2012 IEEE Asia-Pacific Services Computing Conference

JRBridge: A Framework of Large-Scale Statistical Computing for R

Xia Xie, Jie Cao, Hai Jin, Xijiang Ke, Wenzhi Cao
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
email: hjin@hust.edu.cn

***Abstract*—Demands for highly scalable parallel data processing platforms is raising due to an explosion in the number of massive-scale data intensive applications both in industry and in sciences. Performing statistical computing over huge data repositories poses a significant challenge to existing statistical software and computational infrastructure. After analyzing various open source computational infrastructures and their programming paradigm APIs, the results have shown that most of them are JVM based, and their APIs are given as Java interfaces or abstract classes. This paper proposes a generic framework *JRBridge*, which can integrate *R* and JVM-based computational infrastructures by generating Java APIs code wrapper around the native *R* code automatically and handling type conversion. Using this framework, we build a distributed statistical computing environment by integrating *R* with Hadoop. With the Hadoop Distributed File System plugin, it brings a way to store and access datasets with millions of objects. With MapReduce plugin, it brings a natural environment to code MapReduce algorithms in *R*. The experiment result shows *JRBridge* scales linearly with the size of the datasets and thus provides a scalable solution for large-scale statistical computing in *R*.**

Keywords- R Language; JVM; Hadoop; MapReduce; Statistical Computing Method

I. INTRODUCTION

Data explosion springs up in many research fields such as astronomy, information retrieval, and social network. Knowledge buried in these enormous datasets is so invaluable that the ability to apply sophisticated statistical analysis methods to the data is becoming more and more essential. Performing statistical computing over huge data repositories poses a significant challenge to existing statistical software and data management systems.

Statistical software provides rich functionalities for data analysis, which only can handle limited amounts of data. For example, popular tools like *R* operate entirely in main memory with a single server. The system provides a comprehensive environment for statistical computing, including a concise statistical language, well tested libraries of statistical algorithms for data exploration and modeling, and also visualization facilities. We focus on the highly popular *R* statistical analysis program, which have overtaken other tools with closer to 43% data miners in 4th annual data miner survey [22-23]. *Comprehensive R Archive Network* (CRAN) contains a library with nearly 3000 add-in packages, covering areas such as linear and generalized linear models, statistical tests, time series analysis, classification, clustering. They simply fail when the data becomes too large. Some analysts try to avoid this shortcoming by using the most powerful machine available, but vertical scalability is inherently limited and expensive. Others try to work only on the subsets or samples of the data [5].

On the other hand, large-scale computational infrastructures can scale to data in petabyte scale, but provide only insufficient analytical functionalities. There are many infrastructures designed for different computational requirements, such as Hadoop and Dryad [14] for dataflow computing, Pregel [7] and Giraph [8] for graph computing, Haloop [26], Twister [10] and Spark [15] for iterative computing, Yahoo! S4 [13] and Twitter Storm [17] for stream computing. Only Hadoop and Storm support multi-language programming by Hadoop Streaming and Storm Multi-Language protocol. Most of them only open their own low-level APIs for users, and users cannot write code in *R* directly to support own their programming paradigm at the same time. Besides this, there are some high-level declarative languages such as Jaql [12], Pig [20] or Hive [1] for Hadoop, but they do not deliver the rich analytic functionality supported by other statistical software like *R*.

In a word, *R* provides rich functionalities for data analysis, but cannot support large datasets. On the contrary, large-scale computational infrastructures scale to large datasets, but have limited analytical functionalities and most of them cannot bring their computational capability to *R*.

We propose a framework *JRBridge* which can integrate *R* and JVM-based computational infrastructures. The idea is to generate Java APIs code wrapper around the native *R* code automatically, handle type interchange between *R* and Java transparently. Using this framework, we build a distributed statistical environment by integrating *R* with Hadoop. With the *Hadoop Distributed File System* (HDFS) plugin, it brings a way to store and access datasets with millions of objects in HDFS. With MapReduce [11] plugin, it brings a natural environment to code MapReduce algorithms in *R*.

II. RELATED WORKS

Prior works on large-scale data process in *R* are classified by scaling out *R*, SQL-based integration and Hadoop Streaming based integration.

978-0-7695-4897-5/12 $26.00 © 2012 IEEE
DOI 10.1109/APSCC.2012.74

A. *Scaling Out R*

Existing approaches to bring *R* into large-scale data process are mainly classified by low-level message passing and high-level automatic parallelization.

Both Rmpi [9] and rpvm [16] are message-passing type. They are simple wrappers for their respective parallel computing middleware. Usage of message-passing systems require significant expertise in parallel programming such as how to define explicit data distribution and inter-processor communication schemes. Most of all, these systems require a *tightly* connected cluster, where fault tolerance, redundancy and elastic are not provided. At the higher level, there have been some promising systems which can support automatic parallelization such as pR [18].

Here are two major innovations in the recent pR design. The first one is runtime analysis/parallelization. They perform dynamic dependency analysis before interpreting *R* statements; identify tasks and loops that can be parallelized. The second one is that they perform incremental analysis which delays the processing of conditional branches and dynamic loop bounds until the related variables are evaluated. Key feature of pR is that it analyses a sequential *R* source script dynamically and transparently, and parallelizes its execution accordingly. The results of partial execution are collected to perform further analysis in runtime. The framework does not require any modifications to the native *R* environment. But this works is only for non-interactive batch jobs and sequential processing tasks where the dataset fits in main memory on a single machine.

B. *SQL-based Integration*

There are many SQL-based data warehouse systems designed to simplify tasks with writing analysis flows on Hadoop, such as Hive, Pig, Jaql. These systems mostly provide a mechanism to project structure schema on data and query the data using a SQL-like language. RHive [19] is the bridge between *R* and Hive, and Ricardo [24] is the bridge between *R* and Jaql. Ricardo is also suboptimal to beat a custom-written Hadoop job in Java, for the inner data serialization and SQL parsing phrases taking additional time.

Ricardo proves that many deep analytical problems can be decomposed into a "small-data part" and a "large-data part". It consists of three components: small-data part executed in *R*, large-data part, and R-Jaql bridge. The large-data part is executed in the Hadoop/Jaql *Data Management System* (DMS). R-Jaql bridge provides communication and data conversion facilities for integrating these two different platforms.

R serves as an interactive environment for the data analyst. For classical *R* usage, the data itself is not memory-resident in R. Instead, the base data is stored in a distributed file system with Hadoop cluster, which consists of a set of worker nodes that both store data and perform computations. Jaql is an open source data flow language, and it provides a high-level language to process and query Hadoop data, compiling into a set of low-level MapReduce jobs with *JSON* views to solve different data type between *R* and Hadoop. The bridge with a *R* package provides integration module between Jaql and *R*.

Moreover, Ricardo does not support writing MapReduce algorithms in *R* directly. If Ricardo packages cannot meet the demands of particular analysis or analysts want to write their own special algorithms, they have to re-implement the lower-level Java MapReduce code and then bind them to Jaql as user defined functions, which means an analyst who uses Ricardo needs to be an expert in Jaql and Java MapReduce programming at the same time.

C. *Hadoop Streaming based Integration*

Hadoop provides an API that allows user to write map and reduce functions in languages except Java. Hadoop Streaming uses UNIX standard streams as the interface between Hadoop and user's executable program, so user can choose any languages which can read standard input and write to standard output for MapReduce program.

With Hadoop Streaming, *R* user only can focus on the input and output data interchange between *R* and Java. Since the type conversion gap exists between *R* and Java, serialization technologies must be used in these multiple-language systems. Both RHIPE [25] and RHadoop [23] are based on Hadoop Streaming. Users write the map and reduce function in native *R* code, which executes on the original *R* environment installed on each node. RHIPE uses Google's Protobuf library and RHadoop uses RJSONIO [6] library to solve the problem of data type transformation between Java and *R*.

So Hadoop Streaming can give a way for writing MapReduce algorithms in *R* directly, which means it is more smart and convenient than SQL-based integration for users to design their own algorithms. Storm Multi-Language protocol is another streaming-like integration method.

Till now, it shows that only few infrastructures are integrated with *R*, such as MPI, Hadoop, and Storm. These methods are too special to use by other infrastructures such as graph computing or streaming computing. Our motivation is to design a generic framework for integrating *R* with various infrastructures to make *R* benefited from decades of experience on large-scale data process.

III. DESIGN PHILOSOPHY

A. *Offline Batch Processing*

Hadoop composes with HDFS bundled with an implementation of Google's MapReduce paradigm and offers a scalable infrastructure for processing massive amounts of data.

HDFS takes care of the files distribution and replication across the nodes in Hadoop cluster. HDFS is written in Java, the Java class *DistributedFileSystem* is an implementation of the abstract class *FileSystem*, which is the way end-user code interacts with. An application that wants to store/fetch data to/from HDFS should construct a *DistributedFileSystem* Java object with Hadoop configuration, and then call the member function (API functions) of this object to access the HDFS.

We name this kind of API with the following features as *Static Function API* (SF API):

- The core functions of this kind of API are static and fully implemented by the backend infrastructures. There

is no need for developer to write any new embedded codes to achieve those functions.
- Most of SF API can be looked as a *Remote Procedure Call* (RPC) with arguments and a return value. Developers only need a runtime environment as a client to send the RPC request with specified arguments; and then the reply will be received and processed by the backend. At the end of RPC, the backend sends a response back to the client.

MapReduce programming model and API are briefly presented as follows:
- The computation takes a set of input key/value pairs <K_1, V_1>.
- Mapper takes an input pair <K_1, V_1>, and produces a set of intermediate key/value pairs <K_2, V_2> by calling the map method written by user.
- Reducer accepts an intermediate key/value pairs <K_2, V_2>, merges together these values to form a possible smaller set of values, and produces a set of output key/value pairs <K_3, V_3> by calling the reduce method written by user.
- If a combination function is used, it is the same form for the reduction function (and is an implementation of reducer in Java), except its output types are the intermediate key and value types <K_2, V_2>, so they can feed the reduce function.

By default, the user must write MapReduce program in Java by extending the *Mapper* and *Reducer* class with re-implemented map and reduce methods.

We name this kind of API with the following features as *User Define Function API* (UDF API):
- The most important difference between SF API and UDF API is that the functions cannot be fully implemented by the backend alone, which is only a framework. Developers should write code embedded into the framework to implement the user define function. Developers should implements the map and reduce method for their special UDF API.
- There must be a runtime environment on each node since the user code must be distributed to each work node of the cluster. So a type conversion mechanism between *R* and Java is also needed here.

B. *Bulk Synchronous Parallel Model Graph Computing*

Bulk Synchronous Parallel (BSP) model is used in many graph computing infrastructures such as Google Pregel [7] and Apache Giraph [8]. User should switch to the "think like a vertex" model of programming. It can be simply defined as the combination of three attributes:
- A number of components just like the vertex in a graph, each vertex performing processing, keeping graph status, or sending message to other components in graph.
- A router that delivers messages point to point between pairs of components.
- Facilities of synchronizing all or subset of the components at special condition. A computation consists of a sequence of super-steps. In each super-step, every component is allocated to a task consisting of several combinations of local computation steps, message transmissions and message arrivals from other components. The machine proceeds to next super-step until a global check is made to ensure the last super-step has been completed by all components.

Take the open source Giraph as an example. At first, the master gets the appropriate number of splits for the application and writes them to Zookeeper [21]. Then the worker will read vertices from one or more splits, and executes the *compute* method for every vertex assigned to this worker per super-step and buffer the incoming messages to every vertex for the next super-step.

C. *Stream Computing*

There are many programming model used in stream computing such as Pub-Sub model for IBM SPC [2], Actors model for Yahoo! S4, Spout\Bolt model for Twitter Storm. The S4 and Storm are open source infrastructures.

S4 is written in Java. Developers write PEs in the Java programming language. PEs are assembles into applications using the Spring Framework. The processing element API is fairly simple and flexible UDF API. Developers essentially implement two primary handlers: an input event handler *processEvent()* and an output mechanism *output()*. In addition, developers can define some state variables for the PE. *processEvent()* is invoked for each incoming event of the types subscribed by PE. This method implements the logic for input event handing, typically an update of the internal PE state. *output()* method is an optional method that can be configured for invoking in variety of ways. It can be invoked either at regular time intervals *t*, or on receiving *n* input events, in the case where *n*=1. *output()* method implements the output mechanism of the PE, typically to release internal state of the PE to some external systems.

D. *SF API and UDF API Integration Pattern*

With above analyses on how to integrate *R* with popular open source infrastructures, we deduce the common pattern as follows:
- Most open source computational infrastructures are JVM-based, and the default APIs are written in Java.
- Most APIs are simple and flexible, and can be classified by SF API and UDF API.

In order to integrate *R* with these open source computational infrastructures with above common patterns, we need to design four key components: an engine to interpret *R* from Java, R2J (convert *R* objects to Java objects), J2R (convert Java objects to *R* objects), Java class loader and executor in *R* environment.

IV. PROTOTYPE SYSTEM IMPLEMENTATION

The module diagram (Fig.1) of system consists of three layers: plugin layer, common framework layer, and computation infrastructures layer. When users write source codes with the familiar syntax in *R* statistical programming language, they can also use the SF API and UDF API provided by the plugin layer, with which *R* code can be transformed by common framework layer to run on the variety of open source computational infrastructures in the computation infrastructure layer. So users can do large-scale statistical computing in *R* by integrating *R* with computation infrastructures.

The four components build a bridge between *R* and JVM-based open source computational infrastructures by cooperation with the SF API and UDF API integration model.

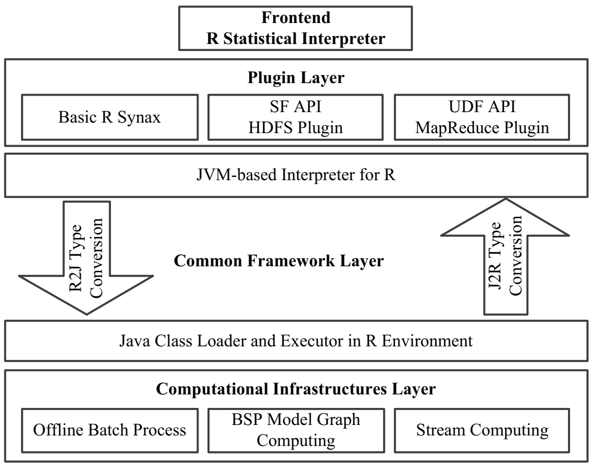

Figure 1. System Module Diagram

A. *JVM-based R Interpreter*

Most of open source computational infrastructures are JVM-based and the default APIs are written in Java, so it is hard to integrate the original *R* written in C and FORTRAN with them. In order to make it easy for integrating, we participate in the open source project Renjin [3], which is a JVM-based *R* interpreter. Renjin's core borrows heavily from the original C code. Indeed, many portions remain literal translations from C to Java. This interpreter can be used in our system at two places. First, the interpreter acts as a frontend of the statistical computing environment so that users can directly write source code with the familiar syntax in *R*. Second, in the UDF API model, the interpreter needs to be invoked by Java code to interpretive execute the embedded *R* code.

Since the implementation of the interpreter is too complex to be illustrated clearly and the most jobs belong to the open source community, we just present the basic interpretive execution flow. Fig.2 is the interpretive execution flow chart.

The frontend of the interpreter consists of two parts: lexer and parser. The lexer is ported directly from the original *R* written in C and the parser is built from the original *gram.y* file using the Java extension of the Bison [4] parser generator. The backend of the interpreter is an evaluator with type system and other libraries supported. In Fig.2, when users write *R* code in the interpreter, the lexer first scans the lines in the *R* code and transforms them into token stream; and then the parser takes the token stream as input and builds them as an *Abstract Syntax Tree* (AST); at last, the evaluator simply descends the AST recursively evaluating symbols, function calls, and promises as they are encountered. All control flow statements are implemented as functions and use exceptions to handle things like break or return.

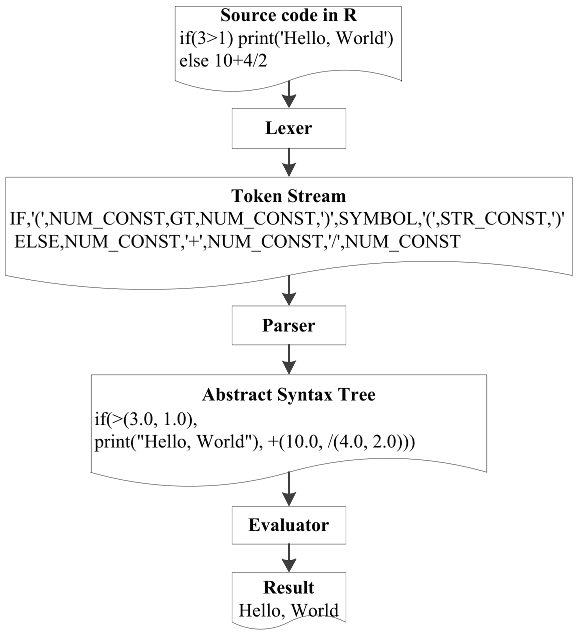
Source code in R
if(3>1) print('Hello, World')
else 10+4/2
Lexer
Token Stream
IF,'(',NUM_CONST,GT,NUM_CONST,')',SYMBOL,'(',STR_CONST,')'
ELSE,NUM_CONST,'+',NUM_CONST,'/',NUM_CONST
Parser
Abstract Syntax Tree
if(>(3.0, 1.0),
print("Hello, World"), +(10.0, /(4.0, 2.0)))
Evaluator
Result
Hello, World

Figure 2. Interpretive Execution Flow

B. *Java Class Loader and Executor in R*

In order to call Java method in *R* environment, Java class should be loaded first. In our system, we use general *Parents Delegation Model* (PDM) to load Java class, which involves 3 kinds of class loaders: bootstrap, extension, and application class loader. The delegation model requires that any request for a class loader to load a given class is first delegated to its parent class loader before the requested class loader tries to load the class itself. If a class loader's parent can load a given class, it returns that class. Otherwise, the class loader attempts to load the class itself. We use the application class loader as default to load the Java class, and we provide *jload*/*import R* methods to load Java class, and *$* operator to make the executor to access methods and fields of Java object.

- *jload*. Given the path of Java library as an argument of *jload*, users can add the path into the search class paths of application class loader. When you want to load the class next time, the application class loader can find it directly.
- *import*. After the path is added by *jload* methods, users can use *import* method to load the class by specifying the full class name. Then users can use the short name of the class in *R* environment.
- *$* operator. We regard Java *HashMap* object as an example in Fig.3. When *import* methods are invoked, an environment *R* object is created at the same time. The information of the fields, methods and constructors of the class is analyzed by reflective mechanism and is transformed into *symbol-value* pairs. Environment is consisted of two things: a frame and a pointer to an

enclosing environment. The frame consists of a set of *symbol-value* pairs, an enclosure (parent environment). When *R* searches the value for a symbol that the frame is examined, its value is returned if a matching symbol is found. If not, the enclosing environment is then accessed and the process repeated. Environments form a tree structure in which the enclosures play the role of parents.

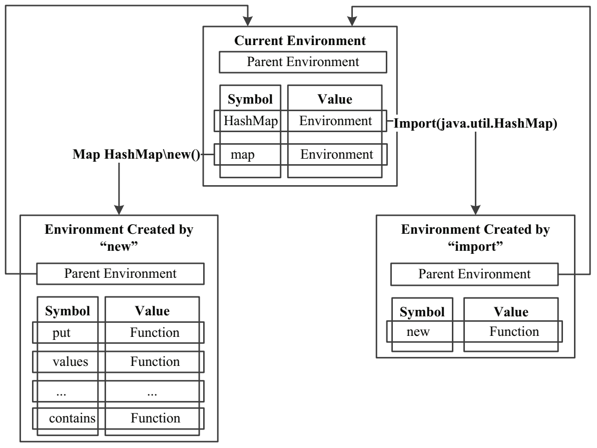

Figure 3. Environment Structure

We provide an *$* operator to access these symbols, e.g., users can use *HashMap$new* to access the construct of *HashMap* and get an instance of *HashMap* named *map*. Then the *map* also exists as an environment object with *HashMap symbol-value pairs*. So users can use *map$put* (1, *foo*) to call *HashMap* methods or fields.

C. R2J and J2R Type Conversion

Automatically type conversion occurs only in the presence of context switching, e.g., when calling Java method from *R* environment or interpretive executing the embedded *R* code from Java. Both the parameters and return values are needed type conversions.

R2J and J2R type conversion only can automatically convert the basic atomic type between *R* and Java. For other types, the Java special object except for the above basic type is transformed into an environment *R* object as mentioned before, and users operate *$* to this Java special object.

R has six basic types: *logical*, *integer*, *real*, *complex*, *character*, and *raw*. The data type in Java can be partitioned into two categories: integer types (*Boolean*, *Char*, *Byte*, *Short*, *Int*, and *Long*) and floating point types (*Float* and *Double*). The corresponding Java classes are *Boolean*, *String*, *Short*, *Integer*, *Long*, *Float*, and *Double*, they can also be taken as basic data types.

For our initial work, as the *complex* is not the basic type in Java, we use the external library. Table I shows the type conversion rules.

R is a dynamic typing language and Java is a static typing language, so J2R is easy but R2J needs some dynamic type conversion in *R* itself.

TABLE I. TYPE CONVERSION RULES

Java Basic Type	R Basic Type
"Boolean", "boolean"	logical
"Short", "short" , "Integer", "int"	integer
"Double", "double", "Float", "float", "Long", "long"	real
"String", "char[]"	character
"Byte" ,"byte"	raw

D. Hadoop Plugin

Plugin layer is at the top of common framework layer. There are two kinds of plugins: SF plugin and UDF plugin. Fig.4 shows the structure of them and usually the plugins are written in *R* and Java language by hybrid programming. The part written in *R* gives user simple *R* API which is used in *R* statistical computing. In SF plugin, such as HDFS or Giraph plugin, the part written in Java can be some capsulation of the complex original API that the computational infrastructure offered. In UDF plugin, the code generator may generate the framework code with user defined *R* code embedded, and byte code generator may compiler these generated Java code into Java jar file which can run on computational infrastructure.

HDFS plugin and MapReduce plugin are used to report our experience on integrating *R* with Hadoop.

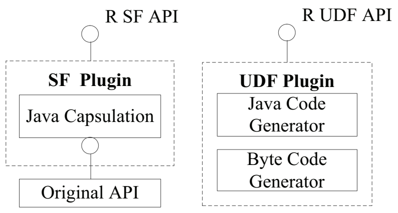

Figure 4. SF Plugin and UDF Plugin Structure

- *HDFS Plugin*

Fig.5 shows the HDFS plugin structure. The *R* API part gives user simple *R* API which can be directly used to access the HDFS clusters; the Java capsulation part shields the complex details of HDFS API and reduce the difficulty of calling Java with the *jload\import\$ operator*.

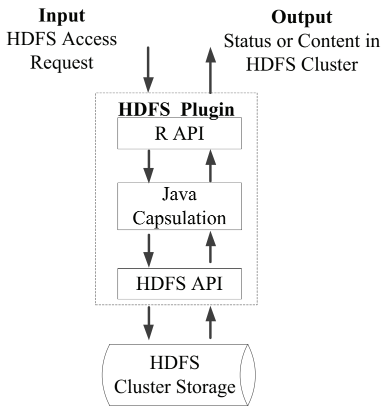

Figure 5. HDFS Plugin Structure

HDFS plugin provides basic connectivity to the HDFS. *R* programmers can browse, read, write, and modify files stored in HDFS.

- *MapReduce Plugin*

Integrating MapReduce with *R* needs the plugin to generate code with embedded *R* method such as *map* or reduce function. Fig.6 shows the structure of MapReduce plugin. With providing Hadoop or MapReduce API, the code of each MapReduce job consists of the following four parts: 1) code for submitting, this part needs to deal with Job submission parameters (including the input and output path, etc.), and specify the configuration of *Mapper* class and *Reducer* class; 2) *mapper* code written in *R*, in which the user defined *map* method; 3) reducer code, in which the user defines *reduce* method written in *R*; 4) combiner code, usually in order to reduce communication overhead. Combined phase can reduce the key/value pair produced by *map* phase in advance.

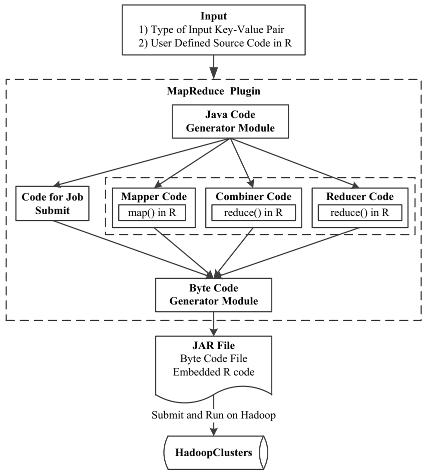

Figure 6. MapReduce Plugin Structure

V. Experiment and Analysis

We have implemented the Hadoop HDFS plugin and MapReduce plugin in *JRBridge* framework..

A. Test Setup

Counting word occurrences in a large document collection is a typical example used to illustrate the MapReduce technique. The data set is split into smaller segments and the *map* function is executed on each data segments. The *map* function produces a <*key*, *value*> pair for every word that it encounters. Here, the *same word* is the key and the value is *1*. The framework groups all the pairs, which have the *same key* (*same word*) and invokes the reduce function passing the list of values for a given key. The reduce function adds up all the values and produces a count of values for the particular key, which in this case is the number of occurrences of a particular word in the document set.

Hadoop cluster consists of one master node and five work nodes. Each node is assigned to 4GB memory, and two processors of 2.40GHz with Java Development Kit 6.0 and Apache Hadoop 0.20.203 installed on Fedora 15 operating system. Compared with the original *R*, we deploy *R* 2.14.0 on the master too. Among these 6 nodes, Hadoop Job Tracker and Name Node are deployed on the master node with our *JRBridge* system.

In the experiment, we only need to code in the *JRBridge* system on the master node and watch the Hadoop monitor webpage to get the performance statistics. We use the HDFS plugin to produce 5, 10, 50, 100, 200, 400, and 800 million words into HDFS clusters to test the function of access HDFS from *R* environment. In order to test the function of writing MapReduce in *R* with MapReduce plugin, we write *wordcount* code in *R* to process the datasets produced by HDFS plugin. Table II shows the dataset size in performance experiment when doing *wordcount* statistical computing from small data size to large data size.

TABLE II. Dataset Size in Performance Experiment

Number of Words in Millions	Data Size in MB
5	52.27
10	104.53
50	522.61
100	1045.21
200	2090.35
400	4180.78
800	8361.43

B. Performance and Scalability

Fig.7 shows the result of the performance comparison among the original *R* code, generated jar with embedded *R* by Hadoop plugin on our *JRBridge* system, and the original *wordcount* program written in Java.

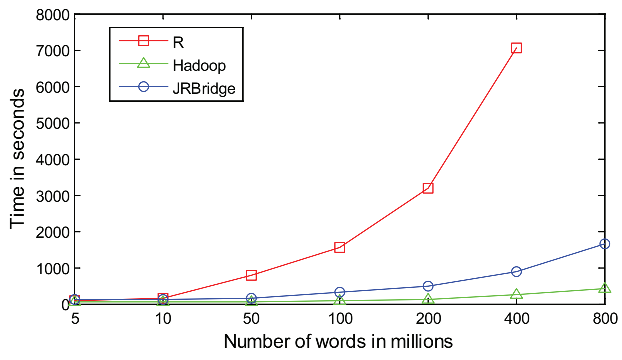

Figure 7. Performance Comparison

As the original *R* can only operate the dataset entirely in main memory, the original *R* code firstly use *scan* method to read the whole dataset into memory and then use *new.env* with *hash=TRUE* option to create a hash table to store the word and its count. The time original *R* consumed involves two parts: one for reading the data into memory, and another for statistic in memory. As the result shows, when size of dataset is reaching 4GB (the memory size of master node), and the time consumed is increased rapidly. When the size of dataset is 8GB, the original *R* nearly crashed. There is a missing data in Figure 7 when the number of words reaches 800 million.

By writing *R* code with the MapReduce plugin API, our *JRBridge* system will generate the jar file with embedded R code to run on Hadoop clusters. As the Hadoop can do distributed parallel computing automatically on 5 work nodes, the time of *JRBridge* is nearly reduced to 1/7 of *R*. As we can see in Figure 7, both *JRBridge* and original Hadoop scale linearly with the size of the dataset and thus provide a scalable solution.

Performance of our *JRBridge* is still not very good enough. The time of *JRBridge* is nearly twice than that of original *R*. The reasons for this are as follows: 1) all input and output key/value pairs need to be transformed between *R* object and Java object; 2) the JVM-based interpreter must be initialized at each setup for the map or reduce tasks, and the embedded *R* code must be interpretive executed which is not so fast as pure Java.

With increasing maturity, both the relative difference between systems as well as the total execution time can decrease further. Indeed, our initial work on *JRBridge* has already led to multiple improvements on large-scale statistical computing in *R*.

VI. Conclusion and Future Work

In this paper, we present *JRBridge*, a framework for large-scale statistical computing in *R*. *JRBridge* provides a feature-rich and scalable framework for integrating *R* and JVM-based open source computational infrastructures. With detailed analysis on how to integrate *R* with popular open source infrastructures, we propose SF and UDF integration models. We build this framework with 4 common components to follow these two integration models, and we also do some initial works on integrating *R* and Hadoop by implementing the HDFS plugin and MapReduce plugin. The experiment results show that the integrating *R* and Hadoop with *JRBridge* scale linearly with the size of the dataset and thus provides a scalable solution on large-scale statistical computing in *R*.

But the experiment results also show that the performance of *JRBridge* is still suboptimal. We expect significant performance improvements in the future as this technology matures. Since we only focus on integrating *R* and Hadoop, it is evident that our framework is potentially applicable to other JVM-based computational infrastructures. Our ultimate vision is to use *JRBridge* to integrate *R* with various infrastructures to make *R* benefited from decades of experience on large-scale data process. For ongoing work, we will improve the performance of *JRBridge*, such as reduce the time of type conversion and interpretive execution, and then improve our integration model when designing the plugin for other infrastructures such as Giraph, S4 or other coming infrastructures.

Acknowledgment

This work is supported by National Natural Science Foundation under grant No.61003007 and Wuhan Chenguang Program under grant No.201271031369.

References

[1] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. "Hive: a Warehousing Solution Over a Map-Reduce Framework," *Proc. VLDB Endowment 2009*, Lyon, France, 2009, Vol.2, pp.1626-1629.

[2] A. Lisa, A. Henrique, B. Ranjita, E. Frank, K. Richard, S. Philipe, P. Yoonho, and V. Chitra. "SPC: a Distributed, Scalable Platform for Data Mining," *Proc. 4th International Workshop on Data Mining Standards, Services and Platforms (DMSSP'06)*, Philadelphia, PA, 2006, pp. 27-37.

[3] A. Bertram (2010) Renjin Project on Google Code website [Online]. Available: http://code.google.com/p/renjin/.

[4] A. Demaille, J. E. Denny, and P. Eggert. (2011) Bison: GNU Parser Generator. [Online]. Available: http://www.gnu.org/software/bison/.

[5] C. T. Chu, S. K. Kim, and Y. A. Lin. "MapReduce for Machine Learning on Multicore", *Proc. Neural Information Processing Systems Conference (NIPS'06)*, 2006, pp. 306-313.

[6] D. T. Lang. (2012) Package RJSONIO [Online]. Available: ftp://h64-50-233-100.mdsnwi.tisp.static.tds.net/pub/cran/web/packages/RJSONIO/RJSONIO.pdf.

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. "Pregel: A System for Large-Scale Graph Processing," *Proc. the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*, Indianapolis, Indiana, 2010, pp.135-146.

[8] L. L. Rocca, J. H. Badsberg, and C. Dethlefsen. "The giRaph package for graph representation in R," *Proc. 2nd International R User Conference*, 2006, Vienna, Austria, p.102

[9] H. Yu. (2010) Rmpi: Interface to MPI [Online]. Available: http://cran.r-project.org/web/packages/Rmpi/index.html

[10] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. H. Bae, J. Qiu, and G. Fox. "Twister: A Runtime for Iterative MapReduce," *Proc. 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*, Chicago, 2010, pp. 810-818.

[11] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters," *Proc. OSDI'04*, San Francisco, CA, 2004, pp. 137-150.

[12] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C. C. Kanne, F. Ozcan, and E. J. Shekita. "Jaql: a Scripting Language for Large Scale Semistructured Data Analysis Data Analysis," *Proc. VLDB'11*, Seattle, Washington, 2011, pp.1272-1283.

[13] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. "S4: Distributed Stream Computing Platform," *Proc. IEEE International Conference on Data Mining Workshops (ICDMW'10)*, Sydney, NSW, 2010, pp.170 - 177.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys'07)*, New York, 2007, pp.59-72.

[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster Computing with Working Sets," *Proc. 2nd USENIX conference on Hot Topics in Cloud Computing (HotCloud'10)*, Boston, 2010, pp.10-16.

[16] M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, H. Tierney, and U. Mansmann, "State of theArt in Parallel Computing with R," *Journal of Statistical Software*, Vol. 31, No. 1, August 2009, pp.219-221.

[17] N. Marz (2011) Storm Project on Github website [Online]. Available: https://github.com/nathanmarz/storm/.

[18] N. Samatova. "pR: Introduction to Parallel R for Statistical Computing," *Proc. of CScADS Scientific Data and Analytics for Petascale Computing Workshop 2009*, pp. 505-509.

[19] NexR. (2011) RHive: R and Hive. [Online]. Available: http://cran.r-project.org/web/packages/RHive/index.html

[20] O. Chistopher, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. “Pig Latin: A Not-So-Foreign Language for Data Processing”, *Proc. 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*, New York, 2008, pp. 1099-1110.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. “Zookeeper: wait-free coordination for internet-scale systems,” *Proc. USENIX ATC’10*, Boston, MA, 2010.

[22] Rexer Analytics (2010) The Rexer Analytics website. [Online]. Available: http://www.rexeranalytics.com/Data-Miner-Survey-Results-2010.html.

[23] K. Rexer. The 4th Annual Data Miner Survey-2010 Survey Summary Report, Presented at Predictive Analytics World (PAW'10), 2010

[24] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. “Ricardo: Integrating R and Hadoop,” *Proc. 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD’10)*, Indianapolis, Indiana, 2010, pp. 987-998.

[25] S. Guha (2011) RHIPE - R and Hadoop Integrated Processing Environment. [Online]. Available: http://ml.stat.purdue.edu/rhipe/.

[26] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, “HaLoop: Efficient Iterative Data Processing on Large Clusters,” *Proc. VLDB Endowment 2010*, Vol.3, pp.285-296.