SparkR: Scaling R Programs with Spark

Shivaram Venkataraman[1], Zongheng Yang[1], Davies Liu[2], Eric Liang[2], Hossein Falaki[2]
Xiangrui Meng[2], Reynold Xin[2], Ali Ghodsi[2], Michael Franklin[1], Ion Stoica[1,2], Matei Zaharia[2,3]
[1]AMPLab UC Berkeley, [2] Databricks Inc., [3] MIT CSAIL

ABSTRACT

R is a popular statistical programming language with a number of extensions that support data processing and machine learning tasks. However, interactive data analysis in R is usually limited as the R runtime is single threaded and can only process data sets that fit in a single machine's memory. We present SparkR, an R package that provides a frontend to Apache Spark and uses Spark's distributed computation engine to enable large scale data analysis from the R shell. We describe the main design goals of SparkR, discuss how the high-level DataFrame API enables scalable computation and present some of the key details of our implementation.

Keywords

Spark; R; Statistical Computing;

1. INTRODUCTION

Recent trends in big data analytics indicate the growing need for interactive analysis of large datasets. In response to this trend, a number of academic [12, 32, 8] and commercial systems [18] have been developed to support such use cases. However, data science surveys [1] show that in addition to relational query processing, data scientists often use tools like R to perform more advanced analysis on data. R is particularly popular as it provides support for structured data processing using data frames and includes a number of packages for statistical analysis and visualization.

However, data analysis using R is limited by the amount of memory available on a single machine and further as R is single threaded it is often impractical to use R on large datasets. Prior research has addressed some of these limitations through better I/O support [35], integration with Hadoop [13, 19] and by designing distributed R runtimes [28] that can be integrated with DBMS engines [25].

In this paper, we look at how we can scale R programs while making it easy to use and deploy across a number of workloads. We present SparkR: an R frontend for Apache Spark, a widely deployed [2] cluster computing engine. There are a number of benefits to designing an R frontend that is tightly integrated with Spark.
Library Support: The Spark project contains libraries for running SQL queries [10], distributed machine learning [23], graph analytics [16] and SparkR can reuse well-tested, distributed implementations for these domains.
Data Sources: Further, Spark SQL's data sources API provides support for reading input from a variety of systems including HDFS, HBase, Cassandra and a number of formats like JSON, Parquet, etc. Integrating with the data source API enables R users to directly process data sets from any of these data sources.
Performance Improvements: As opposed to a new distributed engine, SparkR can inherit all of the optimizations made to the Spark computation engine in terms of task scheduling, code generation, memory management [3], etc.

SparkR is built as an R package and requires no changes to R. The central component of SparkR is a distributed data frame that enables structured data processing with a syntax familiar to R users [31](Figure 1). To improve performance over large datasets, SparkR performs lazy evaluation on data frame operations and uses Spark's relational query optimizer [10] to optimize execution.

SparkR was initially developed at the AMPLab, UC Berkeley and has been a part of the Apache Spark project for the past eight months. SparkR is an active project with over 40 contributors and growing adoption [6, 7]. We next outline the design goals of SparkR and key details of our implementation. Following that we outline some of the efforts in progress to improve SparkR.

2. BACKGROUND

In this section we first provide a brief overview of Spark and R, the two main systems that are used to develop SparkR. We then discuss common application patterns used by R programmers for large scale data processing.

2.1 Apache Spark

Apache Spark [2] is a general purpose engine for large scale data processing. The Spark project first introduced Resilient Distributed Datasets (RDD) [34], an API for fault tolerant computation in a cluster computing environment. More recently a number of higher level APIs have been developed in Spark. These include MLlib [23], a library for large scale machine learning, GraphX [16], a library for processing large graphs and SparkSQL [10] a SQL API for analytical queries. Since the above libraries are closely integrated with the core API, Spark enables complex workflows where say SQL queries can be used to pre-process data and the results can then be analyzed using advanced machine learning algorithms. SparkSQL also includes Catalyst [10], a distributed query optimizer that improves performance by generating the optimal physical plan for a given query. More recent efforts [9] have looked at developing a high level distributed DataFrame API for structured data processing. As queries on DataFrames are executed using the SparkSQL query optimizer, DataFrames provide both better usabil-

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2903740

ity and performance compared to using RDDs [4]. We next discuss some of the important characteristics of data frames in the context of the R programming language.

2.2 R Project for Statistical Computing

The R project [26] consists of a programming language, an interactive development environment and a set of statistical computing libraries. R is an interpreted language and provides support for common constructs such as conditional execution (`if`) and loops (`for`, `while`, `repeat`) etc. R also includes extensive support for numerical computing, with data types for vectors, matrices, arrays and libraries for performing numerical operations.

Data frames in R: In addition to numerical computing, R provides support for structured data processing through data frames. Data frames are tabular data structures where each column consists of elements of a particular type (e.g., numerical or categorical). Data frames provide an easy syntax for filtering, summarizing data and packages like `dplyr` [31] have further simplified expressing complex data manipulation tasks on data frames. Specifically, `dplyr` provides a small number of *verbs* for data manipulation and these include relational operations like selection, projection, aggregations and joins. Given its popularity among users, the concept of data frames has been adopted by other languages like Pandas [21] for Python etc. Next, we look at some of the common workflows of data scientists who use R as their primary programming language and motivate our design for SparkR based on these workflows.

2.3 Application Patterns

Big Data, Small Learning: In this pattern, users typically start with a large dataset that is stored as a JSON or CSV file. Data analysis begins by joining the required datasets and users then perform data cleaning operations to remove invalid rows or columns. Following this users typically aggregate or sample their data and this step reduces the size of the dataset. The pre-processed data is then used for building models or performing other statistical tasks.

Partition Aggregate: Partition aggregate workflows are useful for a number of statistical applications like ensemble learning, parameter tuning or bootstrap aggregation. In these cases the user typically has a particular function that needs to be executed in parallel across different partitions of the input dataset and the results from each partition are then combined using a aggregation function. Additionally in some cases the input data could be small, but the same data is evaluated with a large number of parameter values.

Large Scale Machine Learning: Finally for some applications users run machine learning algorithms on large datasets. In such scenarios, the data is typically pre-processed to generate features and then the training features, labels are given as input to a machine learning algorithm to fit a model. The model being fit is usually much smaller in size compared to the input data and the model is then used to serve predictions.

We next present SparkR DataFrames and discuss how they can be used to address the above use cases.

3. DESIGN

In this section we present some of the design choices involved in building SparkR. We first present details about the DataFrames API and then present an overview of SparkR's architecture.

3.1 SparkR DataFrames API

The central component of SparkR is a distributed data frame implemented on top of Spark. SparkR DataFrames have an API similar to dplyr or local R data frames, but scale to large datasets using Spark's execution engine and relational query optimizer [10].

```
1  # Load the flights CSV file using `read.df`
2  df <- read.df(sqlContext, "./nycflights13.csv",
3                "com.databricks.spark.csv")
4
5  # Select flights from JFK.
6  jfk_flights <- filter(df, df$origin == "JFK")
7
8  # Group and aggregate flights to each destination.
9  dest_flights <- agg(
10     groupBy(jfk_flights, jfk_flights$dest),
11     count = n(jfk_flights$dest))
12
13 # Running SQL Queries
14 registerTempTable(df, "table")
15 training <- sql(sqlContext,
16   "SELECT distance, depDelay, arrDelay FROM table")
```

Figure 1: Example of the SparkR DataFrame API

```
dest_flights <- filter(df, df$origin == "JFK") %>%
    groupBy(df$dest) %>%
    summarize(count = n(df$dest))
```

Figure 2: Chaining DataFrame operators in SparkR

DataFrame Operators: SparkR's DataFrame supports a number of methods to read input and perform structured data analysis. As shown in Figure 1, SparkR's `read.df` method integrates with Spark's data source API and this enables users to load data from systems like HBase, Cassandra etc. Having loaded the data, users are then able to use a familiar syntax for performing relational operations like selections, projections, aggregations and joins (lines 6–11). Further, SparkR supports more than 100 pre-defined functions on DataFrames including string manipulation methods, statistical functions and date-time operations. Users can also execute SQL queries directly on SparkR DataFrames using the `sql` command (lines 15–16). SparkR also makes it easy for users to chain commands using existing R libraries [11] as shown in Figure 2. Finally, SparkR DataFrames can be converted to a local R data frame using the `collect` operator and this is useful for the big data, small learning scenarios described earlier.

Optimizations: One of the main advantages of the high-level DataFrame API is that we can tightly integrate the R API with the optimized SQL execution engine in Spark. This means that even though users write their code in R, we do not incur overheads of running interpreted R code and can instead achieve the same performance as using Scala or SQL. For example, Figure 4 compares the performance of running group-by aggregation on 10 million integer pairs on a single machine using Spark with R, Python and Scala. From the figure we can see that SparkR's performance is similar to that of Scala / Python and this shows the benefits of separating the logical specification in R from the physical execution.

3.2 Architecture

SparkR's architecture consists of two main components as shown in Figure 3: an R to JVM binding on the driver that allows R programs to submit jobs to a Spark cluster and support for running R on the Spark executors. We discuss both these components below.

3.2.1 Bridging R and JVM

One of the key challenges in implementing SparkR is having support for invoking Spark functions on a JVM from R. The main requirements we need to satisfy here include (a) a flexible approach where the JVM driver process could be launched independently by say a cluster manager like YARN (b) cross-platform support on

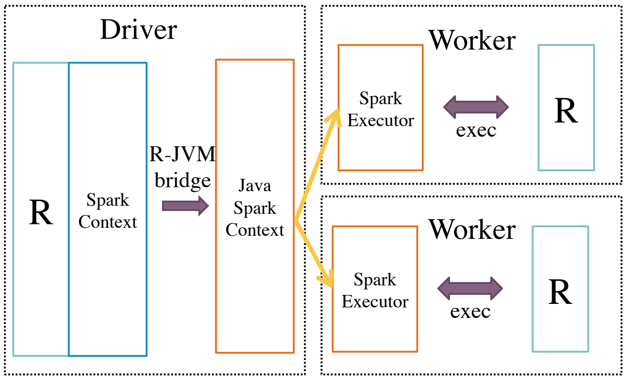

Figure 3: SparkR Architecture

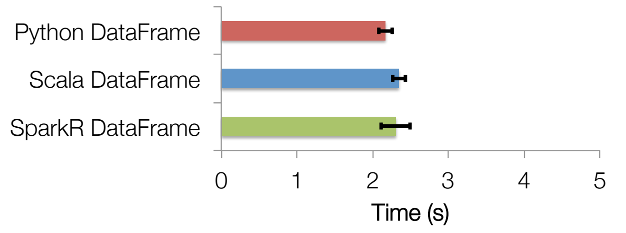

Figure 4: SparkR Performance Comparison with Python, Scala APIs

Windows, Linux, etc. (c) a lightweight solution that does not make it cumbersome to install SparkR. While there are some existing packages which support starting an in-process JVM [27] we found that these methods do not meet all our requirements.

Thus we developed a new socket-based SparkR internal API that can be used to invoke functions on the JVM from R. Our high level design is inspired by existing RPC or RMI-based systems [29] and we introduce a new SparkR JVM backend that listens on a Netty-based socket server. Our main reason for using sockets is that they are supported across platforms (in both Java and R) and are available without using any external libraries in both languages. As most of the messages being passed are control messages, the cost of using sockets as compared other in-process communication methods is not very high.

There are two kinds of RPCs we support in the SparkR JVM backend: method invocation and creating new objects. Method invocations are called using a reference to an existing Java object (or class name for static methods) and a list of arguments to be passed on to the method. The arguments are serialized using our custom wire format which is then deserialized on the JVM side. We then use Java reflection to invoke the appropriate method. In order to create objects, we use a special method name `init` and then similarly invoke the appropriate constructor based on the provided arguments. Finally, we use a new R class 'jobj' that refers to a Java object existing in the backend. These references are tracked on the Java side and are automatically garbage collected when they go out of scope on the R side.

3.2.2 *Spawning R workers*

The second part of SparkR's design consists of support to launch R processes on Spark executor machines. Our initial approach here was to fork an R process each time we need to run an R function. This is expensive because there are fixed overheads in launching the R process and in transferring the necessary inputs such as the Spark broadcast variables, input data, etc. We made two optimizations which reduce this overhead significantly. First, we implemented support for coalescing R operations which lets us combine a number of R functions that need to be run. This is similar to operator pipelining used in database execution engines. Second, we added support for having a daemon R process that lives throughout the lifetime of a Spark job and manages the worker R processes using the `mcfork` feature in `parallel` package [26]. These optimizations both reduce the fixed overhead and the number of times we invoke an R process and help lower the end-to-end latency.

```
# Query 1
# Top-5 destinations for flights departing from JFK.
jfk_flights <- filter(flights, flights$Origin == "JFK")
head(agg(group_by(jfk_flights, jfk_flights$Dest),
         count = n(jfk_flights$Dest)), 5L)

# Query 2
# Calculate the average delay across all flights.
collect(summarize(flights,
                  avg\_delay = mean(flights$DepDelay)))

# Query 3
# Count the number of distinct flight numbers.
count(distinct(select(flights, flights$TailNum)))
```

Figure 7: Queries used for evaluation with the flights dataset

4. EVALUATION

In this section we evaluate some of our design choices described in the previous sections and also study how SparkR scales as we use more machines. The dataset we use in this section is the airline on-time performance dataset[1] that is used to evaluate existing R packages like dplyr [30]. This dataset contains arrival data for flights in USA and includes information such as departure and arrival delays, origin and destination airports etc. We use data across six years (2009-2014) and overall our input has 37.27M rows and 110 columns. The queries we use to evaluate SparkR are listed in Figure 7. The queries make use of filtering, aggregation and sorting and are representative of interactive queries used by R users. We use a cluster of 32 `r3.xlarge` machines on Amazon EC2 for our evaluation. Each machine consists of 2 physical cores, 30GB of memory and 80GB of SSD storage. All experiments were also run using Apache Spark 1.6.0 and we used the `spark-csv`[2] package for reading our input.

4.1 Strong Scaling

We first study the scaling behavior of SparkR by executing the three queries in Figure 7 and varying the number of cores used. In this experiment, the input data is directly processed from HDFS and not cached in memory. The time taken for each query as we vary the number of cores from 8 to 64 is shown in Figure 5. From the figure we can see that SparkR achieves near-linear scaling with the time taken reducing from around 115 seconds with 8 cores to around 20 seconds with 64 cores. However waiting for 20 seconds is often sub-optimal for interactive queries and we next see how caching data in memory can improve performance.

4.2 Importance of Caching

For studying the benefits of caching the input table in memory we fix the number of cores used as 64 and measure the time taken

[1]http://www.transtats.bts.gov/Tables.asp?DB_ID=120
[2]http://github.com/databricks/spark-csv

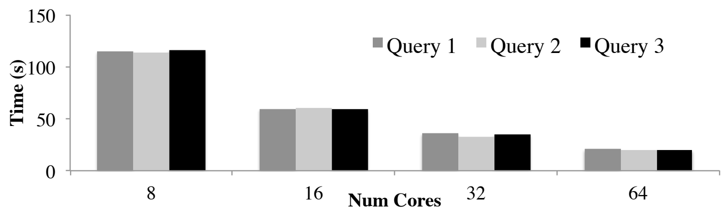

Figure 5: Query performance as we scale the number of cores used for three queries from Figure 7

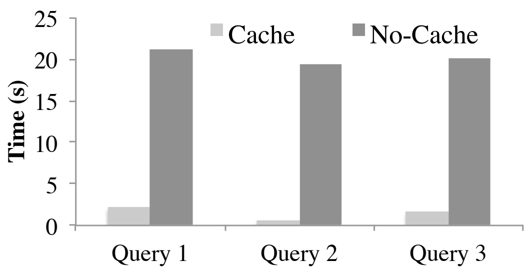

Figure 6: Effect of caching input data

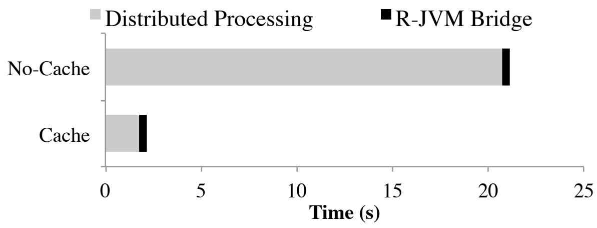

Figure 8: Breakdown of time taken R-to-JVM bridge and distributed processing Spark for Query 1 from Figure 7

```
# Train a GLM model
model <- glm(arrDelay ~ depDelay + distance,
             family = "gaussian", data = training)

# Print model summary
summary(model)

# Compute predictions using model
preds <- predict(model, training)
```

Figure 9: Building Generalized Linear Models in SparkR

by each query when the input data is cached. Results from this experiment are shown in Figure 6. We see that caching the data can improve performance by $10x$ to $30x$ for this workload. These results are in line with previous studies [34, 10] that measured the importance of caching in Spark. We would like to note that the benefits here come not only from using faster storage media, but also from avoiding CPU time in decompressing data and parsing CSV files. Finally, we can see that caching helps us achieve low latencies (less than 3 seconds) that make SparkR suitable for interactive query processing from the R shell.

4.3 Overhead of R-JVM binding

We next evaluate the overhead of using our socket-based R to JVM bridge discussed in Section 3.2.1. To do this we use query 1 from Figure 7 and run the query with both caching enabled and disabled on 64 cores. Using the performance metrics exported by Spark, we compute the time taken to run distributed computation and the time spent in the R-JVM bridge. In Figure 8, we see that the R-JVM bridge adds a constant overhead around 300 milliseconds irrespective of whether the data is cached or not. This overhead includes the time spent in serializing the query and in deserializing the results after it has been computed. For interactive query processing we find having an overhead of a few hundred milliseconds does not affect user experience. However, as the amount of data shipped between R and JVM increases we find that the overheads become more significant and we are exploring better serialization techniques in the R-JVM bridge to improve this.

5. ONGOING WORK

We are continuing work on SparkR in many areas to improve performance and enable more use cases. The two main areas we discuss here relate to large scale machine learning by integration with MLlib [23] and supporting partition aggregate workflows using user-defined functions (UDFs).

5.1 Large Scale Machine Learning

R includes support for a number of machine learning algorithms through the default `stats` package and other optional packages like `glmnet` [14], `cluster` [20] etc. The machine learning algorithms typically operate directly on data frames and use C or Fortran linkages for efficient implementations. One of the most widely used machine learning functions in R is the `glm` method that fits Generalized Linear Models. The `glm` method in R lets users specify the modeling of a response variable in a compact symbolic form using *formulas*. For example, the formula $y \sim a + b$ indicates the response y is modeled linearly by variables a and b. `glm` also lets users specify the loss function to use and can thus be to used to implement linear regression, logistic regression etc. The `glm` method returns a model trained using the input data and users typically use the `summary` function in R to print a number of statistics computed about the model.

To support large scale distributed machine learning in SparkR, we are working on integrating Spark's MLlib API with SparkR DataFrames. Our first focus is `glm` and to provide an intuitive interface for R users, we extend R's native methods for fitting and evaluating models as shown in Figure 9. We support a subset of the R formula operators in SparkR. These include the $+$ (inclusion), $-$ (exclusion), : (interactions) and intercept operators. SparkR implements the interpretation of R model formulas as an MLlib [23] feature transformer and this integrates with the ML Pipelines API [22]. This design also enables the same RFormula transformer to be used from Python, Scala and thereby enables an R-like succinct syntax for GLMs across different Spark APIs.

We are also working on implementing support for model summaries in SparkR to compute (a) minimum and maximum deviance residuals of the estimation (b) the coefficient values for the estimation (c) the estimated standard errors, t-values and p-values. Currently we have implemented these metrics for Gaussian GLMs trained using weighted least squares and we are working towards extending support for such metrics across different different families (Poisson, Gamma etc.) and link functions (logit, probit etc.) using iteratively re-weighted least squares (IRWLS).

5.2 User Defined Functions

To support the partition aggregate usage pattern discussed before, we are working on providing support for running user-defined functions (UDFs) in parallel. Spark supports UDFs written in Scala, Python and these APIs allow UDFs to run on each row of the input DataFrame. However, a number of R packages operate on local R data frames and it would be more user-friendly to support UDFs where R users can directly make use of these packages. In SparkR we plan to support UDFs that operate on each partition of the distributed DataFrame and these functions will in turn return local R columnar data frames that will be then converted into the corresponding format in the JVM.

In addition to the above UDF-based API, we find that for some use cases like parameter tuning, the input dataset is small but there are a number of parameter values that need to be evaluated in parallel. To support such workflows we are working on a parallel execution API, where we take in a local list, a function to be executed and run the function for each element of the local list in one core in the cluster. Finally one of the main features that we aim to support with UDFs is closure capture or support for users to refer to external global variables inside their UDFs. We plan to implement this using R's support for reflection and one of the challenges here is to ensure that we only capture the necessary variables to avoid performance overheads.

5.3 Efficient Data Sharing

One of the main overheads when executing UDFs in SparkR is the time spent serializing input for the UDF from the JVM and then deserialzing it in R. This process is also repeated for the data output from the UDF and thus adds significant overhead to the execution time. Recent memory management improvements [3] have introduced support for off heap storage in Spark and we plan to investigate techniques to use off heap storage for sharing data efficiently between the JVM and R. One of the key challenges here is to develop a storage format that can be parsed easily in both languages. In addition to the serialization benefits, off heap data sharing can help us lower the memory overhead by reducing the number of data copies required.

6. RELATED WORK

A number of academic (Ricardo [13], RHIPE [17], RABID [19]) and commercial (RHadoop [5], BigR [33]) projects have looked at integrating R with Apache Hadoop. SparkR follows a similar approach but inherits the functionality [23] and performance [3] benefits of using Spark as the execution engine. The high level DataFrame API in SparkR is inspired by data frames in R [26], dplyr [31] and pandas [21]. Further, SparkR's data sources integration is similar to pluggable backends supported by dplyr. Unlike other data frame implementations, SparkR uses lazy evaluation and Spark's relational optimizer to improve performance for distributed computations. Finally, a number of projects like DistributedR [25], SciDB [24], SystemML [15] have looked at scaling array or matrix-based computations in R. In SparkR, we propose a high-level DataFrame API for structured data processing and integrate this with a distributed machine learning library to provide support for advanced analytics.

7. CONCLUSION

In summary, SparkR provides an R frontend to Apache Spark and allows users to run large scale data analysis using Spark's distributed computation engine. SparkR has been a part of the Apache Spark project since the 1.4.0 release and all of the functionality described in this work is open source. SparkR can be downloaded from `http://spark.apache.org`.

Acknowledgments: We would like to thank the anonymous reviewers and our shepherd Tyson Condie for their feedback. We would also like to thank Sun Rui, Yu Ishkawa, Chris Freeman, Dan Putler, Felix Cheung, Hao Lin, Antonio Piccolboni, Yanbo Liang, and all other contributors to the open source SparkR project. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, Cisco, Cray, Cloudera, EMC, Ericsson, Facebook, Guavus, HP, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Schlumberger, Splunk, Virdata and VMware.

8. REFERENCES

[1] 2015 data science salary survey. https://www.oreilly.com/ideas/2015-data-science-salary-survey.

[2] Apache Spark Project. http://spark.apache.org.

[3] Project Tungsten: Bringing Spark Closer to Bare Metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[4] Recent performance improvements in Apache Spark: SQL, Python, DataFrames, and More. https://goo.gl/RQS3ld.

[5] Rhadoop. http://projects.revolutionanalytics.com/rhadoop.

[6] Spark survey 2015. http://go.databricks.com/2015-spark-survey.

[7] Visual Analytics for Apache Spark and SparkR. http://goo.gl/zPje2i.

[8] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The Stratosphere platform for big data analytics. *VLDB Journal*, 23(6):939–964, 2014.

[9] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia. Scaling spark in the real world: performance and usability. *Proceedings of the VLDB Endowment*, 8(12):1840–1843, 2015.

[10] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, et al. Spark SQL: Relational data processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[11] S. M. Bache and H. Wickham. *magrittr: A Forward-Pipe Operator for R*, 2014. R package version 1.5.

[12] M. Barnett, B. Chandramouli, R. DeLine, S. Drucker, D. Fisher, J. Goldstein, P. Morrison, and J. Platt. Stat!: An interactive analytics environment for big data. In *SIGMOD 2013*, pages 1013–1016.

[13] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: integrating R and Hadoop. In *SIGMOD 2010*, pages 987–998. ACM, 2010.

[14] J. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22, 2010.

[15] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, et al. SystemML: Declarative machine learning on MapReduce. In *ICDE*, pages 231–242. IEEE, 2011.

[16] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI 2014*, pages 599–613.

[17] S. Guha, R. Hafen, J. Rounds, J. Xia, J. Li, B. Xi, and W. S. Cleveland. Large complex data: Divide and Recombine (d&r) with RHIPE. *Stat*, 1(1):53–67, 2012.

[18] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht,

M. Jacobs, et al. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR 2015*.

[19] H. Lin, S. Yang, and S. Midkiff. RABID: A General Distributed R Processing Framework Targeting Large Data-Set Problems. In *IEEE Big Data 2013*, pages 423–424, June 2013.

[20] M. Maechler, P. Rousseeuw, A. Struyf, M. Hubert, and K. Hornik. *cluster: Cluster Analysis Basics and Extensions*, 2015.

[21] W. McKinney. Data Structures for Statistical Computing in Python . In S. van der Walt and J. Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[22] X. Meng, J. Bradley, E. Sparks, and S. Venkataraman. ML Pipelines: A New High-Level API for MLlib. https://goo.gl/pluhq0, 2015.

[23] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, et al. MLlib: Machine Learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[24] Paradigm4 and B. W. Lewis. *scidb: An R Interface to SciDB*, 2015. R package version 1.2-0.

[25] S. Prasad, A. Fard, V. Gupta, J. Martinez, J. LeFevre, V. Xu, M. Hsu, and I. Roy. Large-scale predictive analytics in vertica: Fast data transfer, distributed model creation, and in-database prediction. In *SIGMOD 2015*.

[26] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[27] S. Urbanek. *rJava: Low-Level R to Java Interface*, 2015. R package version 0.9-7.

[28] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices. In *Eurosys 2013*, pages 197–210.

[29] J. Waldo. Remote Procedure Calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, 1998.

[30] H. Wickham. *nycflights13: Data about flights departing NYC in 2013.*, 2014. R package version 0.1.

[31] H. Wickham and R. Francois. *dplyr: A Grammar of Data Manipulation*, 2015. R package version 0.4.3.

[32] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD 2013*.

[33] L. Yejas, D. Oscar, W. Zhuang, and A. Pannu. Big R: Large-Scale Analytics on Hadoop Using R. In *IEEE Big Data 2014*, pages 570–577.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI*, 2012.

[35] Y. Zhang, W. Zhang, and J. Yang. I/O-efficient statistical computing with RIOT. In *ICDE 2010*, pages 1157–1160.