Ricardo: Integrating R and Hadoop

Sudipto Das[1*] Yannis Sismanis[2] Kevin S. Beyer[2]
Rainer Gemulla[2] Peter J. Haas[2] John McPherson[2]

[1]University of California
Santa Barbara, CA, USA
sudipto@cs.ucsb.edu

[2]IBM Almaden Research Center
San Jose, CA, USA
{syannis, kbeyer, rgemull, phaas, jmcphers}@us.ibm.com

ABSTRACT

Many modern enterprises are collecting data at the most detailed level possible, creating data repositories ranging from terabytes to petabytes in size. The ability to apply sophisticated statistical analysis methods to this data is becoming essential for marketplace competitiveness. This need to perform deep analysis over huge data repositories poses a significant challenge to existing statistical software and data management systems. On the one hand, statistical software provides rich functionality for data analysis and modeling, but can handle only limited amounts of data; e.g., popular packages like R and SPSS operate entirely in main memory. On the other hand, data management systems—such as MapReduce-based systems—can scale to petabytes of data, but provide insufficient analytical functionality. We report our experiences in building Ricardo, a scalable platform for deep analytics. Ricardo is part of the eXtreme Analytics Platform (XAP) project at the IBM Almaden Research Center, and rests on a decomposition of data-analysis algorithms into parts executed by the R statistical analysis system and parts handled by the Hadoop data management system. This decomposition attempts to minimize the transfer of data across system boundaries. Ricardo contrasts with previous approaches, which try to get along with only one type of system, and allows analysts to work on huge datasets from within a popular, well supported, and powerful analysis environment. Because our approach avoids the need to re-implement either statistical or data-management functionality, it can be used to solve complex problems right now.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Design

*The author conducted parts of this work at the IBM Almaden Research Center.

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

1. INTRODUCTION

Many of today's enterprises collect data at the most detailed level possible, thereby creating data repositories ranging from terabytes to petabytes in size. The knowledge buried in these enormous datasets is invaluable for understanding and boosting business performance. The ability to apply sophisticated statistical analysis methods to this data can provide a significant competitive edge in the marketplace. For example, internet companies such as Amazon or Netflix provide personalized recommendations of products to their customers, incorporating information about individual preferences. These recommendations increase customer satisfaction and thus play an important role in building, maintaining, and expanding a loyal customer base. Similarly, applications like internet search and ranking, fraud detection, risk assessment, microtargeting, and ad placement gain significantly from fine-grained analytics at the level of individual entities. This paper is about the development of industrial-strength systems that support advanced statistical analysis over huge amounts of data.

The workflow for a data analyst comprises multiple activities. Typically, the analyst first explores the data of interest, usually via visualization, sampling, and aggregation of the data into summary statistics. Based on this exploratory analysis, a model is built. The output of the model is itself explored—often through visualization and also through more formal validation procedures—to determine model adequacy. Multiple iterations of model-building and evaluation may be needed before the analyst is satisfied. The final model is then used to improve business practices or support decision making. Feedback from model users can lead to further iterations of the model-development cycle.

During this process, the data analyst's indispensable toolkit is a statistical software package such as R, SPSS, SAS, or Matlab. Each of these packages provides a comprehensive environment for statistical computation, including a concise statistical language, well tested libraries of statistical algorithms for data exploration and modeling, and visualization facilities. We focus on the highly popular R statistical analysis program. The Comprehensive R Archive Network (CRAN) contains a library of roughly 2000 add-in packages developed by leading experts and covering areas such as linear and generalized linear models, nonlinear regression models, time series analysis, resampling methods, classical parametric and nonparametric tests, classification, clustering, data smoothing, and many more [13]. We refer to the application of these sophisticated statistical methods as *deep analytics*.

Most statistical software packages, including R, are designed to target the moderately-sized datasets commonly found in other areas of statistical practice (e.g., opinion polls). These systems operate on a single server and entirely in main memory; they simply fail when the data becomes too large. Unfortunately, this means that

data analysts are unable to work with these packages on massive datasets. Practitioners try to avoid this shortcoming either by exploiting vertical scalability—that is, using the most powerful machine available—or by working on only subsets or samples of the data. Both approaches have severe limitations: vertical scalability is inherently limited and expensive, and sampling methods may lose important features of individuals and of the tail of the data distribution [9].

In parallel to the development of statistical software packages, the database community has developed a variety of large-scale data management systems (DMSs) that can handle huge amounts of data. Examples include traditional enterprise data warehouses and newer systems based on MapReduce [11], such as Hadoop. The data is queried using high-level declarative languages such as SQL, Jaql, Pig, or Hive [14, 19, 23]. These systems leverage decades of research and development in distributed data management, and excel in massively parallel processing, scalability, and fault tolerance. In terms of analytics, however, such systems have been limited primarily to *aggregation processing*, i.e., computation of simple aggregates such as SUM, COUNT, and AVERAGE, after using filtering, joining, and grouping operations to prepare the data for the aggregation step. For example, traditional reporting applications arrange high-level aggregates in cross tabs according to a set of hierarchies and dimensions. Although most DMSs provide hooks for user-defined functions and procedures, they do not deliver the rich analytic functionality found in statistical packages.

To summarize, statistical software is geared towards deep analytics, but does not scale to large datasets, whereas DMSs scale to large datasets, but have limited analytical functionality. Enterprises, which increasingly need analytics that are both deep and scalable, have recently spurred a great deal of research on this problem; see Section 6. Indeed, the work in this paper was strongly motivated by an ongoing research collaboration with Visa to explore approaches to integrating the functionality of R and Hadoop [10]. As another example, Cohen at al. [9] describe how the Fox Audience Network is using statistical functionality built inside a large-scale DMS. As discussed in Section 6, virtually all prior work attempts to get along with only one type of system, either adding large-scale data management capability to statistical packages or adding statistical functionality to DMSs. This approach leads to solutions that are often cumbersome, unfriendly to analysts, or wasteful in that a great deal of well established technology is needlessly re-invented or re-implemented.

In this paper, we report our experience in building Ricardo, a scalable platform for deep analytics. Ricardo—which is part of the eXtreme Analytics Platform (XAP) project at the IBM Almaden Research Center—is named after David Ricardo, a famous economist of the early 19th century who studied conditions under which mutual trade is advantageous. Ricardo facilitates "trading" between R and Hadoop, with R sending aggregation-processing queries to Hadoop (written in the high-level Jaql query language), and Hadoop sending aggregated data to R for advanced statistical processing or visualization—each trading partner performs the tasks that it does best. In contrast to previous approaches, Ricardo has the following advantages:

- *Familiar working environment.* Analysts want to work within a statistical environment, and Ricardo lets them continue to do so.

- *Data attraction.* Ricardo uses Hadoop's flexible data store together with the Jaql query language. This combination allows analysts to work directly on any dataset in any format; this property of "attracting" data of any type has been identified as a key requirement for competitive enterprise analytics [9].

- *Integration of data processing into the analytical workflow.* Analysts traditionally handle large data by preprocessing and reducing it—using either a DMS or shell scripts—and then manually loading the result into a statistical package, e.g., as delimited text files. By using an integrated approach to data processing, Ricardo frees data analysts from this tedious and error-prone process, and allows them to leverage all available data.

- *Reliability and community support.* Ricardo is built from widely adopted open-source projects from both the data management community and the statistical community. It leverages the efforts of both communities and has a reliable, well supported, and state-of-the-art code base.

- *Improved user code.* Ricardo facilitates more concise, more readable, and more maintainable code than is possible with previous approaches.

- *Deep analytics.* By exploiting R's functionality, Ricardo can handle many kinds of advanced statistical analyses, including principal and independent component analysis, k-means clustering, and SVM classification, as well as the fitting and application of generalized-linear, latent-factor, Bayesian, time-series, and many other kinds of statistical models.

- *No re-inventing of wheels.* By combining existing statistical and DMS technology, each of which represents decades of research and development, we can immediately start to solve many deep analytical problems encountered in practice.

Ricardo is inspired by the work in [8], which shows that many deep analytical problems can be decomposed into a "small-data part" and a "large-data part." In Ricardo, the small-data part is executed in R and the large-data part is executed in the Hadoop/Jaql DMS. A key requirement for the success of this combined approach is that the amount of data that must be communicated between both systems be sufficiently small. Fortunately, this requirement holds for almost all of the deep analytics mentioned above.

In the following sections, we show how Ricardo can facilitate some key tasks in an analyst's typical workflow: data exploration, model building, and model evaluation, all over a very large dataset. For illustrative purposes, we use the dataset provided for the Netflix movie-recommendation competition [16]. Although the competition itself was based on a subset of just 100M movie ratings, our experiments on a Hadoop cluster in the Amazon Elastic Compute Cloud (EC2) indicate that Ricardo can scale R's functionality to handle the billions of ratings found in practice—over a terabyte of data in our case.

We emphasize that an analyst who uses Ricardo need not necessarily be an expert in Jaql nor understand exactly how to decompose all the deep analytics appropriately. Ricardo can potentially deliver much of its deep-analytics functionality in the form of R packages and functions that hide most of the messy implementation details. For example, we describe in the sequel how Ricardo can be used to efficiently fit a latent-factor model of movie preferences over a massive dataset stored in a Hadoop cluster, as well as how Ricardo can be used for large-scale versions of principal component analysis (PCA) and generalized linear models (GLMs). This functionality can potentially be delivered to the analyst via high-level R functions called, e.g., `jaqlLF`, `jaqlPCA`, and `jaqlGLM`. Of course, if existing Ricardo packages do not meet the needs of a particular

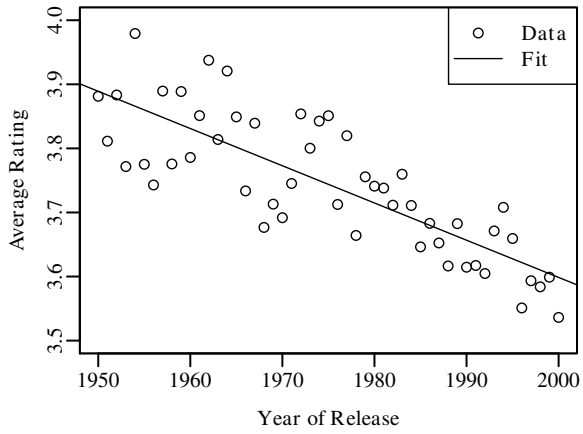

Figure 1: Average rating of a movie depending on its age

analysis, then the analyst will have to implement the lower-level functionality. The hope is that, over time, users will develop a large library of Ricardo packages in the same way that the CRAN repository has been developed for in-memory analytical packages. The XAP project team is trying to kick off this effort by providing packages for the most common types of large-scale analyses.

The remainder of this paper is structured as follows. In Section 2, we drill down into key aspects of a typical analytics workflow, using movie recommendation data as a running example. Section 3 briefly describes the systems of interest: the R package for statistical computing, the Hadoop DMS, and the Jaql query language. In Section 4, we describe Ricardo and illustrate its use on several analysis tasks. Section 5 describes our experimental study on the Netflix data. We discuss prior work in Section 6 and give our conclusions in Section 7.

2. MOTIVATING EXAMPLES

We motivate Ricardo in the context of three examples that cover key aspects of the analyst's workflow. The examples are centered around the Netflix competition [5]. This competition was established in 2006 in order to improve the recommender system that Netflix uses to suggest movies to its ten million customers. Since recommendations play an important role in establishing, maintaining and expanding a loyal customer base, such systems have become the backbone of many of the major firms on the web such as Amazon, eBay, and Apple's iTunes [15, 25]. Unlike search engines, which help find information that we know we are interested in, recommender systems help discover people, places, and things previously unknown to us. We classify the examples by the degree of complexity in the trading between R and Hadoop.

2.1 Simple Trading

Our first two examples concern the exploration and evaluation phases of the analyst's workflow. During the exploration phase, an analyst tries to gain preliminary insights about the dataset of interest. For example, Figure 1 depicts how movies are perceived with respect to their age. We can see that, on average, older movies are perceived more positively than newer movies. To produce this simple data visualization, the analyst first performs a linear regression and then calls upon a plotting facility to display the raw data and fitted model. The analyst might also want to formally verify that the fitted trend is statistically significant by looking at summary test statistics such as the t-statistic for the slope. Statistical software such as R offers a convenient and powerful environment for performing this type of exploration. However, such software cannot scale to the size of the datasets found in practice. Although, for the competition, Netflix published only a small subset of its ratings (100M), the actual number of (explicit and implicit) ratings encountered in practice is orders of magnitude larger. Ricardo allows such data to be efficiently preprocessed, aggregated, and reduced by Hadoop, and then passed to R for regression analysis and visualization.

During the evaluation phase the analyst wants to understand and quantify the quality of a trained model. In our second example, we assume that the analyst has built a model—such as a latent-factor recommendation model as described in the sequel—and wants to identify the top-k outliers, i.e., to identify data items on which the model has performed most poorly. Such an analysis, which must be performed over all of the data, might lead to the inclusion of additional explanatory variables (such as movie age) into the model, to improve the model's accuracy. Ricardo enables such outlier analysis by allowing the application of complex R-based statistical models over massive data. It does so by leveraging the parallelism of the underlying Hadoop DMS.

The foregoing examples illustrate "simple trading" scenarios between R and Hadoop. In the first case, data is aggregated and processed before it is passed to R for advanced analysis; in the second case, an R statistical model is passed to Hadoop for efficient parallel evaluation over the massive dataset. As discussed below, other analyses require more intricate exchanges, which Ricardo also supports.

2.2 Complex Trading

Our third example illustrates the use of Ricardo during the modeling phase of an analyst's workflow. One approach to model building over a huge dataset might use a simple-trading scheme in which Hadoop reduces the data by aggregation or sampling, and then passes the data to R for the model-building step. The downside of this approach is that detailed information, which must be exploited to gain a competitive edge, is lost. As shown below, Ricardo permits a novel "complex trading" approach that avoids such information loss.

The complex-trading approach, like simple trading, requires a decomposition of the modeling into a small-data part, which R handles, and a large-data part, which Hadoop handles—as mentioned in Section 1, many deep analytical problems are amenable to such a decomposition. Unlike simple trading, however, a complex-trading algorithm involves multiple iterations over the data set, with trading back and forth between R and Hadoop occurring at each iteration.

As an instructive example, we consider the problem of building a latent-factor model of movie preferences over massive Netflix-like data. This modeling task is central to the winning Netflix competition technique [18], and enables accurate personalized recommendations for each individual user and movie, rather than global recommendations based on coarse customer and movie segments. Because such a model must discern each customer's preferences from a relatively small amount of information on that customer, it is clear that every piece of available data must be taken into account. As with many other deep analytics problems, the sampling and aggregation used in a simple-trading approach is unacceptable in this setting.

To understand the idea behind latent-factor models, consider the data depicted in Figure 2, which shows the ratings of three customers and three movies in matrix form. The ratings are printed in boldface and vary between 1 (hated the movie) and 5 (loved it). For example, Michael gave a rating of 5 to the movie "About Schmidt"

	About Schmidt (*2.24*)	Lost in Translation (*1.92*)	Sideways (*1.18*)
Alice (*1.98*)	? (*4.4*)	**4** (*3.8*)	**2** (*2.3*)
Bob (*1.21*)	**3** (*2.7*)	**2** (*2.3*)	? (*1.4*)
Michael (*2.30*)	**5** (*5.2*)	? (*4.4*)	**3** (*2.7*)

Figure 2: A simple latent-factor model for predicting movie ratings. (Data points in boldface, latent factors and estimated ratings in italics.)

and a rating of 3 to "Sideways". In general, the ratings matrix is very sparse; most customers have rated only a small set of movies. The italicized number below each customer and movie name is a latent factor. In this example, there is just one factor per entity: Michael is assigned factor 2.30, the movie "About Schmidt" gets 2.24. In this simple example, the estimated rating for a particular customer and movie is given by the product of the corresponding customer and movie factors.[1] For example, Michael's rating of "About Schmidt" is approximated by $2.30 \cdot 2.24 \approx 5.2$; the approximation is printed in italic face below the respective rating. The main purpose of the latent factors, however, is to *predict* ratings, via the same mechanism. Our estimate for Michael's rating of "Lost in Translation" is $2.30 \cdot 1.92 \approx 4.4$. Thus, our recommender system would suggest this movie to Michael but, in contrast, would avoid suggesting "Sideways" to Bob, because the predicted rating is $1.21 \cdot 1.18 \approx 1.4$.

Latent factors characterize the interaction between movies and customers and sometimes have obvious interpretations. For example, a movie factor might indicate the degree to which a movie is a "comedy" and a corresponding customer factor might indicate the degree to which the customer likes comedies. Usually, however, such an interpretation cannot be given, and the latent factors capture correlations between customers and movies without invoking domain-specific knowledge. Under the assumption that the preferences of individual customers remain constant, the more available feedback, the better the modeling power of a latent-factor model [15, 17]. Netflix and other companies typically collect billions of ratings; taking implicit feedback such as navigation and purchase history into account, the amount of data subject to recommendation analysis is enormous.

In Section 4 below, we discuss in detail how Ricardo handles the foregoing exploration, evaluation, and model-building example tasks that we have defined, and we indicate how Ricardo can be applied to other deep analytics tasks over massive data.

3. PRELIMINARIES

In this section, we briefly describe the main components of Ricardo: the R statistical software package, the Hadoop large-scale DMS, and the Jaql query language.

[1]In an actual recommender system, there is a vector of latent factors associated with each customer and movie, and the estimated rating is given by the dot product of the corresponding vectors.

3.1 The R Project for Statistical Computing

R was originally developed by Ross Ihaka and Robert Gentleman, who were at that time working at the statistics department of the University of Auckland, New Zealand. R provides both an open-source language and an interactive environment for statistical computation and graphics. The core of R is still maintained by a relatively small set of individuals, but R's enormous popularity derives from the thousands of sophisticated add-on packages developed by hundreds of statistical experts and available through CRAN. Large enterprises such as AT&T and Google have been supporting R, and companies such as REvolution Computing sell versions of R for commercial environments.

As a simple example of R's extensive functionality, consider the following small program that performs the regression analysis and data visualization shown previously in Figure 1. It takes as input a "data frame" `df` that contains the mean ratings of the movies in the Netflix dataset by year of publication, performs a linear regression, and plots the result:

```
fit <- lm(df$mean ~ df$year)
plot(df$year, df$mean)
abline(fit)
```

(A data frame is R's equivalent of a relational table.) Older movies appear to be rated more highly than recent ones; a call to the R function `summary(fit)` would validate that this linear trend is indeed statistically significant by computing assorted test statistics.

3.2 Large-Scale Data Management Systems

Historically, enterprise data warehouses have been the dominant type of large-scale DMS, scaling to hundreds of terabytes of data. These systems are usually interfaced by SQL, a declarative language for processing structured data. Such systems provide extensive support for aggregation processing as defined in Section 1. Implementations of enterprise data warehouses benefit from decades of experience in parallel SQL processing and, as indicated by the recent surge of new vendors of "analytical databases," continue to provide innovative data management solutions.

Besides being very expensive, these systems suffer from the disadvantage of being primarily designed for clean and structured data, which comprises an increasingly tiny fraction of the world's data. Analysts want to be able to work with dirty, semi-structured or even unstructured data without going through the laborious cleansing process needed for data warehousing [9]. For these situations, MapReduce and its open-source implementation Apache Hadoop have become popular and widespread solutions.

Hadoop comprises a distributed file system called HDFS bundled with an implementation of Google's MapReduce paradigm [11]. Hadoop operates directly on raw data files; HDFS takes care of the distribution and replication of the files across the nodes in the Hadoop cluster. Data processing is performed according to the MapReduce paradigm. The input files are split into smaller chunks, each of which is processed in parallel using a user-defined *map* function. The results of the map phase are redistributed (according to a user-defined criterion) and then fed into a *reduce* function, which combines the map outputs into a global result. Hadoop has been successfully used on petabyte-scale datasets and thousands of nodes; it provides superior scalability, elasticity and fault-tolerance properties on large clusters of commodity machines. Hadoop is an appealing alternative platform for massive data storage, manipulation and parallel processing.

3.3 Jaql: A JSON Query Language

Perhaps the main drawback of Hadoop is that its programming interface is too low-level for most of its users. For this reason, a variety of projects aim at providing higher-level query interfaces on top of Hadoop; notable examples include Jaql [14], Pig [19], Hive [23] and Cascading[2]. Ricardo uses Jaql as its declarative interface to Hadoop. Jaql is an open-source dataflow language with rich data processing features such as transformation, filtering, join processing, grouping and aggregation. Jaql scripts are automatically compiled into MapReduce jobs, which are then executed in parallel by Hadoop. Since Jaql is built around the JSON data model and is highly extensible, it enhances Hadoop's usability while retaining all of Hadoop's flexibility.

Although Jaql operates directly on user data files, it makes use of JSON views to facilitate data processing. For example, consider the following JSON view over the Netflix dataset:

```
[
  {
    customer: "Michael",
    movie: { name: "About Schmidt", year: 2002 },
    rating: 5
  },
  ...
],
```

Square brackets denote arrays (all ratings) and curly braces enclose structured objects (individual rating or movie). The following Jaql query computes average ratings as a function of movie age; this data can then be "traded" to R, which can in turn process the data to produce Figure 1.

```
read("ratings")
-> group by year = $.movie.year
   into { year, mean: avg($[*].rating) }
-> sort by [ $.year ].
```

The operator -> pipes the output of the expression on the left hand side into the right hand side; the current record is accessed via reference to the special variable $. Jaql provides functionality that would be tedious to implement using the native Hadoop API.

4. TRADING WITH RICARDO

In this section, we describe Ricardo in more detail, using the examples of Section 2.

4.1 Architecture Overview

Figure 3 gives an overview of Ricardo's design. Ricardo consists of three components: an R driver process operated by the data analyst, a Hadoop cluster that hosts the data and runs Jaql (and possibly also some R sub-processes), and an R-Jaql bridge that connects these two components.

R serves as an interactive environment for the data analyst, but in contrast to classical R usage, the data itself is not memory-resident in R. Instead, the base data is stored in a distributed file system within the Hadoop cluster, which comprises a set of worker nodes that both store data and perform computation. As described previously, Jaql provides a high-level language to process and query the Hadoop data, compiling into a set of low-level MapReduce jobs.

The third, novel component of Ricardo is the R-Jaql bridge, which provides the communication and data conversion facilities needed to integrate these two different platforms. The bridge not only allows R to connect to a Hadoop cluster, execute Jaql queries, and receive the results of such queries as R data frames, but also allows Jaql queries to spawn R processes on Hadoop worker nodes. The bridge comprises an R package that provides integration of Jaql into R and a Jaql module that integrates R into Jaql. The latter module allows worker nodes to execute R scripts in parallel and R functions to be used inside Jaql queries. The key functions[3] provided by the R package are:

`jaqlConnect()`	Opens a channel connecting R to a Hadoop cluster.
`jaqlSave()`	Saves data from R to HDFS.
`jaqlValue()`	Returns to R arbitrarily complex JSON results from Jaql queries.
`jaqlTable()`	Like `jaqlValue()` but optimized for data frames (i.e. tabular data).
`jaqlExport()`	Exports R objects from the R driver process to R processes running on the worker nodes.

The components of Ricardo work together to support an analyst's workflow. The analyst typically starts by issuing Jaql queries from inside a top-level R process. These queries sample or aggregate the data to allow viewing and visualization of the data from within R. Next, the analyst builds models of the data. The building of these models—e.g., by training the model or fitting parameters—as well as the validation of their quality—e.g., by visualization, by cross validation, or by computing measures of fit or test statistics—is performed as a joint effort of R, Jaql, and the R-Jaql bridge. In the following subsections, we give specific examples of how Ricardo can be used in an analytic workflow.

4.2 Simple Trading

First recall the data-exploration example of Section 2.1. The following R script shows how the `jaqlTable()` function is used to push the aggregation of the data to the Hadoop cluster using Jaql. It computes the data frame `df` that holds the average ratings for movies of different ages; R then processes `df` using the code given in Section 3.1 to fit a regression model and produce Figure 1. As discussed in Section 3, R can use the `summary(fit)` command to determine the adequacy of the regression model.

```
ch <- jaqlConnect()
df <- jaqlTable(ch, '
  read("ratings")
  -> group by year = $.movie.year
     into { year, mean: avg($[*].rating) }
  -> sort by [ $.year ]
')
```

Now recall the second example of Section 2.1, in which the analyst wants to apply a trained model over all the data in order to compute the top-k outliers. The following R script shows how a trained model can be invoked in parallel using `jaqlExport()`. In this example, we use the linear model computed above.

```
mymodel <- function(x) {
  fit$coeff[2]*x[1] + fit$coeff[1]
}
jaqlExport(ch, fit)
jaqlExport(ch, mymodel)
res <- jaqlTable(ch, '
  read("ratings")
  -> transform { $.*,
```

[2] `http://www.cascading.org`

[3] At the time of this writing, the implementation of `jaqlExport` is still underway.

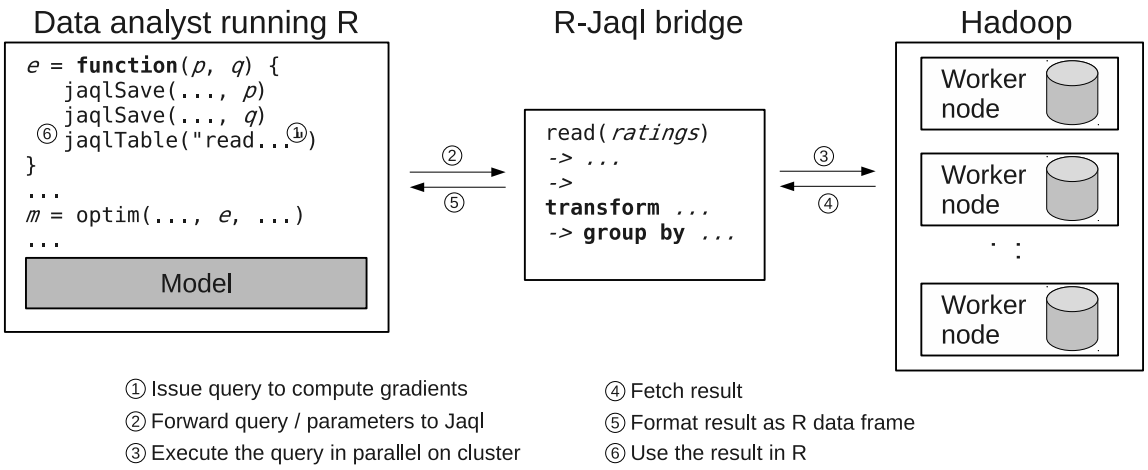

Figure 3: Overview of Ricardo's architecture.

```
        error: pow( R("mymodel", [$.movie.year])
                    - rating, 2) }
  -> top 10 by [ $.error desc ]
')
```

Here `mymodel()` is an R function that implements the linear model evaluation, outputting an estimated average rating corresponding an input movie age. The `jaqlExport()` invocations ensure that both the computed model `fit` and the function `mymodel()` are "known" by the R-processes that are spawned on the worker nodes. During execution of the Jaql query, the worker nodes, in parallel, each execute the `mymodel()` function and compute the corresponding squared errors. The errors are sorted in descending order (using the MapReduce framework), and the top-k outliers are returned back to R as a data frame. Note that the function `mymodel()` is simple enough to be implemented entirely in Jaql, without requiring R to evaluate it. In practice, however, this is seldom the case. See, for example, the time-series models discussed in Section 4.4.

4.3 Complex Trading

In this section, we describe in detail how Ricardo can be used for building statistical models over massive data when simple trading approaches do not suffice. We first illustrate our approach using the latent-factor model mentioned in Section 2.2, and then in Section 4.4 we briefly indicate other types of analysis that can be handled by Ricardo.

Recall that complex-trading scenarios arise when the model of interest must incorporate all of the available (massive) data. Because data reduction prior to modeling is not allowed, the data will not fit in main memory. Ricardo does require that the model itself—e.g., the set of latent factors in our Netflix example—fit into the main memory of the analyst's R process. This assumption holds for all but extremely large models (such as those in PageRank algorithms). In the case of Netflix, the latent factors occupy on the order of tens to hundreds of megabytes of memory, even after scaling up the numbers of customers and movies used in the competition by several orders of magnitude. A model of this size can easily fit in main memory. In contrast, the base data on customers and movies ranges from hundreds of gigabytes to terabytes and is thus significantly larger than the model. Ricardo places no restrictions on the base-data size, as long as the Hadoop cluster is equipped with a sufficient number of worker nodes to process the data.

4.3.1 Computing a Latent-Factor Model

Recall from Section 2 that the latent-factor model for movie recommendations assigns a set of latent factors to each customer and each movie. These factors are used to predict ratings that are unknown; movies with high predicted ratings can then be recommended to the customer. This general scheme also applies to other recommendation problems. In general, our goal is to recommend *items* (movies) to *users* (customers) based on known relationships between items and users.

Indexes u and i denote users and items. Denote by p_u the latent factor associated with user u and by q_i the latent factor for item i. Once learned, these factors allow us to compute a predicted rating $\hat{r}_{ui}$ of u for i via a simple product:

$$\hat{r}_{ui} = p_u q_i. \tag{1}$$

The value of $\hat{r}_{ui}$ predicts how much user u will like (or dislike) item i. As mentioned previously, this latent-factor model is highly simplified for aid in understanding Ricardo; an industrial-strength recommender system would (1) maintain a vector of factors for each user and items, (2) use additional information about the user and item (such as demographic information), and (3) would account for shifts of preferences over time [17]. All of these extensions can be handled by Ricardo.

Recent work [4, 7, 16] has suggested that the latent factors be learned via an optimization-based approach that tries to minimize the sum of squared errors between the observed ratings and the corresponding predicted ratings from the model. Thus we want to solve the problem

$$\min_{\{p_u\},\{q_i\}} \sum_{u,i} (r_{ui} - p_u q_i)^2, \tag{2}$$

where the sum is taken over the set of observed ratings.[4]

The above optimization problem is large and sparse, and has to be solved using numerical methods. The general algorithm is as follows:

1. Pick (random) starting points for the values of p_u and q_i.

[4]In fact, (2) has been simplified; our actual implementation makes use of "regularization" techniques to avoid overfitting. For Netflix data, a simple weight decay scheme—which amounts to adding a penalty term to (2)—has proven successful [16].

2. Compute the squared error and the corresponding gradients with respect to each p_u and q_i, using the current values of p_u and q_i.

3. Update the values of each p_u and q_i according to some update formula.

4. Repeat steps 2 and 3 until convergence.

The idea behind this "gradient descent" algorithm is to use the gradient information to try and reduce the value of the squared error as much as possible at each step. Observe that step 2 corresponds to a summation over the entire dataset and is thus data-intensive, while step 3 requires the use of a sophisticated optimization algorithm that is able to deal with a large number of parameters.

4.3.2 The R Component

We next describe how R contributes to the model-fitting procedure described in Section 4.3.1 by using its sophisticated optimization algorithms. We assume the existence of two functions `e` and `de`, which compute the squared error and the corresponding gradients for specific values of the latent factors. Both functions take a single argument `pq`, which holds the concatenation of the latent factors for users and items. (The description of how we implement these functions is deferred to Sections 4.3.3 and 4.3.4.) R can then use its `optim` function to drive the overall optimization process. The `optim` function operates by using the current squared error value and gradient computed by `e` and `de`. Then `optim` uses this information to update the `pq` values in such a manner as to reduce the squared error as much as possible. This process is repeated until the squared error no longer decreases significantly. The updating process can be viewed as taking a "downhill step" in the space of possible `pq` values. Both the direction and size of the step are determined by the specific optimization method that is chosen, and can be quite complex. As discussed below, R has a number of optimization methods to choose from.

The following snippet contains all of the high-level R code that is required to run the optimization:

```
model <- optim( pq, e, de,
                method="L-BFGS-B",
                ... )
```

where `pq` is the starting point and `model` is the resulting solution. The first three arguments thus describe the input to the optimization problem. The `method` argument determines the optimization method to be used. The `optim` function also takes a number of additional arguments that allow fine-grained control of the optimization algorithm, such as setting error tolerances and stopping criteria or bounding the number of iterations.

Some popular optimization algorithms provided by R include the conjugate-gradient method [3] and L-BFGS [6]. Optimization is a complex field and there are many different methods, each with its own strengths and weaknesses. We experimented with many different optimization methods and we found that the L-BFGS method is very effective. L-BFGS is an optimization technique for solving large non-linear optimization problems; it implements a limited memory quasi-Newtonian optimization algorithm based on the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method. The details of this method are beyond the scope of this paper; in essence, it makes use of the values of `e` and `de` to learn the "curvature" of the objective function (the squared error in our case) by maintaining a low-memory approximation of the inverse of the Hessian Matrix. This information is then used to estimate the best direction and size for the next optimization step.

4.3.3 The Hadoop and Jaql Component

In Ricardo, Hadoop and Jaql are responsible for data storage and large-scale aggregation processing. For the latent-factor example, this latter processing corresponds to the computation of `e` and `de` at each step of the R-driven optimization algorithm. For the Netflix data, the ratings are stored in a table `r` and the latent factors are stored in tables `p` and `q`. For simplicity, we assume that the following JSON views have been defined over these datasets:

```
r: schema { u: long, i: long, rating: double }
p: schema { u: long, factor: double }
q: schema { i: long, factor: double }
```

Each record in the ratings table corresponds to a single rating by user u of item i. Each record in p or q corresponds to a latent factor.

We start with a simple "naive" Jaql implementation of the `e` and `de` functions and then discuss some potential optimizations. Our goal is to compute the following quantities:

$$e = \sum_{u,i} (r_{ui} - p_u q_i)^2,$$

$$\frac{\partial e}{\partial p_u} = -2 \sum_{i} (r_{ui} - p_u q_i) q_i,$$

$$\frac{\partial e}{\partial q_i} = -2 \sum_{u} (r_{ui} - p_u q_i) p_u,$$

where, as before, the sums are taken over the observed ratings. For the computation of the squared error e, we have to bring together the rating from the `r` table and the corresponding factors from the `p` and `q` tables. The following Jaql query does the trick:

```
join r, p, q
   where r.u == p.u and r.i == q.i
   into { p_u: p.factor, q_i: q.factor, r_ui: r.rating }
-> transform { error: pow( $.r_ui - $.p_u*$.q_i, 2 )}
-> sum( $.error );
```

This query first performs a three-way join between `r`, `p`, and `q`, then computes the squared error for each of the observed ratings, and finally accumulates all of these squared errors. The computation of the gradients is performed in a similar way, but requires an additional grouping step. The following query computes the derivatives with respect to the user factors:

```
join r, p, q
   where r.u == p.u and r.i == q.i
   into { u: r.u, r_ui: r.rating,
          p_u: p.factor, q_i: q.factor }
-> transform { u: $.u,
               gradient: ($.r_ui-$.p_u*$.q_i)*$.q_i }
-> group by u = $.u
   into { u, gradient: -2.0*sum($[*].gradient)};
```

The query returns a table containing pairs of users `u` and the corresponding gradient $\partial e / \partial p_u$. The query for the gradients with respect to the movie factors is similar.

In our actual implementation, we used an optimized query that computes the error and gradients simultaneously. This has the advantage that the number of MapReduce jobs generated by Jaql—and thus the number of passes over the data—is reduced from three to one. The optimized query is given by:

```
r -> hashJoin( fn(r) r.i, q, fn(q) q.i,
               fn(r,q) { r.*, q_i: q.factor } )
  -> hashJoin( fn(r) r.u, p, fn(p) p.u,
               fn(r,p) { r.*, p_u: p.factor } )
  -> transform { $.*, diff: $.rating-$.p_u*$.q_i }
```

```
-> expand [ { value: pow( $.diff, 2.0 ) },
            { $.u, value: -2.0*$.diff*$.q_i },
            { $.i, value: -2.0*$.diff*$.p_u } ]
-> group by g = { $.u, $.i }
   into { g.u?, g.i?, total: sum($[*].value) }
```

This query, though compact in terms of lines of Jaql code, is rather subtle in the way it exploits the features of the underlying map-combine-reduce framework. To avoid going into onerous detail, we simply mention that `hashJoin` is an efficient function that includes in its argument list a key-extractor function for the probe side, a dataset for the build side, a key extractor for the build side, and a function that pairs matching tuples of the join. The first hash join joins `r` and `p`, and the second hash join takes the result and joins it with `q`. The remainder of the query performs all of the aggregations at once. It works with three different kind of records: (1) records with only a `u` field, (2) records with only an `i` field, and (3) records with neither `u` nor `i`. The records are distinguished using the existence operator (denoted as a question mark in the query above). During the aggregation, records of type (1) contribute to the gradients of user factors, records of type (2) to the gradients of item factors, and records of type (3) to the overall squared error. The execution plan resembles the execution of a "grouping-sets" SQL query. We expect future versions of Jaql to automatically perform some of the above optimizations.

4.3.4 Integrating the Components

In what follows, we show how to integrate the "large-data" computation of Section 4.3.3 with the "small-data" computation of Section 4.3.2, using the bridge. In particular, the naive version of the squared-error function `e` that is passed to R's optimization function `optim` has the following form:

```
e <- function(pq) {
  jaqlSave(ch, pq[1:np], p);
  jaqlSave(ch, pq[(np+1):(np+nq)], q);
  jaqlValue(ch, "<query>");
}
```

The argument `pq` given to the error function is a concatenation of the current customer and movie factors; it is provided to `e` by `optim`. The first two lines of the function body split `pq` into its components and then save the result in Jaql via a call to `jaqlSave`. Here, variables `p` and `q` point to locations in the Hadoop cluster. The final line in the function body then executes the Jaql query that performs the actual computation and returns the result as a floating point value—here `"<query>"` is an abbreviation for the naive Jaql code for computing `e` that was given in Section 4.3.3. The naive function `de` for computing the gradients looks similar, but the call to `jaqlValue()` is replaced by `jaqlTable()`, since multiple numbers are returned. The result is automatically converted into an R data frame. All functions provided by the bridge automatically perform any necessary conversions of data types.

In our implementation we use the optimized Jaql query (described in Section 4.3.3) to compute the squared error and the corresponding gradients concurrently. Thus, the implementation of `e` and `de` must be adjusted slightly. For brevity we omit the details.

4.4 Other Deep Analytics

In the previous sections, we provided examples of how Ricardo enhances the analyst's workflow by scaling deep analytics to massive datasets, exploiting Hadoops parallelism while staying in the analyst-friendly environment that R provides. We discussed how Ricardo handles "simple trading" and "complex trading" scenarios during the key phases of an analyst's workflow.

For purposes of exposition, all of our examples so far have been in the setting of the Netflix dataset, so that our focus has been a bit narrow. E.g., the latent-factor model, while certainly nontrivial and providing a good demonstration of the power of the Ricardo framework, is actually somewhat simple in terms of statistical sophistication. In this section we briefly outline how Ricardo can be used to perform a number of other, more complex analyses, by leveraging R's extensive library of statistical functions.

As one example, we experimented with R's time-series analysis functions and incorporated them in the recommender system. In this scenario, we took the time of the rating into account, since people often change preferences over time. (For simplicity, we take the "time" for a rating to be the total number of movies that the customer has rated up to and including the current movie.) The following R/Jaql code shows how one computes, in parallel, an "autoregressive integrated moving average" (ARIMA) time-series model for each individual customer:

```
my.arima <- function(x) {
  library(forecast)
  auto.arima(x[[1]][ sort.list(x[[2]]) ])
}
jaqlExport(ch, my.arima)
arima.models <- jaqlTable(ch, '
  r -> group by g={ $.u }
       into { g.u,
              model: R("my.arima", [
                       $[*].rating, $[*].date
                     ] ) }
')
```

Here `my.arima()` is an R function that first orders the ratings vector of a customer in increasing time order and then computes an ARIMA model by automatically fitting the model parameters. The embedded Jaql query groups the ratings by customer u and then constructs the corresponding rating and rating-date vectors in parallel. Other extensions of the recommender system are possible, such as incorporation of demographic and movie genres information. By exploiting R's machine learning library functions, one could easily train, in parallel, decision trees and self-organizing maps that capture important properties of demographic and movie genre segments.

We have also demonstrated that Ricardo can perform large-scale principal component analysis (PCA) and generalized linear models (GLM). For PCA, we use the covariance method: Hadoop/Jaql first shifts and scales the data by subtracting the empirical mean and dividing by the empirical standard deviation (after using Hadoop to compute these quantities), and then computes the empirical covariance matrix. Next, R performs an eigenvector decomposition of the covariance matrix, and finally Hadoop/Jaql projects the original data onto the lower-dimensional space spanned by the major eigenvectors.

GLMs generalize classical linear regression models by (1) allowing the response variable to be expressed as a nonlinear function—called the *link function*—of the usual linear combination of the independent variables, and (2) by allowing the variance of each noisy measurement to be a function of its predicted value. To fit a GLM model to a massive dataset, Ricardo exploits Hadoop, together with R's library of distributions and link functions (which includes their derivatives and inverses). The model-fitting algorithm is similar to the algorithm for fitting a latent-factor model, except that R is used not only for the driver process, but also at the worker nodes.

More generally, as pointed out in [8], many deep analytics can be decomposed into a small-data part and a large-data part. Such decompositions are possible not only for linear regression, latent-factor, PCA, and GLM analyses, but also for k-means clustering,

independent component analysis, support vector machines, classification, and many more types of analysis. The general approach to scaling up these analytical tasks is to reformulate them in terms of "summation forms". These sums, which are inherently distributive, can be executed in parallel. Then the results flow into algorithms that typically perform small matrix operations, such as (pseudo)-inversions or eigenvector decompositions, depending upon the specific analysis. Thus, although a relatively small collection of scalable analytics has been implemented so far, Ricardo can potentially be applied much more generally.

4.5 Implementation Details

The design philosophy of Ricardo is to minimize the transfer of functionality between R and Hadoop—letting each component do what it does best—as well as the amount of data transferred. The data that crosses the boundary between R and Jaql is usually aggregated information. As shown by our latent-factor example, complex trading involves multiple iterations over the data, with aggregates crossing the R-Jaql bridge many times. The bridge must therefore be very efficient so as not to become a performance bottleneck. The design of the bridge turned out to be rather intricate, for a couple of reasons. One challenge is that R is implemented entirely in C and Fortran, whereas Hadoop and Jaql belong to the Java world. An even greater challenge arises because the data formats of R and Jaql differ considerably. For example, R uses a column-oriented structure (data frames), whereas Jaql and Hadoop are designed for row-oriented access to data. As another example, R freely intermingles references to data items by position (e.g., `customer[[3]]`) and references by association (e.g., `customer$income`), whereas JSON objects represent data in a way that maintains either position or association, but not both. In this section, we discuss some of the design choices that we made in implementing the R-Jaql bridge and their impacts on the performance of data transfers across the bridge.

Bridging C with Java The problem of interfacing Java and C has been well studied, and the Java Native Interface (JNI) is a common solution for bridging these languages. A straightforward solution would be to use JNI to submit the query from R to the Jaql engine, and when Jaql finishes computation, use JNI calls to iteratively fetch data from Jaql into R, one tuple at a time. But this approach is inefficient for transferring even a few hundred data tuples, i.e., a few megabytes of data. For the Netflix latent-factor model, computing an aggregate rating for a movie (over the customers who rated the movie) requires about 17K data tuples to be transferred from Jaql to R. Using a JNI call for each tuple, the transfer takes tens of minutes, making this approach infeasible for any real-world applications. We therefore designed the bridge to pass only meta- and control-information using JNI calls, and wrappers on both the R side and the Jaql side use this information to perform bulk data transfers using shared files. For instance, when transferring results of a Jaql query to R, a Jaql wrapper materializes the result into a file in a format that can be directly loaded into R, and passes the location of this file to the R process through JNI. A wrapper on the R side reads this file and constructs an R object from the data, which can then be used for further R processing and analysis. This indirection through files (or even through shared memory) results in orders-of-magnitude improvement in performance—the foregoing 17K tuple transfer now takes only a few seconds. A similar approach is used for transferring data from R to Jaql.

Handling data-representation incompatibilities In R, large data objects resembling relational tables are primarily manipulated as data frames, which are column-oriented structures. On the other hand, Jaql and Hadoop are primarily row-oriented. Surprisingly, when large R objects are serialized in a row-oriented format, the cost within R of reading these objects from disk (or from any I/O channel) grows super-linearly in the number of columns transferred. This realization led us to design the bridge to perform the translation from row-oriented to column-oriented formats and vice versa. The Jaql wrapper translates the row-oriented results and outputs one file per column of data, while the wrapper on the R side reads the files one column at a time, and constructs the data frame from the individual columns. This approach ensures that cost of data transfer is linear with respect to the number of rows and the number of columns. Similar approaches were used to circumvent the remaining incompatibilities between R and JSON objects.

Property	Value
`mapred.child.java.opts`	-Xmx700m
`io.sort.factor`	100
`io.file.buffer.size`	1048576
`io.sort.mb`	256
`mapred.reduce.parallel.copies`	20
`mapred.job.reuse.jvm.num.tasks`	-1
`fs.inmemory.size.mb`	100

Table 1: Hadoop configuration used in experiments

5. EXPERIMENTAL STUDY

In this section, we report experimental results and share our experiences in implementing the latent factor model for movie recommendations. Our discussion focuses on performance and scalability aspects, although, as we have argued, Ricardo's technical approach has many advantages beyond simply providing a workable solution.

5.1 Test Setup

All performance experiments were conducted on a cluster of 50 nodes in the Amazon Elastic Compute Cloud (EC2). Each node was of type "High CPU Extra Large," i.e., a 64-bit virtual machine with 7 GB of memory, 20 EC2 Compute Units CPU capacity (8 virtual cores with 2.5 EC2 Compute Units each), and 1690GB of local storage. Aggregated over the cluster, this amounts to about 400 CPU cores, 70TB of aggregated disk space, and 350 GB of main memory. We installed Hadoop version 0.20.0 on each of the nodes. One of the nodes was selected as master and ran the Hadoop namenode and jobtracker processes; the rest of the nodes were worker nodes and provided both storage and CPU. Each worker node was configured to run up to seven concurrent map tasks and one reduce task. This map-heavy configuration was chosen because the bulk of the processing is performed during the map phase; reducers merely aggregate the map output. Table 1 lists other Hadoop parameters that we chose; the remaining parameters match the default configuration of Hadoop. The R driver itself was running on a separate EC2 gateway node of type "Standard Extra Large", which differs from the remaining nodes in that it had more memory (15GB) but less CPU power (8 EC2 Compute Units: 4 virtual cores with 2 EC2 Compute Units each).

Our experiments were performed on "inflated" versions of the original Netflix competition dataset [5]. The original dataset contains data for 100M ratings, including anonymized customer identifiers (about 480K distinct), information about the movies (such as title and date of release; about 18K distinct), and the date that the rating was entered. The inflation process allowed us to evaluate the performance of Ricardo on datasets having real-world scales.

Number of Rating Tuples	Data Size in GB
500M	104.33
1B	208.68
3B	625.99
5B	1,043.23

Table 2: Data set sizes used in the scale out experiments

Specifically, we added six additional attributes to each tuple in order to simulate a real recommender system, where the logged information contains fine-grained information such as demographics and navigation history. We also increased the number of ratings, with the largest inflated dataset containing 5B ratings (roughly 1TB of raw data). Table 2 summarizes the various dataset sizes used in the scale-up experiments. The numbers of distinct customers and distinct movies were retained from the original data, which amounts to a space requirement of about 4MB per factor of the model.

5.2 Performance and Scalability

Virtually all of the processing time for computing the latent factor model was spent on the Jaql/Hadoop side, rather than in R. This is not surprising, since scanning, joining, and aggregating large datasets is both I/O and CPU intensive, whereas R's adjustment of latent-factor values during the optimization procedure is comparably cheap. Nevertheless, the total system performance depends on the time required per iteration over the data and on the number of iterations until convergence. Thus both the DMS and R play an integral part in overall performance.

Figure 4 shows the performance results for a single iteration—i.e., the computation of the squared error and the corresponding gradient—for various dataset sizes. Recall that we presented the latent factor model using only a single latent factor per user and per movie. In the experiments, we actually fit a vector of latent factors per user and per movie during a single pass. For simplicity we omit the details, since the overall response time was only slightly affected by the vector size. We used two different implementations: a baseline implementation using a hand-tuned Hadoop job and a Jaql implementation as used by Ricardo. As can be seen in the figure, both implementations scale linearly with the size of the dataset and thus provide a scalable solution. The execution plans used by raw Hadoop and Jaql are similar, but Jaql requires about twice as much time as raw Hadoop. The reason for this difference lies in Jaql's higher level of abstraction: Jaql is designed to handle arbitrary queries, data formats, and data types, whereas our Hadoop implementation is specifically tailored to the problem at hand. This is also reflected in code complexity: the Jaql code comprises roughly 10 lines per query, whereas the Hadoop implementation takes around 200 lines.

The execution times as shown in Figure 4 should be taken with a grain of salt, as both Hadoop and Jaql are under active development. Each iteration currently takes a relatively large amount of time, up to an hour for large (1TB) data. We believe that with increasing maturity, both the relative difference between systems as well as the total execution time will decrease further. Indeed, our initial work on Ricardo has already led to multiple improvements in Jaql, e.g., better schema support, efficient serialization code, and runtime tuning.

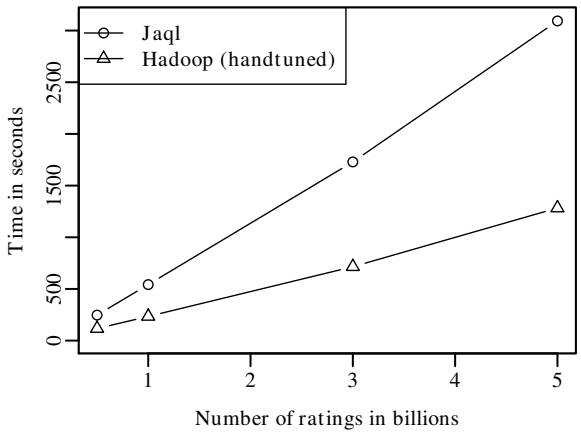

Figure 4: Performance comparison (single iteration)

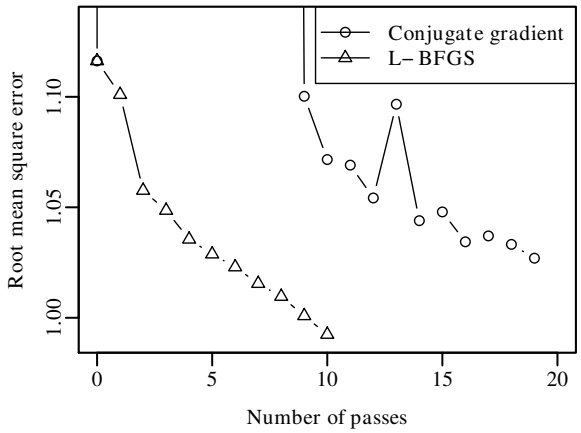

Figure 5: Number of passes and RMSE using one latent factor

5.3 Optimization Evaluation

We experimented with many different optimization routines, including naive gradient descent as well as more sophisticated algorithms such as the conjugate-gradient method or L-BFGS. This experimentation was easy to perform, since we could directly and easily take advantage of R's large optimization library.[5]

Figure 5 displays the root-mean-squared error (RMSE) of the latent-factor model as a function of the number of iterations of the optimization algorithm, i.e., the number of passes over the data. Results are given for both the conjugate-gradient and L-BFGS optimization methods. The conjugate gradient method performs a gradient computation followed by a line search and thus requires multiple passes over the data to arrive at a new point estimate. The line searches account for the fluctuations in the RMSE curve that can be seen in Figure 5. In contrast, L-BFGS is a quasi-Newton method with low memory footprint and exhibits a completely different behavior. Both methods converge, but L-BFGS converges faster for this particular problem.

[5] http://cran.r-project.org/web/views/Optimization.html

6. RELATED WORK

This section discusses prior work on alternative approaches to Ricardo. Unlike Ricardo, these approaches do not trade data and functionality between a DMS and a statistical package, but rather try to either scale out R or incorporate deep analytics into a DMS.

6.1 Scaling Out R

The R community is very active in improving the scalability of R and there are dozens of approaches that aim at parallelizing R across multiple processors or nodes. The main drivers for this work are increased dataset sizes and the increasing demands of scientific research and high-performance computing [21].

The existing approaches to parallelizing R can be classified by their degree of abstraction. These range from low-level message passing to task- and data-parallel processing to high-level (largely) automatic parallelization.

Packages of the *message-passing* type include Rmpi and rpvm, both of which are wrappers for their respective parallel computing middlewares. Direct use of message-passing systems requires significant expertise in parallel programming. More importantly, these packages require a "tightly" connected cluster, where fault tolerance, redundancy and elasticity are not provided. This restriction limits scalability to relatively small clusters. In contrast, Hadoop (and hence Ricardo), provides all of the foregoing functionality, and hence can scale to very large clusters.

Higher-level *task- and data-parallel computing* systems for parallelizing R are usually built on top of a message-passing package, and are easier to use. The most popular representative R package of this type is SNOW [24] (for Simple Network Of Workstations). SNOW provides functionality to spawn a cluster (using, for example, MPI), to distribute values across the cluster, and to apply in parallel a given function to a large set of alternative arguments. SNOW can be used to implement task parallelism (arguments are tasks) or data parallelism (arguments are data). SNOW's data-parallelism functionality is similar to the map operator in Hadoop; reduce operations, however, require more elaborate solutions.

In general, SNOW appears to be too low-level for conveniently implementing scalable deep analytics. This assertion is based on our experience in using the SNOW package to implement the computation of the latent-factor model for movie recommendations. The implementation involved the following steps:

1. *Distribution of data.* The base data had to be distributed across the cluster so that worker nodes can access it. This was achieved by using a distributed file system and by splitting the data into smaller chunks, which are then handled by the individual nodes.

2. *Data processing.* Operations such as the filter, group-by and join were implemented in R.

3. *Parallelization.* The data processing operations were broken down into fragments, executed in parallel, and then the results where combined.

It is evident that the above steps constitute the core functionality of a large-scale DMS. Our implementation of latent-factor computation using SNOW required 50% more code than the implementation in Ricardo, while at the same time being much more complicated and error prone. Additional features such as scalability to a large number of nodes, fault tolerance and elasticity were so tedious to implement using R and SNOW that they were dropped from our implementation. A recent package called RHIPE [12] provides SNOW-like functionality using Hadoop as the runtime environment and thus alleviates some of these problems. Data processing operators and their parallelization are not yet implemented, however. Thus, analysis tasks such as latent-factor modeling are currently easier to implement in Ricardo, and the Ricardo implementations have better runtime properties.

At the highest level, one would like a system that supports *automatic parallelization* of high-level R commands. Recently, there have been some promising results in this direction. Not all sequential R code can be parallelized automatically, of course, but linear-algebra operations can be handled by some systems, including pR [20] and RIOT [26]. Both packages are similar to Ricardo in spirit, but instead of offloading data processing, they offload parallel matrix computations to systems specifically designed for this purpose.

6.2 Deepening a DMS

Whereas the R community has tried to incorporate large-scale DMS functionality, the DMS community has tried to incorporate analytics. Early efforts in this direction incorporated some simple statistical analyses into parallel relational database systems [1]; indeed, the SQL standard now covers computation of basic summary statistics such as standard deviation and covariance, as well as simple (single-variable) linear regression. Moreover, the major commercial vendors have all incorporated some simple vendor-specific analytic extensions. Cohen et al. [9] describe an attempt to incorporate into a relational DBMS such analytical functionality as vector and matrix operations, ordinary least squares, statistical tests, resampling techniques, and parameter fitting via the conjugate-gradient method. The implementation exploits the extensible datatype facilities of Postgres. Although the resulting SQL queries are closer to what an analyst would use than classical SQL, it is unclear whether an analyst would ever be truly comfortable enough with the quasi-SQL syntax to abandon the familiar constructs of R, SPSS, SAS, or Matlab. The statistical functionality would probably need to be hidden in stored procedures.

Moving away from relational databases, we come to the Mahout project [2], which directly implements various machine-learning algorithms in Hadoop. The Mahout implementation currently neither exploits high-level data processing languages built on top of Hadoop nor does it make use of any statistical software. However, Mahout is still at an early stage. As more and more sophisticated methods are added, leveraging such existing functionality adds to the stability and simplicity of the implementation.

Arguably, the functionality of statistically enriched DMSs will always lag behind that of statistical software packages, and will not enjoy the same level of community support as can be found, for example, in the CRAN repository. Statistical packages also provide analysts with a more comfortable interface to access this functionality. In contrast, expressing statistical functionality in a query language is tedious. For example, the SQL implementation of the Mann-Whitney U Test in [9] is much more complex and difficult to read than a call to a function like `wilcox.test` as provided by R. For these reasons, it is highly likely that data analysts will always want to work with statistical software.

We conclude this section by mentioning SciDB [22]. This system embodies a very interesting and radical approach that attempts to completely redesign the data model—by using multidimensional arrays for storage and making vectors and arrays first-class objects—and scale both data management and computation by executing functions and procedures in parallel and as close to the data as possible.

7. CONCLUSION

We have presented Ricardo, a scalable platform for deep analytics. Ricardo combines the data management capabilities of Hadoop and Jaql with the statistical functionality provided by R. Compared to previous approaches, Ricardo provides a feature-rich and scalable working environment for statistical computing, and benefits from decades of experience from both the statistical and the data management communities. Although our experiments indicated that the performance of both Hadoop and Jaql is still suboptimal, we expect significant performance improvements in the future as this technology matures. Our work has focused on Hadoop and R, but it is evident that our approach is potentially applicable to other statistical packages and other DMSs.

In ongoing work, we are identifying and integrating additional statistical analyses that are amenable to the Ricardo approach. Although Ricardo currently requires knowledge of the Jaql query language, a tighter and transparent integration into the R language is possible using packages and functions that hide the underlying data-management details from the analyst. For certain types of analyses, it may be possible to combine our work with automatic compilation technology to integrate R and Hadoop even more seamlessly. Our ultimate vision is to enable analysts to work with massive data just as they work with main-memory-size data today; Ricardo can be seen as a first step in this direction.

8. REFERENCES

[1] N. R. Alur, P. J. Haas, D. Momiroska, P. Read, N. H. Summers, V. Totanes, and C. Zuzarte. *DB2 UDB's High Function Business Intelligence in e-Business*. IBM Redbook Series, ISBN 0-7384-2460-9, 2002.

[2] Apache Mahout. `http://lucene.apache.org/mahout/`.

[3] M. Avriel. *Nonlinear Programming: Analysis and Methods*. Dover Publishing, 2003.

[4] R. Bell, Y. Koren, and C. Volinsky. Modeling relationships at multiple scales to improve accuracy of large recommender systems. In *KDD*, pages 95–104, 2007.

[5] J. Bennett and S. Lanning. The Netflix prize. In *KDD Cup and Workshop*, 2007.

[6] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208, 1995.

[7] J. Canny. Collaborative filtering with privacy via factor analysis. In *SIGIR*, pages 238–245, 2002.

[8] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.

[9] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. *PVLDB*, 2(2):1481–1492, 2009.

[10] J. Cunningham. Hadoop@Visa, Hadoop World NY, 2009. `http://www.slideshare.net/cloudera/hw09-large-scale-transaction-analysis`.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[12] S. Guha. RHIPE - R and Hadoop Integrated Processing Environment. `http://ml.stat.purdue.edu/rhipe/`.

[13] K. Hornik. The R FAQ, 2009. `http://CRAN.R-project.org/doc/FAQ/R-FAQ.html`.

[14] JAQL: Query Language for JavaScript Object Notation (JSON). `http://code.google.com/p/jaql`, 2009.

[15] J. A. Konstan. Introduction to recommender systems. In *SIGMOD Conference*, pages 1373–1374, 2008.

[16] Y. Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *KDD*, pages 426–434, 2008.

[17] Y. Koren. Collaborative filtering with temporal dynamics. In *KDD*, pages 447–456, 2009.

[18] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[19] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[20] N. F. Samatova. pR: Introduction to Parallel R for Statistical Computing. In *CScADS Scientific Data and Analytics for Petascale Computing Workshop*, pages 505–509, 2009.

[21] M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, and U. Mansmann. State of the art in parallel computing with R. *Journal of Statistical Software*, 31(1):1–27, June 2009.

[22] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and SciDB. In *CIDR*, 2009.

[23] 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. *PVLDB*, 2(2):1626–1629, 2009.

[24] L. Tierney, A. J. Rossini, N. Li, and H. Sevcikova. snow: Simple network of workstations. `http://cran.r-project.org/web/packages/snow/`.

[25] M. Wedel, R. T. Rust, and T. S. Chung. Up close and personalized: a marketing view of recommendation systems. In *RecSys*, pages 3–4, 2009.

[26] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O-efficient numerical computing without SQL. In *CIDR*, 2009.