dmapply: A functional primitive to express distributed machine learning algorithms in R

Edward Ma Vishrut Gupta Meichun Hsu Indrajit Roy[+]
HPE Vertica, [+]Hewlett Packard Labs
{ema,vishrut.gupta,meichun.hsu,indrajitr}@hpe.com

ABSTRACT

Due to R's popularity as a data-mining tool, many distributed systems expose an R-based API to users who need to build a distributed application in R. As a result, data scientists have to learn to use different interfaces such as RHadoop, SparkR, Revolution R's ScaleR, and HPE's Distributed R. Unfortunately, these interfaces are custom, non-standard, and difficult to learn. Not surprisingly, R applications written in one framework do not work in another, and each backend infrastructure has spent redundant effort in implementing distributed machine learning algorithms.

Working with the members of R-core, we have created **ddR** (Distributed Data structures in R), a unified system that works across different distributed frameworks. In **ddR**, we introduce a novel programming primitive called `dmapply` that executes functions on distributed data structures. The `dmapply` primitive encapsulates different computation patterns: from function and data broadcast to pair-wise communication. We show that `dmapply` is powerful enough to express algorithms that fit the statistical query model, which includes many popular machine learning algorithms, as well as applications written in MapReduce. We have integrated **ddR** with many backends, such as R's single-node `parallel` framework, multi-node `SNOW` framework, Spark, and HPE Distributed R, with few or no modifications to any of these systems. We have also implemented multiple machine learning algorithms which are not only portable across different distributed systems, but also have performance comparable to the "native" implementations on the backends. We believe that **ddR** will standardize distributed computing in R, just like the SQL interface has standardized how relational data is manipulated.

1. INTRODUCTION

R is one of the top choices for statisticians and data scientists [26]. While R is commonly used as a desktop tool, data science teams in most enterprises install RHadoop, SparkR, Distributed R, and other R-based interfaces to run analytics on the corresponding backend distributed system, such as Hadoop MapReduce and Spark [4, 5, 30, 34]. By using R, data scientists can continue to program in the familiar R language, even on massive data. Unfortunately, each backend exposes a custom R API, which leads to non-portable applications. Many interfaces also expose low-level details specific to a particular backend, making them difficult to learn. For example, SparkR, which is an R interface for the Apache Spark engine, exposes dozens of Spark's functions that can be non-intuitive to data scientists, and a program written in SparkR's API will not run on other frameworks.

The core reason for this disarray is the lack of any standardized way to perform distributed computing in R. While SQL semantics have unified how relational data is manipulated, there has been little effort in providing a similar standard for advanced analytics. Even though the R community has contributed more than 6000 algorithms packages, there are hardly any parallel or distributed versions of these algorithms. A simple, standard interface for distributed computing in R can potentially kickstart contributions on R-based parallel machine learning applications without the fear of vendor lock-in.

There are two key challenges in providing a standardized system for advanced analytics. First, the system should implement an interface that is not only *easy to use* by data scientists but also *flexible* enough to express many advanced analytics tasks. Second, generic interfaces typically have high overheads due to their flexible nature. One needs to ensure that applications expressed in this system, such as machine learning algorithms, have *good performance*.

In this paper we describe **ddR** (Distributed Data structures in R), a system that unifies distributed computing in R. We created **ddR** in collaboration with members of R-core, those who maintain and release the R language. **ddR** defines the following three distributed versions of R data structures, providing an intuitive way to partition and store massive datasets: `dlist`, `darray`, and `dframe`, which are the distributed equivalents of the fundamental R containers: list, matrix, and data frame, respectively. These data structures store different types and shapes of data such as structured, unstructured, sparse and dense data.

To manipulate these distributed data structures, **ddR** introduces `dmapply`, a single programming primitive with functional semantics. We chose functional semantics because it ensures there are no side-effects, thus making it easy to parallelize a computation. The `dmapply` operator is multivariate, which means programmers can use it to operate

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

on multiple data structures at the same time, e.g., by using `dmapply(A,B,C..)`. Much of the power of `dmapply` comes from the fact that it can express different kinds of data computation and communication patterns that are common in analytics tasks. For example, programmers can easily express embarrassingly parallel computations, where a function operates on each partition of data and no data movement is required. Programmers can also express cases where certain data structures, or parts of them, are broadcast to all computations. In fact, users can even express patterns in which computation on a worker node receives data from any subset of workers, and thus implement different kinds of aggregations. In section 4.2 we confirm the expressive power of `dmapply` by showing that it can implement algorithms that fit the statistical query model [16], which includes most of the popular machine learning algorithms. We also show that `dmapply` can express applications written in the popular MapReduce paradigm [10].

A key advantage of **ddR** is that it can be integrated with backends by writing a simple driver code, typically without making any modification to the underlying system. We have integrated **ddR** with several popular data processing engines, such as R's `parallel` package (which leverages multiple cores on a single server), a socket-based backend called `SNOW`, HPE's open-sourced Distributed R [24, 30], and Spark [34]. **ddR** includes a growing list of common R operators such as distributed `sum`, `colMeans`, and others, that have been implemented using `dmapply`. These operators ensure that any backend that integrates with **ddR** gets these operators for free. For example, programmers who use SparkR's dataframes or HPE Distributed R's dataframes can call distributed `colMeans` (average of each individual column) even if the respective backend does not have a native implementation of that function. These helper operators ensure that R programmers can continue to use the same standard functions that they expect from single-threaded R. These operators also provide an incentive to different backend vendors to integrate with **ddR** and help the standardization effort.

We have implemented a number of reference machine learning algorithms in **ddR** that perform clustering, classification, and regression. Our empirical evaluation indicates that these algorithms have very good performance. On a single server, regression with the `parallel` backend on 12M rows and 50 columns converges in 20 seconds, while clustering 1.2M 500-dimensional points takes only 7 seconds per-iteration. Notably, these **ddR** algorithms have performance similar to, and sometimes even better than, machine learning products such as H2O [3] and Spark's MLlib, whose algorithms are custom written and not portable. In a distributed setting, such as with HPE Distributed R as the backend, **ddR** algorithms such as K-means and regression show near linear scalability and can easily process hundreds of gigabytes of data. When used with SparkR or HPE Distributed R, **ddR** algorithms have similar performance as Spark's native MLlib algorithms or Distributed R's algorithms. These results show that users will benefit from the ease of a standardized interface to express applications, and yet obtain performance at par with custom or non-portable implementations.

The core components of **ddR** are available as an R package from R's public package repository called CRAN [20]. In the last four months, the **ddR** package has been downloaded more than 2,500 times [25].

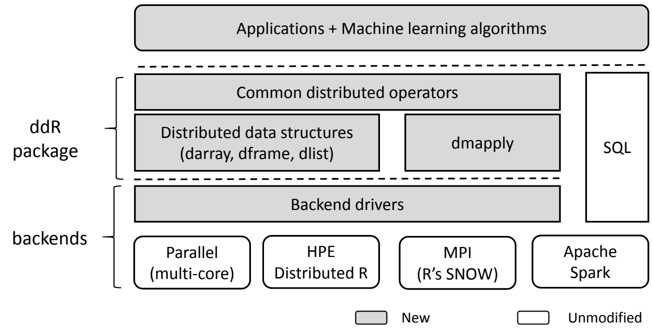

Figure 1: High level architecture. Shaded regions show new components.

The contributions of this paper are:

- Design of a standard interface, **ddR**, for distributed computing in R. We introduce a simple yet powerful functional primitive, `dmapply`, and show that it is *expressive* enough to implement many machine learning algorithms.
- Implementation of **ddR** and its integration with multiple backends including R's `parallel` and `SNOW` backends, HPE Distributed R, and Spark. The **ddR** package makes distributed systems *easy to use* for data scientists.
- Implementation of multiple machine learning applications in **ddR**, which run on different backends, and have *good performance*. We demonstrate that these applications have performance comparable to native versions of these algorithms written in their respective backends.

While our focus in this paper is to improve R, we believe that the concepts in **ddR**, such as `dmapply`, are general enough that they may help other analytics languages, such as Python, that are popular wrapper interfaces for distributed systems. Following the paradigm of dplyr [32] where SQL-like functions in R are transparently pushed into a database, a standard primitive for distributed computing in R could also lead to more intelligent integration between an R application and a parallel database system.

2. ARCHITECTURAL OVERVIEW

The **ddR** project started as a collaboration between R-core, i.e., maintainers of the R language, HPE, and the open-source Distributed R community, which has benefited from cross-industry feedback through workshops and the R Consortium. The goal of the project is to extend R, streamline integration with popular advanced analytics systems, and create portable distributed application and algorithms in R. In this section we provide an overview of **ddR** and how it interacts with different backends.

Figure 1 provides a high level view of **ddR**. The **ddR** programming model exposes distributed data structures, the `dmapply` function to operate on these data structures, and a set of commonly used operators. Programmers use these APIs to create distributed applications, such as machine learning algorithms, and utility libraries, such as parallel

data loaders. One beneficial outcome of implementing applications in **ddR** is that users can choose to switch backends on which their applications execute. For example, a customer may initially use a **ddR** clustering algorithm on their laptop using Rs `parallel` package, which provides efficient multi-core processing, and then run the exact same code on HPE Distributed R or SparkR, in production, and on massive data.

We have implemented the **ddR** programming model on top of multiple backends. To integrate with a backend, the key effort is to implement the three distributed data structures, the `dmapply` function, and a couple of basic operators that provide functionality such as moving all contents of a data structure from workers to the master node. This code lives in the "backend driver" (see Figure 1), a piece of connecting logic that binds the frontend interface to the backend by translating invocations to the `dmapply` primitive to the backend's native API.

The amount of effort required to integrate a backend is low if the backend already has an implementation of the core data structures. For example, Distributed R already implements the three types of distributed data structures and we only had to implement the `dmapply` function. Similarly, Spark provides data frames and we were able to use its RDD API to implement `dmapply`. However, for **ddR**'s default multi-core engine, which uses R's `parallel` package, we had to implement everything from scratch since standard R does not have a notion of distributed data structures.

As shown in Figure 1, programmers who use **ddR** to express their R applications can continue to also use APIs, such as the SQL interface, that the underlying backend exposes. As an example, on the Spark backend users will still be able to call SQL operators on RDDs, while on the `parallel` backend users will be able to invoke the `dplyr` package that implements a subset of SQL operators on R objects [32]. Section 5 provides more details about different components.

Example. Figure 2 shows how programmers can invoke **ddR**'s distributed clustering algorithm. Line 1 imports the **ddR** package, while line 2 imports a distributed K-means library written using the **ddR** API. Line 4 determines the backend on which the functions will be dispatched. In this example the backend used in the default `parallel` backend, which is single-node but can use multiple cores. In line 6, the input data is generated in parallel by calling a user-written function `genData` using `dmapply`. The input is returned as a distributed array with as many partitions as the number of cores in the server. Finally, in line 8, the **ddR** version of the K-means algorithm is invoked to cluster the input data in parallel. The key advantage of this **ddR** program is that the same code will run on a different backend, such as HPE Distributed R, if line 4 is simply changed to `useBackend(distributedR)`.

```
1 library(ddR)
2 library(kmeans.ddR)
3 #Specify which backend to use
4 useBackend(parallel)
5 #Populate a distributed array with synthetic data
6 features <- dmapply(genData,id = 1:ncores,
      MoreArgs = list(centers = cen, nrow = R, ncol
      = C), output.type = "darray", combine = "rbind
      ", nparts = c(ncores,1))
7 #Call distributed K-means library
8 model <- dkmeans(features,K)
```

Figure 2: A distributed K-means application.

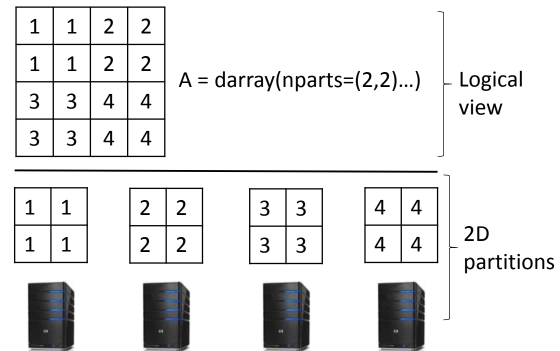

Figure 3: Storing distributed arrays as a set of partitions.

3. DISTRIBUTED DATA STRUCTURES

Data scientists use a variety of data structures and operators to express their analytics workflows. With **ddR**, our goal is to ensure that data scientists can continue to use similar interfaces for distributed computing in R. The **ddR** programming model exposes three types of distributed data structures: arrays, data frames, and lists. Distributed arrays (`darray`) store data of a single type. Arrays can represent vectors and matrices. While it is common to use dense arrays, where each element in the cell has a value, many applications need sparse arrays to efficiently store their data. Therefore, **ddR** also supports sparse matrices which are stored in the column compressed format [13]. By default, distributed arrays are stored in column major order and each partition of a distributed array can be operated by highly optimized matrix libraries such as BLAS [17]. Distributed data frames (`dframe`) are similar to arrays, with the exception that each column can store elements of a different type. Distributed data frames are also stored in columnar format. Distributed lists (`dlist`) store a sequence of elements where each element can be a complex data type such as a matrix.

The combination of these three data structures has been sufficient to express many real world applications. Consider the example of graph analysis. The connectivity in a graph can be represented as a `dlist` of edges or a sparse `darray` for fast matrix manipulation (e.g., PageRank is a series of sparse matrix-vector multiplication). If each vertex has attributes, a `dframe` can be used to store those attributes.

Collection of partitions. Each distributed data structure is internally stored as a set of partitions of the same type. For example, a `darray` is a collection of array partitions, while a `dlist` is a collection of local lists. A partition is always stored in a single server, and the distributed object contains metadata information, such as locations and sizes of partitions, to manage data. Unlike `dlist`, a `darray` and `dframe` may be partitioned not only by rows or columns, but also in blocks (i.e., 2-D partitioning). Figure 3 shows how a `darray` may be stored across multiple servers. In this example, the `darray` is partitioned into 4 blocks, and each server holds only one partition. The `darray` argument `nparts` in the figure specifies how the partitions are located

```
1 #Create a distributed list. By default each
      element becomes a partition
2 A <- dlist(1,2,3,4,5)
3 #Access partitions
4 p <- parts(A)
5 #Multiply elements in each partition by a constant
6 B <- dmapply(function(x) {2*x[[1]]}, p)
7 #Fetch the result (={2,4,6,8,10}) on the master
8 print(collect(B))
```

Figure 4: Declaring and using distributed data structures.

in a grid. For example, if the array was row partitioned instead of blocks, one would use `nparts(4,1)` to state that the four partitions should be stitched row wise by placing them one below the other.

Write-once semantics. A key feature of distributed objects is that they have *write-once* semantics. Once created, a data structure is immutable. However, a data structure can be read-shared, even concurrently, by any number of operators. This choice of immutable data structures is a result of embracing the functional programming approach in R. Data structures are operated on by functions dispatched using `dmapply`, which is side effect free, and returns a new data structure as the result. Therefore, the programming model prevents in-place updates to data structures, thus making it much easier to parallelize a computation.

Accessing partitions. Code may also operate on only a subset of the partitions in a distributed object. In **ddR**, programmers can use `parts(A)` to access the partitions of a data structure. The `parts` function returns a list that contains references to individual partitions. Note that there is no data movement between the master and workers when using the `parts` function; it just returns a reference, which is primarily metadata, about the partitions. In the next section, we will explain in greater detail how partitions provide flexibility in expressing algorithms. Finally, programmers can use the `collect` keyword to gather the whole data structure, or a subset of partitions, and materialize them at the master.

Example. Figure 4 shows a simple example that creates a distributed list and accesses its partitions. Line 2 declares a distributed list which holds the numbers 1 to 5. By default it will be stored as five partitions, each containing one number. In line 4, `p` is a local R list (not a distributed list) which has five elements and each element is a reference to a partition of `A`. Line 6 executes a function on each element of `p`, which means each partition of A, and multiplies each partition by 2. The result `B` is a `dlist`, has five partitions, and is stored across multiple nodes. Line 8 gathers the result into a single local R list and prints it.

4. DISTRIBUTED PROCESSING

The **ddR** programming model introduces only one parallelism primitive, distributed multivariate apply (`dmapply`), for distributed processing. When used in combination with distributed data structures, it provides a flexible way to express multiple types of computation and communication patterns. Therefore, programmers can use it to implement many kinds of analytics applications, as well as utility libraries.

```
1 A <- dlist(1,2,3,4)
2 B <- dlist(11,12,13,14)
3 #C will be a dlist={12,14,16,18}
4 C <- dmapply(FUN=sum, A, B)
5 #D will be a dlist={13,15,17,19}
6 D <- dmapply(FUN=sum, A, B, MoreArgs=list(z=1))
```

Figure 5: Using `dmapply`.

The key arguments of distributed apply, `dmapply(FUN, A,B,..., MoreArgs)`, are (1) `FUN` that is a function, (2) multiple R data structures on whose elements the function is applied, and (3) a list of R data structures passed as `MoreArgs` that are additional input to the function FUN. Note that the R data structures can be both basic R types, such as vectors and lists, as well as distributed data structures. There are three additional arguments that are used to define the type and shape of the output, but we will not focus on them in this paper.

The `dmapply` primitive applies the function `FUN` to each element of the input data structure. Let us assume there are only two distributed lists, `A` and `B`, as inputs, and the function is `sum`. The runtime will extract the first element of `A` and `B` and apply `sum` on it, extract the corresponding second elements, and so on. Line 4 in Figure 5 shows the corresponding program and its results. The `MoreArgs` argument is a way to pass a list of objects that are available as an input to each invocation of the function. As an example, in line 6 of Figure 5, the constant `z` is passed to every invocation of `sum`, and hence 1 is added to each element of the previous result `C`.

4.1 Communication and computation patterns

The `dmapply` interface is powerful because it can express many types of computation patterns.

Function broadcast. A common programming paradigm is to apply a function on each element of a data structure similar to the map function in prior systems [10]. This can be achieved in **ddR** by simply calling `dmapply(FUN,A)`. In fact, programmers can also express that a function should be applied to each partition at a time instead of each element at a time by calling `dmapply(FUN, parts(A))`. In this case, the runtime will invoke `FUN` on a full partition. If there is only one input data structure, the runtime ensures that there is no data movement by shipping the function to the corresponding location of the data. Figure 6(A) illustrates how a function is broadcast to different partitions of a data.

Data broadcast. In some cases, programmers need to include the same data in all invocations of a function. As an example consider the K-means clustering algorithm that iteratively groups input data into K clusters. In each iteration, the distance of the points to the centers has to be calculated, which means the centers from the previous iteration have to be available to all invocations of the distance calculation function. In `dmapply`, programmers use `MoreArgs` to specify what data structures need to be present at all function invocations. The **ddR** runtime supports both standard R objects as well as distributed data structures, and partitions of it, in the `MoreArgs` field. The runtime may reassemble a distributed data structure or parts of it, and then broadcast it to all worker nodes so that the data is available during computation. Figure 6(B) shows how data may be broadcast to all workers.

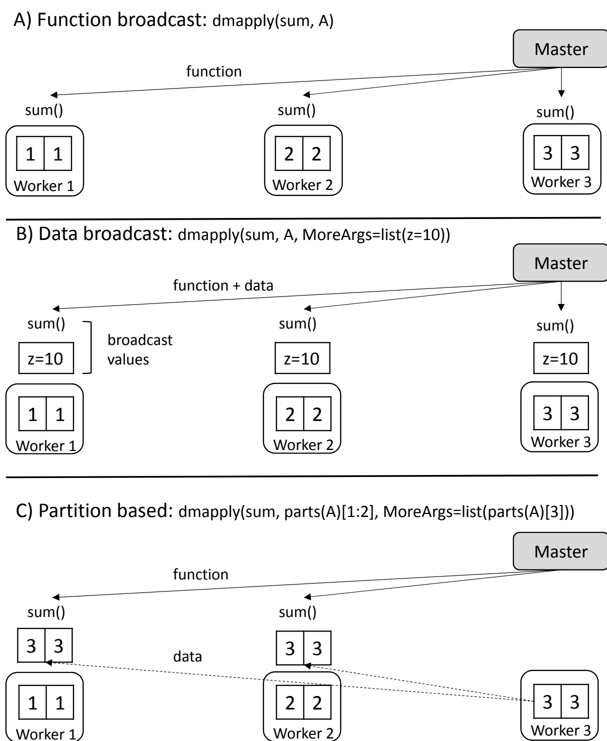

Figure 6: Example computation patterns in ddR.

Figure 7 shows one implementation of distributed randomforest using **ddR**. Randomforest is an ensemble learning method that is used for classification and regression. The training phase creates a number of decision trees, such as 500 trees, that are later used for classification. Since training on large data can take hours, it is common to parallelize the computationally intensive step of building trees. Line 3 in Figure 7 uses a simple parallelization strategy of broadcasting the input to all workers by specifying it in `MoreArgs`. Each worker then builds 50 trees in parallel (`ntree=50`) by calling the existing single threaded randomforest function. At the end of the computation, all the trees are collected at the master and combined to form a single model (line 5). In this example, the full contents of a single data structure are broadcast to all workers.

Accessing a subset of data partitions. The `dmapply` approach allows programmers to operate on any subset of partitions that contain distributed data. As an example, `dmapply(length, parts(A)[1:2])` will find the lengths of the two partitions. In this example, the user invokes the `length` function only on two partitions of the complete data structure. Similarly, a subset of the data can be passed to each invocation of the function by referring to the relevant partitions in `MoreArgs`. In general, this kind of usage can result in data movement, since the runtime has to aggregate the corresponding data partitions. Figure 6(C) shows how a function is applied to a subset of data partitions, after a certain partition from a remote worker is made available.

Multivariate traversal. An important aspect of `dmapply` is that it supports simultaneous iteration over multiple data structures. Irrespective of the number of data structures, the invariant is that the function `FUN` will be invoked on the

```
1 #Input data is a distributed data frame
2 input<-dframe(..)
3 dmodel <- dmapply(randomforest, 1:10, MoreArgs =
      list(data=input, ntree=50))
4 #Combine distributed trees
5 model <- do.call(randomForest::combine, collect(
      dmodel))
```

Figure 7: Distributed randomforest.

first element of each data structure, followed by the second set of elements, and so on. If the number of elements in all the data structures is not the same, then the user can specify `dmapply` to repeat elements of the smaller data structure or throw an error.

There is a subtle point about what constitutes an element. For a `dlist A`, the first element is `A[[1]]`, even if that element is another data structure such as a vector or matrix (since a list can hold complex data types). For a `darray B`, each cell is an element, and the elements are iterated in column major order. For example, the first element in a two dimensional distributed array is `B[1,1]`. For a `dframe C` each column is an element, i.e., the first element is the first column, the second element is the second column, and so on. In the above example if a user writes a statement `dmapply(FUN, A,B,C)` then the runtime will aggregate the first element of `A`, first cell of `B`, and first column of `C`, and apply `FUN` on them , and so on. Ideally, the number of elements in list `A` should equal the number of cells in `B` (no. of rows $\times$ no. of cols), which should equal the number of columns in `C`. These iteration rules of the distributed data structures follow those of their non-distributed counterparts in R (lists, matrices, and data.frames), so R users can easily reason about them as though they were standard R data structures.

In addition to iterating on the the data structure element by element, the user can also iterate on respective partitions of different data structures. For example, `dmapply(FUN, parts(A), parts(B), parts(C))` applies `FUN` to the corresponding partitions of the `dlist`, `darray`, and `dframe`. This iteration policy conforms to the rules we mentioned earlier since `parts` returns a list and iteration occurs on each element of a list, which is a partition in this case. When iterating over multiple data structures, the physical layout of the data, i.e., how data is stored on each machine, determines whether data movement will occur. If corresponding elements of the data structures are co-located, then no data movement is required. Otherwise, the runtime will move data before dispatching the function `FUN`.

4.2 Expressiveness

Since **ddR** advocates the use of a single parallelism primitive, `dmapply`, a natural question to ask is whether it is sufficient to express common machine learning algorithms. We use two approaches to show the expressiveness of **ddR**. First, we show how algorithms that fit the statistical query model [16] can be expressed in **ddR**. Since a number of common algorithms, such as K-means, regression, PCA and others, adhere to the statistical query model, it implies that these algorithms can be expressed in **ddR**. Second, we take the MapReduce programming model as an example and show how its functionality can be achieved using `dmapply`.

4.2.1 Machine Learning algorithms

A statistical learning query is a query made over data that computes $P_{Data}[F(x,y)=1]$, where F is an arbitrary boolean-valued function, x is an observation, and y is the corresponding label for that sample. Intuitively, the query runs the function F over the entire training dataset, and returns the proportion of samples on which F evaluates to one. Algorithms that can be expressed via repeated application of these learning queries are said to fit the statistical learning query model. While it may appear that the statistical query model is restrictive, it has been shown that most learning problems can be reduced to a statistical learning query formulation [16, 27].

Implementation using `dmapply`. Chu et. al. have shown how algorithms that fit the statistical query model can be written as summations over data, and hence be implemented in MapReduce [27]. We use a similar reduction strategy to show that algorithms in the statistical query model can be expressed in **ddR**. In **ddR**, we use a series of `dmapply` and `collect` statements to execute statistical learning queries. Let us assume that the training samples X and their respective classes Y are sampled from the distribution D. Assuming the function F returns a boolean value, we can compute $P_{Data}[F(x,y)=1]$ using the **ddR** code in Figure 8.

```
1 FUN <- function(X,Y,F) sum(sapply(1:nrow(X) ,
        function(i,x,y,f) f(x[i,],y[i,]),X,Y,F))
2 P <- collect(dmapply(FUN, parts(X),parts(Y)))
3 Q <- sum(P)/nrow(X)
```

Figure 8: Computing summations in parallel.

In Figure 8, the function `FUN` computes the summation $\sum_{x \in X_k} F(x,y)$ for each partition k of the input data `X` and `Y`. We use `dmapply` to execute this summation in parallel and then collect the total sum $P = \sum_{x \in X} F(x,y)$. Finally, to calculate the probability we divide the summation by the number of rows of X using `nrow`. Note this same formulation can be used to calculate $E[F(x,y)]$ when F is not a boolean function.

Let's consider a concrete example such as linear regression to show how the summation formulation can be used to implement algorithms. The goal of linear regression is to find β such that $(X\beta - y)^T(X\beta - y)$ is minimized. One may use the gradient descent method to iteratively calculate β^i, which is the estimate of β at iteration i. Since the gradient of $(X\beta - y)^T(X\beta - y)$ is $2X^TX\beta - 2X^Ty$, $\beta^{i+1} = \beta^i - (2X^TX\beta^i - 2X^Ty)$. Therefore, the main computation in linear regression is to calculate $X^TX = \sum_{x \in X} x^Tx$ and $X^Ty = \sum_{x \in X} X^Ty$, both of which are summation problems. These summations can be calculated in a distributed fashion which significantly reduces the overall execution time when X is large. As it turns out, common machine learning algorithms such as K-means, PCA, Gaussian mixture models and others can also be expressed as series of simple summations [27].

Tree-based algorithms. Many learning algorithms, such as randomforest and gradient boosted trees, are an ensemble of decision trees. We show that such tree-based algorithms can also be expressed in **ddR**. The main goal of these algorithms is to segment the sample space Ω into $\{\Omega_i\}$ where $\bigcup \Omega_i = \Omega$, and minimize the loss function $F\left(Y, \hat{Y}\right)$ where $\hat{Y}_i := E_{\Omega_i}[Y_i | X_i, X_i \in \Omega_i]$ on each subset Ω_i. Common loss functions are information entropy and variance. We show that the evaluation of these loss functions can be expressed using $P[F(x,y)=1]$ and $E[F(x,y)]$, which are statistical queries on data and can be calculated in **ddR** using the code in Figure 8.

If the response variable Y is numeric, one can use variance as the loss function. This loss function can be evaluated by calculating, in parallel, the first and second moment of Y, $E_{\Omega_i}[Y]$ and $E_{\Omega_i}\left[Y^2\right]$, on a subset Ω_i. $E_{\Omega_i}[Y]$ and $E_{\Omega_i}\left[Y^2\right]$ can be calculated with the same **ddR** code in Figure 8 since $E_{\Omega_i}[y] = E[y]/P((x \in \Omega_i) = 1)$ and $E_{\Omega_i}\left[y^2\right] = E\left[y^2\right]/P((x \in \Omega_i) = 1)$.

If the response Y is a categorical variable, then one can use gini or information gain as the loss function. The first step in such a computation is to obtain the probability distribution of Y on Ω_i which requires computing the density function $P_{\Omega_i}[y=k]$ for all values of k on the sets Ω_i. Again, $P_{\Omega_i}[y=k]/P((x \in \Omega_i) = 1)$ can be calculated in parallel with the same **ddR** code as calculating $P[F(x,y)=1]$ (Figure 8) where the function F returns 1 if $y = k, X \in \Omega_i$ and 0 otherwise.

Performance. We have shown that **ddR**'s primitives are powerful enough to express a number of parallel algorithms. However, expressiveness does not imply efficient implementation or high performance. Therefore, we use the extensive evaluation in Section 6 to confirm that these **ddR** algorithms are indeed scalable and have good performance, even when compared to custom machine learning libraries.

4.2.2 Relationship with MapReduce

MapReduce is a functional programming interface popularized by the open source Hadoop ecosystem. We show that **ddR** can express algorithms and applications written in MapReduce. Our goal is not to promote the use of the MapReduce interface or port applications written in MapReduce to **ddR**. Instead, we simply focus on the expressive power of `dmapply` by demonstrating its relationship with MapReduce.

Let us assume that a program uses `M` as the map function followed by `R` as the reduce function. The following code in **ddR** gives the same result as running MapReduce with `M` and `R`:

```
1 P = length(parts(X))
2 mapped = dmapply(M,X)
3 temp = dmapply(H(p,i), p = rep(parts(mapped),P), i
      = sapply(1:P, function(ii) rep(ii,P)))
4 reduced = dmapply(R,lapply(0:(P-1), function(i)
      parts(temp,P*i + 1:P)))
```

In the above code, the first `dmapply` statement applies the map function `M` on the input data and stores the results in the distributed object `mapped`. The second `dmapply` call uses the splitting function `H` (typically a hash function on keys) to divide the contents of each partition of `mapped` into sub-partitions that are stored in the distributed object `temp`. Finally, the third `dmapply` call gathers all the partitions of `temp` which have the same key and applies the reduction function `R`.

5. IMPLEMENTATION

As shown earlier in Figure 1, **ddR** is implemented in three layers. The top layer is the application code, such as a distributed algorithm, which makes calls to the **ddR** API (e.g.,

dmapply) and associated utility functions (e.g., colSums). The second layer is the core **ddR** package, which contains the implementations of the **ddR** API. This layer is responsible for error checking and other tasks common across backends, and invokes the underlying backend driver to delegate tasks. It consists of about 2,500 lines of code that provide generic definitions of distributed data structures and classes that the backend driver can extend. Finally, the third layer consists of the backend driver (usually implemented as a separate R packaged such as distributedR.ddR) and is responsible for implementing the generic distributed classes and functions for that particular backend. Typically, a backend driver implementation may involve 500–1,000 lines of code.
Distributed algorithms. We have implemented four **ddR** algorithms and made them available on CRAN: K-means clustering, generalized linear models, PageRank, and randomforest. Each algorithm was implemented using the abstractions provided by **ddR** (e.g., dmapply, parts, collect), and store their data in dlists, darrays, or dframes. Some of the algorithms were implemented in only 250 lines of R code while others required more than 2,000 lines. The core of these distributed algorithms require only a hundred or so lines of code. Most of the remaining code is related to providing functionality similar to existing single-threaded R algorithms, such as robust error handling.
Distributed operators. A major limitation of existing distributed frameworks is that they do not provide common utility functions that R users expect. As an example, a data scientist will routinely use the summary function on an array to obtain the min, max, and quantile like statistical measures of the data. Similarly, a data scientist may require functions to combine two arrays (rbind), find sum of each row (rowSums), or list the last few elements in the data structure (tail). **ddR** removes this limitation of distributed frameworks by expressing common utility functions using distributed data structures and dmapply. As long as the existing frameworks implement dmapply, the users will be able to benefit from these utility functions. We have implemented a number of utility functions in **ddR**. As an example, the distributed implementation of rowSums in **ddR** first collects the local summation for each partition using collect(dmapply(rowSums, parts(x))) and then merges the results at the master node.
Extensibility. **ddR** follows an object-oriented programming pattern, implemented as S4 classes in R [7]. The main **ddR** package defines the abstract classes for distributed objects, while backend drivers are required to extend these classes via inheritance. This permits drivers to override default generic operators in **ddR** (e.g., sum, rowMeans) if the backend has a more optimized implementation of the same functionality. For example, one can implement a generic groupBy operation in **ddR** by using a dlist with $k \times p$ partitions (where k is the number of grouping classes and p is the number of partitions of the distributed object to be grouped), and then shuffle the partitions using parts and dmapply. However, Spark already provides a groupBy operation that is more efficient, and the **ddR** driver for Spark reuses the version provided by Spark instead of the generic implementation.

6. EVALUATION

The strength of **ddR** is its unified interface for distributing computing, and the flexibility to choose the best backend for the task in hand. In this section, we empirically show three aspects of **ddR**:

- We show that the same **ddR** algorithm can indeed be executed on a variety of backends such as R's parallel, SNOW, HPE Distributed R, and Spark, both in single-server and multi-server setups.
- We show that these **ddR** algorithms have good performance and scalability, and are competitive with algorithms available in other products.
- We show there is very little overhead of using **ddR**'s abstractions. Our algorithms implemented in **ddR** have similar performance to algorithms written directly in the respective backend.

Setup. All experiments use a cluster of 8 HP SL390 servers running Ubuntu 14.04. Each server has 24 hyperthreaded 2.67 GHz cores (Intel Xeon X5650), 196 GB of RAM, 120 GB SSD, and are connected with full bisection bandwidth on a 10Gbps network. We use R 3.2.2 with parallel and SNOW, Distributed R 1.2.0, and Spark 1.5. We also compare against established open source machine learning products such as H2O 3.6 [3] and Spark MLlib. H2O is a high-performance machine learning library written in Java. It provides an R interface to invoke the H2O algorithms. Spark MLlib is the machine learning project of Spark and uses optimized linear algebra libraries for performance. We use three machine learning algorithms in our evaluation: (a) randomforest, a decision tree based ensemble learning method, (b) K-means clustering algorithm, and (c) linear regression.

6.1 Single server setup

We first evaluate the case when a user requires only a single server to run computations, either because the input data is small enough or the CPU cores in a single server provide sufficient performance benefits. When using **ddR**, the user can choose R's parallel or SNOW as the backend which run only on a single server. One can also use HPE Distributed R or Spark in a single node mode, though these systems are primarily targeted at multi-server computations. The main difference between parallel and SNOW is in the mechanisms they use for managing multi-process execution. The parallel backend uses Unix fork command to start multiple processes that communicate through inter-process communication. Because of its reliance on Unix system calls, the fork based parallel package does not run on Windows servers. The SNOW package starts multiple processes that communicate using sockets, and can be used even in a Windows environment.

6.1.1 Scalability

Figure 9 shows the performance of parallel randomforest as we increase the number of cores. We use a dataset with 1M observations, 10 features, and run the randomforest algorithm to create 500 decision trees. Randomforest is a compute-intensive algorithm, and the single-threaded default algorithm in R takes about 28 minutes to converge. Using **ddR**, we can parallelize the tree building phase by assigning each core to build a subset of the 500 trees. By using multiple cores, each of the backends parallel, SNOW, and HPE Distributed R can reduce the execution time to about 5 minutes with 12 cores. For this algorithm all the

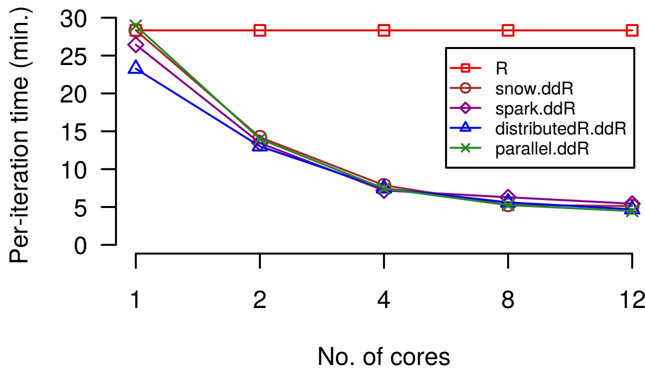

Figure 9: Single server randomforest. Lower is better.

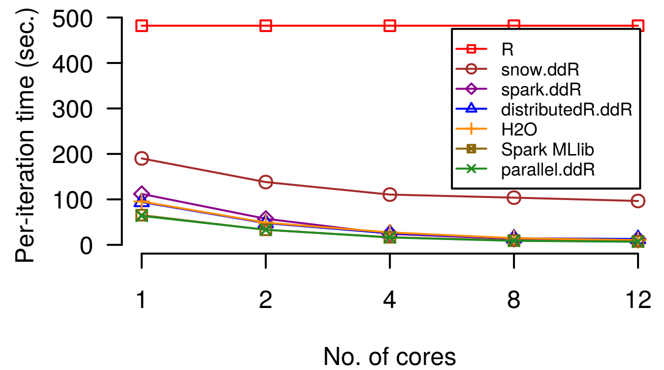

Figure 10: Single server K-means. Lower is better.

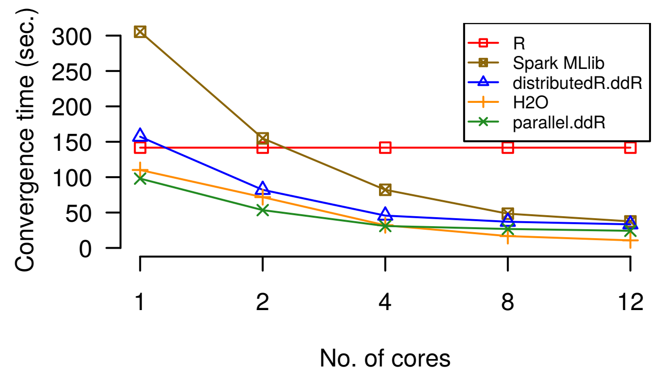

Figure 11: Single server regression. Lower is better.

backends have similar performance, and achieve substantial speedups, about 6× by going from a single core to 12 cores.

Figure 10 shows how long it takes to cluster 1.2M points with 100 attributes into 500 groups. R's default single-threaded algorithm takes 482s for each iteration of K-means. When using `SNOW`, the **ddR** version of K-means takes 96s with 12 cores. HPE Distributed R and `parallel` provide the best performance in this setup, completing each K-means iteration in just 10s with 12 cores. In addition these systems show near linear speedup on K-means, where the performance improves by 10× when going from a single core to 12 cores. The performance of `SNOW` is worse than others because of its inefficient communication layer. `SNOW` incurs high overheads when moving the input data from the master to the worker processes using sockets. In comparison, the `parallel` workers receive a copy-on-write version of the input data when the parent process creates child processes using the fork system call.

Finally, Figure 11 shows the scalability of parallel linear regression algorithm. Since regression is less compute intensive than K-means or randomforest, we use a larger dataset with 12M records each with 50 features. R's single-threaded regression algorithm converges in 141s. The **ddR** regression algorithm on HPE Distributed R takes 155s with a single core but converges in 33s with 12 cores. The `parallel` version is faster and converges in around 20s with 12 cores, which corresponds to about 5× speedup over its single core performance. Since this dataset is multi-gigabyte, `SNOW` takes tens of minutes to converge, of which most of the time is spent in moving data between processes. Therefore, we exclude `SNOW` from the figure.

6.1.2 *Comparison with H2O and Spark MLlib*

In addition to scalability of **ddR** algorithms, we also measure how these algorithms compare to state-of-the-art machine learning products such as H2O and Spark MLlib.

H2O is a multi-threaded machine learning product and has been embraced by the R community for its performance. Unlike other parallel R packages, H2O provides parallel implementations of algorithms itself instead of a generic interface to write parallel programs. Figure 10 and Figure 11 include the performance of the corresponding H2O algorithms. Our evaluation shows that the **ddR** version of K-means on `parallel` is about 1.5× faster than H2O's K-means (Figure 10). For example, **ddR** can complete each iteration in less than 7s with `parallel` with 12 cores compared to more than 10s by H2O. Figure 11 shows that **ddR**'s regression implementation with `parallel` is comparable to H2O, with the H2O implementation slightly faster at 8 and 12 cores. The reason for the slight performance advantage is becasue H2O uses multi-threading instead of multi-processing, as by `parallel`, which lowers the cost of sharing data across workers. However, the **ddR** algorithms on `parallel` and HPE Distributed R outperform H2O in their ability to handle very large data. As an example, **ddR** algorithms on these backends have similar scalability even on 5× the input data size, while the H2O algorithms crash on large datasets.

Spark MLlib provides native implementation of machine learning algorithms using Spark's Scala API. Figure 10 shows that Spark MLlib's K-means algorithm has similar performance as H2O, and is slightly slower than the **ddR** algorithm running on `parallel`. Figure 11 shows that the regression implementation in Spark MLlib, when using 4 or less cores, is about 2× slower than both H2O and **ddR**'s implementation on `parallel` or HPE Distributed R. At 8 or more cores the performance of Spark MLlib is comparable, but still less, than the other systems.

6.2 Multi-server setup

The same **ddR** algorithms that work on a single server also run in multi-server mode with the appropriate backend. In this section we show that **ddR** algorithms can process hundreds of gigabytes of data and provide similar scalability as custom implementations.

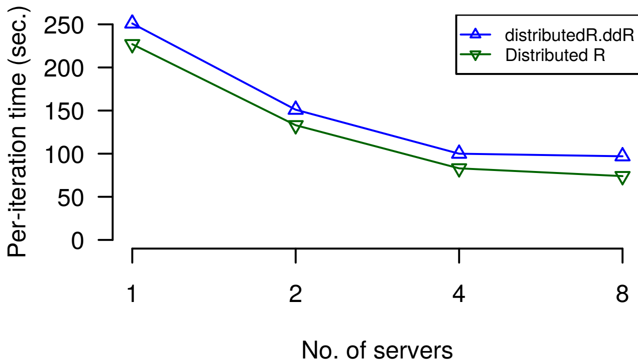

Figure 12: Regression: Strong scaling. Lower is better.

6.2.1 Strong scaling results

Regression is a popular algorithm in the financial sector and is commonly applied on millions of records. We use Distributed R to show how linear regression with **ddR** can be scaled horizontally to utilize multiple servers. For the experiments, we synthetically generated a dataset with 120M rows by creating vectors around coefficients that we expect to fit the data. This methodology ensures that we can check for accuracy of the answers. The dataset size is approximately 95GB and has 100 features per record. Both Distributed R and **ddR**'s implementation of regression use the Newton-Raphson method. Figure 12 shows how distributed linear regression scales on upto 8 nodes, each using 12 cores. The custom regression algorithm in Distributed R takes 227s per-iteration with a single server which reduces to 74s with 8 servers. Therefore, on this dataset the Distributed R algorithm shows about 3× speedup as the number of servers is increased to 8. The **ddR** version of regression, running on Distributed R as the backend, shows similar performance and scalability. On a single server it takes about 251s to complete an iteration which reduces to 97 seconds with 8 servers. Therefore, the custom regression algorithm in Distributed R is only 23% faster than the **ddR** version. The **ddR** algorithm has the added advantage that it runs on other backends as well, thus giving a single interface to the R users.

6.2.2 Weak scaling results

Next, we show the scalability of **ddR**'s K-means clustering algorithm as the dataset size increases. We use synthetic datasets, with 30M, 60M, 120M, and 240M rows. Each dataset has 100 features, and we set the number of centers (K) to 1,000. The 240M row datasets corresponds to approximately 180GB of data. Figure 13 compares the performance of **ddR**'s K-means clustering algorithm on Spark and Distributed R. Additionally, we plot the execution time of the custom K-means algorithms that ship with Distributed R and Spark MLlib. As we increase the number of nodes from 1 to 8, we proportionally increase the number of rows in the dataset from 30M to 240M (i.e., 8×). In an ideal distributed system the per-iteration execution time should remain constant for this setup.

There are three interesting observations from Figure 13. First, the **ddR** version of K-means, on both Spark and Distributed R, scales almost perfectly as the dataset size and

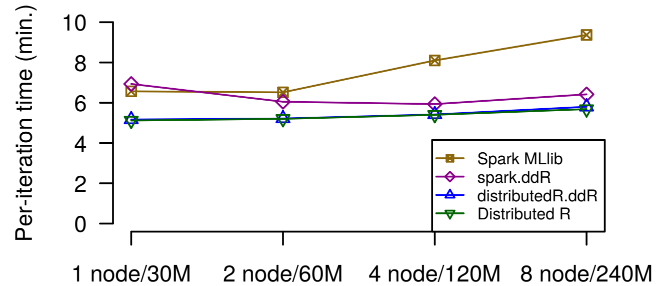

Figure 13: K-means: Weak scaling. Lower is better.

the number of machines is increased. When Spark is used as the backend, the **ddR** algorithm takes around 7 minutes per iteration. With Distributed R as the backend the per-iteration time of **ddR** is around 6 minutes. Second, on this dataset the Distributed R backend outperforms the Spark backend. Therefore, if a user has both the backends installed, it can choose to run the application written in **ddR**, without any modifications, on Distributed R for better performance. Finally, our evaluation shows that the **ddR** version of the algorithm gives the same or sometimes even better performance than the custom algorithm implementation on the same framework. The **ddR** algorithm with Distributed R as the backend (`distributedR.ddR`) is within 5% of the K-means library that ships with Distributed R. The **ddR** algorithm with Spark as the backend is in fact slightly faster than Spark's MLlib algorithm when the dataset size is 120M and 240M.

6.3 Summary

The single server and multi-server results validate our hypothesis that **ddR** provides application portability along with good performance. For example, algorithms written in **ddR** can run on different backends without any code modifications or the user fearing loss of accuracy in the results. The performance of the same **ddR** algorithm will depend on the backend used (e.g., how well a backend handles data transfers), but we found that most backends, with the exception of `SNOW`, result in good performance. For the single server case, we recommend using `parallel` which provides the best performance, and is readily available since it ships with R. On multi-server environment, a user may choose to run **ddR** with Distributed R or Spark, based on availability of the infrastructure. Even though the **ddR** algorithms have been written independent of the backend, our results show that their performance is comparable to, and sometimes even better than, the native algorithms in other machine learning frameworks such as H2O and Spark MLlib.

7. RELATED WORK

Distributed frameworks. MapReduce started the wave of programming models and infrastructures that provide a simple yet fault tolerant way to write distributed applications [10]. Even though simple, many found the MapReduce model too low level to program. As a result, the second wave of distributed systems, such as Pig [23], HIVE [28] and DryadLINQ [33], focused on programmer productivity by providing a SQL like interface. These systems are batch processing systems and not well suited for iterative machine

learning or streaming tasks. Even the machine learning Mahout library, based on the open source MapReduce implementation, has fallen out of favor due to its poor performance [1]. Finally, the third wave of distributed systems focus on domain specific applications and leverage hardware, such as main memory, to improve performance. Spark uses in-memory processing and lineage based fault tolerance to improve performance of distributed applications [34]. Pregel [21], GraphLab [19], and Concerto [18] advocate a vertex-centric programming model to express both graph algorithms as well as machine learning algorithms. Storm [29], Spark Streaming [35] and Naiad [22] provide support for streaming applications.

For data scientists, implementing applications on these distributed frameworks with custom interfaces is a nightmare. There have been efforts to provide R-friendly interfaces to many of these systems. Notable examples are Ricardo [9], RHadoop [6], and SparkR for the MapReduce and Spark programming model. Unfortunately, these interfaces typically expose the custom underlying functions of the framework but in R, such as `map` and `reduce` in the case of RHadoop. The data scientist still needs to learn the underlying system. Moreover, applications written using these interfaces are not portable across frameworks. **ddR** solves these issues by exposing a simple unified interface with similar semantics as R's current single-threaded data structures.

SystemML provides a declarative approach for machine learning which initially focused on MapReduce and now on Spark [12]. By relying on a declarative syntax it inherits certain powerful aspects of databases such as a plan optimizer [8]. **ddR** differs from the approach of SystemML by improving R itself, instead of proposing a new declarative language, introducing a simple but expressive distributed parallelism primitive, `dmapply`, and implementing portable distributed algorithms and utility functions. One could envision integrating **ddR** and certain optimization techniques of SystemML similar to how compilers provide program optimizations.

Databases and machine learning. Most databases provide support for machine learning algorithms either through the ability to call custom UDFs or by exposing in-built algorithms or by integrating with external tools. Many vendors such as Oracle, HPE Vertica, Microsoft SQL server, and others embed R in their database. Users can call any of the thousands of R packages inside a single UDF, though performance is limited by the single threaded R package. As an example, customers may funnel a table through a single threaded R K-means UDF, but they cannot create a distributed K-means function by simply invoking multiple K-means UDFs. In all these cases, the R user interacts with the database primarily through a thin SQL wrapper, such as the hugely popular `dplyr` package [32]. These wrapper R packages are natural for data wrangling in SQL but do not provide a portable way to implement machine learning algorithms. MADlib like approaches show how in-database, machine learning algorithms can be implemented via user-defined functions, and SQL statements [15, 2]. Unfortunately, to contribute an in-database algorithm one needs to follow the programming paradigm and the low level language API proposed by MADlib. Finally, many databases support fast connectors to external frameworks, from single threaded R with SAP HANA [14] to distributed frameworks like Spark and HPE Distributed R with Vertica [24]. When using external frameworks to run machine learning applications outside of the database, the user still has to express her application using the custom interface exposed by Spark or Distributed R.

Parallel libraries in R. R is inherently single-threaded, but there are over 25 packages that provide some form of parallelism extensions to R [11]. Some well known packages include `parallel`, `SNOW`, `foreach`, and `Rmpi`. All of these packages expose their custom syntax for expressing parallelism, and none of them have the concept of distributed data structures. Not surprisingly, even with so many parallel extensions, R has hardly any parallel machine learning algorithms written using them. While the `parallel` package exposes parallelism based on the functional semantics of the apply class of functions, it is a single node solution and does not tackle distributed systems issues such as partitioned distributed data structures. HPE Distributed provides an infrastructure for distributed computing in R, but has similar shortcomings: it exposes a custom syntax and applications written on it cannot run on other frameworks such as SparkR [30, 31].

8. ONGOING WORK

While we have focused on expanding the data structures available in R and integrating with recent distributed analytics systems, one could imagine using a parallel database as a **ddR** backend. By using a parallel database and **ddR**, R users can continue to program in their familiar API, while benefiting from the optimized execution engines of databases. In our ongoing work to integrate **ddR** with a database (such as Vertica), we leverage the fact that many databases support R based user-defined functions. Here we outline one way to implement a database driver for **ddR**.

Maintaining order and partitions. Elements of R objects (e.g., matrix) have implicit structure and ordering with respect to other elements. We expect a database table to include specific columns that denote row or column indices of each element. Similarly, the **ddR** interface also maintains a correspondence between data and the partitions to which they belong. One could create these partitions on the fly using indices or store an explicit database column for partition ids.

Database query. We can express a multivariate `dmapply` by using joins and R user-defined functions. One way to implement the generic `dmapply(FUN(x,y,z), X, Y, MoreArgs = list(z=3.0))` function in Vertica, while ensuring parallel UDF processing, is as follows:

```
SELECT FUN(COMBINED.data1,COMBINED.data2,3.0) OVER
       (PARTITION BY COMBINED.partition)
FROM (SELECT X.data AS data1,Y.data AS data2,X.
       partition AS partition FROM
X JOIN Y ON X.index=Y.index) AS COMBINED;
```

9. CONCLUSION

R is a powerful tool for single-threaded statistical analysis, but lacks proper language bindings for distributed computing. Our design and implementation of **ddR** is a first step in extending the R language and providing a unified interface for distributed computing. We have shown the expressive power of the **ddR** interface, and how algorithms implemented in this interface have performance comparable to custom implementations. Customers will no longer

need to struggle with custom interfaces for each distributed backend; instead, they can write portable R applications using **ddR**, and leverage the already implemented **ddR** algorithms across their favorite distributed computing infrastructure.

10. ACKNOWLEDGMENTS

We thank Michael Lawrence from R-core who worked with us to design and implement **ddR**. We thank Arash Fard and other members of the HPE Vertica team for contributing algorithms to **ddR**. We also thank the R Consortium and attendees of the *HPE Workshop on Distributed Computing in R* for their support and feedback.

11. REFERENCES

[1] Apache Mahout. http://mahout.apache.org.

[2] Comparing pattern mining on a billion records with HP Vertica and Hadoop. http://www.vertica.com/2013/04/08/comparing-pattern-mining-on-a-billion-records-with-hp-vertica-and-hadoop/.

[3] H2O: Machine learning library. http://www.h2o.ai/.

[4] MATLAB–parallel computing toolbox. http://www.mathworks.com/products/parallel-computing/.

[5] Revolution R enterprise scaler. http://www.revolutionanalytics.com/revolution-r-enterprise-scaler.

[6] RHadoop and MapR. https://www.mapr.com/resources/rhadoop-and-mapr.

[7] The s4 object system. http://adv-r.had.co.nz/S4.html.

[8] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in systemml. *Proc. VLDB Endow.*, 7(7):553–564, Mar. 2014.

[9] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 987–998, New York, NY, USA, 2010. ACM.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[11] D. Eddelbuettel. High-performance and parallel computing with r. https://cran.r-project.org/web/views/HighPerformanceComputing.html.

[12] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 231–242, Washington, DC, USA, 2011. IEEE Computer Society.

[13] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, Jan. 1992.

[14] P. Große, W. Lehner, T. Weichert, F. Färber, and W.-S. Li. Bridging two worlds with RICE integrating R into the SAP in-memory computing engine. *PVLDB*, 4(12), 2011.

[15] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib analytics library: Or MAD skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, Aug. 2012.

[16] M. Kearns. Efficient noise-tolerant learning from statistical queries. *J. ACM*, 45(6):983–1006, Nov. 1998.

[17] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, Sept. 1979.

[18] M. M.-J. Lee, I. Roy, A. AuYoung, V. Talwar, K. R. Jayaram, and Y. Zhou. Views and transactional storage for large graphs. In *Middleware*, volume 8275 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2013.

[19] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[20] E. Ma, I. Roy, and M. Lawrence. ddR: Distributed data structures in R. https://cran.r-project.org/web/packages/ddR/index.html.

[21] 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. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[22] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[24] 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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1657–1668, New York, NY, USA, 2015. ACM.

[25] D. Robinson. Package downloads over time. https://dgrtwo.shinyapps.io/cranview/.

[26] D. Smith. New surveys show continued popularity of R. http://blog.revolutionanalytics.com/2015/11/new-surveys-show-continued-popularity-of-r.html.

[27] C. tao Chu, S. K. Kim, Y. an Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Y. Ng. Map-reduce for machine

learning on multicore. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, 2007.

[28] 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 Endow.*, 2(2):1626–1629, Aug. 2009.

[29] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[30] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: Distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 197–210, New York, NY, USA, 2013. ACM.

[31] S. Venkataraman, I. Roy, A. AuYoung, and R. S. Schreiber. Using R for iterative and incremental processing. In *4th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'12, Boston, MA, USA, June 12-13, 2012*, 2012.

[32] H. Wickham. dplyr:A grammar for data manipulation. `https://cran.r-project.org/web/packages/dplyr/index.html`.

[33] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[35] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.