Large-scale Predictive Analytics in Vertica: Fast Data Transfer, Distributed Model Creation, and In-database Prediction

Shreya Prasad Arash Fard Vishrut Gupta Jorge Martinez
Jeff LeFevre Vincent Xu Meichun Hsu Indrajit Roy[+]

HP Vertica, [+]HP Labs

{shreya.prasad,afard,vishrut.gupta,jms}@hp.com
{jeff.lefevre,vincent.xu,meichun.hsu,indrajitr}@hp.com

ABSTRACT

A typical predictive analytics workflow will pre-process data in a database, transfer the resulting data to an external statistical tool such as R, create machine learning models in R, and then apply the model on newly arriving data. Today, this workflow is slow and cumbersome. Extracting data from databases, using ODBC connectors, can take hours on multi-gigabyte datasets. Building models on single-threaded R does not scale. Finally, it is nearly impossible to use R or other common tools, to apply models on terabytes of newly arriving data.

We solve all the above challenges by integrating HP Vertica with Distributed R, a distributed framework for R. This paper presents the design of a high performance data transfer mechanism, new data-structures in Distributed R to maintain data locality with database table segments, and extensions to Vertica for saving and deploying R models. Our experiments show that data transfers from Vertica are 6× faster than using ODBC connections. Even complex predictive analysis on 100s of gigabytes of database tables can complete in minutes, and is as fast as in-memory systems like Spark running directly on a distributed file system.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems; H.3.4 [**Information Storage and Retrieval**]: Systems and software—*distributed systems*

General Terms

Design, Performance

Keywords

HP Vertica; R; Machine Learning; In-database

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00
http://dx.doi.org/10.1145/2723372.2742789.

1. INTRODUCTION

Data scientists rely on advanced statistical tools, such as R and MATLAB, for exploratory analysis: modeling historical data using different algorithms and then applying these models on new data. Since operational data is generally stored in databases, data scientists extract data from databases, import it in their statistical tool, and then use their tool for both creating and deploying models. This workflow is slow and cumbersome. Extracting data using ODBC connectors can take hours, and statistical tools are typically single process, in-memory systems that cannot handle 100s of gigabytes of data.

HP Vertica, Teradata, IBM Netezza, Pivotal Greenplum and others currently implement a handful of data mining algorithms inside the database [22, 14, 2]. Yet, data scientists continue to prefer their external statistical tools because of ease-of-use, the rigor of the algorithm implementations in these tools (for regulatory compliance), and availability of thousands of functions (such as 6000 packages in R). Given the popularity of statistical tools, vendors of these tools have enhanced their systems for parallel processing, which means the number of scalable algorithms will continue to grow in these systems [4, 9, 26]. Additionally, when in-database implementation of an algorithm is unavailable, data scientists are anyway forced to extract data from a database. Hence, in addition to extending in-database capabilities, there is a need to improve the performance of workflows that use both database and statistical tools. In this paper, we describe how HP Vertica integrates with R to solve the challenges of customers who use both database and statistical tools.

Using statistical tools, such as R, with a database is challenging for multiple reasons. First, there is high overhead in transferring data from the database and converting to R's format. Second, even if machine learning models are created in R, it is ill-suited for applying the model on large amounts of data in a timely manner. Third, when database and R instances run on the same node, resource contention can degrade performance.

1.1 Challenges

To understand the challenge in data transfer, consider the case when 50GB to 150GB of table data has to be transferred from a database to R. Figure 1 compares two cases (1) when data is transferred from Vertica to a single R instance and (2) when extensions to R, such as Distributed

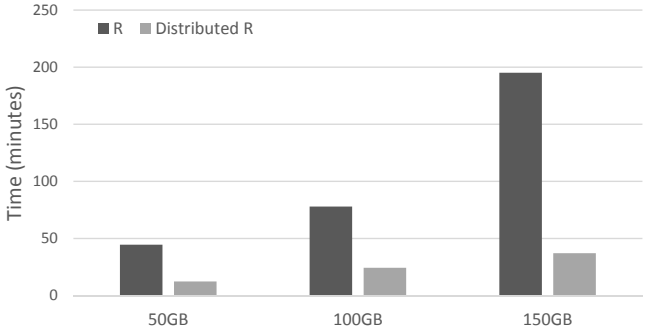

Figure 1: Extracting data from a database is slow. R and Distributed R use ODBC connections in this setup. Lower is better.

R [3], are used to extract data in parallel from a database. We use a 5-node database installation, where both R and Distributed R make connections via ODBC. Using a single R instance (a common scenario with customers), loading even 50 GB takes close to an hour. Parallel analytics engines, such as Distributed R (prior name Presto [26]) or ScaleR [9] may improve performance by starting multiple ODBC connections to the database. Even this approach has limitations. Multiple simultaneous SQL queries can overwhelm the database. For example, an installation of Distributed R on five 24-core servers will simultaneously start 120 ODBC connections, stressing the database. Additionally, ODBC connections destroy data locality when data is moved from the database to concurrent R instances. Figure 1 shows what happens if we launch Distributed R with 120 instances where each R instance concurrently requests $1/120^{th}$ of the table rows in the database. Even with all of this parallelism, loading 150 GB of data can still take close to 40 minutes. Ironically, many machine learning analyses, such as regression, can complete in a few minutes even at these data sizes (Section 7). Using these current approaches means that data scientists may spend more time waiting for data to load than to perform analysis.

Regarding the second challenge, extensions to R provide parallel algorithms that speed up model creation. However, models are typically created by training on a subset of data, while deployment of models can occur on terabytes of new data, and may have real-time constraints. Consider the case of media buying platforms (such as RocketFuel [7]), which bid on digital media advertisement in real time. These platforms may create offline regression models on user characteristics (such as websites visited and demographics), and then use these models to bid, in real-time, on advertisement slots auctioned by Google, Facebook, and other online services. While R is a good fit for offline model creation, it is ill-suited for real-time model deployment–transferring data from a database is slow, and so is managing newly arriving terabytes of data in R.

Finally, brokering shared resources between the database and R is challenging. Machine learning computations are generally compute as well as memory intensive. As an example, a single R instance can easily use 100% of a CPU core during computations, and have a large memory footprint. Running R computations on the same nodes as the database can lead to resource starvation and poor performance for both R jobs and database queries.

1.2 Contributions

This paper describes how HP Vertica solves the above challenges by integrating the Vertica database with Distributed R, HP's open source enhancement to single-threaded R. The integrated product leverages the strengths of both Vertica and R to provide the following features:

- Fast, parallel data transfer between Vertica and R
- Creation of distributed machine learning models in R using database tables
- In-database model deployment and prediction

For fast, parallel data data transfer, our key idea is to reduce the number of simultaneous database queries issued to fetch data, and still allow parallel transfer from Vertica to Distributed R. Only the master node in Distributed R needs to issue a SQL query and initiate data transfer. The new transfer mechanism works irrespective of whether R instances are on the same or different nodes as the database. Our methods enable users to choose between different policies to indicate their preference for maintaining data locality versus ensuring evenly partitioned data in Distributed R.

For model creation, data scientists can leverage high performance parallel algorithms in Distributed R to create models. We have added new data-structures to Distributed R so that parallel algorithms work on data loaded from the database.

For model deployment, we have extended Vertica to save R models and apply them on tables. Since models can sometimes be very large, and inappropriate for storing in table format, we serialize R objects and store them in Vertica's internal distributed file system. Models in Vertica can be accessed by the query engine to run distributed prediction functions. We use Hadoop YARN to broker shared resources when Vertica and Distributed R are installed on the same machines.

Our evaluation shows dramatic increase in performance of workflows that use R on database data. For instance, data transfer is now about 6× faster than using a traditional ODBC connector. As an example, 400GB of data can be loaded into Distributed R in less than 10 minutes. Additionally, in-database prediction takes only a few minutes on billions of rows. We also compare against Spark, an in-memory platform, which is 100× faster than MapReduce for iterative algorithms [28]. Spark loads data directly from Hadoop file system (HDFS) and is expected to be faster than executing machine learning via external tools (such as R) on database tables. Our results show that with Vertica integration, Distributed R executes machine learning algorithms as fast as Spark even when data resides in a database. These results are encouraging because, unlike the low overhead of loading data directly from HDFS, interfacing with a database is slower because the database first loads data from the local filesystem, deserializes and decompresses data, converts it to a standard format and then hands it to Distributed R for parsing and converting into R objects.

2. OVERVIEW

Vertica is a disk-based, columnar store with MPP architecture [23]. Distributed R, on the other hand, extends R

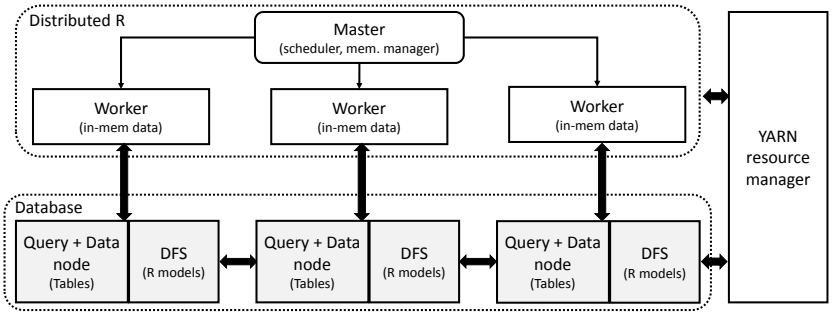

Figure 2: Architectural diagram of Vertica and Distributed R integration

with distributed, in-memory data structures (such as arrays and lists) to execute parallel machine learning algorithms [26]. Distributed R currently handles only data that fits in the aggregate memory of the cluster. For larger datasets, the operating system may invoke virtual memory swapping and substantially increase overheads. In this work we have integrated Vertica with Distributed R by adding functionality to both systems. The new techniques support fast data transfers, maintain data locality, and include the ability to deploy R models inside the database.

Figure 2 shows the architecture of the integrated product. Both Vertica and Distributed R run on commodity Linux nodes. For good performance, we recommend well-provisioned network such as nodes with 10Gb ethernet connection. Distributed R can be installed on either the same nodes as the Vertica database or on remote nodes. We use Hadoop YARN, a resource manager, to broker shared resources between Vertica and Distributed R. YARN uses Linux containers for isolation between processes, and can enforce memory and CPU usage restrictions. YARN ensures that Vertica queries and Distributed R jobs do not adversely interfere with each other.

To begin an R session, users connect to the Distributed R master node shown in Figure 2. During the workflow, data is transferred from Vertica's on-disk tables to Distributed R's in-memory data structures. The data transfer occurs in parallel. For high performance, data is read from local disks, adhering to the table segmentation scheme of Vertica. Once data is loaded in Distributed R, users can manipulate it using R functions and also apply distributed machine learning algorithms. After creating machine learning models, users can store them in the database for later use. If the model has a reference to data spread across Distributed R nodes, then the data is first fetched from Distributed R workers to the master, embedded into an R model object, and then transferred from the Distributed R master to the Vertica node. Within the Vertica node, R models are stored in an internal distributed file system (DFS) and hence accessible to the database query engine including user-defined functions.

The seamless integration between Vertica and Distributed R ensures that users can leverage the strengths of both systems. In a typical enterprise scenario, customers use standard ETL processes to first load data into Vertica. During a predictive analytics workflow, pre-processing steps such as feature extraction can be accomplished inside Vertica itself using SQL operators, sometimes in conjunction with user-defined functions. After pre-processing, machine learning

```
#Start Distributed R session
1 : library(distributedR)
2 : library(HPdregression)
3 : distributedR_start()
4 : ...

#Load features
5 : data<-db2darray(‘‘mytable’’,
          list(‘‘def’’), list(‘‘A’’,’’B’’))

#Run distributed regression and cross validation
6 : model<-hpdglm(data$Y, data$X,
            family=binomial(link=logit))
7 : cv.hpdglm(data$Y, data$X, model)
8 : print(coef(model))

#Deploy model to database
9 : deploy.model(model, ‘rModel’)

#Run in-db predictions
10: q<-‘‘SELECT glmPredict(A,B
         using PARAMETERS model=‘rModel’)
         OVER (PARTITION BEST) from mytable2’’
11: res<-sqlQuery(conn,q)
```

Figure 3: Regression analysis. Illustrates data transfer, distributed model creation, and model deployment.

models can be created in Distributed R. For instance, Figure 3 shows an example R code which uses Vertica and Distributed R to perform distributed regression analysis. In lines 1-3, the user starts Distributed R from an R console. In line 5, the `db2darray` function selects features from a Vertica table and loads it in a distributed array in R. In line 6, `hpdglm` executes distributed logistic regression R. The user can then inspect the regression model, such as the coefficients using `coeff`, and include other types of analysis in R (line 8). Using `deploy.model` in line 9, the regression model is serialized and stored in Vertica. Finally, the user can run predictions inside the database using SQL commands, which can be invoked at the Vertica SQL prompt or through R (line 10-11).

3. VERTICA FAST TRANSFER

We have implemented a new data transfer mechanism, called Vertica Fast Transfer (VFT), that solves two challenges. First, it overcomes the problem that hundreds of simultaneous ODBC connections from Distributed R can

```
select ExportToDistributedR(col1, col2, col3,
         USING PARAMETERS DR_workers =
         'worker1:port|worker2:port',
         partitionSize=100000,
         policy = 'uniform')
       over(PARTITION BEST) from Samples;
```

Figure 4: SQL query issued internally to initiate data transfer

overwhelm the database and make transfers slow. Second, it provides the ability to optimize data transfer for different partitioning policies such as those preserving locality or balancing load. To understand the second challenge, consider the case of multiple R instances connecting to a database. As each instance loads data from the database table, data locality is destroyed. For example, the first R instance will request the first N rows of a table (e.g., 1 to 1 million rows), the second instance will request the next N rows (e.g., 1 million to 2 million rows), and so on. However, the first million rows may be spread across the database nodes (depending upon the segmentation scheme). Extracting these rows from multiple nodes and transferring it to each R instance hurts data locality and increases transfer overheads.

Vertica Fast Transfer has two components. On the database side, once a data request arrives from Distributed R, Vertica starts parallel data streams to Distributed R nodes. Vertica user-defined functions handle these streams and also determine the data partitioning scheme. On the Distributed R side, data streams are stored in-memory, optionally combined, and then converted into R objects. All of these steps are hidden from the user by a single line of R function, i.e., `db2darray` on line 5 of Figure 3.

3.1 Extracting tables

Initially data resides as tables in Vertica and is stored as *segments* on the database nodes. After transfer to Distributed R, the data becomes part of distributed data-structures (such as arrays). To depict that the data distribution may be different in Distributed R compared to the original database table, we call chunks of data local to Distributed R nodes as *partitions*. As an example, a 5-node Distributed R cluster may load a database table as an R array with 5 partitions. The union of these partitions correspond to the complete data in the table.

To initiate data loading, the Distributed R master node issues a single SQL query to Vertica. At the same time, Distributed R workers start listening for network connections from Vertica processes. The SQL query invokes a Vertica user-defined function, `ExportToDistributedR` to start data transfer. The query also relays meta-data information that is used in subsequent steps to perform the parallel data transfer. Figure 4 shows the three key arguments that are included in the `ExportToDistributedR` function. First, network information such as the hostname and ports of the Distributed R workers, is passed. This information tells Vertica which R instances to connect to. Second, approximate sizes of partitions that R instances expect, is provided. Partition sizes are calculated by dividing the the number of rows in the Vertica table by the total number of R instances waiting to receive the data. This calculation is performed by the Distributed R code that invokes the data transfer. Partition sizes are used as hints by Vertica to determine how much

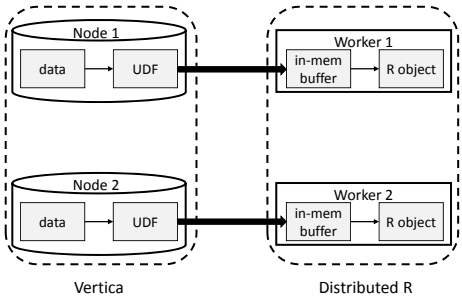

Figure 5: Locality preserving data transfer policy

data should be buffered before transferring to R instances. Third, the transfer policy argument determines how data should be spread across R instances.

Once all data transfer related information is received, Vertica spawns multiple instances of user-defined functions (UDFs) to extract data from its columnar storage. UDFs on each database node read a unique segment of the table stored on that node. The partitioning policy determines how the database table should be split for each UDF instance to operate. Generally, the `PARTITION BY` clause on a particular table column is used. However, in many cases there is no natural column to partition the table, and the only requirement is to transfer table contents to Distributed R without ordering constraints. In such cases, Vertica's `PARTITION BEST` scheme is appropriate as it processes data that is local to each Vertica node and sends it to the corresponding R instance. Using this scheme, we avoid the costly data movement of simultaneous ODBC connections that fetch ordered sequence of rows from a table. Vertica's `PARTITION BEST` takes into account resource availability, such as CPU and memory usage, to determine the optimal number of UDF instances to spawn. The UDFs use in-memory buffers to stage table contents before pushing them to R instances.

3.2 Data distribution policies

Vertica can currently transfer data to Distributed R using one of the two policies– *locality preserving* and *uniform distribution*. As shown in Figure 5 and 6, these policies provide users control over how data is distributed. Figure 5 illustrates the locality preserving policy. This policy adheres to the data segmentation on Vertica, i.e., there is one-to-one mapping of data transfer between Vertica nodes and Distributed R workers. In this policy, all UDF instances executing on Vertica node 1 will send data to Distributed R worker 1 and so on. This approach is used when Vertica and Distributed R have the same number of nodes. In the common case, where Vertica and Distributed R are installed on the same nodes, there is additional advantage as network overhead is minimized.

While data locality is important, it can sometimes lead to load imbalance and hence poor performance of machine learning algorithms in Distributed R. For example, if tables in Vertica have skewed segmentation, once loaded in Distributed R, some R instances will hold more data than others. During the execution of distributed machine learning algorithms, this data skew can lead to straggler tasks, and hence poor performance [20]. Figure 6 illustrates the uniform distribution policy, which addresses the issue of data

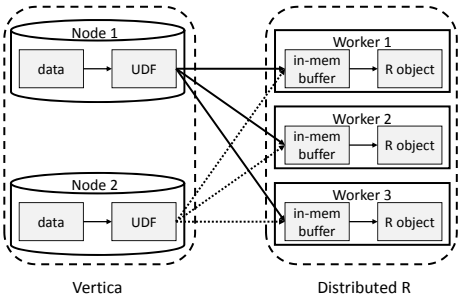

Figure 6: Uniform distribution data transfer policy

skew. The key idea is to sprinkle data across Distributed R workers such that data distribution is largely uniform, i.e., each R instance will contain the same amount of data. In this policy, each UDF instance connects and transfers data equally to Distributed R workers in a round-robin fashion until all local table data has been processed. This policy can be employed irrespective of the relative number of Distributed R and Vertica nodes.

3.3 Receiving data in R

As each Distributed R node receives data from Vertica, it stores them as in-memory data files (typically in /dev/shm). For parallelism, each Distributed R node uses a thread-pool to accept data streams from multiple Vertica UDF instances. The conversion of incoming data into distributed data-structures happens in two steps. First, an empty distributed data-structure (such as a distributed array) is creates in R. This empty data-structure will have a symbol table in the Distributed R master node, which points to empty partitions on worker nodes. Once R instances on the workers receive enough rows from Vertica, the in-memory files are converted into R objects and assembled into partitions. Thus, the empty pointers in the distributed data-structure start pointing to the now filled partitions. At the end of this stage, data from Vertica is available for use by Distributed R algorithms.

4. NEW DATA STRUCTURES ADDED TO DISTRIBUTED R

Distributed R enhances R's data structures such as arrays, data-frames, and lists to store data in-memory across nodes. Data in Distributed R is partitioned by rows, columns, or blocks (sub-matrices). Using these data-structures, R programmers can manipulate remote data and express distributed algorithms. For example, to add two distributed arrays, programmers write a R function that takes corresponding partitions of the two arrays and adds them. In this section, we describe novel additions to Distributed R for integrating with Vertica. To simplify the discussion we focus on distributed arrays. We first discuss the existing state of distributed arrays and then describe the enhancements.

In current distributed arrays, all partitions, except for the last one, are of the same size. The last partition may be slightly smaller if the total number of rows in the data is not divisible by the number of partitions. Figure 7 shows an example where the input table has 2 columns and 6

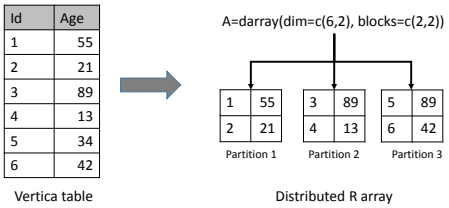

Figure 7: Distributed arrays originally supported only equal sized partitions

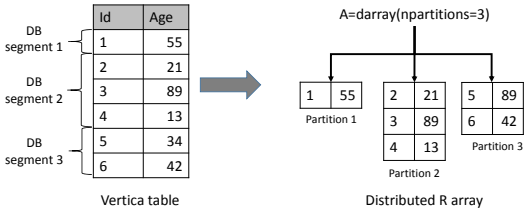

Figure 8: New data-structures now handle arrays with different partition sizes

rows (ignoring the names of the columns). The programmer can declare an array using `A = darray(dim=c(6,2), blocks=c(2,2))` and load the input data into Distributed R. This array definition creates a distributed array of size 6×2, where each partition is a 2×2 sub-array. Therefore, the array has equal sized partitions.

Forcing partitions to be the same size has multiple advantages. First, R programmers can treat distributed arrays just like normal R arrays. For example, the syntax of declaring distributed arrays (`darray(dim=,blocks=)`) is same as R arrays, except that `blocks` represents the partition size. Second, in many distributed algorithms different arrays may need to be co-partitioned, i.e., partitioned in the same manner for parallel processing to occur. As an example, to add distributed arrays, the arrays should be partitioned in the same manner (such as by rows) and be of the same size. It is fairly simple to use `blocks` to declare arrays that have the same partition size. Finally, knowledge of array dimension and partition sizes during declaration simplifies how Distributed R allocates and manages its memory space.

To integrate with Vertica, we need to handle arrays with different partition sizes. In fact, partition sizes are not known at the time distributed arrays are declared. Instead, partition sizes become available once data has been transferred from Vertica to Distributed R. The reason for the different partition sizes is that the Vertica table segments present on each node depend on the segmentation scheme and may contain different numbers of elements. Therefore, when using the *locality preserving* data transfer policy, partitions of the distributed array correspond to table segments and will be of uneven size. Figure 8 shows an example where an input table is initially stored in a 3-node Vertica database. In this case the first row of the table is in database node one, the next three rows are in the second node, and the remaining in the last node. If we want to maintain data locality while loading data in Distributed R, we need distributed data-structures that can store different sized partitions in R.

Functionality	Description
`darray(npartitions=)`	Create a distributed array with specified number of partitions
`dframe(npartitions=)`	Create a distributed data-frame with specified number of partitions
`dlist(npartitions=)`	Create a distributed list with specified number of partitions
`partitionsize(A,i)`	Return size of i^{th} partition or of all partitions if i is missing.
`clone(A,nrow=,ncol=)`	Return another object with the same structure, such as number partitions, as input A .

Table 1: New language constructs in Distributed R

To handle different sized partitions, we have added new data-structures in Distributed R. Table 1 lists the new data-structures and helper functions in Distributed R. Users can declare distributed arrays without providing any size information (`darray(npartitions=)`). After declaration, meta-data related to `darray` is created on the Distribute R master node, but no memory is reserved on the workers to store data contents. Users can call R functions to either generate data for each partition or load data from Vertica. Even though each partition can be of different size, Distributed R checks for conformity between adjacent partitions. For example, if data is row partitioned, each partition may have variable number of rows, but the same number of columns. These checks ensure that arrays constitute well-formed matrices.

Figure 8 illustrates an example of array partitions when data has been loaded from Vertica to Distributed R. The Distributed R array has three partitions, each of which is of a different size. The first array partition contains only one row which is the data present in the first segment of the database. The second and third array partitions contain three and two rows respectively, which correspond to the remaining two database segments.

We have implemented multiple machine learning algorithms in Distributed R that use these new data-structures. In many of these algorithms, intermediate steps require arrays which have the same partitioning scheme as other arrays. Since arrays can have hundreds of different sized partitions, we provide high level functions such as `clone` to copy the structure of an existing array.

Figure 9 illustrates how a table from Vertica is loaded into a distributed array `X`, in order to run algorithms. The first step, `X<-db2darray(..)`, loads data from the table *Samples* to the array `X`. The array `X` has two partitions but each with different number of rows. Next, a user may run the `Y<-clone(..)` command to create a vector `Y`. `Y` has the same number of partitions as `X`, same number of rows, one column, and the partitions are co-located with those of array `X`. In the figure, the Distributed R memory manager is located on the master node. The memory manager tracks the location and meta-data of each partition.

5. IN-DATABASE MODEL DEPLOYMENT AND PREDICTION

After creating machine learning models in Distributed R, data scientists typically use the model for predictions. We have added new features in Vertica to store and apply ma-

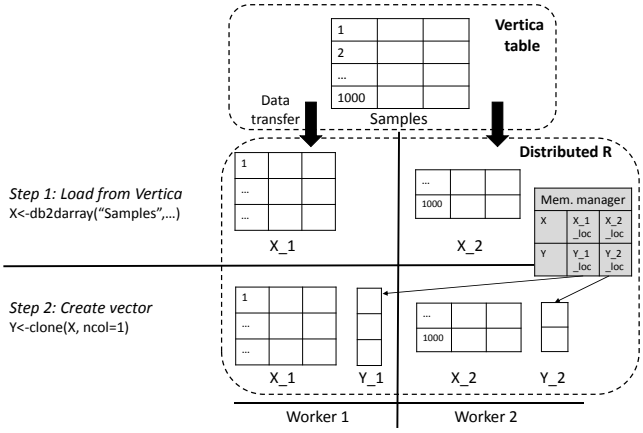

Figure 9: Distributed arrays with different partition sizes

```
=> select * from R_Models;

model  | owner | type       | size | description
-------------------------------------------------
model1 | X     | kmeans     | 100  | clustering
model2 | Y     | regression |  20  | forecasting
```

Figure 10: Vertica R models table

chine learning models in the database. Saving models in Vertica eliminates the need to move data to an external analytical tool to perform predictions. Therefore, models can be applied on terabytes of data as well as on newly arriving data which is streamed into the database.

Once a model has been created in Distributed R, users can deploy them in the database using the `deploy.model` function. Users simply pass the R model as well as a name with which the model can be referred to (e.g., Figure 3 line 9). Internally, models are first serialized and then transferred to the database using a Vertica user-defined function (UDF). Since models can be large (sometimes gigabytes), we don't store them as part of a regular table. Instead, models are stored as binary blobs in Vertica's distributed file system (DFS). The Vertica DFS was primarily created for storing unstructured data that are accessible by the query engine. The DFS can replicate files across nodes to ensure that they are available at all nodes. While models are stored in the DFS, meta-data related to the models are stored in a database table called `R_Models`. Figure 10 shows an example of what the model table may contain. Models can be assigned security permissions to grant access or modification rights to database users. Models stored in the DFS provide the same fault-tolerance guarantees as Vertica tables.

Users can explore models and apply them on tables by providing R functions that perform predictions. Prediction functions are algorithm specific because both the data contained in the model, and how it should be used depends upon the machine learning algorithm. As an example, a K-means clustering model may contain information about centers while a regression model may contain only coefficients. Applying a K-means model on new data involves calculating distance from the centers while for linear regres-

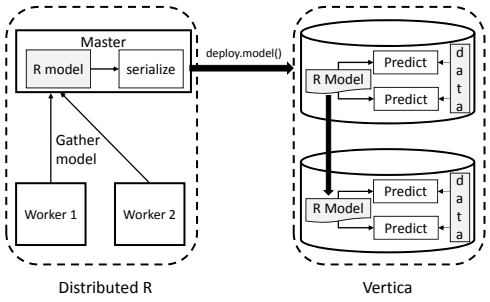

Figure 11: Model deployment and prediction

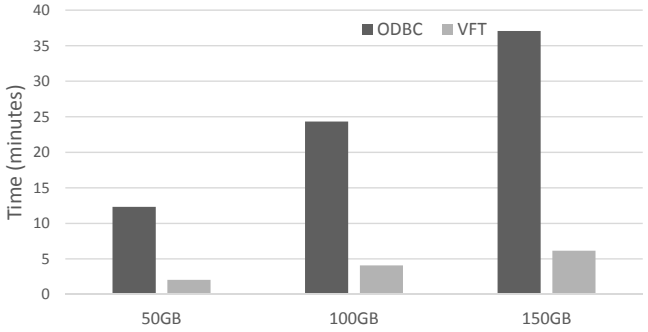

Figure 12: ODBC vs. Vertica Fast Transfer in a 5-node cluster. Lower is better.

sion coefficients have to be multiplied with values in a row. We have added prediction functions in Vertica for common machine learning models such as clustering, regression, and randomforest. Users have the flexibility to create their own prediction functions for custom models and register them with Vertica.

Figure 11 illustrates the complete workflow of how models are created in Distributed R and then deployed in Vertica for predictions. If the content of a machine learning model is distributed across Distributed R workers, the master first gathers the model from R workers, and then sends them to one of the Vertica nodes. The Vertica DFS internally replicates the model, and makes it available to different nodes for prediction functions. When prediction functions are invoked, Vertica starts user-defined functions that first retrieve the models from DFS, deserialize and load them in R, and call the prediction function on the input data. The Vertica query planner starts many parallel instances of user-defined functions. The amount of parallelism is dependent on resources available and how the input table is partitioned. When the table is well partitioned among the nodes of the Vertica cluster, a near linear speedup can be achieved through parallel prediction.

6. RESOURCE MANAGEMENT

Distributed R and Vertica can be deployed on the same node or on separate nodes. There are advantages and disadvantages to both setup options. Using just a single set of nodes means that in many cases network overhead is minimized and transferring data from Vertica to Distributed R is fast. Node consolidation also leads to better resource utilization so that the database and R processes can share resources when needed. The disadvantage of such an approach is that resource isolation is a thorny issue. Under heavy usage, both SQL queries and machine learning processes will see performance degradation, especially when resources are insufficient. Deploying Vertica and Distributed R on separate nodes can be considered a case of static resource allocation. It ensures that machine learning analysis does not affect production SQL queries. However, if machine learning analysis is infrequent or SQL queries occur in bursts, node resources remain idle and underutilized.

We use Hadoop's YARN resource manager for allocating and isolating resources. YARN uses a two level scheduler, supports different allocation policies such as capacity and fairness, and is cognizant of data locality. To use YARN, each framework requires an application manager which makes requests to YARN for resources. Instead of creating a new resource manager, we leverage YARN because it is a natural choice for multi-engine software stacks.

Since releasing resources and tearing down a database is costly, Vertica requests resources from YARN for long term use. Distributed R, on the other hand, requests resources from YARN whenever a user starts a session. When starting a Distributed R session, users can specify resources such as the number of cores or amount of memory to be used in that session. These user specified resources are requested from YARN, with a preference for data locality with Vertica. When scheduled on the same nodes, Vertica and Distributed R processes are isolated using Linux `cgroups` [6]. These enforcement mechanisms ensure that each process is restricted to the allocated amount of CPU and memory usage.

7. PERFORMANCE BENEFITS

By integrating Vertica and Distributed R, we address two major customer complaints– extracting data from databases is slow and R cannot be used for analysis on really large data. In this section we empirically evaluate the benefits of our integration.

Setup. Our experiments use a cluster of 24 HP SL390 servers running CentOS 6.4. Each server has 24 hyper-threaded 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 Vertica 7.1 and Distributed R 1.0.0. For the comparison study with Spark, we use Spark 1.1.0 running on HDFS. HDFS is set to the default 3-way data replication. Spark machine learning algorithms are from the MLlib package which uses optimized linear algebra libraries.

7.1 Vertica Fast Transfer

The aim of our new Vertica Fast Transfer (VFT) mechanism is to ensure that data can be loaded in an external analytics engine in a matter of minutes. Figure 12 compares the performance of loading data in Distributed R using parallel ODBC connections versus VFT. In this experiment, we use a 5-node Vertica cluster, with Distributed R installed on a separate 5-node cluster. For VFT we use the locality-preserving policy. Distributed R starts 24 R instances on each node, and each R instance connects using an ODBC connection or uses the VFT mechanism. We vary the Vertica table size from 50 GB to 150 GB data (approximately 1 billion to 3 billion rows). The results show that VFT can load datasets as large as 150 GB in less than 6 minutes, com-

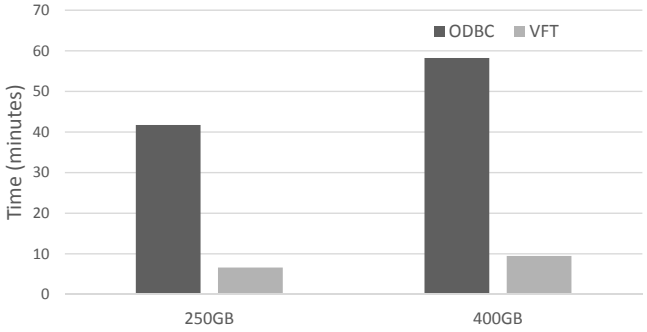

Figure 13: ODBC vs. Vertica Fast Transfer in a 12-node cluster. Lower is better.

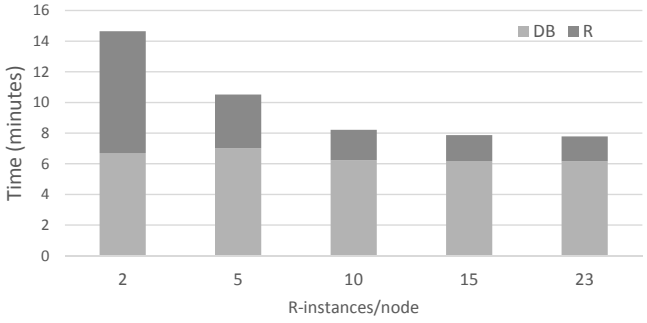

Figure 14: Time breakdown in Vertica Fast Transfer. 12-node cluster setup with 400 GB data.

pared to about 40 minutes in the case of ODBC connectors. We noticed that running Distributed R and Vertica on the same servers has similar performance, which means that the network is not a bottleneck. Instead, disk, database, and R execution are the bottlenecks.

Figure 13 shows similar results on a 12-node cluster and with up to 400 GB size tables. For the ODBC case, Distributed R will spawn 288 (12*24) connections to the database. Unfortunately, even with these many parallel connections and a large database cluster, it takes almost an hour to load 400 GB of data. With VFT, the load time is less than 10 minutes. Figure 14 shows a breakdown of the time spent in VFT, as we increase the number of R instances on each server. The *DB* part includes time taken by Vertica to read 400 GB data from disk, serialize, and send it across the network. The R part includes the time taken by Distributed R instances to receive data, buffer it, and finally convert to an R object. The results show that when there are only a couple of R instances per server, almost half of the transfer time is spent in buffering data and converting into R objects. Time taken by the database is constant and independent of the parallelism in Distributed R. The reason is that the database, irrespective of Distributed R, uses the same amount of parallelism and resources as specified by its query planner. As we increase the number of R instances (more parallelism in Distributed R), the time to create R objects decreases.

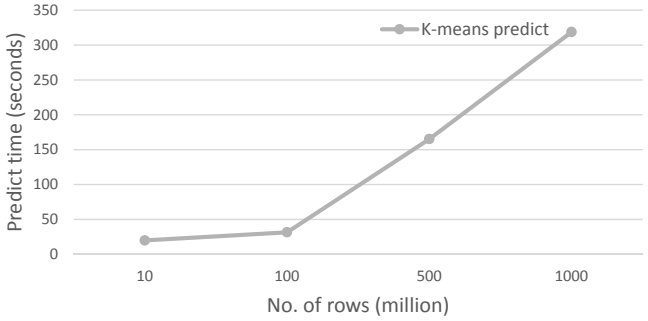

Figure 15: Scalability of in-database K-means prediction.

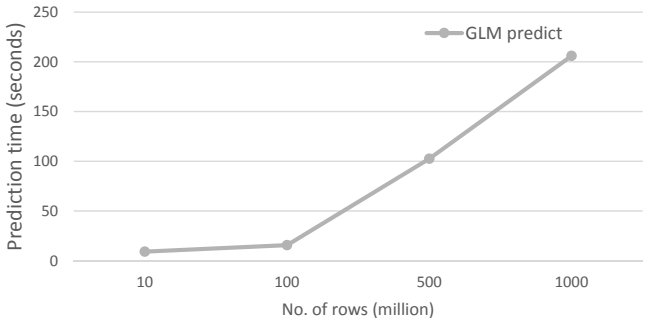

Figure 16: Scalability of in-database linear regression prediction.

7.2 Scalable model deployment in Vertica

Once machine learning models are created, they can be stored in the database for later use. Customers can, therefore, use Vertica itself for predictions. To measure the performance of in-database prediction, we use a 5-node Vertica cluster, and populate tables with six columns and up to a billion rows. We test the performance of prediction on two machine learning models, K-means and linear regression.

Figure 15 shows the time taken to apply the K-means model on Vertica tables. A K-means model contains the final centers into which the training data was clustered. By applying the predict function (`KmeansPredict`), each point in the table is mapped to its nearest cluster center. For each row in the table, `KmeansPredict` calculates the euclidean distance of the point to the centers. The prediction function runs in parallel inside Vertica and, as shown in Figure 15, takes less than 20 seconds to execute on 10 million rows. As we increase the table size to a billion rows, the prediction time increases to 318 seconds. The execution time shows close to linear scaling because both the dataset and execution time grows by approximately 100×.

Figure 16 shows a similar behavior for linear regression. The linear regression model consists of coefficients that define a line best fitting the training data. The Vertica predict function (`GlmPredict`) applies these coefficients on the input table and outputs the response value. As shown in Figure 16, Vertica takes less than 10 seconds to run prediction on a 10 million row table. Even on a 1 billion row table, it takes only 206 seconds to complete prediction. Similar to K-means pre-

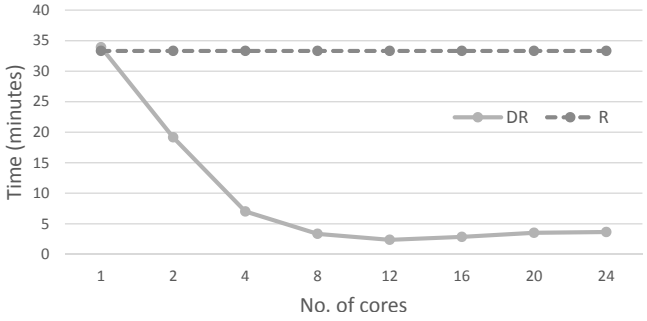

Figure 17: Distributed R vs. R: K-means clustering on a single node. Lower is better.

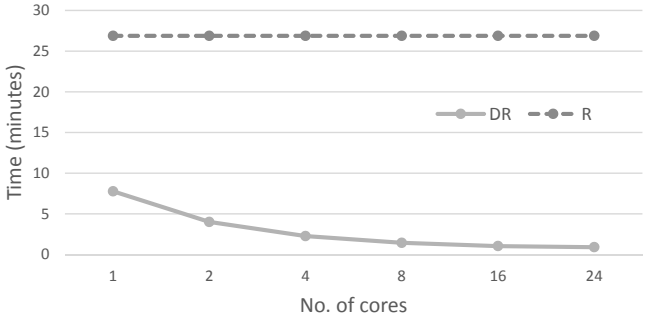

Figure 18: Distributed R vs. R: Linear regression on a single node. Lower is better.

diction, the linear regression prediction function shows near linear scalability as the dataset size is increased.

7.3 Comparison with R and Spark

Vertica's integration with Distributed R not only helps users overcome the limitations of R, but also allows them to efficiently analyze their data stored in the database. We use experiments to show how our solution compares against the state-of-the-art.

7.3.1 Advantages over R

R, due to its single-threaded nature, has limitations regarding both the dataset size and performance. HP Vertica Distributed R solves many of these limitations by scaling R to multi-core and multi-node settings. We have open sourced different clustering, classification, and graph algorithms in Distributed R, and made them available on GitHub [3]. This section provides evidence of the performance and scalability benefits of Distributed R. Here, we use K-means and linear regression as example machine learning applications. We exclude the time taken to load data from Vertica because single-threaded R, using one ODBC connection, takes a long time to load from a database (as shown earlier in Figure 1).

K-means clustering assigns points to one of K groups based on their similarity. In each iteration, points are first mapped to their closest centers and then new centers are calculated by averaging the groups. In this experiment, we set the number of centers, i.e., K, to 1000 and use a synthetic dataset with 1 million points, each with 100 features, to compare the performance of R and Distributed R on a single node as we vary the number of cores from 1–24. Figure 17 plots the time taken per iteration for both R and Distributed R . Figure 17 shows that R takes approximately 35 minutes to complete an iteration of K-means, even as the number of cores is increased. In contrast, on a single node Distributed R can reduce the per-iteration time to less than 4 minutes by using 12 or more cores. The performance plateaus beyond 12 cores because the node has only 12 physical cores and the K-means algorithm is compute bound. Overall, Distributed R shows 9× speedup over stock R by using 12 cores.

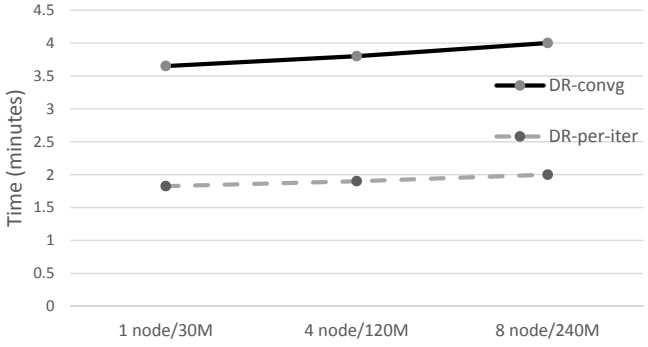

Figure 19: Scalability of linear regression in Distributed R. Both per-iteration and convergence time are shown. Lower is better.

We also compare against R using regression analysis, which is widely used by financial firms for forecasting, such as predicting sales based on customer characteristics. Figure 18 compares the performance of regression on a dataset with 100 million rows and 7 columns. On this dataset, R takes more than 25 minutes to converge to the result. In comparison, Distributed R takes less than 10 minutes even with a single core. The reason for this difference is because R uses matrix decomposition to implement regression, while Distributed R uses the Newton-Raphson technique [27]. Even though the final answer is the same, these techniques result in different running time. Irrespective of the implementation technique, Distributed R shows good single-core performance as well as scalability with the number of cores. As the number of cores is increased from 1 to 24, the execution time of Distributed R drops from about 8 minutes to less than a minute, representing a 9× speedup.

Scaling on multiple nodes. While R is restricted to a single node and single core, Distributed R can scale horizontally and utilize multiple nodes to analyze multi-gigagbyte datasets. Figure 19 shows how distributed linear regression scales on upto 8 nodes. For the experiments, we synthetically generated datasets by creating vectors around coefficients that we expect to fit the data. This methodology ensures that we can check for accuracy of the answers by Distributed R. We use 100 features per dataset, and the number of rows are 30M, 120M, and 240M for 1, 4, and 8 nodes respectively, representing a proportional data increase per node. Figure 19 shows that Distributed R completes each Newton-Raphson iteration in less than 2 minutes, and converges in just 4 minutes (2 iterations).

7.3.2 Comparison with Spark on HDFS

Due to the prevalent belief that loading data into a database and then exporting data out of a database is slow,

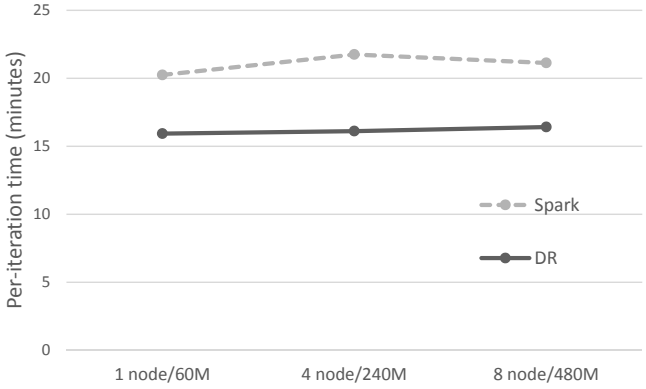

Figure 20: Distributed R vs. Spark: K-means algorithm. Lower is better.

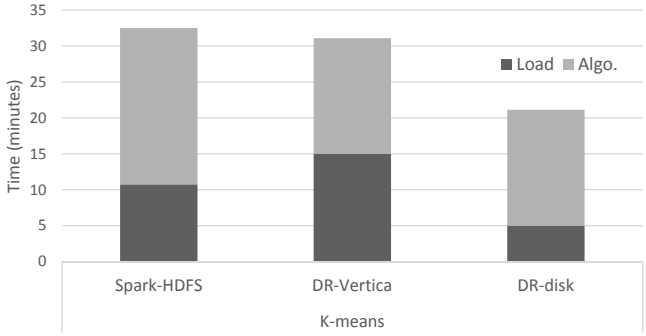

Figure 21: End-to-end comparison between Vertica-Distributed R and Spark. We use a 4-node cluster setup. Lower is better.

Hadoop, especially the HDFS filesystem, has become a common choice for storing raw data. On this data, analysts may use MapReduce, Spark, and other programming paradigms for different kinds of analysis. While MapReduce and systems built on top of it are known to be slow, Spark provides a fast, in-memory computation layer, and is an order of magnitude faster.

In this section we show that Vertica with Distributed R can be as fast as Spark running on top of HDFS. These numbers are encouraging because loading data directly from a distributed file system is expected to be much faster than reading data out of a database and running analytics in an external tool. This is because the database has to read data from the local filesystem, deserialize and decompress data, convert it into a standard format (such as csv), and may even need to serialize again if the client is using a network connection. The external tool, such as R, will then parse the incoming data and convert it into objects. In comparison, Spark which is tightly integrated with HDFS, reads the data directly from the local HDFS node and optionally deserializes the data before converting into its own data-structures.

We run K-means on multi-gigabyte datasets and compare the end-to-end application performance between Vertica and Spark. We use three setups: 1 node cluster, 4 node cluster, and 8 node cluster running Distributed R on Vertica and Spark on HDFS.

K-means. We use three synthetic datasets, with 60M, 240M, and 480M rows. Each dataset has 100 features, and we set the number of centers (K) to 1000. These datasets correspond to approximately 45GB, 180GB, and 360GB on-disk data.

Figure 20 compares the performance of Spark and Distributed R. We exclude the load time to first measure the scalability of each system. As we increase the number of nodes from 1 to 4 and finally 8, we proportionally increase the number of rows in the dataset from 60M to 240M, and then to 480M. Therefore, in an ideal distributed system the per-iteration execution time should remain constant. In the figure, `Spark` and `DR` denote the same implementation of the K-means algorithm, and hence an apples-to-apples comparison.

Figure 20 shows that Distributed R takes approximately 16 minutes to complete each iteration at 8 nodes while Spark requires 21 minutes or more. Both Distributed R and Spark scale well as the number of servers and the dataset size is increased. Overall, Distributed R and Spark have similar performance, with Distributed R faster about 20%.

End-to-end experiments. While the previous experiment shows that machine learning algorithms available in Distributed R are slightly faster than Spark, the results exclude one important aspect–time to load data. Our final experiment compares end-to-end performance, i.e, application time plus the time taken to load data from Vertica compared to Spark loading data from HDFS.

We run K-means in the 4-node setup. The datasets are the same as in the previous experiment, 240M rows and 100 features. For K-means, Distributed R takes 15 minutes to load the data from Vertica and 16 minutes per-iteration. Spark, on the other hand, needs 11 minutes to load the data from HDFS and 21 minutes per-iteration. Overall, this means that Spark and Distributed R take almost the same time to run K-means end-to-end. We also measure the case when data resides as files in the local ext4 filesystem of each node, and Distributed R loads data directly from these files. `DR-disk` in the figure shows that it takes just 5 minutes to load all the data from files in ext4, about 2× faster than Spark on HDFS and 3× faster than loading via Vertica. These numbers indicate the higher overheads involved in extracting data from distributed filesystems and databases.

Summary. Our comparison with Spark shows that interfacing an external tool, such as R, with a database is not necessarily slow. With proper integration, R applications can read data from a database at comparable speeds to systems like Spark that are tightly integrated with HDFS, and read directly from the distributed file system.

8. RELATED WORK

There are many systems that provide the ability to perform machine learning analysis. We briefly survey the relevant systems below.

Machine learning infrastructure. Matlab [5], SAS [11], R [8], and others are traditional statistical tools used by data scientists for different types of analysis including machine learning and graph processing. Each of these tools has a wide user base, sometimes in the order of millions of customers. R, given it's open source nature, has become increasingly popular and has more than 6000 packages available. All of these tools are generally single-threaded

and known to have limitations on large datasets. Therefore, the common usage model is to install these tools on laptops and desktops, and perform analysis on sampled data to avoid hitting the data size limitations. Several recent efforts have extended these tools for parallel processing on a single machine(parallel Matlab [4], ScaleR [9]), distributed computing (Distributed R [26]), and even integration with Hadoop (SAS on Hadoop [12], RHadoop [10]).

The last few years has seen a surge in distributed computing infrastructure for machine learning and graph processing. Google's MapReduce pushed the scale of distributed computing to thousands of commodity, but unreliable, servers [18]. A prominent use of the MapReduce infrastructure is for graph analysis, such as calculating PageRank of the Web graph [15]. The Hadoop ecosystem, an open source version of MapReduce, added many machine learning and graph analysis algorithms under the Mahout library [1]. However, the MapReduce paradigm is inefficient for machine learning algorithms, since they are iterative in nature. There have been efforts to improve performance of iterative algorithms by using techniques such as caching (HaLoop [16], Twister [19]). Another line of research has abandoned the MapReduce interface completely for specialization, such as a vertex centric programming for graph processing (Pregel [24], GraphLab [20]). Spark generalizes the programming paradigm of MapReduce by supporting LINQ like API which include aggregation and filter functions [28]. Spark improves over the performance of other systems by using main memory for computations, avoiding disk accesses, but still guaranteeing fault-tolerance. Spark's machine learning and SQL interface are an order of magnitude faster than similar systems on Hadoop.

Many of the techniques in this paper, such as extracting data from databases, are independent of the choice of the external analytics tool. For example, one could use the mechanisms in this paper to integrate Vertica with Spark instead of Distributed R. The integration of Vertica and Distributed R ensures that customers can continue to use the popular tool R along with industrial strength SQL by Vertica, and get competitive performance.

Databases and machine learning. Database vendors use three main approaches to support machine learning analysis (1) provide ways to extract data from database to R like tools, (2) support R within the database, and (3) implement distributed algorithms in the database. We contrast these approaches below.

ODBC based connectors are commonly used to extract data from databases. As mentioned in this paper, ODBC connections are slow and do not meet the performance requirements on large data. Solutions such as Teradata Parallel Transporter (TPT) FastExport supports parallel data transfer from Teradata to a client [13]. It uses multiple sessions to extract partitioned table data, and is similar to using multiple ODBC connections. Unlike mechanisms expressed in this paper, TPT does not transfer data to the memory of remote analytics engines, nor does it support different transfer policies such as partitioning data for load balance or data locality. RICE shows how to bridge SAP HANA with single-threaded R [21]. The key idea is to use shared memory on the node to convert database tables to R data-frames with low overhead. The data format conversion techniques in RICE are complementary to Vertica's fast transfer mechanism. As an example, Distributed R uses shared memory to not only stage data arriving from Vertica but also to share data across multiple R processes. However, unlike RICE, Vertica also focuses on reducing the overhead of transferring data in parallel to multiple R instances, and supports different data distribution policies.

Many prominent database vendors such as Oracle, Vertica, and others embed R in the database. This approach provides the ability to call R with user defined functions (UDF). Unfortunately, such an approach is limited by the single threaded implementation of the algorithm in the UDF. As an example, a customer can call single threaded R K-means by funneling a table through a single R UDF, but they cannot create a distributed K-means function by simple invoking multiple K-means UDFs. Ricardo is an example system that combines R and Hadoop [17]. Ricardo performs coarse grained decomposition by delegating large scale aggregation queries to Hadoop and small-scale statistical analysis to single-threaded R. Vertica has similar goals of delegating pre-processing and model deployment to the database, but provides new mechanisms for integrating with a distributed system such as Distributed R.

Finally, MADlib exemplifies the approach of expressing distributed, in-database, machine learning algorithms via user-defined functions, and SQL statements [22]. SciDB goes a step further by providing support for arrays in the database instead of retrofitting them on relational tables [25]. MADlib like approaches can leverage the SQL query optimizer and reduce data movement overhead. However, to contribute algorithms one needs to follow the programming paradigm proposed by MADlib. Vertica's integration with Distributed R is an orthogonal approach. It leverages the strengths of both relational databases and R. Our approach uses the database for model deployment, but does not rule out implementing distributed algorithms inside the database (such as pattern mining in Vertica [2]). However, by integrating Vertica with Distributed R, Vertica customers get the familiarity and power of R. Additionally, since most data scientists use R like tools to implement their analysis, one expects R based package contributions to grow rapidly, and these new algorithms will become automatically available to Vertica customers.

9. CONCLUSIONS

By integrating Vertica with Distributed R, we have not only expanded the functionality of Vertica but also solved multiple performance issues that customers face when using R with a database. This paper shows how customers can now use Vertica's industrial strength SQL engine for preprocessing data, deploying machine learning models, and applying predictions on database tables. Our new data transfer mechanism and parallel algorithms in Distributed R improve the end-to-end execution time of workflows by an order of magnitude compared to using R on a database.

10. ACKNOWLEDGMENTS

We are thankful to Sunil Venkayala, Dan Huang, Hua Zhang, Elena Chan, and Kyungyong Lee for their contributions to the HP Distributed R product. We are grateful to the reviewers, especially our shepherd Mehul Shah, for their valuable feedback and help in improving this paper.

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] HP Vertica Distributed R. https://www.vertica.com/distributedR.

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

[5] MATLAB–the language of technical computing. http://www.mathworks.com/products/matlab.

[6] Process containers. http://lwn.net/Articles/236038/.

[7] Programmatic media buying. http://rocketfuel.com/.

[8] The R project for statistical computing. http://www.r-project.org.

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

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

[11] SAS analytics. http://www.sas.com.

[12] SAS in-memory statistics for Hadoop. http://www.sas.com/en_us/software/sas-hadoop/in-memory-hadoop.html.

[13] Teradata parallel transporter. http://www.teradata.com/tools-and-utilities/parallel-transporter/.

[14] Teradata Warehouse Miner. http://www.teradata.com/products-and-services/teradata-warehouse-miner/.

[15] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *WWW7*, 1998.

[16] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1–2), 2010.

[17] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD*, 2010.

[18] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[19] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *HPDC*, 2010.

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[21] 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.

[22] 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. *PVLDB*, 5(12), 2012.

[23] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *PVLDB*, 5(12), 2012.

[24] 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 *SIGMOD*, 2010.

[25] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A storage manager for complex parallel array processing. In *SIGMOD*, 2011.

[26] 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.

[27] T. J. Ypma. Historical development of the Newton-Raphson method. *SIAM Rev.*, 37(4), 1995.

[28] 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 *NSDI*, 2012.