# Oracle R Advanced Analytics for Hadoop 2.8.2 Release Notes

**REVISION HISTORY**

| NUMBER | DATE | DESCRIPTION | NAME |
|---|---|---|---|
| | | | |

# Contents

1 Introduction 1

2 What's new in ORAAH release 2.8.2 1

3 Known issues in ORAAH release 2.8.2 6

4 Parallelization Packages in R compared to those in ORAAH 7

5 Architecture 7

6 Working With the ORAAH's Hadoop Interface 9

7 Working with ORAAH's Apache Hive/Impala Interface 10

8 Working with ORAAH's Spark Interface 10

9 Native Analytic Functions 11

10 Copyright Notice 12

11 3rd Party License Notice 13

# 1 Introduction

Oracle R Advanced Analytics for Hadoop (ORAAH) is a collection of R packages that enable big data analytics from the R environment. With ORAAH, Data Scientists and Data Analysts who are comfortable within R gain the benefit of a broad reach from the R environment into data resident across multiple platforms (HDFS, Apache Hive, Apache Impala, Oracle Database, and local files). This gain is boosted by the power to leverage the massively distributed Hadoop computational infrastructure to analyze potentially rich cross-platform datasets.

ORAAH provides some big advantages for big data analytics:

1. A general computation framework where you can use the R language to write custom logic as mappers or reducers. These execute in a distributed, parallel manner using the compute and storage resources available within a Hadoop cluster. Support for binary RData representation of input data enables R-based MapReduce jobs to match the I/O performance of pure Java-based MapReduce programs.

2. Tools ready for work "right out of the box" to provide 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 of these techniques have R interfaces, they have been implemented either in Java, or, in R as distributed, parallel MapReduce jobs that leverage all the nodes of your Hadoop cluster. This enables these analytics tools to work with 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 standalone Spark cluster. One can switch Spark execution on or off with the new `spark.connect()` and `spark.disconnect()` functions. For more information about Apache 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, distributed 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 2.8.2

1. Support for Cloudera Distribution of Hadoop (CDH) release up to 6.0.0. Both "classic" MR1 and YARN MR2 APIs are supported.

2. ORAAH 2.8.2 extends support for Apache Spark 2.3 and higher. Select predictive analytic functions can be executed on a Hadoop cluster using YARN to dynamically form a Spark cluster or a dedicated standalone Spark cluster. Users can switch Spark execution on or off using new `spark.connect()` and `spark.disconnect()` functions. For more information about Spark, refer to https://spark.apache.org/.

3. Another major ORAAH feature in ORAAH 2.8.2 is the support for Apache Impala in the transparency layer along with Apache Hive. Apache Impala is a distributed, lighting fast SQL query engine for huge data stored in an Apache Hadoop cluster. It is a massively parallel and distributed query engine that allow users to analyse, transform and combine data from a variety of data sources. It is used when there is need of low latency result. Unlike Apache Hive, it does not convert queries to MapReduce tasks. MapReduce is a batch processing engine, so by design Apache Hive, which relies on MapReduce, is a heavyweight, high-latency execution framework. MapReduce Jobs have all kinds of overhead and are hence slow. Apache Impala on the other hand does not translate a SQL query into another processing framework like map/shuffle/reduce operations, so it does not suffer from latency issues. Apache Impala is designed for SQL query execution and not a general purpose distributed processing system like MapReduce. Therefore it is able to deliver much better performance for a query. Apache Impala, being a real time query engine, is best suited for analytics and for data scientist to perform analytics on data stored in HDFS. However not all SQL queries are supported in Apache Impala. In short, Impala SQL is a subset of HiveQL and might have few syntactical changes. For more information about Apache Impala refer to https://impala.apache.org/. Support for this new connection type has been added to `ore.connect(..., type="IMPALA")`. For more details and examples on connecting to Apache Hive and Apache Impala on non-kerberized and kerberized (with SSL/ without SSL enabled) clusters can be checked by `help(ore.connect)`.

4. In ORAAH 2.8.2 we introduced a new package `ORCHmpi`. This package has distributed MPI-backed algorithms which run over the Apache Spark framework. `ORCHmpi` needs MPI libraries made available to ORAAH either by making MPI available system-wide, or by setting ORCH related environment variables on the client node. See "Change List" document for more details on the list of new functions available in `ORCHmpi` package.

5. ORAAH 2.8.2 adds support for new quasi-Newton algorithms, L-BFGS and L-BFGS-TR, to solve the underlying optimization models. L-BFGS with polynomial extrapolation is an improvement upon previous ORAAH version where in addition to the early Armijo-Wolfe termination criteria and cubic interpolation, ORAAH introduced polynomial extrapolation, thereby reducing the number of iterations (number of data passes). L-BFGS-TR introduces the trust region reducing the number of iterations even further. Currently only GLM has support for L-BFGS in ORAAH R client, but ORAAH Java anaytics library include L-BFGS support for both GLM and LM solvers. L-BFGS is recommended for models with very large number (e.g. more than 10000) of numerical features, or whenever memory available to each Spark worker is severely limited.

6. L-BFGS* are sophisticated and remarkably important numerical optimization algorithms for unconstrained nonlinear optimization. This class of algorithms have three main advantages over alternatives. Firstly, L-BFGS* is a gradient-based algorithm with minimal memory requirements (as compared to the second order techniques such as iteratively reweighted least squares). Secondly, L-BFGS* enjoys superlinear convergence, which results in dramatically fewer number of iterations as compared to a much simpler gradient descent. And, thirdly, both algorithms often require 10%~30% fewer iterations than most open source and ORAAH 2.7.1 implementations.

7. All Spark based algorithms in ORAAH can now read input data from multiple sources. These sources include CSV data from HDFS (`hdfs.id`), Spark data frame, Apache Hive tables (as `ore.frame` object), Apache Impala tables (as `ore.frame` object) or any other Database using JDBC connection. See "Change List" document for more details about function `orch.jdbc()` which enables JDBC input.

8. Many new functions have been added in ORAAH 2.8.2 to handle Spark data frame operations. The new functions are `orch.df.fromCSV()`, `orch.df.scale()`, `orch.df.summary()`, `orch.df.describe()`, `orch.df.sql()`, `orch.df.createView()`, `orch.df.collect()`, `orch.df.persist()` and `orch.df.unpersist()`. For more details on these functions see "Change List" document.

9. A very important addition in ORAAH 2.8.2 is the support for scaling and aggregate functions. Normalization can be of paramount importance for successful creation and deployment of machine learning models in practice. ORAAH 2.8.2 also introduces a general scaling function `orch.df.scale()` featuring 12 standard scaling techniques specifically: "standardization", "unitization", "unitization_zero_minumum", "normalization", and so forth, which can be applied to all numerical columns of a Spark data frame at once. Currently, `orch.df.scale()` only supports Spark data frame as input.

10. `spark.connect()` now has new parameters *logLevel* and *enableHive* to change the Apache Spark message logging level, and change the Spark default, and to enable Apache Hive support in Spark Session. With Apache Hive support enabled in the Spark session, all Spark based algorithms are able to read Apache Hive tables directly as input source. See "Change List" document for more details.

11. `spark.connect()` now resolves the active HDFS Namenode for the cluster if the user does not specify the *dfs.namenode* parameter. An INFO message is displayed which prints the Namenode being used. If for any reason, auto-resolution of Namenode fails, users can always specify *dfs.namenode* value themselves.

12. `spark.connect()` now loads the default Spark configuration from the *spark-defaults.conf* file if it is found in the *CLASSPATH*. If such a file is found, an INFO message is displayed to the user that defaults were loaded. These configurations have the lowest priority during the creation of a new Spark session. Properties specified within `spark.connect()` will overwrite the default values read from the file.

13. New defaults have been added to `spark.connect()` that are used when not specified. *spark.executor.instances*, which specifies the number of parallel Spark worker instances to create, is set to `2` by default. Also, *spark.executor.cores*, which specifies the number of cores to use on each Spark executor, is set to `2` by default.

14. With this release, the loading time of `library(ORCH)` has been significantly reduced. The mandatory environment and Hadoop component checks on startup have been made optional. These checks are now one time activity and the cached configuration is used everytime ORAAH is loaded. But if you wish to run checks within the current R session then you can invoke `orch.reconf()` anytime after ORAAH has been loaded. This will remove the existing cached configuration

save the new configuration. It is recommended to run `orch.reconf()` everytime you upgrade any of your Hadoop component like Hadoop/Yarn, Hive, Sqoop, OLH, Spark, etc. You can also use `orch.destroyConf()` which destroys the currently cached ORAAH configuration that was saved when `library(ORCH)` was loaded for the first time after install or the last time `orch.reconf()` was run. This will not cause the checks to run immediately in the current R session. In the next R session, when library is loaded, the configuration and component checks will run again and the configuration will be cached. For more details on these functions see "Change List" document.

15. ORAAH 2.8.2 has a new client side HDFS file system interface, which is implemented using Java. This new feature improves the performance of many `hdfs.*()` functions like `hdfs.get()`, `hdfs.put()`, `hdfs.upload()`, `hdfs.exists()`, `hdfs.meta()`, `hdfs.attach()`, `hdfs.rm()`, `hdfs.rmdir()`, `hdfs.tail()` and `hdfs.sample()` significantly. This change is transparent to the users and no exported API has changed. If your Hadoop configuration directory *HADOOP_CONF_DIR* is present in the *CLASSPATH/ORCH_CLASSPATH* this improvement will set itself up. If for any reason the java client fails to initialize, the older version of `hdfs.*()` functions will work without any speed ups.

16. `hdfs.toHive()` and `hdfs.fromHive()` now work with the newly supported Apache Impala connection as well.

17. If the CSV files being uploaded to HDFS using `hdfs.upload()` have a header line which provides the column names for the data, then ORAAH now uses this information in the generated metadata instead of treating the data with no column names (which made ORAAH to use generated column names *VAL1*, *VAL2*, and so on). User need to specify the parameter `header=TRUE` in `hdfs.upload()` for correct header line handling.

18. A new field *queue* has been added to `mapred.config` object which is used to specify the MapReduce configuration for the MapReduce jobs written in R using ORAAH. This new configuration helps a user select the MapReduce queue to which the MapReduce task will be submitted.

19. Previously, ORAAH used the *Hive CLI* interface in a few functions, which was not desired and caused slowdowns in those functions which came from the *Hive CLI* startup time. In this release, *Hive CLI* usage was removed entirely. Users will notice significant performance gains with Apache Hive related functions in ORAAH, like `hdfs.toHive()`, `hdfs.fromHive()` and `orch.create.parttab()`.

20. To improve the configuration options for the `rJava` JVM, which runs alongside ORAAH in R, support for `-Xms` has been added. To change the default value of `256m`, the user can set the environment variable *ORCH_JAVA_XMS*. Earlier versions of ORAAH already had the option to set `-Xmx` with a default of `1g` using the environment variable *ORCH_JAVA_XMX*. For more details and examples, check `help(ORCH_JAVA_XMS)`.

21. Most of the hadoop commands in earlier versions of ORAAH would failed if user had set *JAVA_TOOL_OPTIONS* environment variable. This has been fixed in this release.

22. While running any java application in R through `rJava` package (i.e. while running Spark based algorithms in ORAAH) pressing *Ctrl + C* Keyboard keys would interrupt `rJava` JVM, which in turn crashed R process. This issue has been fixed in this release.

23. The signatures for almost all Spark MLlib algorithms `orch.ml.*()` in ORAAH have been changed in ORAAH 2.8.2. For more details on these changes see "Change List" document.

24. A new Spark MLlib algorithm was added in ORAAH 2.8.2, `orch.ml.gbt()`. This function is used to invoke MLlib Gradient-Boosted Trees (GBTs). MLlib GBTs are ensembles of decision trees. GBTs iteratively train decision trees in order to minimize a loss function. Like decision trees, GBTs handle categorical features, extend to the multiclass classification setting, do not require feature scaling, and are able to capture non-linearities and feature interactions. MLlib supports GBTs for binary classification and for regression, using both continuous and categorical features.

25. Support for a new activation function *softmax* has been added to `orch.neural2()` in this release.

26. ORAAH 2.8.2 introduces a new summary function which is used for Apache Hive tables, `orch.summary()`. This function improves performance over `summary()` function for Apache Hive tables. This function calculates descriptive statistics and supports extensive analysis of columns in an `ore.frame` (when connected to Apache Hive), along with flexible row aggregations. It provides a relatively simple syntax as compared to the SQL queries that produces the same result.

27. MLlib machine learning and statistical algorithms require a special row-based distributed Spark data frame for training and scoring. A new function `orch.mdf()` is used to generate such MLlib data frames that can be provided to any of the MLLib algorithms supported in ORAAH. If you wish to run multiple MLlib algorithms on same data, then it is recommended to create such a MLlib data frame upfront using `orch.mdf()` to save repeated computation and creation of these MLlib data frames by every Mllib algorithm.
28. MLlib model objects have target factor levels associated with them (if available) to help the user better understand the prediction results and labels.
29. Oracle R formula engine was redesigned to include major enhancements and improvements in release 2.8.2. It introduces support for main statistical distributions in Oracle R Formula:
    a. Beta distribution
    b. Binomial distribution
    c. Cauchy distribution
    d. Chi-squared distribution
    e. Exponential distribution
    f. F-distribution
    g. Gamma distribution
    h. Geometric distribution
    i. Hypergeometric distribution
    j. Log-normal distribution
    k. Normal distribution
    l. Poisson distribution
    m. Student t-distribution
    n. Triangular distribution
    o. Uniform distribution
    p. Weibull distribution
    q. Pareto distribution

for each of the aforementioned distributions, ORAAH 2.8.2 introduces .. density, .. cumulative density, .. quantile, and .. random deviates functions.

1. ORAAH Oracle R formula engine in 2.8.2 also introduces support for the following special functions:
    a. `gamma(x)` - gamma function Gamma(x)
    b. `lgamma(x)` - natural logarithm of the gamma function
    c. `digamma(x)` - the first derivative of the lgamma(x)
    d. `trigamma(x)` - the second derivative of the lgamma(x)
    e. `lanczos(x)` - Lanczos approximation of the gamma function
    f. `factorial(x)` - factorial n!
    g. `lfactorial(x)` - natural logarithm of the factorial function
    h. `lbeta(a, b)` - natural logarithm of the Beta function
    i. `lchoose(n, k)` - natural logarithm of the binomial coefficient
2. Lastly, Oracle R Formula introduces new aggregate functions:
    a. `avg(colExpr)` - mean (average) value
    b. `mean(colExpr)` - mean value (synonym for avg)
    c. `max(colExpr)` - maximum value

   d. `min(colExpr)` - minimum value
   e. `sd(colExpr)` - standard deviation
   f. `stddev(colExpr)` - standard deviation (synonym for sd)
   g. `sum(colExpr)` - sum
   h. `variance(colExpr)` - variance
   i. `var(colExpr)` - variance (synonym for variance)
   j. `kurtosis(colExpr)` - kurtosis
   k. `skewness(colExpr)` - skewness where `colExpr` is an arbitrary (potentially nonlinear) column expression. Above aggregate functions supports only Spark data frame input at this moment.

3. This release makes a major improvement in ORAAH integration capabilities with other services and products. Before this 2.8.2 release, the only way to interact and integrate with ORAAH was via the R language and R runtime, starting from ORAAH 2.8.2, all the core analytics functionality was decoupled and consolidated into a standalone ORAAH core analytics Java library that can be used directly without the need of R language from any Java or Scala platform. This allows direct, efficient and easy integration with ORAAH core analytics. For more information, refer to "oraah-analytics-library-2.8.2-scaladoc" packed as a *zip* file in the installer package.

4. Since 2.7.1, ORAAH can connect to Oracle Databases running in Multitenant Container Database (CDB) mode. A new parameter *pdb* can be used to specify the service name of the pluggable database (PDB) to which the connection has to be established.

5. Since 2.7.1, ORAAH supports a new environment configuration variable *ORCH_CLASSPATH*. You can set the *CLASSPATH* used by ORAAH using this variable. *ORCH_CLASSPATH* can be set for the shell or in *Renviron.site* file to make it available for R and ORAAH. *rJava* does not support wildcard characters in *CLASSPATH* environment variable. This led to users adding exact jar files in the *CLASSPATH* before they could use Apache Spark analytics. Now ORAAH 2.7.1 resolves this issue and will support the wildcards in path. For example, having a path */usr/lib/hadoop/lib/*.jar* in *CLASSPATH* adds all jars from */usr/lib/hadoop/lib* to rJava *CLASSPATH*. This makes the process of getting started with ORAAH's Spark features easier and faster. You can use wildcards with *ORCH_CLASSPATH* too, if you are going to use *ORCH_CLASSPATH* instead of *CLASSPATH* for ORAAH configuration.

6. ORAAH 2.7.0 introduced distributed model matrix and distributed R formula support used in all Spark MLlib-based functionality. These greatly 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 have their own implementation of R formula, which closely follows the R formula syntax implemented in R base functionality, yet is much more efficient from a performance perspective. Internally Distributed Model Matrices are stored as Spark RDDs (Resilient Distributed Datasets). This version adds new features to these components of ORAAH. See "Change List" document for more details.

7. Note that the new MKL is added to the installer package: Intel® Math Kernel Library Version 2019 for Intel® 64 architecture applications.

8. This version of ORAAH comes with new installers and un-installers for both ORAAH server and client. See "Install Guide" and "Change List" document for more details on improvements and new features.

9. Bug fixes and updates across the platform improve stability and ease-of-use. For more information see the "Change List" document.

10. Since ORAAH 2.5.0, support for data load to Spark cache from HDFS files was introduced. This is provided by the function hdfs.toRDD(). ORAAH `hdfs.id` objects were also extended to support data residing in both HDFS and in Spark memory.

11. New implementations on Spark for GLM, LM and Neural Networks were added in the release 2.7.0. Because these new functions are not compatible with the currently available functions on Hadoop API, they are implemented as orch.glm2() , orch.lm2() and orch.neural2(), separate functions that can only be executed when ORAAH is connected to Spark.

# 3 Known issues in ORAAH release 2.8.2

There are a few known issues in this release.

1. Attaching an HDFS folder that contains only `0`-sized data files results in an exception. To work around this, prior to using the `hdfs.attach()` API on such files, use the `hdfs.size()` function to check if the size is 0.
2. Executing a MapReduce job that produces no output at all may result in exception that stops script execution. Set the parameter `cleanup=TRUE` to avoid this 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, connections to Spark will fail. See the installation guide for detailed information and examples for configuring ORAAH and the system environment.
5. Only text-based HDFS objects can be loaded into Spark distributed memory and used for Spark-based analytics. Binary Rdata HDFS objects are not supported in Spark.
6. To enable connection to Apache Impala, you must have Impala JDBC driver version 4 libraries on your client node. They can be downloaded from Cloudera website. The path to these libraries must be set as *IMPALA_JAR* environment variable Apart from *IMPALA_JAR* you must have *IMPALA_HOME* defined as well. See ORAAH install guide for configuration details and examples.
7. To enable Apache Impala as input to all Spark based algorithms in `ORCHstats` and `ORCHmpi` packages, you must do an additional setup. The *IMPALA_JAR* library must be added to *SPARK_DIST_CLASSPATH*. See ORAAH install guide for configuration details and examples.
8. To enable JDBC input to all Spark based algorithms you must have the respective database's JDBC driver library on all nodes on the cluster at the same path. And this path must be a part of your *CLASSPATH* and *SPARK_DIST_CLASSPATH*. Since, Apache Spark executors ingest data over JDBC connection in a distributed manner, the success depends on availability of libraries on all nodes.
9. On a kerberized cluster Apache Impala table input to Spark algorithms might fail. A simple workaround is to switch to an Apache Hive connection and use the Apache Impala table as input to Spark based analytics. This is possible since both Apache Hive and Apache Impala share a common Metastore.
10. Using HDFS data after invoking `hdfs.fromHive()` on an Apache Hive table, might cause Spark-based analytics to fail if the NA identifier in the table data files is not "NA". Such failure can be resolved by setting the correct value for "na.strings" in the HDFS metadata manually by using hdfs.meta(). For example, if the table *iris_tab* has "NULL" as its NA identifier value then you can set the metadata as: `hdfs.meta(iris_hdfs, na.strings ="NULL")`, where *iris_hdfs* is an `hdfs.id` object created by `iris_hdfs <- hdfs.fromHive(iris_tab)`. After setting the correct metadata you should have no problems using data from HDFS with Spark-based analytics.
11. `ore.create()` on large R `data.frame` objects fails due to a specific *hiveserver2* behavior, and can be avoided by running ORAAH client on the same node of the cluster which is running the *hiveserver2* process. If your usage does not involve creating Apache Hive tables from large R `data.frame`'s then you can run ORAAH client on any node of your cluster.
12. MPI process management has a single point-of-failure master which also handles cleanup, and is a part of the driver process. Normally, even if there are errors during an MPI processing stage, the MPI cluster would terminate on its own and clean up its space. That includes situations when some tasks fail due to severe errors, such as segmentation faults or resource starvation. However, in any case the driver process additionally takes steps to verify that all MPI resources are released, in case if MPI cluster fails to relinquish them on its own. However, if the driver application also goes defunct at the same time, then only the manual cleanup routine (`orch.mpi.cleanup()`) is available to remedy the situation.
13. Current MPI algorithms only work with the dense algebra. It means that sparse problems are densified by the MPI stages, which in turn may mean a dramatic increase in both memory and computational requirements. Combined with the fact that MPI tasks cannot be serialized and may start computations only when all of the MPI tasks are loaded, this may create a resource starvation. Future releases will address the sparse algebra support.

# 4 Parallelization Packages in R compared to those in ORAAH

ORAAH is often compared/contrasted with other options available in the R ecosystem, in particular, with 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 MapReduce 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.

# 5 Architecture

ORAAH was initially built upon the Hadoop streaming utility. Hadoop streaming enables creation and execution of MapReduce 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 has since evolved to work with the Apache Spark framework alongside MapReduce. Many Spark-based native analytics have been added in ORAAH, see "Native Analytic Functions" section for the complete list.

ORAAH is designed for R users to work with a Hadoop cluster in a client-server configuration. Client configurations must conform to the requirements of the Hadoop distribution that ORAAH is deployed against. This is because ORAAH uses command line interfaces (CLI) to communicate from the client node to HDFS running on Hadoop cluster. Many HDFS related functions have been moved from CLI to java-based HDFS operations, which is transparent to the user in terms of usage, but the performance improvements are noticeable. For the list of these functions see "Change List" document.

ORAAH allows R users to move data from an Oracle Database table/view into HDFS files in Hadoop. To do this, ORAAH uses the SQOOP utility. Similarly data can be moved from an HDFS file into Oracle Database. For the reverse data movement, a choice of using SQOOP or the Oracle Loader for Hadoop (OLH) utility is available and the choice depends on the size of data being moved and security requirements for 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 to use ORAAH on a different distribution see the help regarding *ORCH_HAL_VERSION* for details.

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

1. Access to HDFS objects: ORAAH HDFS APIs map 1:1 to the corresponding HDFS command line interfaces that explore HDFS files. Additionally, ORAAH exposes new functions to explore contents of HDFS files.
2. Access to Apache Hive and Apache Impala SQL functionality by overloading certain R operations and functions on `data.frame` objects that map to Apache Hive/Impala 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 (see `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/Apache Hive/Apache Impala and Oracle Database. 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.
- Apache Impala tables.
- Proprietary binary RData representations.
- Apache Spark data frame objects.

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

Before ORAAH works with delimited text files in a HDFS directory, it determines the metadata associated with the files and captures this metadata in a separate file named *ORCHMETA*, which is stored alongside the data files.

*ORCHMETA* contains the following information:

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

b. the delimiter that is the value separator;

c. the number and data types of columns in the file;

d. optional names of columns (which are available in a header line);

e. dictionary information for categorical columns (in R terminology, the 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 Apache Hive or Apache Impala tables, the *ORCHMETA* file is auto-created from the table definition.

ORAAH can optionally convert input data into R's binary RData representation (See the documentation for `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 MapReduce 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 varying key delimiter and value delimiter. Read performance of files with the same key and value delimiter is roughly two times better than that of files with different such delimiters. The key delimiter and the 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 for R based MapReduce jobs in ORAAH. When possible, it is suggested to use this binary data representation for performance sensitive analytics.

From release 2.5.0, ORAAH had included support for Spark Resilient Distributed Dataset (RDD) objects. Users can 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. In this release support for Apache Spark data frames have been added. The results of Spark-enabled analytics are returned as a Spark data frame object. Users have an option to save data from Spark's distributed memory to an HDFS directory in order to preserve it between ORAAH sessions. For more information, review the documentation of the `orch.df.fromCSV()`, `hdfs.toRDD()` and `hdfs.write()` functions.

With the 2.8.2 release we introduce a hybrid Spark/MPI platform. In this architecture, computational pipelines combine Spark-backed and MPI-backed execution stages. Normally, this happens intermittently when data is handed over from Spark tasks to MPI tasks via mapped memory shared segments and vice versa, and so Spark execution idles when MPI execution happens, and vice versa. MPI process cluster is created exclusively for each MPI stage, and is torn down after stage is done. MPI stages are used primarily to handle super-linear computations, in terms of I/O, for which Spark shuffle processesing strategies proved to be less efficient (such as matrix multiplications, some factorizations and linear system solutions). For some numerical problems that are highly-interconnected in terms of I/O, we have observed 3 and more orders of magnitude speed-ups compared to Spark shuffles.

The trade-off for MPI I/O efficiency is twofold. First, the datasets must fit into the available free shared memory of the cluster in their entirety. That is true not only for all inputs of computation, but also for the outputs, and sometimes some intermediate processing. Unlike shared-nothing architectures, the message-passing task order of execution cannot be serialized. This means that some large enough problems may be impossible to solve on limited resources, even if sufficient off-virtual-ram capacity is available for temporary needs. The fact that memory maps are used for the bulk data partitions, may alleviate some low-memory situations at expense of I/O cache and OS swap, but it is generally recommended to have sufficient extra capacity in the cluster to fit all the MPI-based stages into cluster memory comfortably.

Second, unlike spark tasks, and for similar reasons, MPI tasks cannot be re-tried (or opportunistically executed for that matter) in case of a single task or node failure for whatever reason. The entire MPI cluster of processes will have to be torn down, and MPI stage may have to be re-attempted. Also, for the same reason, the results of MPI stages are not resilient or persistent, so if a subsequent Spark computational stage loses a partition, it cannot be re-computed individually without re-running the entire MPI stage. We take care of resilience issue by ensuring proper caching strategy is chosen in the Spark pipelines once MPI results are handed over to the Spark executors, as soon as possible. However, this is limited to reliability of the Spark cache itself.

Also, any hardware failures during MPI stage execution would result in tearing down the entire MPI stage, and possibly, another re-run. There are also further architectural limitations as addressed in the "known issues" section. With that, in this release Spark/MPI hybrid executions should be considered experimental. At his point they are better suited for dense problems. Categorical predictors with high number of categories and one-hot extraction may result in computational inefficiencies and/or overloads. Further releases will work toward improving overall robustness and efficiency of Spark/MPI hybrid pipelines.

# 6 Working With the ORAAH's Hadoop Interface

ORAAH's MapReduce job definition is accomplished through `hadoop.exec()` or `hadoop.run()` APIs. The `hadoop.exec()` function is a subset of `hadoop.run()` in that `hadoop.run()` additionally supports appropriate staging of the final result from a MapReduce job. Use the `hadoop.exec()` function to build a pipeline of an ORAAH MapReduce job that streams data from one job stage to another when all of the following is true:

- The job is not required to bring intermediate results back to ORAAH environment.
- The 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 the `hadoop.run()` or `hadoop.exec()` functions as a `data.frame` object with input values and a vector object with input keys. Optionally, input can be supplied to the mapper as an R matrix, vector, or a list object (see `help(mapred.config)`). If the input data to a MapReduce job is in RData representation, set the `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 an 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 the mapper's output and hence the 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 `help(orch.keyvals)` and `help(orch.keyval)` for additional details.

Complex output object, such as a variable length character vector, 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 provides the benefit of allowing 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 space and improve performance. See the `COMPRESS` argument description of the `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 the reducer's output via the `reduce.output` field of `mapred.config` class. If this specification is unavailable, ORAAH can 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, then the `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.

## 7 Working with ORAAH's Apache Hive/Impala Interface

ORAAH allows Apache Hive and Apache Impala tables/views from default and non-default databases to be used as 'special' data frames called `ore.frame` objects. `ore.frame`'s have enough metadata to be able to generate HiveQL or Impala SQL queries when data in the Apache Hive/Impala tables/views is processed. However, not all SQL queries are supported in Apache Impala. See the following demos to understand ORAAH-Hive interface:

```
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
```

Connections to both Kerberized and non-kerberized Apache Hive/Impala servers are fully supported. For more details and examples check `help(ore.connect)`.

## 8 Working with ORAAH's Spark Interface

ORAAH allows users to create and destroy Spark sessions using `spark.connect()` and `spark.disconnect()` functions. You need to create a Spark session in R to be able to run any of the available Apache Spark-based analytic functions. For more details and examples for connection to different Spark master types check `help(spark.connect)`. Few MapReduce-based functions, namely `orch.getXlevels()`, `orch.neural()` and `hdfs.dim()`, switch over to Spark execution framework if you are already connected to Spark in your R session.

Spark-based analytics in 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 (as `ore.frame` object).
- Apache Impala tables (as `ore.frame` object).
- Any other Database table using JDBC connection (check `help(orch.jdbc)` for details and examples on how to create JDBC connection object).
- Apache Spark data frame objects created using ORAAH functions like `orch.df.fromCSV()`, running `predict()` on Spark algorithm models, or any other external means.

See the following demos to understand ORAAH-Spark interface:

```
R> demo(package="ORCH")
orch_mllib ORCH's Spark mllib algorithms
orch_formula Demonstrates the creation of design (or model) matrices using ORAAH parallel ↩
    distributed formula engine.
```

# 9 Native Analytic Functions

The following analytic functions are available out-of-the-box in ORAAH. These functions are parallel and distributed, which execute utilizing all nodes of the Hadoop cluster. All these functions have examples present in their help documentation, which can be run using `example(function_name)`.

1. Hadoop MapReduce based functions:
    a. Covariance Matrix computation (`orch.cov`)
    b. Correlation Matrix computation (`orch.cor`)
    c. Principal Component Analysis (`orch.princomp`, `orch.predict`)
    d. K-means Clustering (`orch.kmeans`, `orch.predict`)
    e. Linear Regression (`orch.lm`, `orch.predict`)
    f. Generalized Linear Model including Logistic Regression (`orch.glm`)
    g. Multilayer Feed-Forward Neural Network (`orch.neural`)
    h. Matrix completion using Low Rank Matrix Factorization (`orch.lmf`)
    i. Non-negative Matrix Factorization (`orch.nmf`)
    j. Reservoir Sampling (`orch.sample`)
2. Spark based functions:
    a. Generalized Linear Model (`orch.glm2`)
    b. Linear Regression (`orch.lm2`)
    c. Multilayer Feed-Forward Neural Network (`orch.neural2`)
3. Spark MLlib based functions:
    a. Linear regression (`orch.ml.linear`)
    b. Lasso regression (`orch.ml.lasso`)
    c. Ridge regression (`orch.ml.ridge`)
    d. Logistic regression (`orch.ml.logistic`)
    e. Support Vector Machine (`orch.ml.svm`)
    f. Decision Trees based Classification and Regression (`orch.ml.dt`)
    g. Random Forest based Classification and Regression (`orch.ml.random.forest`)
    h. Gradient Boosted Tree based Classification and Regression (`orch.ml.gbt`)
    i. Gaussian Mixture Model based Clustering (`orch.ml.gmm`)
    j. K-means Clustering (`orch.ml.kmeans`)
4. Spark with MPI based functions:
    a. Extreme Learning Machine model for Multiclass Classification and Regression (`orch.elm`)
    b. Extreme Learning Machine for Mutilayer Perceptron model for multiclass Classification and Regression (`orch.helm`)
    c. `k`-rank SVD decomposition (`orch.dssvd`)
    d. Princial Component Analysis (`orch.dspca`)

# 10 Copyright Notice

# 11 3rd Party License Notice

Oracle R Advanced Analytics for Hadoop (ORAAH)


This product includes 3rd party open source software components and libraries which are licensed under the **Apache License, Version 2.0** (the "License"). The full copy of the License can be obtained at http://www.apache.org/licenses/LICENSE-2.0.

This project includes:

- JavaCPP under Apache License 2.0
- Apache Commons Math under Apache License 2.0
- Apache Log4J under Apache License 2.0
- Apache Mahout under Apache License 2.0
- Akka under Apache License 2.0