Book chapter final draft

Book info

Editor: Olivier Terzo, Lorenzo Mossucca

Booker: Cloud Computing with E-science Applications

ISBN-10: 1466591153

Publisher: CRC Press/Taylor & Francis

http://www.amazon.co.uk/Cloud-Computing-Science-Applications-Olivier/dp/1466591153

RPig: Concise Programming Framework by Integrating R with Pig for Big Data Analytics

MingXue Wang, Sidath B. Handurukande

Mingxue.Wang@Ericsson.com, Sidath.Handurukande@Ericsson.com

Network Management Lab, Ericsson, Ireland.

1 Introduction

With the explosive growth in the use of ICT, applications that involve deep analytics need to be shifted to scalable solutions for big data. Our work is motivated by the big data analytic capabilities of network management systems, such as network traffic analysis, in the Telecom domain. More specifically, the work is an extension of Apache Pig/Hadoop frameworks which are commonly used to build cost effective big data system in industry. The design, the developed software implementation and the solution we describe here are general and applicable to other domains.

To build a scalable system, one approach is to use distributed parallel computing models, such as MapReduce[1], that allow to add more (computer) nodes into the system to scale horizontally. MapReduce has been recently applied to many Data Management Systems (DMS), such as, Hadoop and Pig. These systems target the storage and querying of data for top layer applications. However, they lack the necessary statistical and machine learning algorithms and therefore can only be used for simple data analysis. For advanced or deep analysis, Mahout[2] contains a limited number of machine learning algorithms implemented in the MapReduce model. Because of the large number of and complexity of machine learning and statistical algorithms, the redesign and redevelopment of these algorithms in MapReduce model are difficult tasks. Various algorithms are still missing in Mahout in comparison with matured statistical and machine learning frameworks. For example, Support Vector Machines (SVM), one of commonly used algorithms is still under development in the Mahout. On the other hand, traditional statistical software, such as R, has a very rich and extensive set of machine learning and statistical processing functionalities for advanced analysis, but it is not distributed and not scalable on its own. In general, it only runs within a single computer and requires all data to be loaded into memory for processing. Some solutions have been proposed to scale out this traditional statistical software, such as RHadoop[3], but limitations still exist. For example, some require writing key-value paired map and reduce functions leading to difficulties in usage and longer development time. More detail of related work is described in section 6. Our approach addresses the problem by integrating traditional and matured statistical software (R) with a scalable DMS (Pig) to scale out deep analytic.

In this chapter, we present RPig, an integrated framework with R and Pig for scalable machine learning and advanced statistical functionalities, which makes it feasible to use high-level languages to develop analytic jobs easily in concise programming. RPig takes advantage of both the deep statistical analysis capability of R and parallel data processing capability of Pig. Both data storage and processing for deep data analysis are distributed and scalable. The framework has the following main advantages:

- The statistical and machine learning functions of R can be easily wrapped and be directly used with Pig statements. This allows developing advanced parallel analytic jobs with two high level languages R and Pig (Latin), without needing to learn new languages, APIs or rewrite complex statistical algorithms. The development effort can be significantly reduced for the user.

- The framework is able to parallelize both R and Pig executions automatically at the execution stage. The necessary low level operations such as data conversion, fault handling, are handled by the framework itself. The framework offers automatic parallel execution for advanced data analysis.

In the rest of the chapter, we describe two scenarios that we encounter in section 2, that neither R nor Pig can handle independently. Section 3 describes the foundation frameworks: R, Hadoop and Pig. The overall RPig framework and its components are explained in section 4. Experiments and results are in section 5. Finally, we talk about related work and give our conclusion (section 6 and 7).

2 Motivating Scenarios

To demonstrate the need and usefulness of our RPig framework, we describe two example use cases in the context of network management systems where scalable statistical processing is necessary.

2.1 Both IO and CPU intensive scenario with EMA

In this first use case, vast amount of events are collected from a given mobile network, and stored as event log files. An event is a report about a particular service client (e.g., Viber-VoIP service client) and contains information such as:

ID|period_start| period_end| IMSI|IMEISV|RAT|…| packets_downlink| packets_uplink|…

Exponential Moving Average (EMA) is a simple forecasting algorithm based on historical sample data. Using the EMA, an analytic feature of a network management system can forecast the amount of traffic of selected service clients in the next time window when a request is sent. Because of the vast number of events, it is impossible for R to load all data into memory for a simple EMA calculation. However, Pig does not have the EMA function, which R has.

The above problem can be addressed by RPig, which allows log files to be efficiently loaded, preprocessed (filtering, aggregating, etc.) by Pig in parallel, and then directly pass the data to R for a final EMA calculation. In this case, it is both IO and CPU intensive scenario as it requires loading and preprocessing massive log files from hard disks.

2.2 A CPU intensive scenario with SVM

SVM machine learning algorithms can be used for advanced classification and regression analysis. Unknown data can be predicted by an SVM model, which is built from training data in the training phase.

An increasing amount of phone calls are made by various VoIP clients, such as Viber, Skype. One approach for monitoring the service quality of VoIP is using network level key performance indicators (N-KPIs) at IP layer, such as packet loss, jitter, to predict the mean opinion score (MOS), which is a standard speech quality measurement parameter [4]. An SVM based regression algorithm is used in this case, but it is a complex algorithm usually involving long computation times on a relatively small amount of data in the training phase. RPig enables us to define and execute the SVM algorithms in MapReduce model for both SVM training and prediction phases without writing any key-value pair MapReduce functions. As a result, the performance becomes scalable to cluster size, and development effort is reduced.

This use case deals with a complex machine learning algorithm, which is CPU intensive rather than IO intensive. R's in-memory computation takes most of the overall computation time with a small size of data in an analysis job. RPig supports parallelism for various requirements in different scenarios.

3 Background

Big data[5] is data in volumes so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications. Since Google published its MapReduce technology and Apache started the Hadoop project in 2004 and 2005, MapReduce and Hadoop have become a generic and foundational approach for developing scalable, cost effective, flexible, fault tolerant big data systems[6]. Many frameworks, such as Pig and Hive, have been developed based on Hadoop, adding features on it. As Hadoop systems are more widely adopted in industry, the requirements of the real world problems are driving the Hadoop ecosystem to become even richer. For example, Oozie and Azkaban provide workflow and scheduling management. Impala and Shark aim on low latency real-time queries. Our work, RPig, is one of many frameworks, such as Mahout and DataFu[7], targeting deep analytics. In the following sub sections, we briefly describe the frameworks on which the RPig is based.

3.1 R and R packages

R is a programming language and software environment widely used for statistical computing and deep data analysis, such as classification, and regression. R is extensible through R packages. There are thousands of R packages that implement massive specialized machine learning and statistical algorithms.

R's data model contains simple data structure types, such as, *scalars, vectors, lists,* also special compound data structures types: *factors* are used to describe items that can have a finite number of values; *data frames* are *matrices,* may contain different data types (*numeric, factor,* *etc*.). All data structures of R are R objects, which also include other statistical specific models or functions, etc.

The following code snippet shows a simple example of EMA calculation using R. *TTR* is an R package implementing various moving average calculations. The *temp* is a series for EMA calculation with 20 periods to average over.

```
Library(TTR); results <- EMA(temp, 20)
```

3.2 Hadoop and MapReduce

Hadoop offers a distributed file system (HDFS) managing data storage and a MapReduce [5] based distributed parallel programming framework for data processing. Computations are defined in *Map* and *Reduce* functions, which have key-value pairs for input. A map function takes one pair of data which can be processed in parallel *Map(k1,v1)→list(k2,v2)*. A reduce function aggregates related results of map functions *(k2, list(v2))→list(v3)*. Programs need to be written as map and reduce programs to enable parallel computing through Hadoop MapReduce Java APIs.

3.3 Pig and Pig Latin

Pig is built on top of Hadoop and gives a high-level dataflow language called Pig (Latin) [8] for expressing data query and processing. It is similar to SQL of RDBMS, but it is procedural style and gives more control and optimization over the flow of the data. Pig scripts are compiled into sequences of MapReduce jobs by Pig, and they are executed in the Hadoop MapReduce environment.

Pig data model contains scalar types which contain a single atomic value (*integer*, *long*, etc.), and three complex types which can contain other types: *Tuple* is a data record consisting of a sequence of 'fields', which can be any data type; *Bag* is a set of tuples, similar to a “table”; *Map* is a map of a string key to a value, which can be any data type.

Pig provides a set of operators for data processing. For example: *LOAD* and *STORE* can be used for reading and writing data from HDFS. Processing every tuple of a data set can use the FOREACH operator. Many operators are similar to SQL, such as *JOIN*, *GROUP BY* and *UNION* for standard data operations. As with many SQL implementations, Pig supports User-Defined Functions (UDF) which allows performing tasks written in low level language (Java or Python) to extend Pig. The following Pig script shows how to aggregate traffic consumption (both up/down link) on selected VoIP clients (e.g., Skype, Viber) in a time window on events described in Section 2.1.

```
Events = LOAD '$load_par' USING PigStorage('|') AS (ID, period_s:LONG, ..., );
Events = FILTER Events BY (client == 'Viber' OR ...);
Traffics = FOREACH (GROUP Events BY (period_s, period_e, client)) GENERATE
            FLATTEN (group), (SUM(Events.downlink)+SUM(Events.uplink)) AS links:DOUBLE;
```

4 The framework

An initial version of the RPig framework[9] was implemented as a proof-of-concept prototype. The framework provides the RPig script for users to write analytic jobs. The RPig script inherits Pig script syntaxes as the language skeleton, but allows defining inline R scripts as R functions. An R function element will be interpreted as an input payload of a predefined Pig extended function or Pig UDF, which handles the payload at the execution stage. This design gives us a quick implementation by only using the Pig UDF APIs without going through the Pig source code. However, it is not an optimal approach for integrating Pig and R. RPig script has its own constructs and it needs to generate additional Pig supporting statements in execution. The initial version also has the large performance overhead of the data exchange between R and Pig.

To improve the performance of RPig and to integrate R and Pig in an optimal way, we completely redesigned and rewrote the source code to overcome the aforementioned disadvantages of the initial version. By doing so, we have brought the research prototype to an early production stage. Some of the main advantages of the current version over the initial proof-of-concept version are:

- Seamlessly integrated with Apache Pig by having a built-in R script extension similar to other Pig script extensions, such as Python and JavaScript.
- Only standard R and Pig language syntaxes are used without any new language constructs. It allows the use of any existing R and Pig script IDEs.
- Supporting two types of R engines. R UDFs can be executed on the JVM or a standalone R engine.
- Much faster performance. Optimized data conversion and verbosity XML messages are not involved as the intermediate data format.

In the following subsections, we will describe the current version of framework in detail.

4.1 The R script engine extension

To integrate R and Pig and take advantage of both, the R language is expected to be supported to define Pig UDFs for specifying custom processing in Pig data flows. Pig already supports a number of languages, such as Python and JavaScript for UDFs. They are implemented as different script engine extensions in Pig. I.e., an R script engine extension (*RScriptEngine*) is required for our case. It wraps the R engine in the back-end which can interpret R scripts at runtime (Figure 1). The user defines R functions as UDFs in an R script and makes Pig aware of the R script by using the Pig *REGISTER* statement in a data flow (step 1 of Figure 1). An *RScriptEngine* will be initialized and it will register the defined R functions. The *RScriptEngine* will be shipped within Pig generated MapReduce programs to all Hadoop task nodes during execution (step 2 of Figure 1). *RScriptEngine* can execute the registered R functions in the back-end R engine by providing a bridge function for interactions between Pig and R (step 3 of Figure 1).

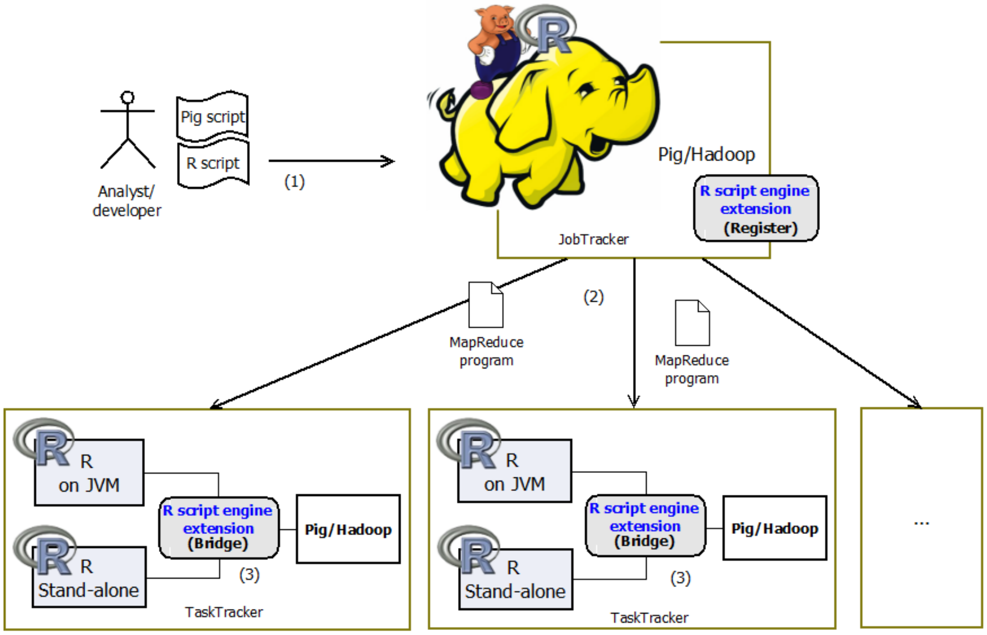

Figure 1: The framework overview

The back-end script engine is usually selected from the Java implementations of the script language. For example, Jython and Rhino are used for Python and JavaScript back-end engines respectively. This enables running the script languages on the JVM where Hadoop and Pig are running. Hence, no additional back-end script engine is required to be installed on every host along with the JVM. However, this is not a case for the R language since there is no mature JVM-based R interpreter. Some preliminary implementations are available, such as Renjin[10], but it is incompatible with most R packages except for some basic libraries. As a result, two types of back-end R engine are supported in our implementation (JVM-based and Standalone R). JVM based R can be used for some basic statistical functions e.g. standard deviation, without requiring the installation of a script engine on all nodes. Standalone R is able to use any R functions and packages. However, the R must be pre-installed on all hosts.

4.2 Data type conversions

It is necessary for data to be passed back and forth between Pig and R during the R functions executions. Since two languages Pig and R, have very different data models, the data must go through a conversion process, which is one of the main responsibilities of the R bridge function. The data type conversion is done automatically based on the following set of predefined rules.

4.2.1 From Pig to R:

- Simple data type
    - *int: integer; long/float/double: double; chararray/datetime: character bytearray: raw; boolean: logical* (e.g. null: NULL); *datatime: POSIXlt,POSIXt*;
- Complex data type
    - *tuple: list*, e.g. (19,2): list(19,2); *dataBag: nested list*, e.g. {(19,2), (18,1)}: list(list(19,2), list(18,1)); *map: named list*, e.g., [apache#pig]: list(key="apache", value="pig")
- Anything else raises an exception

Any nested data objects in Pig, such as nested tuples, will be converted to nested lists in R. Due to the different purposes of the two languages there is no exact semantic match between all data types in the their data models. For example, the *map[key#value]* type of Pig is hardly used in statistical computing, so we convert it to a named *list(key=key, value=value)* which is an ordered collection in R. Users can still convert the converted R object to other R data types via R operations (inside R functions) if necessary. For example it would be possible to convert a nested list to a data frame or a matrix.

4.2.2 From R to Pig:

When the data must be sent back to Pig after R execution, a user defined output schema of the R function is needed. This allows the user to specify what they expected from the R output and remove ambiguity during the conversion. For example, the *logical* value in R could be True, False, or NA (Not Available), but the Pig *boolean* type can only be either True or False. By using the output schema, the *logical* value can be converted to a boolean value. Alternatively the user may specify an *int* or *chararray* value and no semantic information is lost. The following rules are used for type conversion from R to Pig:

- Simple data type
    - *(schema: int) numeric/integer/logical/factor: int* (T:1; F:0; NA:128); *(schema: float/double) numeric/double: double* ; (schema: chararray) *character/logical/factor: chararray*; *(schema: bytearray) raw: bytearray*; *(schema: boolean) logical: boolean* (T: T, F/NA: F); *NULL: null; (schema: datetime) POSIXlt/POSIXt: datetime*
- Complex data type
    - (schema: tuple) *numeric array/character array/logical array/factors/list*: *tuple*, e.g. structure(c(1L, 2L, 1L), .Label = c("a", "b"), class = "factor"): (a, b, a); (schema: bag) *nested list: bag*; (schema: map) *list: map*
- Anything else raises an exception

4.3 Execution and Monitor

At the parallel execution stage, the defined R functions or UDFs are transformed into map functions which are automatically generated by taking advantage of Pig. They are executed in parallel in different Hadoop task nodes. Each R or map task will take a piece of split data and execute independently on an R engine on one task node. If the Hadoop cluster is configured with more than one map task capacity per node, each map or R function will have an isolated session. When a task is completed and a result is returned, the data stored in the R session will be cleared and the process will be killed by the RPig framework. As a consequence, no R session will be kept alive after the R execution is complete, and all data that needs to be saved or persisted from R must be saved in HDFS through Pig operations. This design is chosen because an R session only exists in a single task node, which is replaceable by any other task node in a Hadoop cluster at any time. The R session cannot be retrieved by other nodes at a later time. Pig stores data, including temporary data generated between MapReduce jobs during processing, in HDFS to guarantee that data can be retrieved later from every node of the cluster. The results of all R functions will be collected through Reduce tasks for continuous processing. Users do not need to develop key-value pair map and reduce functions within RPig. They only need to assign the number of map and reduce tasks in parallel execution through Hadoop and Pig configuration.

With regard to fault tolerance, the fault handling happens in two different layers, the node layer and the R engine layer. The underlying Hadoop framework provides failure handling on nodes of the cluster. If a node fails during the execution of an RPig job, Hadoop will restart the task of the failed node in an alternate node. Within a node task, the RPig framework allows the user to define the fault policy to handle errors from an R engine execution on an R function. For example, by default *func_name.fault.ignore←T*. This policy ignores any exceptions and continues. Also *func_name.fault.retry←1*. This allows at most one retry when an exception occurs. If an R execution fails during the map task, then a remedy action defined in a failure policy of the named R function will be applied and the failure event will be logged by the RPig framework. The user still can use R's *tryCatch()* function within an R function to define the fault handling mechanisms within the R session, but the fault policy of RPig allows the user to restart the R function in a brand new R session.

R functions may run exceedingly slowly on occasion and the user would expect a way to monitor the UDF execution time and terminate its execution if it runs too long. RPig offers the facility for monitoring long running R functions. For example, *func_name.monitoredUDF.duration←10* will terminate the named R function if it runs for more than 10 seconds, and return the default value of null.

4.4 Implementation

There are several libraries used for the RPig implementation. Renjin[10] is used for the JVM based R engine. Since the standalone R is implemented in C and Fortan, and Pig is written in Java, Rsession[11] is adopted as the Java interface of R to use the Pig APIs. Pig offers Java annotation based implementation for a monitored Java UDF. To build the same function for R UDFs, we need to create new Java class with annotations for each R functions at runtime. The Javassist[12] is used for defining a new class at runtime and to modify a class file when the JVM loads it.

5 Use case and experiment

In this section, we describe the usage of RPig with the examples we discussed in section 2. To provide valuable comparative experiment results, we also describe and experiment with one alternative framework or implementation for each use case. Although the use cases here are from the telecom domain, the design and the solution we describe are general and applicable to other domains.

Our experiments are conducted in Amazon EMR, where we have all nodes with the same configuration (m1.medium instance, Hadoop 1.0.3, R 2.14.1). One node from Master Instance Group has the extended Pig 0.11 with RPig feature deployed to generate MapReduce programs. The rest of the nodes are from Core Instance Group providing both data storage and MapReduce task execution services. As R requires data be loaded in memory, each node is configured to have a maximum capacity of one map task and one reduce task, so an R session could take the maximum memory available in a single node. We also assign a larger heap-size limit to the child JVMs of map tasks, as these are where R statistical functions are executed. The reduce task is allocated a lower value.

5.1 Summary statistics with quantiles

Before going to complex examples that use different R packages, we would like to show a simple quantiles statistic task to give a “hello world” example in the first case. Quantiles are used to summarize a set of observations by giving the boundary values between the divided distributions. For example, a large number of values for a network parameter observed over time can be summarized in a few numbers, or quantiles, for reporting or comparing with thresholds.

5.1.1 Design and implementation

DataFu[7] is a collection of useful Pig add-ons (UDFs) developed by LinkedIn for data mining and statistics and it is used to for the comparison study in this use case. DataFu is used in many off-line workflows for data derived products like “People You May Know” and “Skills” at LinkedIn. The following shows the main lines of implementation using DataFu[1].

```
DEFINE Quantile datafu.pig.stats.Quantile('0.5','0.75','1.0');
-- Computing the quantiles for each network nodes
Quantiles = FOREACH B { sorted = ORDER values BY val;  GENERATE id, Quantile(sorted);  };
```

[1] Detail explanation of the DataFu quantile example is available at: http://engineering.linkedin.com/open-source/introducing-datafu-open-source-collection-useful-apache-pig-udfs

DataFu uses the *DEFINE* statement to specify a *Quantile* UDF function with string parameters for the function constructor. *('0.5','0.75','1.0')* yields the 50th, 75th percentiles, and the max. The function takes a sorted bag as the input.

The following shows the RPig version for the same computational task.

```
REGISTER 'RFuncs.r' using rsession as RFuncs;                    -- or using 'renjin' for JVM R
%DECLARE q_probs '0.5, 0.75, 1'; %declare q_type '2';
Quantiles = FOREACH A GENERATE id, RFuncs.Quantile(values, '$q_probs', '$q_type');
```

In addition to the above Pig statements, the following R UDF is defined in the *RFuncs.r* script.

```
Quantile.outputSchema ← "q:double";
Quantile ← function (x, probs, type) {
    probs ← as.numeric(unlist(strsplit(probs, split=",")))              # parse the parameter value
    q ← quantile(unlist(x), probs= probs, names= T, type= type);  # call the R quantile() function
    return (as.list(q));  }
```

The *Quantile* UDF is a simple wrapper for the *quantile()* function of the R *stats* library. *x* is a numeric vector whose sample quantiles are desired. Its value is converted from the Pig input tuple by the framework. The function parameters *(probs, type)* value can be supplied in different ways, e.g., a Declare statement which is used in the example or a Parameter File, etc.

To summarize this use case of RPig, any original R function can be easily wrapped and exposed as a Pig R UDF. The necessary input parameter of the original R function can be exposed by the UDF to make the function more generic for reusability. Still, all the input data for a single function call will be executed in one R engine and some partitioning might be necessary (e.g., group by 'week') if the data is too large.

5.1.2 Result and discussion

In this use case of computing quantiles, both DataFu and RPig only require a few lines of Pig (and R) code, as the user does not need to write the quantile algorithm. However the RPig implementation of the function is much more flexible regarding the data input and output formats than DataFu. The DataFu quantile function only takes a sorted input bag and each numeric value is a tuple inside the bag. We have to pre-format the raw data before calling the function in this case. In contrast, the RPig version can handle any format of bag or tuple input. Numeric values can be either in one tuple/bag or separated tuple/bags, since the data always will be flattened into numeric vectors in the R function before computing quantiles.

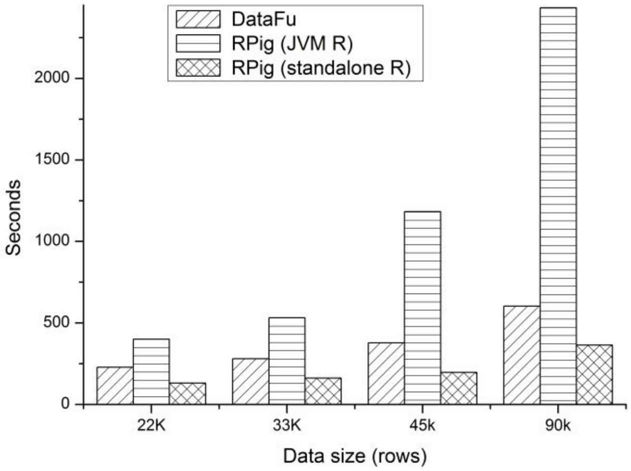

Figure 2: The performance comparison on DataFu and RPig

Figure 2 shows the performance comparison with a fixed 20 node Hadoop cluster. Each row of input data contains more than 10k double values for one network node and that makes around 1GB raw data for every 10k rows. The RPig version implementation with the JVM R engine (Renjin 0.7.0) has the slowest performance. It becomes very slow when input data size gets larger and it consumes almost all available memory for the map task. It might relate to the internal memory management problem of Renjin, since it is only in a very early stage. The RPig with standalone R has the best performance. DataFu (v 0.0.10) is in the second, since it needs to pre-format and sort the data through Pig operations and these take more time before calling the quantile function.

DataFu has some convenience bag (e.g. enumerating bags) and utility functions, but the availability of statistical functions in DataFu is extremely limited. It only includes common statistics tasks (e.g., quantile, variance), PageRank, etc. algorithms which are relevant to the Linkedin use cases. Even for the quantile function, the DataFu only implements the type R-2 estimation, which is one of several algorithms for estimating quantiles. RPig allows the use of nine quantile algorithms implemented in R, selected by the *type* parameter in the example above. With RPig it is easy to wrap and expose any statistical function of R as a Pig UDF. The statistical functions available in RPig are as many and as comprehensive as in the original R.

In summary, RPig provides very extensive statistical and machine learning algorithms by wrapping any original R function in a Pig UDF, and the UDF is flexible with input and output data formats and gives the best performance (with standalone R) in the above case. In contrast, DataFu is ready to use without needing additional installation of a script engine since it runs on the JVM, but the number of functions is extremely limited.

5.2 Forecasting with EMA

5.2.1 Design and implementation

EMA is used for forecasting data traffic on selected VoIP service clients for a use case described in Section 2. Since EMA is a light algorithm, and the aggregated data *Traffics* (from subsection 3.3) is already small enough in this case. We can group all the data together and send it to one R engine using RPig. The following shows the Pig statements:

```
Results = FOREACH (GROUP Traffics ALL) GENERATE RFuncs.ema_all($1, n);
```

ema_all() is a defined R function processing the grouped data, as in the following.

```
ema_all.outputSchema ← toString(lapply(seq(1,11), function(x) {paste("map[tuple(double)]", sep="")}))
ema_all ← function(x, n) {
    xDf ← as.data.frame(do.call(rbind, x[[1]]))       # convert to a data frame
    ...                                               # sorted the data and initial variables
```

```
        library('TTR')
        for(i in 1:length(clients)){
                t ← xDf[xDf[,c(3)]==clients[i], c(4)]
                results ← append(results,list(list(as.character(clients[i]), EMA(t,n)))) }
    return (results) }
```

In this case, the data passed to R is a nested list (x), which contains aggregated traffic data for all service clients in different time windows, $((x_1^1, x_2^1, x_3^1),(x_1^2, x_2^2, x_3^2),....)$. The first line of the R script converts the nested list to a data frame called xDf, so the input data can be easily sorted and selected as a data table. A sorted numeric list containing traffic data of previous time windows for each service client is selected, and is used as input for the R *EMA()* function of the *TTR* package. Results of all service clients as a nested list *results* will be subsequently converted to a Pig map data structure specified by the output schema. The name of the service client is the key of the map, and the forecasted result is the value of the map. In this case, the Pig statement is used as the query language for accessing the data from the HDFS file system, and then the converted data will be sent to R for analytic tasks. Afterwards, the data analytic result is printed on screen or stored in HDFS through Pig statements. Hence, RPig can be used as a way for R programmers to read, write data and files in HDFS.

To summarize this use case of RPig, the Pig operations are used as preprocessing steps to extract and summarize only the necessary information needed for R processing. When the summarized data is small enough to be handled in R in a single node, then we can use any statistical algorithm implementations of R, directly on the summarized data similar to the traditional single machine approach of R.

5.2.2 Result and discussion

The necessary data must be converted and loaded into R first when an R function is involved in a pig data flow, and we consider this as performance overhead. Minimizing this overhead was one of our main tasks after the initial version of RPig development. As a consequence, the initial version of RPig is used for a comparison study in this case. The code implementation for this use case based on the initial version can be found in[9] and it is very similar to the implementation using the current version.

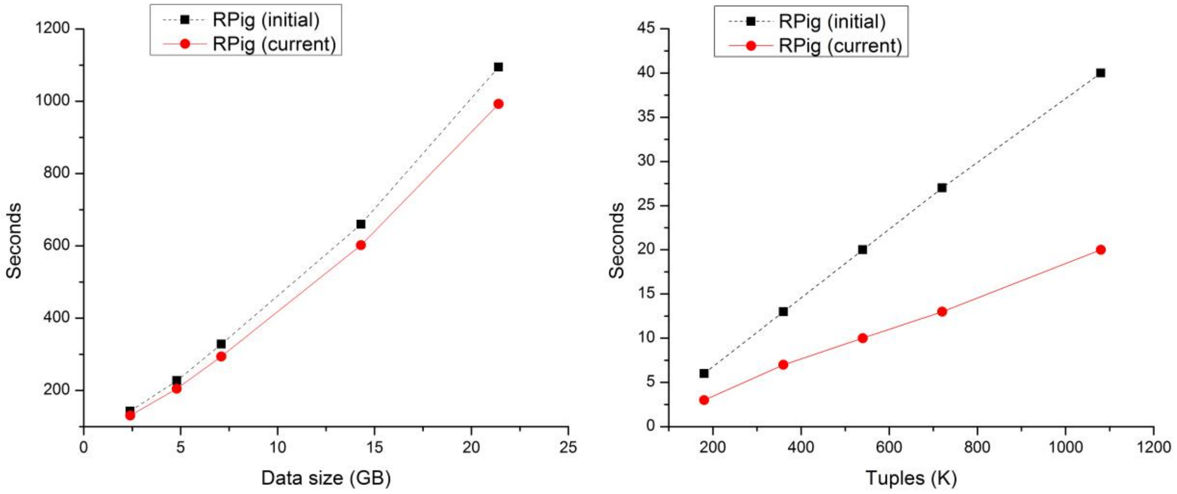

Figure 3 : 1) Overall performance comparison 2) Overhead comparison on data exchange (Pig to R)

Figure 3 (1) shows the results with a fixed 20 nodes. The data size represents the initial raw data size loaded in to Pig. With both versions of the implementations (both with standalone R engines), the performance is decreasing with the increasing data size as expected. In this scenario, the performance mainly depends on Pig/Hadoop, which needs to handle a large amount of raw data, where R only plays a small part in the overall process. We can see the current version has better overall performance and the improvement becomes larger when more data is involved. Figure 3 (2) shows the improvement in detail when sending data from Pig to R in a single node. In this case, summarized data with more than half million data tuples and 4 data fields in each tuple, will take 20 seconds in the initial version, but only takes 10 seconds in current version. Overhead is reduced 50% in the current version. This is achieved by sending data directly to R through the socket connection and many code optimizations in the current version. The initial version of RPig streams the data to the disk as an R source file, then makes

R load the source file. But still, when more data needs to be exchanged between R and Pig, then the overhead becomes larger. This overhead can be considered as a trade-off between user development effort and processing efficiency. We only have 10 lines of R code in the R functions in this use case, but we or the user had to write around 100 lines of Java code for the EMA Pig UDF without using R for same calculation. In another example, DataFu has around 200 lines of Java code for the Quantile function in the first use case. This shows the significant reduction of coding and code maintenance effort with RPig.

In summary, RPig offers concise programming for data analytics by utilizing existing implementations of algorithms in R. However, the necessary data required must be converted and loaded into R and this causes the performance overhead. As a result, data should always be minimized as much as possible before exchanging data between Pig and R to reduce overhead. The current RPig has a significant performance improvement over the initial implementation, it has reduced the overhead of data exchange by 50%.

5.3 Prediction with SVM

5.3.1 Design and implementation

Because an SVM model is constructed based on determined support vectors in SVM algorithms, an SVM training dataset can be represented by data samples as support vectors. The remaining data of the dataset that does not directly contribute to the final SVM model can be viewed as redundant, even though minor inaccuracies may occur in some cases[13] [14]. Therefore, if we have two map functions, one (map_{sv}) is for extracting samples marked as support vectors from a dataset, another (map_{svm_m}) is for having an SVM model from a dataset, and a generic reduce function (*reduce*) is only to aggregate a list of results from map functions, then the SVM training phase to get a model for a dataset D can be defined in MapReduce model as the following.

Training Phase:

```
repeat a number of time if required:
    split D to $\{D_1, D_2, ..., D_n\}$
    D ← in parallel execution: reduce $(map_{sv}(D_1), map_{sv}(D_2), ..., map_{sv}(D_n))$
Model ← reduce $(map_{svm\_m}(D))$
```

Since support vectors are often only a small data subset of the original input dataset $map_{sv}(D) < D$, and map and reduce functions are executed in parallel in Hadoop, building an SVM model from a data subset would be much faster than the original dataset. Hence, the overall SVM training is expected to be scalable with the size of the cluster. A parallel algorithm can also be structured as multiple rounds of map and reduce. Collected samples as support vectors can be treated as a new dataset, hence, the map_{sv} can also be applied repeatedly to further reduce the data size if it is required.

In the prediction phase, it takes the trained model and network KPIs at IP layer, such as packet loss, as input then gives a predicted MOS value instantly. In case, we want to do MOS value prediction in parallel for a large amount of VoIP call sessions *S*, then a map function $map_{predict}$can be defined to take a subset of call sessions to increase the scalability.

Training Phase:

```
split S to $\{S_1, S_2, ..., S_n\}$
results ← in parallel execution:
        $reduce((map_{predict}(Model, S_1), map_{predict}(Model, S_2), ..., map_{predict}(Model, S_2))$
```

As analyzed above, we have the following script for our scenario. We first describe the training phase implementation. The following code fragment shows the step of extracting support vectors from a split dataset *StatisticEvents_s* in MapReduce model.

```
SV = FOREACH StatisticEvents_s GENERATE FLATTEN(RFuncs.svm_sv($1));
```

```
svm_sv.outputSchema ← "bag{tuple(double, double)}";
svm_sv ← function(x) {
        xDf ← as.data.frame(do.call(rbind, x[[1]]))        # data frame of training dataset
        …                                                  # extracting the support vector sv
        return (list(sv))
}
```

The R function *svm_sv* is an implementation of the map map_{sv} function. Extracting support vectors is same as building an SVM model. It covers cross validation, parameter tuning, etc. for complete SVM training. However, instead of getting a final SVM model, we only fetch out the samples as support vectors after the SVM training. In our case using a radial kernel based SVM regression, SVM computation can be represented to solve the following optimization problem.

$$\begin{array}{ll} \min\limits_{\alpha,\alpha^*} & \frac{1}{2}(\alpha-\alpha^*)^\top y_i y_j exp(-\gamma\|x_i-x_j\|^2)(\alpha-\alpha^*)+ \\ & \varepsilon\sum\limits_{i=1}^{n}(\alpha_i-\alpha_i^*)+\sum\limits_{i=1}^{n}y_i(\alpha_i-\alpha_i^*) \\ \text{subject to} & 0\leqslant \alpha_i,\alpha_i^*\leqslant C,^{\sim} i=1,...,n, \\ & \sum\limits_{i=1}^{n}(\alpha_i-\alpha_i^*)=0^{\sim}. \end{array}$$

where, x_i, x_j are the input dataset, *gamma* is a parameter. In R, we use the *e1071* package to supply the above SVM implementation. We first use a tuning function to find the best parameter over a parameter range, and then we train a support vector machine. The following R code fragment shows the part of extracting *sv*.

```
        tuned ← tune.svm(V2 ~ ., data= xDf, gamma= 10^(-2:2), kernel= 'radial')   # turn the parameter
        svmModel <- svm(xDf[2], xDf[1], kernel= 'radial',  gamma=tuned$best.parameters$gamma,
cross=10)
        sv ← xDf[c(svmModel$index),]
        sv ← apply(sv, 1,  function(x){as.list(x)})          # put each row to a list
```

sv is extracted samples as support vectors from the input date fame *xDf*. However, the data frame of R is a column-oriented structure. All of the values of a column are grouped together, and then the values of the next column are in a second group, and so on. Data tables stored in Hadoop and Pig are the same as the commonly used CSV format, which is primarily row-oriented. If we want to use the Pig *SPLIT* operator to split the collected support vectors to repeat the *sv* extracting process again, we need to convert the collected data representation to be row-oriented. And hence, we use the *apply()* function put each row to a list. Finally, all lists will be put as tuples into a bag sent back to Pig. We flatten the bag in Pig and convert the data back to the 'table' format to continue processing.

In the last step of the training phase, we group the finalized datasets or support vectors *SV* and send them to one R engine to get a final SVM model, and then store the model for the prediction phase. The MapReduce model is still applied, but only one map and one reduce functions will be created at this stage.

```
Model = FOREACH (GROUP SV ALL) GENERATE R.svm_m(*);

svm_m.outputSchema ← "model:bytearray";
svm_m ← function(x) {
        ...                                   # get the svmModel
        return (serialize(svmModel, NULL))   }
```

The R UDF *svm_m* is almost the same as above *svm_sv*, but returns an SVM model *svmModel* this time. The serialized model will be saved as a bytearray or original R object in HDFS, so we can use the model directly in R for prediction later on. In the SVM prediction phase, the SVM model can be directly loaded into R from Pig in parallel execution for a huge number of VoIP call sessions.

RHadoop[3] is a popular open source project from Revolution Analytics that allows users to manage and analyze data with Hadoop in R. The *rmr2* is an R package from RHadoop; it offers the user to write MapReduce functions in R. We implemented above parallel SVM design with *rmr2* for the comparative study in this case. The following shows the MapReduce implementation to get support vectors *SV*. We use this function as the example to show the difference in implementation with regard to the RPig version. The *SV* has exactly the same value as the RPig implementation we described above.

```
svDfs ← mapreduce( input=inputPath,
       map = function(dummy, input) {
                  ...           # extracting the support vector sv
              keyval(1, list(sv))      },
      reduce = function(k, sv){
             val ← do.call("rbind", sv);   keyval(1, val)   }
)
SV ←from.dfs(svDfs)$val
```

The *map* function extracts the support vectors of every data subset and outputs a key-value pair, the value part is the support vectors, *sv*. All outputs of the map functions have the same key value, integer *1*, so the extracted *sv* of different data subsets will be collected and aggregated together by the *reduce* function. The final result *SV* can be retrieved for the value part of the key-value pair output of the reduce function.

To summarize this use case with RPig, parallel or iterative statistical algorithms for distributed data sources are expressed as parallel R executions in a Pig data flow. Input data is treated as a number of distributed data sources with no centralized information during parallel R executions for each data source, aggregated results of distributed R executions as stepping stones are relative to a final result of a final centralized R execution. Pig operations are used to distribute the data and tasks for parallel processing with multiple R engines as Map tasks. This approach allows parallel R executions to reduce the processing time. However, statistical errors may be caused by the iterative and incremental statistical algorithms as a trade-off and are acceptable in most cases[13] [14].

5.3.2 Result and discussion

Both RPig and RHadoop allow having a parallel SVM implementation in the MapReduce model. RPig just uses FOREACH statement to parallelize the tasks as the Map functions. RHadoop allows the user to code the entire analytic job in R, but the user has to design the key-value pairs based Map and Reduce functions. This creates complexity for the user in code design and development compared to RPig, especially when multiple MapReduce functions are necessary for complex analytic jobs. In the example described above for getting *SV*, we wrote 16 lines of Pig and R code by using RPig, but it needs 21 lines of R code for RHadoop because of the writing of key-value pair functions (Figure 4 (1)). This again shows the concise programing of RPig.

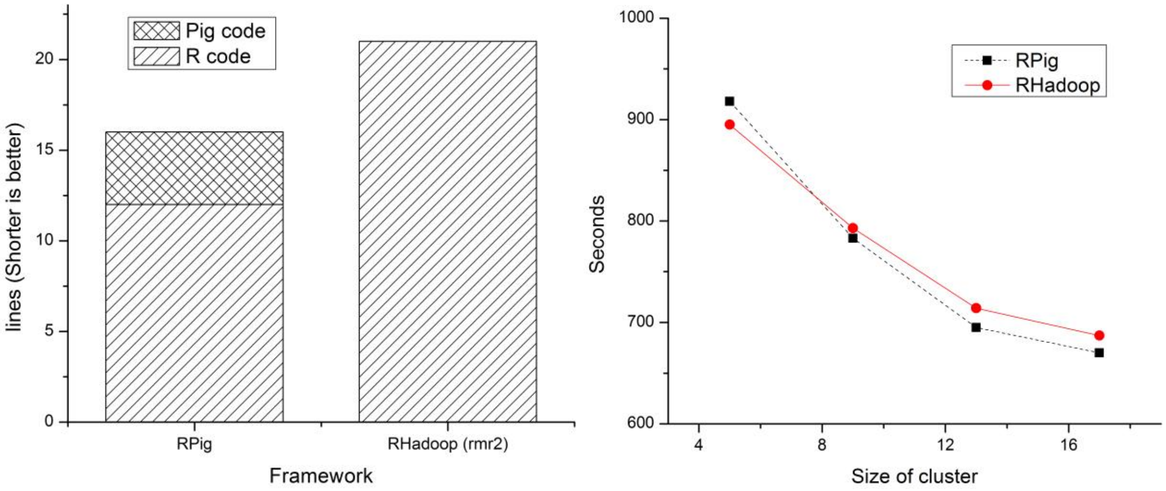

Figure 4: 1) line of code comparison 2) Scalability on SVM training

A relatively small size data is used in this case, since it is very CPU intensive as described earlier. We split the data containing 12K training samples into 16 pieces to get support vectors and then we obtain the final SVM model at the end of the training. The performance of the SVM training phase with respect to the cluster size is shown in Figure 4 (2). RPig has almost identical performance compared to the RHadoop (rmr2 v2.2.2) implementations. There is a reduction of processing time as the cluster size increases, but the decrease in processing time is not exactly linear, as there is a higher communication cost with a larger cluster.

In summary, RPig is able to scale out machine learning functionalities for deep analytics. We demonstrate this through an SVM use case. RPig is less complex to use and requires less development effort for writing parallel machine learning algorithms compared to RHadoop or others (e.g. RHIPE[15]), which require designing and writing key-value paired Map and Reduce functions manually.

6 Related work

6.1 Related to R

With the emergence of big data analytics, many researchers are addressing the scalability issues of R. The existing approaches can be classified in three different categories.

1. Scaling memory size: All data used in R calculations such as lists and data frames, need to be loaded into memory; however a single computer only has a very limited memory size that restricts large data set to be loaded into R. RevoScaleR[16] and bigmemory[17] are R packages that allow R to use hard disk as external memory for calculations. This approach allows R to handle a data size much bigger than its memory size, since the size of hard disk is generally much larger than memory size of a computer.

2. Scaling storage size: Terabyte level big data is generally stored in distributed file systems, such as Hadoop clusters. To enable R to directly read/write data in these large scale data warehouses, interfaces between these warehouses and R are developed, such as Ricardo[18] that offers a bridge between R and Hadoop HDFS. Comparing Ricardo to R bridging work on traditional RDBMS such as RJDBC[19] and RMySQL[20], SQL is replaced by a query language (Jaql) which can be executed in MapReduce model in Ricardo. These approaches allow R to directly access data from database or file systems, but the R script execution remains in a single computer. For parallel data analysis, it requires re-implementing most part of a statistic algorithm in the query language. In other words, we have to re-implement the *SVM()* function of R in Jaql for our second use case to parallelize the process.

3. Scaling CPU power: Approaches for scaling out CPU power for R can be divided into MapReduce and non MapReduce based implementations. MapReduce based approaches are generally running on top of Hadoop. For example, both RHIPE[15] and RHadoop[3] extend R to allow writing key-value pair map and reduce functions within an R script. The MapReduce jobs of R can be submitted to a Hadoop cluster for parallel executions. However, these frameworks require users to manually design complex key-value pair based map and reduce functions, making them very difficult to use and inefficient for analysis job development. In our case with RPig, the key-value pair based map and reduce functions are automatically generated by leveraging the Pig framework. The user only needs to define R functions for a single task node; the execution of the R functions is parallelized automatically based on Pig data flows. RHive[21] has the same concept as our work. It is an R extension facilitating distributed computing via HiveQL/SQL queries. However, it is restricted for Hive data warehouse. And considering the natural differences between Pig and SQL language, RHive is an alternative to RPig, but it cannot be a replacement.

Many approaches utilize non MapReduce based parallel frameworks, such as Open MPI[22] Packages such as Rmpi[23] and snow[24] provide bridge interfaces between R and MPI. CloudRmpi[25] supports to manage an EC2 cluster and access an R session on the master MPI node. Elastic-R[26] allows users to send data to any R engine in an R engine pool. However, above solutions do not support parallel data read/write as Hadoop, hence they are not suitable for IO intensive scenarios. Furthermore, these solutions are very difficult to use as the use must code send/receive message functions for master and slave nodes through complex MPI API.

6.2 Other related solutions

Some approaches try to build new systems without using traditional statistical frameworks, such as R. For example, Mahout[2] is a framework built on top of Hadoop with MapReduce based machine learning algorithms. However, Mahout is only at an early stage; many commonly used algorithms, such as SVM are not available yet. Secondly, it does not provide a high level language, such as R and Pig; instead complex Java APIs are provided. As a result, developing analytic jobs in Mahout is complex and difficult. SystemML[27] proposes a new declarative machine learning Language (DML) for machine learning on MapReduce. However, DML is not flexible as R language; it does not support object oriented features, advanced data types (such as, lists and arrays), etc. in comparison with R. More importantly, SystemML is same as other new developed frameworks such as MLbase[28] and Cloudera ML[29], also lacks on commonly used statistic and machine learning algorithm implementations.

7 Conclusion

R provides comprehensive machine learning and statistical algorithms. However the R execution environment is not distributed, and is not considered as scalable. In contrast, Pig supports parallel data processing using high level language; but it does not provide implementations of common statistical algorithms; it lacks the necessary features for advanced statistical analysis. In this paper, we have presented an integrated RPig framework that takes the advantage of both R and Pig allowing scalable deep analysis while minimizing the development effort with concise programming.

We have described the design, implementation of RPig framework. Based on the use case scenarios, we have demonstrated the usage of our framework. We have shown experimental results related to scalability and coding effort reduction with examples. We also did comparison study in each use case experiment to show the difference or improvement over related work. Our future work will create an R package which would allow calling Pig in R.

1. Dean, J. and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters.* Communications of the ACM, 2008. **51**(1): p. 107-113.
2. *Apache Mahout*. Available from: http://mahout.apache.org/.
3. *RHadoop*. Available from: https://github.com/RevolutionAnalytics/RHadoop/wiki/.
4. Handurukande, S., et al. *Magneto approach to QoS monitoring*. in *IFIP/IEEE International Symposium on Integrated Network Management*. 2011.
5. White, T., *Hadoop: The Definitive Guide*. 2 ed. 2011: O Reilly Media.
6. Eaton, C., et al., *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. 2012: McGraw-Hill.
7. *DataFu*. Available from: http://data.linkedin.com/opensource/datafu.
8. Olston, C., et al. *Pig Latin: A Not-So-Foreign Language for Data Processing*. in *ACM International conference on Management of Data*. 2008.
9. Wang, M., S.B. Handurukande, and M. Nassar. *RPig: A Scalable Framework for Machine Learning and Advanced Statistical Functionalities*. in *IEEE International Conference on Cloud Computing Technology and Science*. 2012. IEEE.
10. *Renjin*. Available from: http://www.renjin.org/.
11. *rsession*. Available from: http://code.google.com/p/rsession/.
12. Chiba, S. and M. Nishizawa. *An Easy-to-Use Toolkit for Efficient Java Bytecode Translators*. in *International Conference on Generative Programming and Component Engineering*. 2003.
13. Syed, b.N.A., et al. *Incremental Learning with Support Vector Machines*. in *International Knowledge Discovery and Data Mining conference*. 1999.
14. Graf, H.P., et al., *Parallel Support Vector Machines: The Cascade SVM.* Advances in neural information processing systems, 2004. **17**: p. 521-528.
15. Guha, S., *Computing environment for the statistical analysis of large and complex data*, in *Statistics*. 2010, Purdue University.
16. Rickert, J., *Big Data Analysis with Revolution R Enterprise (white paper).* Revolution Analytics, 2011.
17. *bigmemory*. Available from: http://cran.r-project.org/web/packages/bigmemory/index.html.
18. Das, S., et al. *Ricardo: Integrating R and Hadoop*. in *ACM International conference on Management of Data* 2010.

19. *RJDBC*. Available from: http://www.rforge.net/RJDBC/index.html.
20. *RMySQL*. Available from: http://cran.r-project.org/web/packages/RMySQL/.
21. *RHive*. Available from: https://github.com/nexr/RHive.
22. *Open MPI*. Available from: http://www.open-mpi.org/.
23. *Rmpi*. Available from: http://www.stats.uwo.ca/faculty/yu/Rmpi/.
24. *snow*. Available from: http://cran.r-project.org/web/packages/snow/index.html.
25. *cloudRmpi*. Available from: http://norbl.com/cloudrmpi/cloudRmpi.html.
26. Chine, K., *Open Science in the Cloud: Towards a Universal Platform for Scientific and Statistical Computing*, in *Handbook of Cloud Computing*, B. Furht and A. Escalante, Editors. 2010, Springer. p. 453-474.
27. Ghoting, A., et al. *SystemML: Declarative Machine Learning on MapReduce*. in *IEEE International Conference on Data Engineering*. 2011.
28. Kraska, T., et al., *MLbase: A Distributed Machine Learning System*, in *The Conference on Innovative Data Systems Research*. 2013.
29. *Cloudera ML*. Available from: https://github.com/cloudera/ml.