Oracle R Advanced Analytics for Hadoop 2.7.0 Release Notes

Contents

1 Introduction.

Oracle R Advanced Analytics for Hadoop (ORAAH) is a collection of R packages that enable Big Data analytics from R environment. It enables a Data Scientist/Analyst to work on data straddling multiple data platforms (HDFS, Hive, Oracle database, local files) from the comfort of the R environment and benefit from the R eco system while leveraging advantages of massively distributed Hadoop computational infrastructure.

ORAAH provides:

1. A general computation framework, where a user can use the R language to write custom logic as mappers or reducers that can execute in a distributed parallel manner using available compute and storage resources on a Hadoop cluster. The I/O performance of such R based mapreduce jobs matches that of pure Java based map reduce programs by adding support for binary RData representation of input data.

2. Out of box predictive analytic techniques for linear regression, generalized linear models, neural networks, matrix completion using low rank matrix factorization, non-negative matrix factorization, k-means clustering, principal components analysis and multivariate analysis. While all these techniques have R interfaces, they have been implemented either in Java or in R as distributed parallel map reduce jobs leveraging all nodes of your Hadoop cluster. So these analytics can work on big data.

3. Support for Apache Spark MLlib functions, Oracle Formula support, and Distributed Model Matrix data structure. New predictive analytic functions from MLlib are exported as R functions and are fully integrated with ORAAH platform. MLlib R functions can be executed either on a Hadoop cluster using YARN to dynamically form a Spark cluster, or on a dedicated stand-alone Spark cluster. Users can switch on or off Spark execution using new spark.connect() and spark.disconnect() function. For more information about Apahe Spark refer to https://spark.apache.org/.

ORAAH's architecture and approach to Big Data analytics is to leverage the cluster compute infrastructure for parallel, distribution computation. ORAAH does this while shielding the R user from Hadoop's complexity through a small number of easy to use APIs.

2 What's new in ORAAH release 2.7.0.

1. Support for Cloudera Distribution of Hadoop (CDH) release 5.7.0. Both "classic" MR1 and YARN MR2 APIs are supported.

2. ORAAH 2.7.0 extends support for Apache Spark. Select predictive analytic functions can be executed on a Hadoop cluster using YARN to dynamically form a Spark cluster or on a dedicated stand-alone Spark cluster. Users and switch on or off Spark execution using new spark.connect() and spark.disconnect() function. For more information about spark refer to https://spark.apache.org/.

3. In this version of ORAAH we are introducing support for selected algorithms from Apache MLlib with support of our proprietary optimized data transformation and data storage algorithms built on top of Apache Spark and designed to improve performance and interoperability of MLlib and ORAAH machine learning functionality with R.

4. ORAAH 2.7.0 also introduces distributed model matrix and distributed formula that are used in all Spark MLlib-based functionality to greaty improve performance and enhance functional compatibility with R. Machine learning and statistical algorithms require a Distributed Model Matrix (DMM) for their training phase. For supervised learning algorithms DMM captures a target variable and explanatory terms; for unsupervised learning DMM captures explanatory terms only. Distributed Model Matrices has its own implementation of R formula, which closely follows R formula syntax, and much more efficient from performance perspective. Internally Distributed Model Matrices are stored as Spark RDDs (Resilient Distributed Datasets).

5. New functions `orch.save.model` and `orch.load.model` were added that allows to save and load models created using Apache Spark MLlib in ORAAH to HDFS for scoring/prediction later on. It also enables model sharing amongst different users if the other users have access to the path where models are saved.

6. Distributed Model Matrix (DMM) can be saved in Comma-Separated Values (CSV) format on HDFS via a call to `hdfs.write()`. For instance, the below example solves a linear regression model, generates predictions, and saves them on HDFS in CSV format.

7. Another major improvement introduced in ORAAH 2.7.0 is support of HiveServer2 in Hive transparency layer. Now ORAAH will use JDBC/Thrift connection in order to communicate with Hive and execute Hive queries on a remote or local Hadoop cluster instead of error-prone and unstable CLI execution layer used in previous versions of ORAAH. This greatly improves performance of Hive transparency layer as well as lowers latency of Hive queries providing user a fluent and robust integration of ORAAH/R with Apache Hive. For more information about Apache Hive refer to https://hive.apache.org/.

8. R’s base predict() function was extended to support prediction on Spark when a model generated by GLM on Spark or by MLlib-base analytics functions is used.

9. New MKL added to the installer package: Intel® Math Kernel Library Version 11.1.0 Product Build 20130711 for Intel® 64 architecture applications.

10. Bugfixes and updates accross the platform for improved stability and ease of use. For more information see the "Change List" document.

3 What’s new in ORAAH release 2.7.0.

1. Since ORAAH 2.5.0 introduces support for data load to Spark cache from HDFS files and for data unload from Spark cache back to HDFS files via function hdfs.toRDD() and hdfs.fromRDD(). ORAAH dfs.id objects were also extended to support both data residing in HDFS and in Spark memory.

2. Neural network analytical function (orch.neural) now supports execution on Spark in addition to previously available Hadoop mapReduce execution support, which dramatically improves its performance. There are no changes to the neural network APIs, its functionality transparently switched to Spark execution facilities once ORAAH is connected to Spark (see spar.connect()). Details are available in orch.neural() documentation.

3. New GLM implementation on Spark was added in this release. The new GLM function has incompatible with the currently available GLM on Hadoop API therefore it was implemented in a form of a separate function orch.glm2() that can only be executed when ORAAH is connected to Spark.

4 Known issue in ORAAH release 2.7.0

There are a few known issues in this release:

1. Attaching an HDFS folder which contains only 0-sized data files will result in exception. To avoid attaching such data one can use hdfs.size() function to check if the size is 0 prior using hdfs.attach() API on such files.

2. Executing a mapReduce job that produces no output at all may result in exception stop the script execution. Set the parameter cleanup=TRUE to avoid such exception.

3. Ensure all categorical columns are explicitly converted into factors (as.factor(categorical column)) in the formula before using such columns in orch.neural().

4. Working with Spark requires that all necessary Spark and Hadoop Java libraries are included in CLASSPATH environment variable. Otherwise connection to Spark will fail. See the installation guide for detailed information about configuring ORAAH and system environment.

5. Only text based HDFS object can be loaded into to Spark distributed memory and used for Spark-based analytics, binary Rdata HDFS object are not supported in Spark.

6. `hdfs.fromHive` function still uses CLI to interract with Hive metastore that may cause issues if Hive client is not installed or misconfigured on the client’s host.

5 Parallelization packages in R vs. ORAAH

ORAAH is often compared/contrasted with other options available in the R eco-system, in particular the popular open source R package called “parallel”. The parallel package provides a low-level infrastructure for “coarse-grained” distributed and parallel computation. While it is fairly general, it tends to encourage an approach that is based on using the aggregate RAM in the cluster as opposed to using the file system. Specifically, it lacks data and resource management components, a task scheduling component and an administrative interface for monitoring. Programming, however, follows the broad Map Reduce paradigm.

The crux of ORAAH is read parallelization and robust methods over parallelized data. Efficient parallelization of reads is the single most important step necessary for Big Data Analytics because it is either expensive or impractical to load all available data into memory addressable by a single process. The rest of this document explains the architecture, design principles and certain implementation details of the technology.

6 Architecture

ORAAH is built upon Hadoop streaming, a utility that is a part of Hadoop distribution and allows creation and execution of Map/Reduce jobs with any executable or script as mapper and/or reducer. ORAAH automatically builds the logic required to transform an input stream of data into an R data frame object that can be readily consumed by user-provided snippets of mapper and reducer logic written in R.

ORAAH is designed for R users to work with Hadoop cluster in a client-server configuration. Client configurations must conform to the requirements of the Hadoop distribution that ORAAH is deployed in. This is because, ORAAH uses command line interfaces to communicate from client node to HDFS and HIVE running on Hadoop cluster.

ORAAH allows R users to move data from an Oracle database table/view into Hadoop as HDFS file. To do this, ORAAH uses the SQOOP utility. Similarly data can be moved back from a HDFS file into Oracle database. For the reverse data movement, a choice of using sqoop or Oracle Loader for Hadoop utility is available depending on the size of data being moved and security requirements when connecting to Oracle database.

For performance sensitive analytic workloads, ORAAH supports R’s binary RData representation for both input and output. Conversion utilities from delimiter separated representation of input data to/from RData representation is available as part of ORAAH.

ORAAH includes a Hadoop Abstraction Layer (HAL) which manages the similarities and differences across various Hadoop distributions. ORCH will auto-detect the Hadoop version at startup but if you’d like to use ORAAH on a different distribution please look into help (ORCH_HAL_VERSION) for details.

ORAAH consists of 5 distinct sets of APIs with the following functionality:

1. Access to HDFS objects: ORCH HDFS APIs map 1:1 to corresponding HDFS command line interfaces that explore HDFS files. Additionally, ORAAH exposes new functions to explore contents of HDFS files.
2. Access to HIVE SQL functionality by overloading certain R operations and functions on data frame objects that map to HIVE tables/views.
3. Launching Hadoop jobs from within a user’s R session using map/reduce functions written in R language. Such functions can be tested locally (help(orch.dryrun) and help(orch.debug()). ORAAH mimics Hadoop style computation but on the client’s R environment optionally enabling debugging capability.
4. Native Hadoop-optimized analytics functions for commonly used statistical and predictive techniques.
5. File movement between local user’s R environment (in-memory R objects), local file system on a user’s client machine, HDFS/HIVE and Oracle database. Inputs to ORAAH ORAAH can work with the following types of input:
   - Delimited text files that are stored in a HDFS directory. ORAAH
   - recognizes only HDFS directory names as input and processes all
   - files inside the directory.
   - Apache HIVE tables.

- Proprietary binary RData representations.
- Apache Spark RDD objects.

If data that is input to an ORAAH orchestrated map-reduce computation does not reside in HDFS (e.g., a logical HiveQL query), ORAAH will create a copy of the data in HDFS automatically prior to launching the computation.

Before ORAAH can work with delimited text files in a HDFS directory it determines metadata associated with the files and captures the metadata in a file stored alongside of the data files. The metadata file is named *ORCHMETA*. The metadata contains the following information:

a. If the file contains key(s) and if so the delimiter that is the key separator

b. The delimiter that is the value separator

c. Number and data types of columns in the file

d. Optional names of columns (as available in a header line)

e. Dictionary information for categorical columns (in R terminology - levels of a factor column)

f. Other ORAAH-specific system data

ORAAH runs an automatic metadata discovery procedure on HDFS files as part of hdfs.attach() invocation. When working with HIVE tables, *ORCHMETA* file is auto-created from the HIVE table definition.

ORAAH can optionally convert input data into R's binary RData representation (see hdfs.toRData and hdfs.fromRData). RData representation is recommended because I/O performance of R based mapreduce jobs on such representation is on par with a pure Java based map-reduce implementation.

ORAAH captures row streams from HDFS files and delivers them formatted as a data frame object (or optionally matrix/vector/list objects generated from the data frame object or AS IS if RData representation is used) to mapper logic written in R. To accomplish this, ORAAH must recognize the tokens and data types of the tokens that become columns of a data frame. ORAAH uses R's facilities to parse and interpret tokens in input row streams. If missing values are not represented using R's “NA” token, they can be explicitly identified by the na.strings argument of hdfs.attach().

Delimited text files with the same key and value separator are preferred over files with a different key delimiter and value delimiter. Read performance of files with the same key and value delimiter is roughly 2X better than that of files with different key and value delimiter. The key delimiter and value delimiter can be specified through the key.sep and val.sep arguments of hdfs.attach() or when running a mapReduce job for its output HDFS data.

IMPORTANT: Binary RData representation is the most performance efficient representation of input data in ORAAH. When possible, users are encouraged to use this binary data representation for performance sensitive analytics.

Starting from ORAAH 2.5.0 we also have added support for Spark Resilient Distributed Dataset (RDD) objects. User can now load data from HDFS to Spark's RDD and optionally cache it in Spark distributed memory in order to improve performance of Spark-enabled analytics run on this data set. Results of Spark-enabled analytics will be retuned as another Spark RDD object and user will have an option to save data from Spark's distributed memory to an HDFS directory in order to preserve it between ORAAH sessions. For for information look at documentation of hdfs.toRDD() and hdfs.fromRDD() functions.

7 Working with ORAAH Hadoop interface

ORAAH's mapReduce job definition is accomplished through hadoop.exec() or hadoop.run() APIs. hadoop.exec() is a subset of hadoop.run() in that the latter additionally supports appropriate staging of the final result from a map-reduce job. The former is used if the data when building a pipeline of ORCH mapReduce jobs that are streaming data from one job stage to another and does not require to bring intermediate results back to ORCH environment and require that input to the mapReduce job is already in HDFS. See help(“hadoop.exec”) and help(“hadoop.run”) for more details.

By default, ORAAH prepares the rows streamed into the mappers and reducers defined as part of hadoop.run() or hadoop.exec() APIs as a data.frame object with input values and a vector object with input keys. Optionally input can be supplied to the mapper as a R matrix, vector or a list object (see mapred.config()). If the input data to a map-reduce job is in RData representation, set "direct.call" parameter to TRUE – this will cause ORAAH to pass input data as is to the user supplied R mapper code. The R code that is meant to execute as mapper logic requires no special coding – it is nothing more than a R closure that accepts a data.frame object as input (by default) together with other variables exported from the user's client R session. Optionally, several configuration parameters may be specified by instantiating an R object of "mapred.config" class.

The mapper has a choice of generating either structured output in the form of a data.frame or any arbitrary data structure. By default, the mapper is assumed to perform any filtering/computation on data it receives as input and generate as output a data frame with the same number of columns and same data type as its input. However this can be overridden through defining the structure of mapper's output and hence reducer's input (in a map/reduce sequence) object using "map.output" parameter in the "mapred.config" class. Additionally, if the mapper's output is 'pristine' (i.e. all null values are represented either as "NA" or "") then, optionally setting yet another "mapred.config" parameter called mapred.pristine can enable better read performance in the reducer. See orch.keyvals() and orch.keyval() for additional details.

Complex output objects from the mapper can be packed using orch.pack() (and read using corresponding orch.unpack()) interfaces before being passed to orch.keyval[s]() functions. Packing of objects has a benefit of allowing to a user to store unstructured, semi-structured or variable-structured data in a structured output of mapReduce jobs. Also, packed data can be compressed in order to minimize used space and improve performance, see the "COMPRESS" argument description of orch.pack() function.

By default, the reducer is not assumed to generate output data in the same format as its input. The user can specify the structure of reducer's output via "reduce.output" field of "mapred.config" class. If this specification is unavailable ORAAH will attempt to discover the metadata by sampling the reducer's output.

If the mapper task is computationally expensive and likely to take longer than the default Hadoop timeout of 600 seconds, "task.timeout" parameter of "mapred.config" class must be set to allow longer timeouts for a specific task. The timeout value must be specified in seconds.

8 Working with ORAAH Hive interface

ORAAH allows HIVE tables/views from default and non-default databases to be used as 'special' data frames in that they have enough metadata to be able to generate HiveQL queries when data in the HIVE tables/views is processed. See the following demos to understand ORAAH-Hive interfaces:

```
R> demo(package="ORCH")
hive_aggregate Aggregation in HIVE
hive_analysis Basic analysis & data processing operations
hive_basic Basic connectivity to HIVE storage
hive_binning Binning logic
hive_columnfns Column function
hive_nulls Handling of NULL in SQL vs. NA in R
hive_pushpull HIVE <-> R data transfer
hive_sequencefile Creating and using HIVE table
```

9 Native Analytic Functions

The following analytic functions are available out of box in ORCH. These functions are parallel / distributed and execute utilizing all nodes of Hadoop cluster.

1. Covariance and Correlation matrix computation (orch.cov, orch.cor)
2. Principal Component Analysis (orch.princomp, orch.predict)
3. K-means clustering (orch.kmeans, orch.predict)
4. Linear regression (orch.lm, orch.predict)
5. Generalized linear models including logistic regression (orch.glm)
6. Neural Networks (orch.neural)

7. Matrix completion using low rank matrix factorization (orch.lmf)
8. Non negative matrix factorization (orch.nmf)
9. Reservoir sampling (orch.sample)

10 Copyright Notice