ORAAH 2.8.2 Change List Summary

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

ORAAH 2.8.2 release marks another major step to bring Spark-based high performance analytics to the Oracle R Advanced Analytics platform. ORAAH now supports Apache Spark 2.3 and higher with added support of Spark data frames as input to all Spark based algorithms.

ORAAH now also works with Apache Impala as a data source. All supported operations on Apache Impala table have the same syntax as that of Apache Hive table. Along with Impala and Hive tables, you can also ingest data into Spark analytics from other JDBC sources like Oracle, MySQL, PostgreSQL, etc provided you have the respective JDBC drivers in your *CLASSPATH* and *SPARK_DIST_CLASSPATH*.

ORAAH continues to support legacy Hadoop mapReduce based analytics as well in the 2.x product releases, but we encourage our customers to move their code base to use the Spark versions (when available) of same functionality.

1 ORAAH 2.8.2 Changes

The following components were upgraded: * Intel MKL upgraded to 2019u5 * Apache Spark upgraded to 2.4.0 * Apache MLlib upgraded to 2.4.0 * Scala upgraded to 2.11.12 * mpich upgraded to 3.3.2 * Apache Commons Math upgraded to 3.6.1 * Apache Log4J upgraded to 2.13.0 * Antlr upgraded to 4.8 * rJava upgraded to 0.9-12 * rJDBC upgraded to 0.2-8

The following issues were fixed: * Fixed misleading error message in the intall postcheck. * spark.connect always reports 2 cores/instances.

1.1 Non backward-compatible API changes:

- None.

1.2 Backward-compatible API changes:

- None.

2 ORAAH 2.8.1 Changes

Support for CDH version 6.0.0 and a number of bug fixes.

2.1 ORCHcore

ORCHcore includes a number of code improvements and bug fixes as listed below.

2.1.1 Non backward-compatible API changes:

- None.

2.1.2 Backward-compatible API changes:

- spark.connect()

ORAAH now resolves the active HDFS Namenode for the cluster if the user does not specify *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.

`spark.connect()` has a new logical parameter *enableHive*. If set to `TRUE`, it enables Hive support in the new Spark Session. Hive support enables usage of Hive tables directly as input source for Spark based algorithms in ORAAH. If there is an active Hive connection when `spark.connect()` is invoked with the default value (NULL), `enableHive` will be set to `TRUE`.

For Hive supported to be enabled in the Spark session, you need to have your Hive configuration available at the client side in the form of *hive-site.xml* file in your *CLASSPATH*. You can do so by adding your *HIVE_CONF_DIR* to the *CLASSPATH* before starting R and loading ORCH library.

Alternatively, you can specify the Hive metastore details as part of `spark.connect()` by specifying parameters like: `hive.metastore.uris="thrift://METASTORE_NAME:METASTORE_PORT"`

Also, if your Hive is kerberized then you need to additionally specify other parameters like: `spark.connect(..., enableHive=TRUE, hive.metastore.uris="thrift://METASTORE_NAME:METASTORE_PORT" hive.metastore.sasl.enabled="true", hive.metastore.kerberos.principal=HIVE_PRINCIPAL, hive.security.authorization.enabled="false", hive.metastore.execute.setugi="true", ...)`.

A new parameter *logLevel* was also added to `spark.connect()`. This value overwrites the logging category specified in Apache Spark's *log4j.properties* file.

`spark.connect()` now loads the default Spark configuration from *spark-defaults.conf* file if it is found in the *CLASSPATH*. If such 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.

New defaults were 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.

For more details check `help(spark.connect)` after loading `library(ORCH)`.

- orch.connect()

On encountering an error while connecting to an Oracle database `orch.connect()` now prints the exact failure message, instead of a generic failure message. For example, *invalid username/password*, *Network adapter couldn't connect to host*, *ORACLE not available*, etc.

- hdfs.dim()

When new CSV data files are loaded in HDFS using `hdfs.upload()` or a new HDFS directory is attached using `hdfs.attach()` the dimensions of this data are unknown. Invoking `hdfs.dim()` on such HDFS path starts a mapReduce program (on user prompt) to compute the dimensions. In ORAAH 2.8.1, if you are connected to Spark using `spark.connect()` then instead of a slower Hadoop mapReduce computation, dimensions are determined using a Spark task. This results in a faster reponse time of `hdfs.dim()`.

- hdfs.fromHive()

`hdfs.fromHive()` now works with both Apache Hive and Apache Impala tables. Support for Apache Impala connection has been newly included in this release.

- hdfs.toHive()

`hdfs.toHive()` now works with both Apache Hive and Apache Impala tables. Support for Apache Impala connection has been newly included in this release.

Also, the *ore.frame* object is returned invisibly now. This prevents `hdfs.toHive()` from printing the entire *ore.frame* if the return value was not assigned to another R object.

- hdfs.push()

Specifying paramter *driver="olh"* with `hdfs.push()` caused failures in earlier versions of ORAAH. This has been fixed in this release.

- orch.create.parttab()

Previously, `orch.create.parttab()` would fail in a few cases. If the *input* was one either an Apache Hive table with first column of *string* type, or an *hdfs.id* object returned by `hdfs.push()` function. It would also fail if *partcols* specified a column that had string values containing spaces. All these issues have been fixed in this release.

- hdfs.write()

There was an issue with `hdfs.write(..., overwrite=TRUE)` wherein after overwriting an HDFS path the ORAAH's HDFS cache for that particular path would go out of sync in some cases. Causing a failure on invoking `hdfs.get()` on that HDFS path. This problem has been resolved in this release.

- hdfs.upload()

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

- mapReduce Queue selection for ORAAH mapReduce jobs

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 user to select the mapReduce queue to which the mapReduce task will be submited. For more details, check `help(mapred.config)`.

2.1.3 New API functions:

- orch.jdbc()

The Apache Spark based analytics in ORAAH 2.8.1 can read input data over a JDBC connection. `orch.jdbc()` facilitates creation of one such JDBC connection identifier object. This object can be provided as *data* in any of the Spark based algorithm available from `ORCHstats` and `ORCHmpi` packages.

User must have the JDBC driver libraries for specific database, you want to read the input data from, in your *CLASSPATH*. Also, since Spark reads the JDBC input in a distributed manner, this JDBC library jar file should be present on all of your cluster nodes at the same path, and make sure to add this path to *SPARK_DIST_CLASSPATH* in *Renviron.site* configuration file. See install guide manual for more details on configuration.

For more details and examples, check `help(orch.jdbc)`.

- orch.jdbc.close()

`orch.jdbc.close()` is used to close the JDBC connection started by `orch.jdbc()`. It is recommended to close the connections once the data has been read and the connection will no longer be used.

For more details and examples, check `help(orch.jdbc.close)`.

- orch.df.scale()

`orch.df.scale()` is used to scale the numerical columns of a data frame.

For a list of available scaling techniques and other details, check `help(orch.df.scale)`.

- orch.df.summary()

Creates and returns a summary of the Spark data frame.

For more details and examples, check `help(orch.df.summary)`.

- orch.df.sql()

`orch.df.sql()` executes a Spark SQL query. This function is used to run an Apache Spark SQL query on a Spark SQL view created using `orch.df.createView()`. The results of the SQL query can then be collected in R session using `orch.df.collect()`.

For more details and examples, check `help(orch.df.sql)`.

- orch.df.createView()

`orch.df.createView()` creates or replaces a temporary Spark SQL view. Creating a Spark SQL view is needed is you wish to run Spark SQL query on an exisiting Spark data frame. This functions registers a Spark data frame as a SQL view. The SQL queries can be submitted using `orch.df.sql()`.

For more details and examples, check `help(orch.df.createView)` and `help(orch.df.sql)`.

- orch.df.collect()

Collects a Spark data frame to client's memory, and returns an equivalent R data frame.

For more details and examples, check `help(orch.df.collect)`.

- orch.df.describe()

Computes and returns statistics for numeric columns of a Spark data frame. If no columns are specified, then `orch.df.describe()` computes statistics for all numerical columns present in the Spark data frame.

For more details and examples, check `help(orch.df.describe)`.

- orch.df.fromCSV()

Creates a Spark data frame from comma-separated values (CSV) data source. It can read data from local file system and HDFS both. If the HDFS file was previously attached in ORAAH, the existing metadata that was created during the previous `hdfs.attach()` operation will also be used.

For more details and examples, check `help(orch.df.fromCSV)`.

- orch.df.persist()

`orch.df.persist()` persists a Spark data frame with the desired Spark storage level specified in the function call. This makes the Spark data frame independant from the source it was created from. For example, a Spark data frame created from CSV files in HDFS using `orch.df.fromCSV()` will still be usable if the HDFS file was deleted, if it was persisted using `orch.df.persist()`.

For more details and examples, check `help(orch.df.persist)`.

- orch.df.unpersist()

`orch.df.unpersist()` unpersists a Spark data frame that was persisted using `orch.df.persist()`.

For more details and examples, check `help(orch.df.unpersist)`.

- orch.reconf()

With this release, the loading time of `library(ORCH)` has been significantly reduced. The mandatory environment and Hadoop component checks on the 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 invoked `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.

- orch.destroyConf()

`orch.destroyConf()` 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.

2.1.4 Other changes:

- New java based HDFS FileSystem interface

ORAAH now has a new client side HDFS File system interface 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 initialise, then the older version of hdfs.* functions will work without any speed ups.

- Disabled mandatory ORCH loading checks

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.

- ORCH_JAVA_XMS environment variable

To improve the configuration options for the `rJava` JVM that runs alongside ORAAH in R, support for `-Xms` has been added. To change the default value of `256m` 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)`.

- Handling of JAVA_TOOL_OPTIONS

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

- Handling of [Ctrl + C] interrupt

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.

- Removes HIVE CLI usage from ORCHcore

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

- Improved install and uninstall scripts for client

ORAAH 2.8.1 has a new package ORCHmpi, which exposes `MPI` based algorithms. They rely on `MPI` libraries present on the nodes to proceed with the computations. It is recommended you build MPI on your nodes (at the same path) and make it available to ORAAH using *ORCH_MPI_LIBS* and *ORCH_MPI_MPIEXEC* environment variables. ORAAH installer comes with a pre-built version (for Oracle Linux 6) of these MPI libraries and the installer places them alongside the other libraries installed with ORAAH. You can specify your *Renviron.site* configuration file path to the client installer, if it is present at a non-default location. The client installer will add the above two setting in the same file.

For more details check `help(ORCH_MPI_LIBS)`, `help(ORCH_MPI_MPIEXEC)` and the install guide manual.

- Improved install and uninstall scripts for server

If the *dcli* tool needed for server installation (on a Big Data Appliance cluster, in case the file list containing node names is not specified) is not present in *PATH*, then a few known locations for the tool are probed before giving up and throwing the error during installation.

2.2 ORCHstats

With the ORAAH 2.6.0 release we had introduced support for selected algorithms from Apache MLlib with support from our proprietary optimized data transformation and data storage algorithms. This interface is built on top of Apache Spark and designed to improve performance and interoperability of MLlib and ORAAH machine learning functionality with R. Building on the initial success of the Apache MLlib integration, ORAAH 2.7.0 expanded coverage of exported MLlib algorithms and greatly improved support for the R formula engine.

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 the R formula, which closely follows the R formula syntax, but is much more efficient from the performance perspective. See the reference manual for detailed information about features supported in the Oracle formula.

With ORAAH 2.8.1, we now support Apache Spark 2.3 and higher. All Spark based algorithms now work with Spark data frame as input. The details on the various supported input types for these algorithms is present in the help of each such algorithm available.

ORCHstats includes a number of code improvements and bug fixes as listed below.

2.2.1 Non backward-compatible API changes:

- orch.model.matrix()

Parameter *maxBlockSize* has been removed.

A parameter *storageLevel* has been added to control the storage level of the input Spark data frame, which can be generated from various sources like CSV data, Apache Hive tables, Apache Impala table, any other JDBC source. Also, `"labeledPoint"` and `"vector"` *type* are no longer needed or supported.

For more details and examples, check `help(orch.model.matrix)`.

- orch.ml.logistic()

Parameter *nClasses* has been removed, since number fo target classes is determined automatically. Other parameters that were removed are *convergenceTol*, *featureScaling*, *regParam* and *validateData*.

New parameters *threshold*, *maxBlockRows* and *storageLevel* have been added.

For more details and examples, check `help(orch.ml.logistic)`.

- orch.ml.linear()

Parameters *miniBatchFraction*, *featureScaling*, *stepSize* and *validateData* have been removed.

New parameters *elasticNetParam*, *regParam*, *standardization*, *maxBlockRows* and *storageLevel* have been added.

For mode details and examples, check `help(orch.ml.linear)`.

- orch.ml.ridge()

Parameters *miniBatchFraction*, *featureScaling*, *stepSize* and *validateData* have been removed.

New parameters *standardization*, *maxBlockRows* and *storageLevel* have been added.

For more details and examples, check `help(orch.ml.ridge)`.

- orch.ml.lasso()

Parameters *miniBatchFraction*, *featureScaling*, *stepSize* and *validateData* have been removed.

New parameters *standardization*, *maxBlockRows* and *storageLevel* have been added.

For more details and examples, check `help(orch.ml.lasso)`.

- orch.ml.svm()

Parameters *miniBatchFraction*, *featureScaling*, *stepSize* and *validateData* have been removed.

New parameters *threshold*, *maxBlockRows* and *storageLevel* have been added.

For more details and examples, check `help(orch.ml.svm)`.

- orch.ml.gmm()

Parameter *convergenceTol* has been removed.

New parameters *maxBlockRows* and *storageLevel* have been added.

For more details and examples, check `help(orch.ml.gmm)`.

- orch.ml.kmeans()

Parameter *nParallelRuns* has been removed.

New parameters *maxBlockRows* and *storageLevel* have been added.

For more details and examples, check `help(orch.ml.kmeans)`.

- orch.ml.dt()

Parameter *nClasses* has been removed, since number fo target classes is determined automatically.

New parameters *minInstancesPerNode*, *minInfoGain*, *maxCategories*, *threshold*, *maxBlockRows* and *storageLevel* have been added.

For more details and examples, check `help(orch.ml.dt)`.

- orch.ml.random.forest()

Parameter *nClasses* has been removed, since number fo target classes is determined automatically. Parameter *seed* has also been removed.

New parameters *threshold*, *maxCategories*, *minInstancesPerNode*, *minInfoGain*, *maxBlockRows* and *storageLevel* have been added.

For more details and examples, check `help(orch.ml.random.forest)`.

- orch.ml.pca()

This function has been removed from this release.

2.2.2 Backward-compatible API changes:

- orch.lm2()

A parameter *storageLevel* has been added to control the storage level of the input Spark data frame, which can be generated from various sources like CSV data, Hive tables, Impala table, any other JDBC source.

- orch.glm2()

A parameter *method* has been added to switch between algorithms to solve the underlying optimization model. You can choose between `"irls"` iteratively reweighted least squares (recommended and default) or `"lbfgs"` for a limited memory environments.

A parameter *storageLevel* has been added to control the storage level of the input Spark data frame, which can be generated from various sources like CSV data, Apache Hive tables, Apache Impala table, any other JDBC source.

- orch.neural2()

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

A parameter *storageLevel* has been added to control the storage level of the input Spark data frame, which can be generated from various sources like CSV data, Apache Hive tables, Apache Impala table, any other JDBC source.

2.2.3 New API functions:

- 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.

For more details and examples, check `help(orch.ml.gbt)`.

- orch.summary()

`orch.summary()` 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. `orch.summary()` returns an `ore.frame` in all cases except when the *group.by* argument is used. If the *group.by* argument is used, then `orch.summary()` returns a list of `ore.frame` objects, one object per stratum.

For more details about the function arguments and examples, check `help(orch.summary)`.

- orch.mdf()

MLlib machine learning and statistical algorithms require a special row-based distributed Spark data frame for training and scoring. `orch.mdf()` is used to generate such MLlib data frames that can be provided to any of the MLLib algorithms supported in ORAAH.

If not created by the user, the MLlib algorithms will generate these input data frames themselves. But if you want to run multiple 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.

For more details and examples, check `help(orch.mdf)`.

2.2.4 Other changes:

- Target Factor Levels in MLlib models

MLlib model objects have target factor levels associated with them (if available) to help the user better understand the prediction results and labels.

2.3 OREbase

OREbase now supports connection to Apache Impala databases. Apache Impala is a distributed, lighting fast SQL query engine for huge data stored in Apache Hadoop clusters. It is a massively parallel and distributed query engine that allows 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.

OREbase now also includes better support for Apache Hive than the earlier releases.

This release includes a number of code improvements and bug fixes as listed below.

2.3.1 Non backward-compatible API changes:

- None.

2.3.2 Backward-compatible API changes:

- ore.connect()

Support for new connection type for Apache Impala is 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)`.

- ore.create()

With this release, the `NA` values in R's `data.frame` are converted to `NULL` in Apache Hive tables uniformly (irrespective of the size of R's `data.frame` object).

Previously, `ore.create(view="<view_name>")` failed after assigning row.names to R's `data.frame` object. This issue has been fixed in this release.

Also, creating Apache Hive table from large R `data.frame` objects could cause an error of type: `SemanticException Line 1:23 Invalid path "tmp_file_path":No files matching path file:tmp_file_path)`. This issue has been resolved in this release.

- ore.sync()

In earlier releases, syncing a paritioned HIVE table raised an error regarding unsupported data types. Unless you had the property *hive.display.partition.cols.separately* set in *hive-site.xml* to *false*. Setting this property is no longer required since it interferes with other Hadoop components like Hue.

- gsub() support in Apache Hive

The subexpression selection in *replacement* string has been fixed in this release. `gsub()` now works well with `ore.character` types.

- Division operator

The Division operator previously generated `NA` in Apache Hive when faced with values causing division by zero, or zero by zero, and such other corner cases. In this release handling of these cases has been improved and `Inf`, `-Inf` and `NaN` are generated, which is similar to the behavior of R's division operator.

2.3.3 New API functions:

- ore.impalaOptions()

Similar to `ore.hiveOptions()` this function sets Apache Impala options, namely, field delimiters for the Apache Impala tables and the current database name.

For more details and examples, check `help(ore.impalaOptions)`.

- ore.showImpalaOptions()

Similar to `ore.showHiveOptions()` this function displays the current value of all the Apache Impala options, namely, field delimiters for the Apache Impala tables and the database name.

For more details and examples, check `help(ore.showImpalaOptions)`.

2.3.4 Other changes:

- Automated JDBC driver lookup

From release 2.7.0, ORAAH started checking for the Hive and Hadoop libraries at certain known locations for different type of Hive installations like RPM, or Cloudera Parcel. These locations are scanned and the relevant java libraries are added to the CLASSPATH for the driver when initiating the RJDBC connection. If you have Hive running from a custom install location you can specify environment variables ORCH_HADOOP_HOME (or default HADOOP_HOME) and ORCH_HIVE_HOME (or default HIVE_HOME) and the required libraries will be picked from there. Also, if any jar file is missing, an appropriate warning message is reported.

This technique is also used for newly supported Apache Impala connections. It prevents any additional library from being added to the *CLASSPATH*.

2.4 ORCHmpi

In ORAAH 2.8.1 we have introduced a new package `ORCHmpi`. This new 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. For more information on setting up MPI, check `help(ORCH_MPI_LIBS)` and `help(ORCH_MPI_MPIEXEC)`. If MPI is configured properly, `orch.mpiAvailable()` and `orch.scalapackAvailable()` functions will return `TRUE`.

Environment variable *ORCH_MPI_MAX_GRID_SIZE* is used to specify the maximum number of MPI workers (not counting the leader process) that MPI computation may spawn on the cluster per submission. It is recommended to set this to an integer value, with maximum being no more than 60 percent of available cluster CPU cores.

The list of new functions exported from `ORCHmpi` are listed below.

2.4.1 Non backward-compatible API changes:

- None.

2.4.2 Backward-compatible API changes:

- None.

2.4.3 New API functions:

- orch.mpiAvailable()

This functions checks if proper MPI subsystem is available. For more details and examples, check `help(orch.mpiAvailable)`.

- orch.scalapackAvailable()

This functions checks if the proper Scalapack subsystem is available. For more details and examples, check `help(orch.scalapackAvailable)`.

- orch.mpi.cleanup()

This function cleans up stuck MPI processes and shared memory segments on the cluster using Apache Spark tasks. This is only necessary as a last recourse if less-than-graceful crash occurred during MPI phase execution, and the driver process (which otherwise automatically cleans up failed MPI jobs) has failed as well.

For more details and examples, check `help(orch.mpi.cleanup)`.

- orch.mpi.options()

This function is used to set MPI stage execution options, like number of MPI computation retries in case of failures, the number of MPI cleanup tasks to start on `orch.mpi.cleanup()` and the maximum number of MPI workers.

For more details and examples, check `help(orch.mpi.options)`.

- orch.elm()

This function is used to create an *Extreme Learning Machine* model based on *L1 FISTA* fitting. This algorithm can be used for multiclass classification and regression. The functions `predict()`, `summary()` and `coef()` can be used on the `orch.elm` models to generate predictions on new data, display summary of the model, and extract different coefficients available in the model.

For more details and examples, check `help(orch.elm)`.

- orch.elm.load()

This function is used to load a saved `orch.elm` model from HDFS.

For more details and examples, check `help(orch.elm.load)`.

- orch.helm()

This function is used to create an *Extreme Learning Machine for Mutilayer Perceptron* model. This algorithm can be used for multiclass classification and regression. The functions `predict()`, `summary()` and `coef()` can be used on the `orch.helm` models to generate predictions on new data, display summary of the model, and extract different coefficients available in the model.

For more details and examples, check `help(orch.helm)`.

- orch.helm.load()

This function is used to load a saved `orch.helm` model from HDFS.

For more details and examples, check `help(orch.helm.load)`.

- orch.dssvd()

This function computes reduced, approximate `k`-rank SVD decomposition. The function coef() can be used on `orch.dssvd` model to extract the coefficients from the model.

For more details and examples, check `help(orch.dssvd)`.

- orch.dssvd.load()

This function loads the `orch.dssvd` model from HDFS.

For more details and examples, check `help(orch.dssvd.load)`.

- orch.dspca()

This function run a distributed MPI backed Princial Component Analysis (PCA). The function `coef()` can be used to extract coefficients from the `orch.dspca` model.

For more details and examples, check `help(orch.dspca)`.

- foldInMx.orch.dspca()

The fold-in PCA operation computes aproximation of new datapoints folded into PCA space. Note that result of this routine is unscaled PCA space.

For more details and examples, check `help(foldInMx.orch.dspca)`.

- foldIn.orch.dspca()

The fold-in PCA operation computes aproximation of new datapoints folded into PCA space. Note that result of this routine is unscaled PCA space. The formula used for approximation of new data points is different than the one used by `foldInMx.orch.dspca`.

For more details and examples, check `help(foldIn.orch.dspca)`.

2.4.4 Other changes:

- None.

3 Copyright Notice