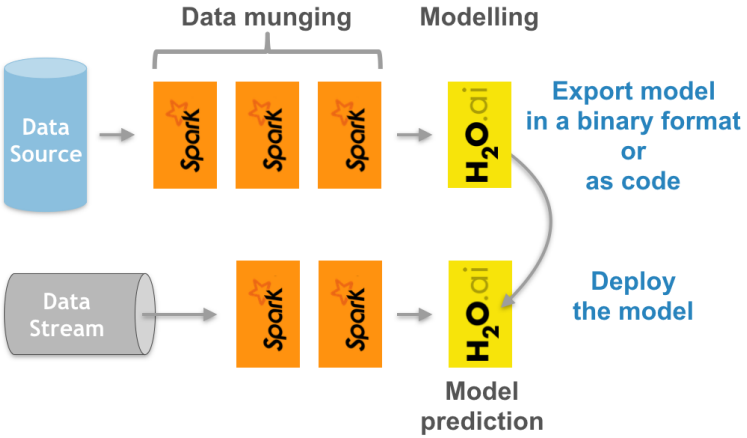


Figure 3: Sparkling Water used as an off-line model producer feeding models into a stream-based data pipeline.

- Use of H2O algorithms in Spark workflow
- Transformation between H2O and Spark data structures
- Use of Spark RDDs, DataFrames and Datasets as input for H2O algorithms
- Use of H2OFrames as input for MLlib algorithms
- Transparent execution of Sparkling Water applications on top of Spark
- Use H2O algorithms in Spark pipelines

Supported Data Sources

Currently, Sparkling Water can use the following data source types:

- Standard Resilient Distributed Dataset (RDD) API for loading data and transforming it into H2OFrames
- H2O API for loading data directly into H2OFrame from file(s) stored on:
  - local filesystems
  - HDFS
  - S3
  - HTTP/HTTPS

For more details, please refer to the H2O documentation at `http://docs.h2o.ai`.

Supported Data Formats

Sparkling Water can read data stored in the following formats:

- CSV
- SVMLight
- ARFF
- Parquet

For more details, please refer to the H2O documentation at `http://docs.h2o.ai`.

Supported Spark Execution Environments

Sparkling Water can run on top of Spark in the following ways:

- as a local cluster (where the master node is `local` or `local[*]`)
- as a standalone cluster[1]
- in a YARN environment[2]

[1] Refer to the Spark standalone documentation `http://spark.apache.org/docs/latest/spark-standalone.html`

[2] Refer to the Spark YARN documentation `http://spark.apache.org/docs/latest/running-on-yarn.html`

Design

Sparkling Water is designed to be executed as a regular Spark application. It provides a way to initialize H2O services on Spark and access data stored in data structures of Spark and H2O.

Sparkling Water supports two type of backends. In the internal backend, Sparkling Water is launched inside a Spark executor, which is created after application submission. At this point, H2O starts services, including distributed key-value (K/V) store and memory manager, and orchestrates them into a cloud. The topology of the created cloud matches the topology of the underlying Spark cluster exactly. The following figure represents the Internal Sparkling Water cluster.

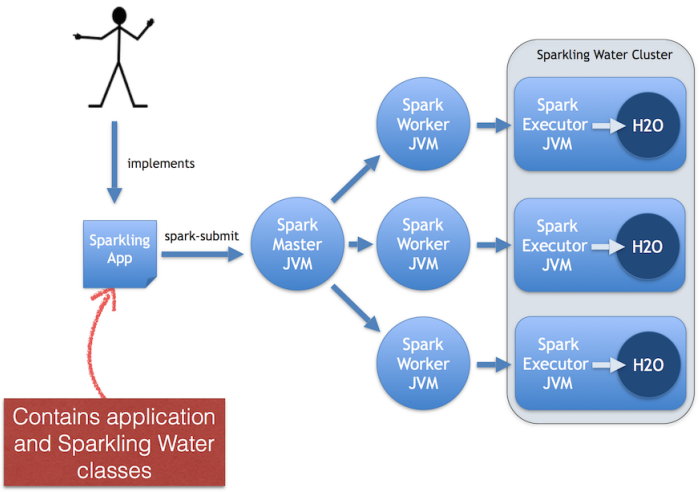


Figure 4: Sparkling Water design depicting deployment of the Sparkling Water in internal backend to the standalone Spark cluster.

In external backend, the H2O cluster is started separately and is connected to from the Spark driver. The following figure represents the External Sparkling Water cluster.

More information about backends is available in the next section.

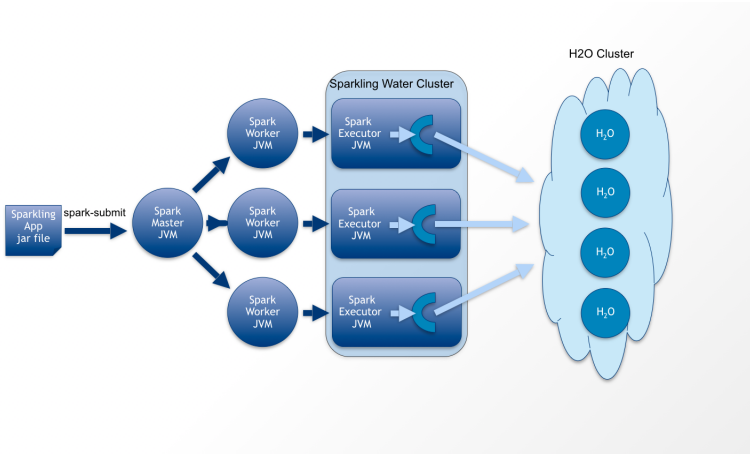


Figure 5: Sparkling Water design depicting deployment of the Sparkling Water in internal backend to the standalone Spark cluster.

Data Sharing between Spark and H2O

Sparkling Water enables transformation between different types of RDDs and H2O's `H2OFrame`, and vice versa.

When converting from an `H2OFrame` to an RDD, a wrapper is created around the `H2OFrame` to provide an RDD-like API. In this case, data is not duplicated but served directly from the underlying `H2OFrame`.

Converting from an RDD/DataFrame to an `H2OFrame` requires data duplication because it transfers data from the RDD storage into `H2OFrame`. However, data stored in an `H2OFrame` is heavily compressed and does not need to be preserved in RDD.

Provided Primitives

Sparkling Water provides several primitives (Table 1), which are the basic components used by Spark components.

Before using H2O algorithms and data structures, the first step is to create and start the `H2OContext` instance using the `val hc = new H2OContext.getOrCreate(spark)` call. `H2OContext` contains the necessary information for running H2O services and exposes meth-

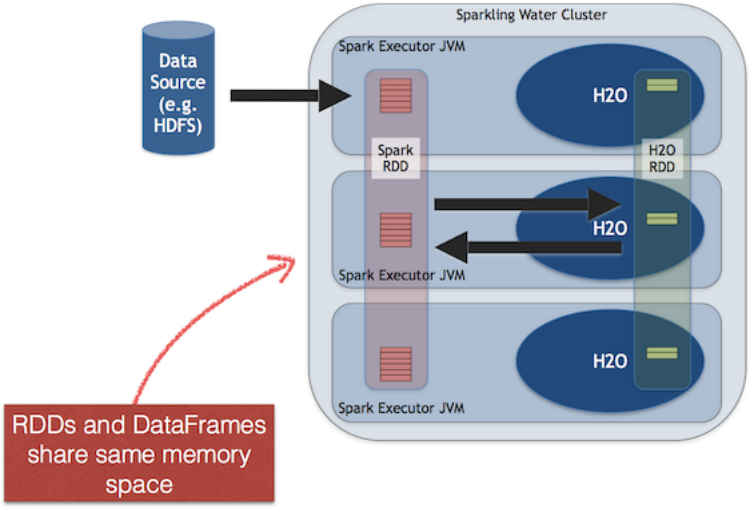


Figure 6: Sharing between Spark and H2O inside an executor JVM.

ods for data transformation between the Spark RDD, `DataFrame` or `Dataset`, and the `H2OFrame`. Starting `H2OContext` involves an operation that:

- In case of internal backend, is distributed and contacts all accessible Spark executor nodes and initializes H2O services (such as the key-value store and RPC) inside the executors' JVMs.
- In case of external backend, either starts H2O cluster on YARN and connects to it or connects to existing H2O cluster right away (depends on the configuration).

Concept	API Implementation	Description
H2O Context	`H2OContext`	Contains H2O and Sparkling Water state, provides primitives to publish data in Spark as `H2OFrame` and vice versa. It follows design principles of Spark primitives such as `SparkSession`, `SparkContext` or `SQLContext`.
H2O Entry Point	`water.H2O`	Represents the entry point for accessing H2O services. Contains information about running H2O services, including a list of nodes and the status of the distributed K/V datastore.
H2O Frame	`water.fvec.H2OFrame`	A distributed data structure representing a table of values. The table is column-based and provides column and row accessors.
H2O Algorithm	package `hex`	Represents the H2O machine learning algorithms library, including DeepLearning, GBM, GLM, DRF, and other algorithms.

Table 1: Sparkling Water primitives

When `H2OContext` is running, H2O data structures and algorithms can be manipulated. The key data structure is `H2OFrame`, which represents a distributed table composed of vectors. A new `H2OFrame` can be created using one of the following methods:

- loading a cluster local file (a file located on each node of the cluster):

```
1 val h2oFrame = new H2OFrame(new File("/data/iris.
    csv"))
```

- loading a file from HDFS/S3/S3N/S3A:

```
1 val h2oFrame = new H2OFrame(URI.create("hdfs://
    data/iris.csv"))
```

- loading multiple files from HDFS/S3/S3N/S3A:

```
val h2oFrame = new H2OFrame(
    URI.create("hdfs://data/iris/01.csv"),
    URI.create("hdfs://data/iris/02.csv")
)
```

- transforming Spark RDD, `DataFrame` or `Dataset`:

```
val h2oFrame = h2oContext.asH2OFrame(sparkData)
```

- referencing existing `H2OFrame` by its key

```
val h2oFrame = new H2OFrame("iris.hex")
```

When `H2OContext` is running, any H2O algorithm can be called. Most of the provided algorithms are located in the `hex` package. Calling an algorithm is composed of two steps:

- Specifying parameters:

```
val train: H2OFrame = new H2OFrame(new File("
   prostate.csv"))
val gbmParams = new GBMParameters()
gbmParams._train = train
gbmParams._response_column = "CAPSULE"
gbmParams._ntrees = 10
```

- Creating the model builder and launching computations. The `trainModel` method is non-blocking and returns a job representing the computation.

```
val gbmModel = new GBM(gbmParams).trainModel.get
```

Sparkling Water Backends

Internal Backend

In internal backend, H2O cloud is created automatically during the call of `H2OContext.getOrCreate`. Since it's not technically possible to get the number of executors in Spark, Sparkling Water tries to discover all executors at the initiation of `H2OContext` and starts H2O instance inside of each discovered executor. This solution is the easiest to deploy; however when Spark or YARN kills the executor, the whole H2O cluster goes down since H2O doesn't support high availability. The same happens also for the case when a new executors join the cluster as the shape of the H2O cluster can't be changed later. Internal backend is default for behaviour for Sparkling Water. It can be changed via spark configuration property `spark.ext.h2o.backend.cluster.mode` to **external** or **internal**. Another way how to change type of backend is by calling `setExternalClusterMode()` or `setInternalClusterMode()` method on `H2OConf` class instance. `H2OConf` is a simple wrapper around `SparkConf` and inherits all properties in spark configuration.

`H2OContext` can be explicitly started in internal backend mode as

```
1 val conf = new H2OConf(spark).setInternalClusterMode()
2 val h2oContext = H2OContext.getOrCreate(spark, conf)
```

If `spark.ext.h2o.backend.cluster.mode` property was set to **internal** either on the command line or on the `SparkConf`, the following call is sufficient:

```
1 val h2oContext = H2OContext.getOrCreate(spark)
```

or

```
1     val conf = new H2OConf(spark)
2     val h2oContext = H2OContext.getOrCreate(spark,
          conf)
```

if we want to pass some additional H2O configuration to Sparkling Water.

External Backend

In external backend, H2O cluster running separately from the rest of Spark application is used. This separation gives the user more stability since Sparkling

Water is no longer affected by Spark executors being killed, which can lead, as in previous mode, to H2O cloud kill as well. If H2O cluster is deployed on YARN, it is required to start it on a YARN queue with YARN preemption disabled to ensure H2O nodes can't be killed by competing jobs.

There are two deployment strategies of external cluster: manual and automatic (YARN only). In manual mode, the user is responsible for starting H2O cluster and in automatic mode, the cluster is started automatically based on our configuration. In both modes, regular H2O/H2O driver jar can't be used as main artifact for external H2O cluster. Instead a special, JAR file extended by classes required by Sparkling Water need to used.

For the released Sparkling Water versions, the extended H2O jar can be downloaded using the `./bin/get-extendend-h2o.sh` script. This script expects a single argument which specifies the Hadoop version for which the jar is to be obtained.

The following code downloads H2O extended JAR for the cdh5.8:

```
1    ./bin/get-extended-h2o.sh cdh5.8
```

If we don't want to run on hadoop but you want to run H2O in standalone mode, we can get the corresponding extended H2O standalone jar as:

```
1    ./bin/get-extended-h2o.sh standalone
```

If you want to see the list of supported Hadoop versions, just run the shell script without any arguments as:

```
1    ./bin/get-extended-h2o.sh
```

The script downloads the jar to the current directory and prints the absolute path to the downloaded jar.

The following sections explain how to use external cluster in both modes. Let's assume for later sections that the path to the extended H2O/H2O driver jar file is available in `H2O_EXTENDED_JAR` environmental variable.

Manual Mode of External Backend

In this mode, we need to start H2O cluster before connecting to it manually. In general, H2O cluster can be started in two ways - using the multicast discovery of the other nodes and using the flatfile, where we manually specify the future locations of H2O nodes. We recommend to use flatfile to specify the location

of nodes for production usage of Sparkling Water, but in simple environments where multicast is supported, the multicast discovery should work as well. Let's have a look on how to start H2O cluster and connect to it from Sparkling Water in multicast environment. To start H2O cluster of 3 nodes, run the following line 3 times:

```
java -jar $H2O_EXTENDED_JAR -md5skip -name test
```

Don't forget the -md5skip argument, it's additional argument required for the external backend to work. After this step, we should have H2O cluster of 3 nodes running and the nodes should have discovered each other using the multicast discovery.

Now, let's start Sparkling Water shell first as `./bin/sparkling-shell` and connect to the cluster:

```
import org.apache.spark.h2o._
val conf = new H2OConf(spark)
    .setExternalClusterMode()
    .useManualClusterStart()
    .setCloudName("test")
val hc = H2OContext.getOrCreate(spark, conf)
```

To connect to existing H2O cluster from Python, start PySparkling shell as `./bin/pysparkling` and do:

```
from pysparkling import *
conf = H2OConf(spark)
    .set_external_cluster_mode()
    .use_manual_cluster_start()
    .set_cloud_name("test")
hc = H2OContext.getOrCreate(spark, conf)
```

To start external H2O cluster where the nodes are discovered using the flatfile, you can run:

```
java -jar $H2O_EXTENDED_JAR -md5skip -name test -
   flatfile path_to_flatfile
```

The flatfile should contain lines in format ip:port of nodes where H2O is supposed to run. To read more about flatfile and it's format, please see H2O's flatfile configuration property available at `https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/product/howto/H2O-DevCmdLine.`

md#flatfile. To connect to this external cluster, run the following commands in the corresponding shell (Sparkling in case of Scala, PySparkling in case of Python):

Scala:

```
import org.apache.spark.h2o._
val conf = new H2OConf(spark)
    .setExternalClusterMode()
    .useManualClusterStart()
    .setH2OCluster("ip", port)
    .setCloudName("test")
val hc = H2OContext.getOrCreate(spark, conf)
```

Python:

```
from pysparkling import *
conf = H2OConf(spark)
    .set_external_cluster_mode()
    .use_manual_cluster_start()
    .set_h2o_cluster("ip", port)
    .set_cloud_name("test")
hc = H2OContext.getOrCreate(spark, conf)
```

We can see that in this case we are using extra call `setH2OCluster` in Scala and `set_h2o_cluster` in Python. When the external cluster is started via the flatfile approach, we need to give Sparkling Water ip address and port of arbitrary node inside the H2O cloud in order to connect to the cluster. The ip and port of this node are passed as arguments to `setH2OCluster/set_h2o_cluster` method.

It's possible in both cases that node on which want to start Sparkling Watter shell is connected to more networks. In this case it can happen that H2O cloud decides to use addresses from network A, whilst Spark decides to use addresses for its executors and driver from network B. Later, when we start `H2OContext`, the special H2O client, running inside of the Spark Driver, can get the same IP address as the Spark driver and thus the rest of the H2O cloud can't see it. This shouldn't happen in environments where the nodes are connected to only one network, however we provide configuration how to deal with this case as well.

We can use method `setClientIp` in Scala and `set_client_ip` in Python available on `H2OConf` which expects IP address and sets this IP address for the H2O client running inside the Spark driver. The IP address passed to this

method should be address of the node where Spark driver is about to run and should be from the same network as the rest of H2O cloud.

Let's say we have two H2O nodes on addresses 192.168.0.1 and 192.168.0.2 and also assume that Spark driver is available on 172.16.1.1 and the only executor is available on 172.16.1.2. The node with Spark driver is also connected to 192.168.0.x network with address 192.168.0.3.

In this case there is a chance that H2O client will use the address from 172.168.x.x network instead of the 192.168.0.x one, which can lead to the problem that H2O cloud and H2O client can't see each other.

We can force the client to use the correct address using the following configuration:

Scala:

```
import org.apache.spark.h2o._
val conf = new H2OConf(spark)
    .setExternalClusterMode()
    .useManualClusterStart()
    .setH2OCluster("ip", port)
    .setClientIp("192.168.0.3")
    .setCloudName("test")
val hc = H2OContext.getOrCreate(spark, conf)
```

Python:

```
from pysparkling import *
conf = H2OConf(spark)
    .set_external_cluster_mode()
    .use_manual_cluster_start()
    .set_h2o_cluster("ip", port)
    .set_client_ip("192.168.0.3")
    .set_cloud_name("test")
hc = H2OContext.getOrCreate(spark, conf)
```

There is also a less strict configuration `setClientNetworkMask` in Scala and `set_client_network_mask` in Python. Instead of its IP address equivalent, using this method we can force the H2O client to use just a specific network and leave up to the client which IP address from this network to use.

The same configuration can be applied when the H2O cluster has been started via multicast discovery.

Automatic Mode of External Backend

In automatic mode, H2O cluster is started automatically. The cluster can be started automatically only in YARN environment at the moment. We recommend this approach as it is easier to deploy external cluster in this mode ans it is also more suitable for production environments. When H2O cluster is start on YARN, it is started as map reduce job and it always use the flatfile approach for nodes to cloud up.

For this case to work, we need to extend H2O driver for the desired hadoop version as mentioned above. Let's assume the path to this extended H2O driver is stored in `H2O_EXTENDED_JAR` environmental property.

To start H2O cluster and connect to it from Spark application in Scala:

```
import org.apache.spark.h2o._
val conf = new H2OConf(spark)
    .setExternalClusterMode()
    .useAutoClusterStart()
    .setH2ODriverPath("path_to_extended_driver")
    .setNumOfExternalH2ONodes(1)
    .setMapperXmx("2G")
    .setYARNQueue("h2o_yarn_queue")
val hc = H2OContext.getOrCreate(spark, conf)
```

and in Python:

```
from pysparkling import *
conf = H2OConf(spark)
    .set_external_cluster_mode()
    .use_auto_cluster_start()
    .set_h2o_driver_path("path_to_extended_driver")
    .set_num_of_external\_h2o\_nodes(1)
    .set_mapper_xmx("2 G   )
    .set_yarn_queue("h2o_yarn_queue")
hc = H2OContext.getOrCreate(spark, conf)
```

In both cases we can see various configuration methods. We explain only the Scala ones since the python equivalents are doing exactly the same.

- `setH2ODriverPath` method is used to tell Sparkling Water where it can find the extended H2O driver jar. This jar is passed to hadoop and used to start H2O cluster on YARN.

- `setNumOfExternalH2ONodes` method specifies how many H2O nodes we want to start.
- `setMapperXmx` method specifies how much memory each H2O node should have available.
- `setYarnQueue` method specifies YARN queue on which H2O cluster will be started. We highly recommend that this queue should have YARN preemption off in order to have stable H2O cluster.

When using `useAutoClusterStart` we do not need to call `setH2ODriverPath` explicitly in case when `H2O_EXTENDED_JAR` environmental property is set and pointing to that file. In this case Sparkling Water will fetch the path from this variable automatically. Also when `setCloudName` is not called, the name is set automatically and H2O cluster with that name is started.

It can also happen that we might need to use `setClientIp/set_client_ip` method as mentioned in the chapter above for the same reasons. The usage of this method in automatic mode is exactly the as in the manual mode.

Programming API

Starting H2O Services

```
1 import org.apache.spark.h2o._
2 val hc = new H2OContext.getOrCreate(spark)
```

This initiates and starts `H2OContext` in one call and can be used to obtain the already existing `H2OContext`.

Memory Allocation

In case of internal backend, H2O resides in the same executor JVM as Spark and the memory provided for H2O is configured via Spark; refer to Spark configuration for more details. Note that in the external backend, only the H2O client running in the Spark driver is affected by Spark memory configuration. Memory has to be configured explicitly for the rest of the H2O nodes in the external backend.

Generic configuration

- Configure the Executor memory (i.e., memory available for H2O in internal backend) via the Spark configuration property `spark.executor.memory`. For example, `bin/sparkling-shell --conf spark.executor.memory=5g` or configure the property in `$SPARK_HOME/conf/spark-defaults.conf`
- Configure the Driver memory (i.e., memory available for H2O client running inside the Spark driver) via the Spark configuration property `spark.driver.memory`. For example, `bin/sparkling-shell --conf spark.driver.memory=4g` or configure the property in `$SPARK_HOME/conf/spark-defaults.conf`.

YARN-specific configuration

- Refer to the Spark documentation `https://spark.apache.org/docs/latest/running-on-yarn.html`
- For JVMs that require a large amount of memory, we strongly recommend configuring the maximum amount of memory available for individual mappers.

Converting H2OFrame into RDD[T]

The `H2OContext` class provides the explicit conversion, `asRDD`, which creates an RDD-like wrapper around the provided `H2OFrame`:

```
1 def asRDD[A <: Product: TypeTag: ClassTag](fr:
    H2OFrame): RDD[A]
```

The call expects the type `A` to create a correctly-typed RDD. The conversion requires type `A` to be bound by `Product` interface. The relationship between the columns of `H2OFrame` and the attributes of class `A` is based on name matching.

Example

```
1 val df: H2OFrame = ...
2 val rdd = asRDD[Weather](df)
```

Converting H2OFrame into DataFrame

The `H2OContext` class provides the explicit conversion, `asDataFrame`, which creates a `DataFrame`-like wrapper around the provided `H2OFrame`. Technically, it provides the `RDD[sql.Row]` RDD API:

```
1 def asDataFrame(fr: H2OFrame)(implicit sqlContext:
    SQLContext): DataFrame
```

This call does not require any type of parameters, but since it creates `DataFrame` instances, it requires access to an instance of `SQLContext`. In this case, the instance is provided as an implicit parameter of the call. The parameter can be passed in two ways: as an explicit parameter or by introducing an implicit variable into the current context.

The schema of the created instance of the `DataFrame` is derived from the column name and the types of `H2OFrame` specified.

Example

Using an explicit parameter in the call to pass `sqlContext`:

```
1 val sqlContext = new SQLContext(sc)
2 val schemaRDD = asDataFrame(h2oFrame)(sqlContext)
```

or as implicit variable provided by actual environment:

```
implicit val sqlContext = new SQLContext(sc)
val schemaRDD = asDataFrame(h2oFrame)
```

Converting RDD[T] into H2OFrame

The `H2OContext` provides implicit conversion from the specified `RDD[A]` to `H2OFrame`. As with conversion in the opposite direction, the type `A` has to satisfy the upper bound expressed by the type `Product`. The conversion will create a new `H2OFrame`, transfer data from the specified RDD, and save it to the H2O K/V data store.

```
implicit def asH2OFrame[A <: Product: TypeTag](rdd:
    RDD[A]): H2OFrame
```

The API also provides explicit version which allows for specifying name for resulting `H2OFrame`.

```
def asH2OFrame[A <: Product: TypeTag](rdd: RDD[A],
    frameName: String): H2OFrame
```

Example

```
val rdd: RDD[Weather] = ...
import h2oContext._
// Implicit call of h2oContext.asH2OFrame[Weather](rdd
    ) is used
val hf: H2OFrame = rdd
// Explicit call of of H2OContext API with name for
    resulting H2OFrame
val hfNamed: H2OFrame = h2oContext.asH2OFrame(rdd, "
    hfNamed")
```

Converting DataFrame into H2OFrame

The `H2OContext` provides **implicit** conversion from the specified `DataFrame` to `H2OFrame`. The conversion will create a new `H2OFrame`, transfer data from the specified `DataFrame`, and save it to the H2O K/V data store.

```
implicit def asH2OFrame(rdd: DataFrame): H2OFrame
```

The API also provides explicit version which allows for specifying name for resulting `H2OFrame`.

```
1 def asH2OFrame(rdd: DataFrame, frameName: String):
    H2OFrame
```

Example

```
1 val df: DataFrame = ...
2 import h2oContext._
3 // Implicit call of h2oContext.asH2OFrame(srdd) is
    used
4 val hf: H2OFrame = df
5 // Explicit call of h2oContext API with name for
    resulting H2OFrame
6 val hfNamed: H2OFrame = h2oContext.asH2OFrame(df, "
    hfNamed")
```

Creating H2OFrame from an Existing Key

If the H2O cluster already contains a loaded `H2OFrame` referenced by the key `train.hex`, it is possible to reference it from Sparkling Water by creating a proxy `H2OFrame` instance using the key as the input:

```
1 val trainHF = new H2OFrame("train.hex")
```

Type Map Between H2OFrame and Spark DataFrame Types

For all primitive Scala types or Spark SQL types (see `org.apache.spark.sql.types`) which can be part of Spark RDD/DataFrame/-Dataset, we provide mapping into H2O vector types (numeric, categorical, string, time, UUID - see `water.fvec.Vec`):

Type mapping between H2O H2OFrame types and RDD[T] types

As type `T` we support following types:

Scala type	SQL type	H2O type
NA	BinaryType	Numeric
Byte	ByteType	Numeric
Short	ShortType	Numeric
Integer	IntegerType	Numeric
Long	LongType	Numeric
Float	FloatType	Numeric
Double	DoubleType	Numeric
String	StringType	String
Boolean	BooleanType	Numeric
java.sql.TimeStamp	TimestampType	Time

T
NA
Byte
Short
Integer
Long
Float
Double
String
Boolean
java.sql.Timestamp
Any scala class extending scala `Product`
org.apache.spark.mllib.regression.LabeledPoint
org.apache.spark.ml.linalg.Vector
org.apache.spark.mllib.linalg

Calling H2O Algorithms

1. Create the parameters object that holds references to input data and parameters specific for the algorithm:

```
1 val train: RDD = ...
2 val valid: H2OFrame = ...
3
4 val gbmParams = new GBMParameters()
5 gbmParams._train = train
6 gbmParams._valid = valid
7 gbmParams._response_column = "bikes"
```

```
8 gbmParams._ntrees = 500
9 gbmParams._max_depth = 6
```

2. Create a model builder:

```
1 val gbm = new GBM(gbmParams)
```

3. Invoke the model build job and block until the end of computation (`trainModel` is an asynchronous call by default):

```
1 val gbmModel = gbm.trainModel.get
```

Using Spark Data Sources with H2OFrame

Spark SQL provides configurable data source for SQL tables. Sparkling Water enable `H2OFrame` to be used as data source to load/save data from/to Spark SQL table.

Reading from `H2OFrame`

Let's suppose we have a `H2OFrame`. The shortest way to load a `DataFrame` from `H2OFrame` with default settings is:

```
1 val df = spark.read.h2o(frame.key)
```

There are two more ways to load a `DataFrame` from `H2OFrame` allowing us to specify additional options:

```
1 val df = spark.read.format("h2o").option("key", frame.
    key.toString).load()
```

or

```
1 val df = spark.read.format("h2o").load(frame.key.
    toString)
```

Saving to `H2OFrame`

Let's suppose we have `DataFrame` df. The shortest way to save the `DataFrame` as `H2OFrame` with default settings is:

```
df.write.h2o("new_key")
```

There are two more ways to save the `DataFrame` as `H2OFrame` allowing us to specify additional options:

```
df.write.format("h2o").option("key", "new_key").save()
```

or

```
df.write.format("h2o").save("new_key")
```

All three variants save the `DataFrame` as `H2OFrame` with the key "new_key". They won't succeed if a `H2OFrame` with the same key already exists.

Loading and Saving Options

If the key is specified as 'key' option, and also in the load/save method, the option 'key' is preferred:

```
val df = spark.read.from("h2o").option("key", "key_one
    ").load("key_two")
```

or

```
val df = spark.read.from("h2o").option("key", "key_one
    ").save("key_two")
```

In both examples, "key_one" is used.

Specifying Saving Mode

There are four save modes available when saving data using Data Source API- see `http://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes`

- If **append** mode is used, an existing `H2OFrame` with the same key is deleted, and a new one created with the same key. The new frame contains the union of all rows from the original `H2OFrame` and the appended `DataFrame`.

- If **overwrite** mode is used, an existing `H2OFrame` with the same key is deleted, and new one with the new rows is created with the same key.
- If **error** mode is used, and a `H2OFrame` with the specified key already exists, an exception is thrown.
- If **ignore** mode is used, and a `H2OFrame` with the specified key already exists, no data are changed.

Deployment

Since Sparkling Water is designed as a regular Spark application, its deployment cycle is strictly driven by Spark deployment strategies (refer to Spark documentation[3]). Spark applications are deployed by the `spark-submit` [4] script that handles all deployment scenarios:

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --conf <key>=<value> \
  ... # other options \
  <application-jar> [application-arguments]
```

- `--class`: Name of main class with `main` method to be executed. For example, the `water.SparklingWaterDriver` application launches H2O services.
- `--master`: Location of Spark cluster
- `--conf`: Specifies any configuration property using the format `key=value`
- `application-jar`: Jar file with all classes and dependencies required for application execution
- `application-arguments`: Arguments passed to the main method of the class via the `--class` option

Referencing Sparkling Water

Using Fatjar

The Sparkling Water archive provided at `http://h2o.ai/download` contains a Fatjar with all classes required for Sparkling Water run.

An application submission with Sparkling Water Fatjar is using the `--jars` option which references included fatjar.

[3] Spark deployment guide `http://spark.apache.org/docs/latest/cluster-overview.html`

[4] Submitting Spark applications `http://spark.apache.org/docs/latest/submitting-applications.html`

```
$SPARK_HOME/bin/spark-submit \
  --jars assembly/build/libs/sparkling-water-assembly
     -2.2.2-all.jar \
  --class org.apache.spark.examples.h2o.
     CraigslistJobTitlesStreamingApp \
  /dev/null
```

Using the Spark Package

Sparkling Water is also published as a Spark package. The benefit of using the package is that you can use it directly from your Spark distribution without need to download Sparkling Water.

For example, if you have Spark version 2.2.0 and would like to use Sparkling Water version 2.2.2 and launch example `CraigslistJobTitlesStreamingApp`, then you can use the following command:

```
$SPARK_HOME/bin/spark-submit \
  --packages ai.h2o:sparkling-water-core_2.11:2.2.2,ai
     .h2o:sparkling-water-examples_2.11:2.2.2,no.priv
     .garshol.duke:duke:1.2 \
  --class org.apache.spark.examples.h2o.
     CraigslistJobTitlesStreamingApp \
  /dev/null
```

Note: The duke library has to be explicitly added when using Sparkling Water via the `--packages` option. The library contains a pom file, which the Spark dependency resolver can't properly resolve, and therefore this library will be missing if not explicitly specified. This is only required in Sparkling Water 2.2.2 and older, 2.1.16 and older, and 2.0.17 and older. In newer Sparkling Water versions, the only dependency that needs to be put into `--packages` is `sparkling-water-package_2.11:{version}`. This package contains all the required dependencies.

The Spark option `--packages` points to coordinate of published Sparkling Water package in Maven repository.

The similar command works for spark-shell:

```
$SPARK_HOME/bin/spark-shell \
 --packages ai.h2o:sparkling-water-core_2.11:2.2.2,ai.
    h2o:sparkling-water-examples_2.11:2.2.2,no.priv.
    garshol.duke:duke:1.2
```

Note: When you are using Spark packages, you do not need to download Sparkling Water distribution. Spark installation is sufficient.

Target Deployment Environments

Sparkling Water supports deployments to the following Spark cluster types:

- Local cluster
- Standalone cluster
- YARN cluster

Local cluster

The local cluster is identified by the following master URLs - `local`, `local[K]`, or `local[*]`. In this case, the cluster is composed of a single JVM and is created during application submission.

For example, the following command will run the ChicagoCrimeApp application inside a single JVM with a heap size of 5g:

```
$SPARK_HOME/bin/spark-submit \
  --conf spark.executor.memory=5g \
  --conf spark.driver.memory=5g \
  --master local[*] \
  --packages ai.h2o:sparkling-water-examples_2
     .11:2.2.2,no.priv.garshol.duke:duke:1.2 \
  --class org.apache.spark.examples.h2o.
     ChicagoCrimeApp \
  /dev/null
```

On a Standalone Cluster

For AWS deployments or local private clusters, the standalone cluster deployment[5] is typical. Additionally, a Spark standalone cluster is also provided by Hadoop distributions like CDH or HDP. The cluster is identified by the URL `spark://IP:PORT`.

The following command deploys the `ChicagoCrimeApp` on a standalone cluster where the master node is exposed on IP `machine-foo.bar.com` and port `7077`:

```
1 $SPARK_HOME/bin/spark-submit \
2   --conf spark.executor.memory=5g \
3   --conf spark.driver.memory=5g \
4   --master spark://machine-foo.bar.com:7077 \
5   --packages ai.h2o:sparkling-water-examples_2
        .11:2.2.2,no.priv.garshol.duke:duke:1.2 \
6   --class org.apache.spark.examples.h2o.
        ChicagoCrimeApp \
7   /dev/null
```

In this case, the standalone Spark cluster must be configured to provide the requested 5g of memory per executor node.

On a YARN Cluster

Because it provides effective resource management and control, most production environments use YARN for cluster deployment.[6] In this case, the environment must contain the shell variable `HADOOP_CONF_DIR` or `YARN_CONF_DIR` which point to Hadoop configuration directory (e.g., `/etc/hadoop/conf`).

```
1 $SPARK_HOME/bin/spark-submit \
2   --conf spark.executor.memory=5g \
3   --conf spark.driver.memory=5g \
4   --num-executors 5 \
5   --master yarn \
6   --deploy-mode client
7   --packages ai.h2o:sparkling-water-examples_2
        .11:2.2.2,no.priv.garshol.duke:duke:1.2 \
```

[5] Refer to Spark documentation `http://spark.apache.org/docs/latest/spark-standalone.html`

[6] See Spark documentation `http://spark.apache.org/docs/latest/running-on-yarn.html`

```
8   --class org.apache.spark.examples.h2o.
        ChicagoCrimeApp \
9   /dev/null
```

The command in the example above creates a YARN job and requests for 5 nodes, each with 5G of memory. Master is set to `yarn`, and together with the deploy mode `client` option forces the driver to run in the client process.

DataBricks Cloud

This section describes how to use Sparkling Water and PySparkling with DataBricks. The first part describes how to create a cluster for Sparkling Water/PySparkling and then discusses how to use Sparkling Water and PySparkling in Databricks.

DataBricks cloud is Integrated with Sparkling Water and Pysparkling. Currently, only internal Sparkling Water backend may be used.

Creating a Cluster

Requirements:

- Databricks Account
- AWS Account

Steps:

1. In Databricks, click **Create Cluster** in the Clusters dashboard.
2. Select your Databricks Runtime Version. (Note that in our demos, we are using '3.0 (includes Apache Spark 2.2.0, Scala 2.11)'.)
3. Select at least 3 workers.
4. Select 0 on-demand workers. On demand workers are currently not supported with Sparkling Water.
5. In the SSH tab, upload your public key. You can create a public key by running the below command in a terminal session:

```
1  ssh-keygen -t rsa -b 4096 -C "your_email@example.
       com"
```

6. Click **Create Cluster**

7. Once the cluster has started, run the following command in a terminal session:

```
ssh ubuntu@<ec-2 driver host>.compute.amazonaws.
    com -p 2200 -i <path to your public/private
    key> -L 54321:localhost:54321
```

This will allow you to use the Flow UI.

(You can find the 'ec-2 driver host' information in the SSH tab of the cluster.)

Running Sparkling Water

Requirements:

- Sparkling Water Jar

Steps:

1. Create a new library containing the Sparkling Water jar.
2. Download the selected Sparkling Water version from `https://www.h2o.ai/download/`.
3. The jar file is located in the sparkling water zip file at the following location: 'assembly/build/libs/sparkling-water-assembly_*-all.jar'
4. Attach the Sparkling Water library to the cluster.
5. Create a new Scala notebook.
6. Create an H2O cloud inside the Spark cluster:

```
import org.apache.spark.h2o._
val h2oConf = new H2OConf(spark).set("spark.ui.
    enabled", "false")
val h2oContext = H2OContext.getOrCreate(spark,
    h2oConf)
```

You can access Flow by going to localhost:54321.

Running PySparkling

Requirements:

- PySparkling zip file
- Python Module: request
- Python Module: tabulate
- Python Module: future
- Python Module: colorama

Steps:

1. Create a new Python library containing the PySparkling zip file.
2. Download the selected Sparkling Water version from `https://www.h2o.ai/download/`.
3. The PySparkling zip file is located in the sparkling water zip file at the following location: 'py/build/dist/h2o_pysparkling_*.zip.'
4. Create libraries for the following python modules: request, tabulate, future and colorama.
5. Attach the PySparkling library and python modules to the cluster.
6. Create a new python notebook.
7. Create an H2O cloud inside the Spark cluster:

```
from pysparkling import *
h2oConf = H2OConf(spark).set("spark.ui.enabled", "
    false")
hc = H2OContext.getOrCreate(spark, h2oConf)
```

To prevent a progress bar error, run the following cell at the beginning of the python notebook:

```
# Patch to workaround progress bar error (this cell
    must be run before using h2o)
import sys

def isatty(self):
return False

type(sys.stdout).isatty = isatty
```

```
8 type(sys.stdout).encoding = "UTF-8"
```

Note: This is only required in Sparkling Water 2.2.2 and older, 2.1.16 and older, and 2.0.17 and older. This is fixed in newer Sparkling Water versions.

Sparkling Water Configuration Properties

The following configuration properties can be passed to Spark to configure Sparking Water:

Configuration Properties not Dependent on Selected Backend

Backend-independent generic parameters

Property name	Default	Description
spark.ext.h2o.backend.cluster.mode	internal	This option can be set either to `internal` or `external`. When set to `external`, H2O Context is created by connecting to existing H2O cluster, otherwise H2O cluster located inside Spark is created. That means that each Spark executor will have one H2O instance running in it. The `internal` mode is not recommended for big clusters and clusters where Spark executors are not stable.
spark.ext.h2o.cloud.name	Generated unique name	Name of H2O cluster.
spark.ext.h2o.nthreads	-1	Limit for number of threads used by H2O. -1 means unlimited.
spark.ext.h2o.disable.ga	true	Disable Google Analytics tracking for embedded H2O.
spark.ext.h2o.repl.enabled	true	Decides whether H2O REPL is initiated.

spark.ext.scala.int.default.num	1	Number of parallel REPL sessions started at the start of Sparkling Water.
spark.ext.h2o.topology.change.listener.enabled	true	Decides whether the listener the kills the H2O cluster upon the change of the underlying cluster's topology is enabled or not.
spark.ext.h2o.spark.version.check.enabled	true	Enables check if runtime Spark version matches build time Spark version.
spark.ext.h2o.fail.on.unsupported.spark.param	true	If unsupported Spark parameter is detected, then the application is forced to shutdown.
spark.ext.h2o.jks	None	Path to Java KeyStore file.
spark.ext.h2o.jks.pass	None	Password for Java KeyStore file.
spark.ext.h2o.hash.login	false	Enable hash login.
spark.ext.h2o.ldap.login	false	Enable LDAP login.
spark.ext.h2o.kerberos.login	false	Enable Kerberos login.
spark.ext.h2o.login.conf	None	Login configuration file.
spark.ext.h2o.user.name	None	Override user name for cluster.
spark.ext.h2o.internal_security_conf	None	Path to a file containing H2O or Sparkling Water internal security configuration.
spark.ext.h2o.node.log.level	INFO	Set H2O node internal logging level.
spark.ext.h2o.node.log.dir	user.dir/h2ologs/SparkAppId or YARN container dir	Location of h2o logs on executor machine.
spark.ext.h2o.ui.update.interval	10000ms	Interval for updates of the Spark UI and History server in milliseconds.
spark.ext.h2o.cloud.timeout	60x1000	Timeout (in msec) for cluster formation.

spark.ext.h2o.node.enable.web	false	Enable or disable web on H2O worker nodes. It is disabled by default for security reasons.

Backend-independent H2O client parameters

Property name	Default	Description
spark.ext.h2o.client.flow.dir	None	Directory where flows from H2O Flow are saved.
spark.ext.h2o.client.ip	None	IP of H2O client node.
spark.ext.h2o.client.iced.dir	None	Location of iced directory for the driver instance.
spark.ext.h2o.client.log.level	INFO	Set H2O client internal logging level (running inside Spark driver).
spark.ext.h2o.client.log.dir	user.dir/h2ologs/SparkAppId	Location of h2o logs on driver machine.
spark.ext.h2o.client.port.base	54321	Port on which H2O client publishes its API. If already occupied, the next odd port is tried on so on.
spark.ext.h2o.client.web.port	-1	Exact client port to access web UI. -1 triggers automatic search for free port starting at spark.ext.h2o.port.base.
spark.ext.h2o.client.verbose	false	The client outputs verbose log output directly into console. Enabling the flag increases the client log level to INFO.
spark.ext.h2o.client.network.mask	None	Subnet selector for H2O client, this disables using IP reported by Spark but tries to find IP based on the specified mask.

Internal Backend Configuration Properties

Internal backend generic parameters

Property name	Default	Description
spark.ext.h2o.flatfile	true	Use flatfile instead of multicast approach for creating H2O cluster.
spark.ext.h2o.cluster.size	None	Expected number of workers of H2O cluster. Value None means automatic detection of cluster size. This number must be equal to number of Spark executors.
spark.ext.h2o.dummy.rdd.mul.factor	10	Multiplication factor for dummy RDD generation. Size of dummy RDD is `spark.ext.h2o.cluster.size` * `spark.ext.h2o.dummy.rdd.mul.factor`.
spark.ext.h2o.spreadrdd.retries	10	Number of retries for creation of an RDD spread across all existing Spark executors.
spark.ext.h2o.default.cluster.size	20	Starting size of cluster in case that size is not explicitly configured.
spark.ext.h2o.subseq.tries	5	Subsequent successful tries to figure out size of Spark cluster, which are producing the same number of nodes.
spark.ext.h2o.internal_secure_connections	false	Enables secure communications among H2O nodes. The security is based on automatically generated keystore and truststore. This is equivalent for `-internal_secure_conections` option in H2O Hadoop deployments.

Internal backend H2O node parameters

Property name	Default	Description
spark.ext.h2o.node.port.base	54321	Base port used for individual H2O nodes.

spark.ext.h2o.node.iced.dir	None	Location of iced directory for H2O nodes on the Spark executors.
spark.ext.h2o.node.network.mask	None	Subnet selector for H2O running inside Spark executors. This disables using IP reported by Spark but tries to find IP based on the specified mask.

External Backend Configuration Properties

External backend parameters

Property name	Default	Description
spark.ext.h2o.cloud.representative	None	ip:port of arbitrary H2O node to identify external H2O cluster.
spark.ext.h2o.external.cluster.num.h2o.nodes	None	Number of H2O nodes to start in auto mode and wait for in manual mode when starting Sparkling Water in external H2O cluster mode.
spark.ext.h2o.cluster.client.retry.timeout	60000ms	Timeout in milliseconds specifying how often the check for availability of connected watchdog client is done.
spark.ext.h2o.cluster.client.connect.timeout	180000ms	Timeout in milliseconds for watchdog client connection. If the client is not connected to the external cluster in the given time, the cluster is killed.
spark.ext.h2o.external.read.confirmation.timeout	60s	Timeout for confirmation of read operation (H2O frame => Spark frame) on external cluster.
spark.ext.h2o.external.write.confirmation.timeout	60s	Timeout for confirmation of write operation (Spark frame => H2O frame) on external cluster.

spark.ext.h2o.cluster.start.timeout	120s	Timeout in seconds for starting H2O external cluster.
spark.ext.h2o.cluster.info.name	None	Full path to a file which is used as the notification file for the startup of external H2O cluster.
spark.ext.h2o.hadoop.memory	6G	Amount of memory assigned to each H2O node on YARN/Hadoop.
spark.ext.h2o.external.hdfs.dir	None	Path to the directory on HDFS used for storing temporary files.
spark.ext.h2o.external.start.mode	manual	If this option is set to `auto` then H2O external cluster is automatically started using the provided H2O driver JAR on YARN, otherwise it is expected that the cluster is started by the user manually.
spark.ext.h2o.external.h2o.driver	None	Path to H2O driver used during auto start mode.
spark.ext.h2o.external.yarn.queue	None	YARN queue on which external H2O cluster is started.

Building a Standalone Application

Sparkling Water Example Project

This is a structure of a simple example project to start coding with Sparkling Water. The source code is available at `https://github.com/h2oai/h2o-droplets/tree/master/sparkling-water-droplet`

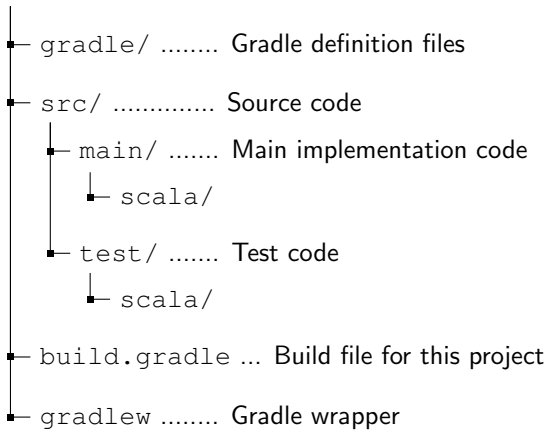
Dependencies

This droplet uses Sparkling Water 2.2, which integrates:

- Spark 2.2
- H2O 3.14.0.7 Weierstrass

For more details see `build.gradle`.

Project structure



Project building

For building, please, use provided gradlew command:

```
./gradlew build
```

Run demo

For running a simple application:

```
./gradlew run
```

Starting with IDEA

There are two ways to open this project in IntelliJ IDEA

Using Gradle build file directly:

Open the project's `build.gradle` in IDEA via File → Open

or using Gradle generated project files:

1. Generate Idea configuration files via `./gradlew idea`
2. Open project in Idea via File → Open

Note: To clean up Idea project files please launch `./gradlew cleanIdea`

Starting with Eclipse

1. Generate Eclipse project files via `./gradlew eclipse`
2. Open project in Eclipse via File → Import → Existing Projects into Workspace

Running tests

To run tests, please, run:

```
./gradlew test
```

Checking code style

To check codestyle:

```
./gradlew scalaStyle
```

Creating and Running Spark Application

Create application assembly which can be directly submitted to Spark cluster:

```
./gradlew shadowJar
```

The command creates jar file `build/libs/sparkling-water-droplet-app.jar` containing all necessary classes to run application on top of Spark cluster.

Submit application to Spark cluster (in this case, local cluster is used):

```
export MASTER="local[*]"
$SPARK_HOME/bin/spark-submit --class water.droplets.
    SparklingWaterDroplet build/libs/sparkling-water-
    droplet-all.jar
```

What is PySparkling Water?

PySparkling Water is an integration of Python with Sparkling water. It allows the user to start H2O services on a spark cluster from Python API.

In the PySparkling Water driver program, the `SparkContext` or `SparkSession` uses Py4J to start the driver JVM, and the JAVA `SparkContext/SparkSession` is used to create `H2OContext` (hc). This in turn starts the H2O cloud in the Spark ecosystem. (In Internal backend only. In external backend, the H2O cloud is running outside of the Spark cluster.) Once the H2O cluster is up, the H2O-Python package is used to interact with the cloud and run H2O algorithms. All pure H2O calls are executed via H2O's REST API interface. Users can easily integrate their regular PySpark workflow with H2O algorithms using PySparkling Water.

PySparkling Water programs can be launched as an application, or in an interactive shell, or notebook environment.

Getting Started:

1. Download Spark (if not already installed) from the Spark Downloads Page.

   Choose Spark release : 2.2.0

   Choose a package type: Pre-built for Hadoop 2.4 and later

2. Point SPARK_HOME to the existing installation of Spark and export variable MASTER.

```
export SPARK_HOME="/path/to/spark/installation"
```

   Launch a local Spark cluster.

```
export MASTER="local[*]"
```

3. From your terminal, run:

```
cd ~/Downloads
unzip sparkling-water-2.2.2.zip
cd sparkling-water-2.2.2
```

   Start an interactive Python terminal:

```
bin/pysparkling
```

The pysparkling shell accepts common pyspark arguments.

Or start a notebook:

```
PYSPARK_DRIVER_PYTHON="ipython"
    PYSPARK_DRIVER_PYTHON_OPTS="notebook" bin/
    pysparkling
```

4. Create an H2O cloud inside the Spark cluster and import H2O-Python package:

```
from pysparkling import *
hc = H2OContext.getOrCreate(spark)
import h2o
```

5. Follow this demo, which imports Chicago crime, census, and weather data. It also predicts the probability of arrest: https://github.com/h2oai/h2o-world-2015-training/blob/master/tutorials/pysparkling/Chicago_Crime_Demo.ipynb

Alternatively, to launch on YARN:

```
wget http://h2o-release.s3.amazonaws.com/sparkling-
    water/rel-2.2/2/sparkling-water-2.2.2.zip
unzip sparkling-water-2.2.2.zip

export SPARK_HOME="/path/to/spark/installation"
export HADOOP_CONF_DIR=/etc/hadoop/conf
export SPARKLING_HOME="/path/to/SparklingWater/
    installation"
$SPARKLING_HOME/bin/pysparkling --num-executors 3 --
    executor-memory 20g --executor-cores 10 --driver-
    memory 20g --master yarn --deploy-mode client
```

Then create an H2O cloud inside the Spark cluster and import H2O-Python package:

```
from pysparkling import *
hc = H2OContext.getOrCreate(spark)
import h2o
```

Or to launch as a Spark Package application:

```
1 $SPARK_HOME/bin/spark-submit --py-files
    $SPARKLING_HOME/py/build/dist/h2o_pysparkling_2
    .2-2.2.2.zip
2 $SPARKLING_HOME/py/examples/scripts/H2OContextInitDemo
    .py
```

Using Spark Data Sources

The way that an `H2OFrame` can be used as Spark's data source differs a little bit in Python from Scala.

Reading from `H2OFrame`

Let's suppose we have an `H2OFrame`. There are two ways how the `DataFrame` can be loaded from `H2OFrame` in pySparkling:

```
1 df = spark.read.format("h2o").option("key", frame.
    frame_id).load()
```

or

```
1 df = spark.read.format("h2o").load(frame.frame_id)
```

Saving to `H2OFrame`

Let's suppose we have a `DataFrame` df. There are two ways how `DataFrame` can be saved as `H2OFrame` in pySparkling:

```
1 df.write.format("h2o").option("key", "new_key").save()
```

or

```
1 df.write.format("h2o").save("new_key")
```

Both variants save `DataFrame` as an `H2OFrame` with key `new_key`. They won't succeed if an `H2OFrame` with the same key already exists.

Loading and Saving Options

If the key is specified as 'key' option, and also in the load/save method, the option 'key' is preferred:

```
df = spark.read.from("h2o").option("key", "key_one").
    load("key_two")
```

or

```
df = spark.read.from("h2o").option("key", "key_one").
    save("key_two")
```

In both examples, `key_one` is used.

A Use Case Example

Predicting Arrival Delay in Minutes - Regression

What is the task?

As a chief air traffic controller, your job is come up with a prediction engine that can be used to tell passengers whether an incoming flight will be delayed by X number of minutes. To accomplish this task, we have an airlines dataset containing ∼44k flights since 1987 with features such as: origin and destination codes, distance traveled, carrier, etc. The key variable we are trying to predict is "ArrDelay"" (arrival delay) in minutes. We will do this leveraging H2O and the Spark SQL library.

SQL queries from Spark

One of the many cool features about the Spark project is the ability to initiate a Spark Session(SQL Context) within our application that enables us to write SQL-like queries against an existing `DataFrame`. Given the ubiquitous nature of SQL, this is very appealing to data scientists who may not be comfortable yet with Scala / Java / Python, but want to perform complex manipulations of their data.

Within the context of this example, we are going to first read in the airlines dataset and then process a weather file that contains the weather data at the arriving city. Joining the two tables will require a Spark Session(SQL Context) such that we can write an INNER JOIN against the two independent `DataFrame`s.

The full source for the application is here: `http://bit.ly/1mo3XO2`

Let's get started!

Data Ingest

Our first order of business is to process both files, the flight data and the weather data:

```
import water.support._
import org.apache.spark.{SparkConf, SparkFiles}
import org.apache.spark.h2o._
import water.support.SparkContextSupport._
import org.apache.spark.examples.h2o.{Airlines,
    WeatherParse}
import java.io.File
// Configure this application
val conf: SparkConf = configure("Sparkling Water: Join
     of Airlines with Weather Data")

// Create SparkSession to execute application on Spark
     cluster
val spark = SparkSession.builder().config(conf).
    getOrCreate()
val h2oContext = H2OContext.getOrCreate(spark)
import h2oContext._
// Setup environment
addFiles(spark.sparkContext,
  absPath("examples/smalldata/chicago/
      Chicago_Ohare_International_Airport.csv"),
  absPath("examples/smalldata/airlines/
      allyears2k_headers.zip"))

val wrawdata = spark.sparkContext.textFile(SparkFiles.
    get("Chicago_Ohare_International_Airport.csv"), 3)
    .cache()
val weatherTable = wrawdata.map(_.split(",")).map(row
    => WeatherParse(row)).filter(!_.isWrongRow())

// Load H2O from zipped CSV file (i.e., access
    directly H2O cloud)
val airlinesData = new H2OFrame(new File(SparkFiles.
    get("allyears2k_headers.zip")))

val airlinesTable: RDD[Airlines] = asRDD[Airlines](
    airlinesData)
```

The flight data file is imported directly into H2O already as an `H2OFrame`. The weather table, however, is first processed in Spark where we do some parsing of the data and data scrubbing.

After both files have been processed, we then take the airlines data that currently sits in H2O and pass it back into Spark whereby we filter for those flights ONLY arriving at Chicago's O'Hare International Airport:

```
val flightsToORD = airlinesTable.filter(f => f.Dest ==
    Some("ORD"))

flightsToORD.count
println(s"\nFlights to ORD: ${flightsToORD.count}\n")
```

At this point, we are ready to join these two tables which are currently Spark RDDs. The workflow required for this is as follows:

- Convert the RDD into a `DataFrame` and register the resulting `DataFrame` as tables.

```
// Import implicit conversions
import spark.implicits._
flightsToORD.toDF.createOrReplaceTempView("
   FlightsToORD")
weatherTable.toDF.createOrReplaceTempView("
   WeatherORD")
```

- Join the two temp tables using Spark SQL

```
val bigTable = spark.sql(
   """SELECT
     |f.Year,f.Month,f.DayofMonth,
     |f.CRSDepTime,f.CRSArrTime,f.CRSElapsedTime,
     |f.UniqueCarrier,f.FlightNum,f.TailNum,
     |f.Origin,f.Distance,
     |w.TmaxF,w.TminF,w.TmeanF,w.PrcpIn,w.SnowIn,w
        .CDD,w.HDD,w.GDD,
     |f.ArrDelay
     |FROM FlightsToORD f
     |JOIN WeatherORD w
     |ON f.Year=w.Year AND f.Month=w.Month AND f.
        DayofMonth=w.Day
     |WHERE f.ArrDelay IS NOT NULL""".stripMargin)
```

- Transfer the joined table from Spark back to H2O to run an algorithm on the data

```
import h2oContext.implicits._
val train: H2OFrame = bigTable
```

H2O Deep Learning

Now we have our dataset loaded into H2O. Recall this dataset has been filtered to only include the flights and weather data on Chicago O'Hare. It's now time to run a machine learning algorithm to predict flight delay in minutes. As always, we start off with the necessary imports, followed by declaring the parameters that we wish to control:

```
import hex.deeplearning.DeepLearning
import hex.deeplearning.DeepLearningModel.
    DeepLearningParameters
import hex.deeplearning.DeepLearningModel.
    DeepLearningParameters.Activation

val dlParams = new DeepLearningParameters()
dlParams._train = train
dlParams._response_column = "ArrDelay"
dlParams._epochs = 5
dlParams._activation = Activation.RectifierWithDropout
dlParams._hidden = Array[Int](100, 100)

val dl = new DeepLearning(dlParams)
val dlModel = dl.trainModel.get
```

More parameters for Deep Learning and all other algorithms can be found in H2O documentation at `http://docs.h2o.ai`.

Now we can run this model on our test dataset to score the model against our holdout dataset:

```
val predictionH2OFrame = dlModel.score(bigTable).
    subframe(Array("predict"))
val predictionsFromModel = asRDD[DoubleHolder](
    predictionH2OFrame).collect.map(_.result.getOrElse
    (Double.NaN))
println(predictionsFromModel.mkString("\n===> Model
    predictions: ", ", ", ", ...\n"))
```

FAQ

Where do I find the Spark logs?

- **Standalone mode**: Spark executor logs are located in the directory `$SPARK_HOME/work/app-<AppName>` (where `<AppName>` is the name of your application). The location contains also stdout/stderr from H2O.
- **YARN mode**: YARN mode: The executor logs are available via the `$yarn logs -applicationId <appId>` command. Driver logs are by default printed to console, however, H2O also writes logs into `current_dir/h2ologs`.

The location of H2O driver logs can be controlled via the Spark property `spark.ext.h2o.client.log.dir` (pass via `--conf`) option.

How can I display Sparkling Water information in the Spark History Server?

Sparkling Water reports the information already, you just need to add the sparkling-water classes on the classpath of the Spark history server. To see how to configure the spark application for logging into the History Server, please see Spark Monitoring Configuration at `http://spark.apache.org/docs/latest/monitoring.html`.

Spark is very slow during initialization, or H2O does not form a cluster. What should I do?

Configure the Spark variable `SPARK_LOCAL_IP`. For example:

```
1 export SPARK_LOCAL_IP='127.0.0.1'
```

How do I increase the amount of memory assigned to the Spark executors in Sparkling Shell?

Sparkling Shell accepts common Spark Shell arguments. For example, to increase the amount of memory allocated by each executor, use the `spark.executor.memory` parameter: `bin/sparkling-shell --conf "spark.executor.memory=4g"`

How do I change the base port H2O uses to find available ports?

H2O accepts the `spark.ext.h2o.port.base` parameter via Spark configuration properties: `bin/sparkling-shell --conf "spark.ext.h2o.port.base=13431"`. For a complete list of configuration options, refer to section 6.4.

How do I use Sparkling Shell to launch a Scala test.script that I created?

Sparkling Shell accepts common Spark Shell arguments. To pass your script, please use `-i` option of Spark Shell: `bin/sparkling-shell -i test.script`

How do I increase PermGen size for Spark driver?

Specify `--conf spark.driver.extraJavaOptions="-XX:MaxPermSize=384m"`

How do I add Apache Spark classes to Python path?

Configure the Python path variable PYTHONPATH:

```
export PYTHONPATH=$SPARK_HOME/python:$SPARK_HOME/python/build:$PYTHONPATH
export PYTHONPATH=$SPARK_HOME/python/lib/py4j-*-src.zip:$PYTHONPATH
```

Trying to import a class from the hex package in Sparkling Shell but getting weird error:

```
error: missing arguments for method hex in object functions; follow this method with '_' if you want to treat it as a partially applied
```

In this case you are probably using Spark 1.5 or newer which is importing SQL functions into Spark Shell environment. Please use the following syntax to import a class from the hex package:

```
import _root_.hex.tree.gbm.GBM
```

Trying to run Sparkling Water on HDP Yarn cluster, but getting error:

```
java.lang.NoClassDefFoundError: com/sun/jersey/api/client/config/ClientConfig
```

The YARN time service is not compatible with libraries provided by Spark. Please disable time service via setting `spark.hadoop.yarn.timeline-service.enabled=false`. For more details, please visit `https://issues.apache.org/jira/browse/SPARK-15343`.

Getting non-deterministic H2O Frames after the Spark Data Frame to H2O Frame conversion

This is caused by what we think is a bug in Apache Spark. When specific kinds of data are combined with higher number of partitions, we can see non-determinism in BroadCastHashJoins. This leads to to jumbled rows and columns in the output H2O frame. We recommend disabling broadcast-based joins, which seem to be non-deterministic as:

```
sqlContext.sql("SET spark.sql.
   autoBroadcastJoinThreshold=-1")
```

The issue can be tracked as PUBDEV-3808 (`https://0xdata.atlassian.net/browse/PUBDEV-3808`). On the Spark side, the following issue is related to the problem: Spark-17806 (`https://issues.apache.org/jira/browse/SPARK-17806`)

How can I configure the Hive metastore location?

Spark SQL context (in fact Hive) requires the use of metastore, which stores metadata about Hive tables. In order to ensure this works correctly, the `${SPARK_HOME}/conf/hive-site.xml` needs to contain the following configuration. We provide two examples, how to use MySQL and Derby as the metastore.

For MySQL, the following configuration needs to be located in the `${SPARK_HOME}/conf/hive-site.xml` configuration file:

```
<property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:mysql://{mysql_host}:${mysql_port}/{
       metastore_db}?createDatabaseIfNotExist=true</
       value>
    <description>JDBC connect string for a JDBC
       metastore</description>
</property>

<property>
    <name>javax.jdo.option.ConnectionDriverName</name>
    <value>com.mysql.jdbc.Driver</value>
    <description>Driver class name for a JDBC
       metastore</description>
</property>

<property>
    <name>javax.jdo.option.ConnectionUserName</name>
    <value>{username}</value>
```

```
16      <description>username to use against metastore
            database</description>
17  </property>
18
19  <property>
20      <name>javax.jdo.option.ConnectionPassword</name>
21      <value>{password}</value>
22      <description>password to use against metastore
            database</description>
23  </property>
```

where:

- {mysql_host} and {mysql_port} are the host and port of the MySQL database.
- {metastore_db} is the name of the MySQL database holding all the metastore tables.
- {username} and {password} are the username and password to MySQL database with read and write access to the {metastore_db} database.

For Derby, the following configuration needs to be located in the `${SPARK_HOME}/conf/hive-site.xml` configuration file:

```
1   <property>
2       <name>javax.jdo.option.ConnectionURL</name>
3       <value>jdbc:derby://{file_location}/metastore_db;
            create=true</value>
4       <description>JDBC connect string for a JDBC
            metastore</description>
5   </property>
6
7   <property>
8       <name>javax.jdo.option.ConnectionDriverName</name>
9       <value>org.apache.derby.jdbc.ClientDriver</value>
10      <description>Driver class name for a JDBC
            metastore</description>
11  </property>
```

where:

- {file_location} is the location to the metastore_db database file.

References

H2O.ai Team. **H2O website**, 2018. URL http://h2o.ai

H2O.ai Team. **H2O documentation**, 2018. URL http://docs.h2o.ai

H2O.ai Team. **H2O GitHub Repository**, 2018. URL https://github.com/h2oai

H2O.ai Team. **H2O Datasets**, 2018. URL http://data.h2o.ai

H2O.ai Team. **H2O JIRA**, 2018. URL https://jira.h2o.ai

H2O.ai Team. **H2Ostream**, 2018. URL https://groups.google.com/d/forum/h2ostream

H2O.ai Team. **H2O R Package Documentation**, 2018. URL http://h2o-release.s3.amazonaws.com/h2o/latest_stable_Rdoc.html