A Comparison of Platforms for Implementing and Running Very Large Scale Machine Learning Algorithms

ABSTRACT

We describe an extensive benchmark of platforms available to a user who wants to run a machine learning (ML) inference algorithm over a very large data set, but cannot find an existing implementation and thus must "roll her own" ML code. We have carefully chosen a set of five ML implementation tasks that involve learning relatively complex, hierarchical models. We completed those tasks on four different computational platforms, and using 70,000 hours of Amazon EC2 compute time, we carefully compared running times, tuning requirements, and ease-of-programming of each.

1. INTRODUCTION

Many platforms have been proposed to provide programming and runtime support for distributed/parallel machine learning (ML) codes, including OptiML [19], GraphLab [11], SystemML [6], and SimSQL [4]. MLBase [10] and ScalOps [20] also address the problem, though the most recent published descriptions indicate that these systems are more immature. Other systems such as Pregel [12], Giraph [2], Hama [17], Spark [23], Ricardo [5], Nyad [14], and DryadLinq [21] may not have been developed only for ML, but count it as an important application.

Unfortunately, there is little objective knowledge regarding the pros and cons of the various platforms (and engineering principles) for implementing and running large-scale, statistical ML codes. The most comprehensive performance studies considering the aforementioned platforms (for example, [8, 14]) do not focus on ML, or consider it only tangentially. Existing studies tend to focus on simple, unrepresentative problems that are very low-dimensional or have only modest CPU requirements relative to data set size.

We seek to remedy this. We describe an objective benchmark of some of the platforms available to a user who wants to run a specific ML inference algorithm over a large data set, but cannot find an existing implementation and thus must "roll her own" ML code. Given the wide variety of ML models, this will not be an uncommon occurrence.[1] We draw a distinction between a user who wants to implement and apply a brand new ML code, and someone who just wants to use a code, and focus on the former. The implementor will want to balance ease of implementation with performance, whereas an end user has little concern for the effort required to engineer the code and will be happy with an intricately constructed C and MPI code as long as it is fast and easy to use.[2]

Our contributions. Our specific contributions are:

(1) First and most obvious, we shed some light on the relative merits of some (quite different) platforms for implementing large-scale ML algorithms. Our results will surprise many readers.

(2) Second, we demonstrate (through example) what a scientific study of a platform for writing large-scale ML codes might look like. We have carefully chosen a set of tasks that involve learning relatively complex, hierarchical statistical models. We have purposely avoided simple, convex models whose parameters can be optimized using easily-implemented techniques such as gradient descent [18]. Their simplicity means they benefit relatively little from the abstractions provided by the platforms we consider.

(3) Finally, we hope that our efforts will grow into a widely used, standard benchmark for this sort of platform. In the future, a implementor of a new or existing platform need only implement these codes and compare with our numbers.

2. STATISTICAL ML: A PRIMER

In statistical ML, one first postulates a generative, statistical model for a data set, characterized by a distribution function $f(X|\theta)$. Here, X is the data set, and θ is the unseen model parameters and hidden variables. *Learning* is the process of choosing a value $\hat{\theta}$ for θ so that $\hat{\theta}$ does a good job of describing how the model could have produced X. Learning is typically quite expensive computationally, especially if the data set is very large. Hence platforms that can support parallel and/or distributed ML are attractive.

We focus our efforts on learning relatively complex, hierarchical models. Such models are often (though not always) characterized by the fact that they take a Bayesian approach to ML. The Bayesian approach is unique in that the user expresses his or her prior belief about the model formally, by supplying a *prior distribution* $f(\theta)$ over θ. The goal of the learning process under the Bayesian approach is to understand something about the so-called *posterior distribution* for θ, written as $f(\theta|X)$. The posterior distribution

[1] For example, it is telling that of the five standard Bayesian ML inference algorithms we consider in this study, it appears that only the collapsed LDA inference algorithm [3] is available as part of an existing package, and even then we are aware of no "non-collapsed" Gibbs sampler implementation (See Section 8 of the paper).

[2] Our focus on giving an implementor an idea of what platform to use means that we do not consider the task of benchmarking ML libraries that are not meant as a platform for writing new codes, such as Mahout [15], MADLib [9] and Vowpal Wabbit [1].

is related to the prior via *Bayes' Rule*:

$$f(\theta|X) \propto f(\theta)f(X|\theta).$$

The simplest, most universal, and the most common way to analyze the posterior is to draw a set of samples from it via Markov chain Monte Carlo (MCMC) [7]. To apply MCMC to the problem, a Markov chain is defined over all possible values for θ. This Markov chain is a random walk over a graph, where each vertex in the graph is a possible value for θ, and an edge between two possible values $\hat{\theta}_1$ and $\hat{\theta}_2$ for θ is labeled with a weight describing the likelihood of transitioning from $\hat{\theta}_1$ to $\hat{\theta}_2$ during the walk. The graph is defined so that observing the current state after an infinite-length walk is equivalent to drawing a sample $\hat{\theta}$ from the target $f(\theta|X)$. In practice a random walk that traverses only a few dozen to a few thousand possible values for θ will suffice to "mix" the chain, or produce a reasonable sample from $f(\theta|X)$.

3. EXPERIMENTAL OVERVIEW

3.1 Platforms Tested

The platforms we evaluate in the paper are:

Spark [23], which embodies the MapReduce/dataflow approach. We choose Spark because it purports to be both higher performance and easier to program than Hadoop MapReduce.

SimSQL [4], which is a parallel, relational database that supports an SQL-based approach to running large-scale MCMC simulations.

GraphLab [11], which supports a graph-based abstraction for writing distributed machine learning codes. For large-scale MCMC, one can map the various components of θ to nodes in a graph, with edges representing statistical dependencies.

Giraph [2], which is another, more general-purpose graph-based processing platform, widely known (for example) to be used extensively by Facebook.

3.2 ML Models Considered

We use each of these platforms to implement five different MCMC codes for learning the following five models:

(1) A Gaussian mixture model (GMM) for data clustering.

(2) The Bayesian Lasso [16], a Bayesian regression model.

(3) A hidden Markov model (HMM) for text.

(4) Latent Dirichlet allocation (LDA), a model for text mining.

(5) A Gaussian model for missing data imputation.

3.3 Balancing Performance and Ease-Of-Use

Since we focus on evaluating the platforms from the point-of-view of a user-implementor, our study focuses both on performance and on programmability. These are frequently at odds. Consider that the platforms we evaluate can execute arbitrary C++ or Java code, so one could simply use the platform to distribute code around a compute cluster, which could then begin communicating via MPI or RPC, bypassing the platform entirely. Hadoop MapReduce is sometimes used in this way. While such a "bare metal" implementation would give high performance, it is going to be labor-intensive to build and would be avoided by most implementors.

Thus, we adopted a simple guideline when considering the question of how to implement the MCMC simulations we study in this paper. We would first attempt to implement each code in the "purest" way possible; that is, the way in which was most in-keeping with the conceptual design of the platform being evaluated. For example, in a graph, this means mapping each variable or data point to a vertex, and using the graph to manage the computation, avoiding the use of mechanisms such as synchronized global variables.

Unfortunately, there were cases where such a "clean" implementation would not work. In such situations, we did what we needed to do in order to render the implementation feasible. This meant writing code that takes some of the burden of managing the data and computation away from the platform.

3.4 Experimental Platform

All of the experiments reported in this paper were performed using Amazon EC2 m2.4xlarge machines, running Ubuntu (each machine had eight virtual cores, two disks, and 68 GB of RAM). In all, we used approximately 70,000 hours of compute time, considering debugging, tuning and testing. At Amazon's current on-demand pricing, this time would cost more than $100,000, though we typically used spot instances, which are (usually) much less expensive.

We were initially concerned about variability in EC2 performance from day-to-day and machine-to-machine. When we tested the same MCMC simulation on five different days using five different compute clusters, we found that the standard deviation in per-iteration running time was only 32 seconds (out of 27 minutes on average) and so we decided that such variations were insignificant.

4. PLATFORMS EVALUATED

4.1 Spark

Spark [23] is an open source cluster computing system designed for large scale data analytics. Spark utilizes Resilient Distributed Datasets (RDDs) [22], which provide a fault-tolerant, distributed memory abstraction. RDDs are built by running coarse-grained transformations over either a data set or other existing RDDs. Spark provides fault tolerance by tracking data lineage, so it can rebuild lost data partitions. The designers of Spark argue that since RDDs allow for in-memory computations without concern for lost data, they can facilitate very fast performance compared to disk-based MapReduce systems such as Hadoop.

To us, one of the most attractive aspects of Spark is that it provides a Python-based programming interface. Python has become a very popular language for writing ML inference codes, providing a nice combination of succinct, beautiful codes and good support for scientific and statistical programming.

4.2 SimSQL

SimSQL [4] is a distributed, relational database system, whose design has been augmented to support stochastic analytics. Specifically, SimSQL has native support for a special type of user-defined function known as a *variable generation* (VG) *function*. VG functions are randomized, table-valued functions. Using SimSQL's dialect of SQL, it is possible to define a random database table whose contents are constructed from one or more VG function invocations, as well as from deterministic data stored within the database. Random table definitions in SimSQL's SQL dialect can be mutually recursive. Hence one can define, in SQL, MCMC simulations.

SimSQL is written mostly in Java, and has a Prolog-based query optimizer. SimSQL VG functions themselves are written in C++. SimSQL compiles SQL simulations into Java classes that are executed as Hadoop MapReduce jobs.

4.3 GraphLab

GraphLab [11] is a graph-based distributed computing platform written in C++. The so-called "graph-parallel" abstraction used by

GraphLab is a useful abstraction for ML, since many ML inference algorithms are naturally centered around variables/data (which map to vertices in a graph) and the statistical relationships between them (which map to edges). GraphLab is unique in that its computational model is pull-based and asynchronous. Each vertex in the graph constantly requests data from its neighbors in order to update its own state. Asynchronicity is the single most defining (and unique) characteristic of the GraphLab approach, and allows for a very appealing computational model, since when writing the code associated with a vertex, one need only consider the computation required to update the state of that vertex; one can more or less ignore the remainder of the computation.

4.4 Giraph

Giraph [2] is also a graph-based, distributed computing platform. It is written in Java and runs on top of Hadoop. It is often viewed as an open-source version of Pregel [12]. Giraph differs fundamentally from GraphLab in that its model is push-based and synchronous, using the so-called bulk synchronous parallel (BSP) model. Giraph divides a graph algorithm into a sequence of *supersteps*. In each superstep, every vertex executes the same user-defined compute function in parallel, where the vertex receives messages from other vertices in superstep $i-1$, accesses and updates its local state, and then sends its messages to the other vertices for superstep $i+1$. Vertices can work together to perform tasks such as aggregation in a tree-based fashion using combiners to speed the computation.

4.5 Versions Tested

We tested version 0.1 of SimSQL, version 2.2 of GraphLab, version 1.0.0 of Giraph. Our GMM implementation ran on version 0.7.3 of Spark and the other models on version 0.8.0 of Spark.

5. GAUSSIAN MIXTURE MODEL

We begin our experimental evaluation by considering a classical data clustering model: the Gaussian mixture model (GMM). A GMM views a data set as being produced by a set of K Gaussian (multi-dimensional normal) distributions; the kth Gaussian is parameterized by a mean vector μ_k and a covariance matrix Σ_k, and has an associated probability π_k. To produce the jth point in the data set, the model first selects a Gaussian by generating a sampled vector $\mathbf{c}_j$ vector from a Multinomial$(\pi, 1)$ distribution ($c_{j,k}$ is then a 1 if and only if the k Gaussian was used to produce the data point x_j, and zero otherwise). The data point itself is then sampled from the Gaussian indicated by $\mathbf{c}_j$. We put a Dirichlet(α) prior on π, a Normal(μ_0, Λ_0^{-1}) prior on each μ_k, and an InvWishart(v, Ψ) prior on each Σ_k. The goal during learning is then to discover all of the unseen c_j values, as well as all of the Gaussian parameters.

Let $\mathbf{p}_j^{(i)}$ denote the unit-length vector whose kth entry is proportional to $\pi_k^{(i)} \times \text{Normal}\left(\mathbf{x}_j | \mu_k^{(i)}, \Sigma_k^{(i)}\right)$. That is, it gives the posterior probability that point k came from the jth cluster, given the parameters at the ith MCMC iteration. A Markov chain to learn the desired, posterior distribution can be derived as:

$$\mu_k^{(i)} \sim \text{Normal}\Bigg(\left(\Lambda_0\mu_0 + n\left(\Sigma_k^{(i-1)}\right)^{-1}\right)^{-1} \times \left(\Lambda_0\mu_0 + \left(\Sigma_k^{(i-1)}\right)^{-1}\sum_j c_{j,k}^{(i-1)}\mathbf{x}_j\right), \left(\Lambda_0\mu_0 + n\left(\Sigma_k^{(i-1)}\right)^{-1}\right)^{-1}\Bigg)$$

$$\Sigma_k^{(i)} \sim \text{InvWish}\left(n + v, \Psi + \sum_j c_{j,k}^{(i-1)}(\mathbf{x}_j - \mu_k^{(i)})(\mathbf{x}_j - \mu_k^{(i)})^T\right)$$

$$\pi_k^{(i)} \sim \text{Dirichlet}\left(\alpha + \sum_j c_j^{(i-1)}\right) \quad c_j^{(i)} \sim \text{Multinom}\left(\mathbf{p}_j^{(i)}, 1\right)$$

The task is to write a distributed code that simulates this chain.

5.1 Spark Implementation

We begin by creating a RDD named `data`, which is read and parsed from data in permanent storage (HDFS in our example):

```
lines = sc.textFile("hdfs://master:54310/data.txt")
data = lines.map(parseLine).cache()
```

The `cache()` function asks the system to store RDD in the memory, which can accelerate subsequent visits and operations on `data`.

Next, we compute the hyper-parameters μ and Λ_0^{-1} as the observed mean and dimensional variance of the data:

```
num = data.count()
hyper_mean = data.reduce(add)/num
hyper_cov_diagonal = data.map(lambda x: square(x -
            hyper_mean)).reduce(add)/num
numpy.fill_diagonal(hyper_cov, hyper_cov_diagonal)
```

Based on those parameters, we initialize the model (π_i, μ_i and Σ_i) for each Gaussian component:

```
c_model = sc.parallelize(range(0, K)).map(lambda x:
         (x, (mvnrnd(hyper_mean, hyper_cov),
         invWishart(hyper_cov, len(hyper_mean)+2))))
         .collectAsMap()
pi = np.zeros(K, float)
pi.fill(1.0/K)
```

Here, `mvnrnd(.)` and `invWishart(.)` are user-defined functions that call PyGSL library functions. `collectAsMap()` is a built-in function that transforms the RDD to a Python dictionary.

Finally, we come to the main loop of the program, which consists of three MapReduce jobs. The first dominates the overall runtime:

```
c_agg = data.map(lambda x:
        sample_mem(x, pi, c_model))
        .reduceByKey(lambda (x1, y1, z1),
        (x2, y2, z2):(x1+x2, y1+y2, z1+z2))
```

On the Map side, the output of the function `sample_mem(x, pi, c_model)` is a tuple `(k, (1, x, sq_x))`, where `k` is the membership of data point `x` and `sq_x` is the matrix $\hat{\Sigma} = (x - \mu_k)(x - \mu_k)^T$, μ_k is the mean of kth Gaussian component. On the Reduce side, tuples with the same membership are aggregated into a single tuple.

The second job is Map-only. This job samples μ_k and Σ_k for each Gaussian component in parallel:

```
c_model = c_agg.mapValues(lambda (c_num, x_sum,
          sq_sum): updateModel(c_num, x_sum, sq_sum,
                      hyper_mean, hyper_cov)
          ).collectAsMap()
```

The third job collects the number of data points assigned to each cluster, which is then used to update π:

```
c_num = c_agg.mapValues(lambda (c_num, x_sum,
         sq_sum): c_num).collectAsMap()
pi = sample_dirichlet(c_num)
```

5.2 SimSQL Implementation

The simulation is implemented in SimSQL using a database schema with four random tables that correspond to the four classes of variables listed above:

```
clus_means[i](clus_id,dim_id, dim_value)
clus_covas[i](clus_id, dim_id1, dim_id2,dim_value)
clus_prob[i](clus_id, prob)
membership[i](data_id, clus_id)
```

The data to be processed are stored in a database table:

```
data(data_id, dim_id, data_val)
```

as are the various entries in the α vector (the hyperparameter for the Dirichlet prior on π):

```
cluster(dim_id, alpha)
```

In addition, several views are used to store the other hyperparameters, which are computed empirically from the data. For example, the vector μ_o is computed as the mean of the entire data set:

```
create view mean_prior(dim_id, dim_val) as
  select dim_id, avg(data_val)
  from data
  group by dim_id;
```

Aside from this, the entire SimSQL code consists of (1) initialization codes for the first three random tables, (2) recursive definitions for all four random tables, and (3) a C++ implementation of the `multinomial_membership` VG function, which is used to update `membership[i]` (the other VG functions are all library functions).

As an example, consider the following initialization:

```
create table clus_prob[0] (clus_id, prob) as
  with diri_res as Dirichlet
    (select clus_id, pi_prior
     from cluster)
  select diri_res.out_id, diri_res.prob
  from diri_res;
```

This code uses the hyperparameters stored in the `cluster` table to parameterize the Dirichlet VG function, which then outputs the value of $\pi^{(0)}$ as a set of $(k, \pi_k^{(0)})$ pairs, that are then stored in the `clus_prob[0]` table.

Here is an example of a recursive definition:

```
create table clus_prob[i](clus_id, prob) as
  with diri_res as Dirichlet
    (select cmem.clus_id,
       cmem.count_num+clus.pi_prior as diri_para
     from (select cm.clus_id as clus_id,
                  count(cm.data_id) as count_num
           from membership[i-1] as cm
           group by clus_id) as cmem, clus
     where cmem.clus_id = clus.clus_id)
  select diri_res.out_id, diri_res.prob
  from diri_res;
```

This code parameterizes the Dirichlet distribution by performing the required $\alpha + \sum_j c_j^{(i-1)}$ computation, in order to re-sample the selection probability for each of the clusters. This computation requires that we compute the number of times that each data point is assigned to each cluster, which is done via SQL aggregation.

5.3 GraphLab Implementation

We define three types of vertices: data vertices, cluster vertices, and the mixture-proportion vertex. There is a one-to-one mapping between data points and data vertices. The clusters are put into a separate cluster vertex, and the vector of mixing proportions π is maintained by the mixture-proportion vertex. The cluster vertices and data vertices define a complete, bipartite graph and the mixture-proportion vertex is connected to each of the data vertices.

Each Gibbs sampler iteration is implemented using GraphLab's *gather-apply-scatter* abstraction. In the gather phase, each node "gathers" the state of all of its neighbors. In our implementation, the jth data vertex exports a view of itself that contains three values: the data point $\mathbf{x}_j$, the vector $\mathbf{c}_j$ identifying the cluster that produced the jth data point; and the matrix $\hat{\Sigma_j} = (\mathbf{x}_j - \mu_{1(\mathbf{c}_j)})(\mathbf{x}_j - \mu_{1(\mathbf{c}_j)})^T$. Here, $1(\mathbf{c}_j)$ returns the identity of the entry in $\mathbf{c}_j$ having the value "1". This is stored as a triple $\langle \mathbf{c}_j, \mathbf{x}_j, \hat{\Sigma_j} \rangle$. This triple is "gathered" by all of the nodes attached to the data vertex. The mixture-proportion vertex sums up all of the $\mathbf{c}_j$ vectors as it examines the triples associated with each data vertex, and the kth cluster vertex sums up all of the $\mathbf{x}_j$ values (and all of the Σ_j values) for those data vertices having a 1 in the kth position in $\mathbf{c}_j$.

Likewise, each data vertex "gathers" the state of the model from the mixture-proportion vertex (where it receives the vector π) and from each of the cluster vertices (where it receives the cluster mean vector μ_k and cluster covariance matrix Σ_k).

Next comes the apply phase, where each one of the vertices re-samples its own state. Each data vertex re-samples $\mathbf{c}_j$, the mixing-proportion vertex re-samples π, and the cluster vertices re-sample the cluster mean and covariance matrix.

In the scatter phase, each vertex signals all of its adjacent vertices, letting them know that the apply phase has completed.

5.4 Giraph Implementation

The graph we use for Giraph is identical to the GraphLab graph. However, as we will describe, the Giraph programming model is more oriented towards message-passing, as opposed to computation over a predefined set of edges. These messages can be constructed case-by-case, whereas in GraphLab a vertex must simple export a single view of its internals to the rest of the world.

The actual learning computation begins with an initialization of the cluster-related parameters $\langle \mu_k^{(0)}, \Sigma_k^{(0)}, \pi_k^{(0)} \rangle$ stored in the each cluster vertex followed by an initialization of the membership of each data point $\mathbf{c}_j^{(0)}$ stored in each data vertex.

After initialization, the cluster-membership vertex sends the current value of π_k to the kth cluster vertex, so K messages are sent in all. Each cluster vertex receives this mixing-proportion message and broadcasts the triple $\langle \mu_k, \Sigma_k, \pi_k \rangle$ to the whole system. The jth data vertex receives these K messages, and samples its membership $\mathbf{c}_j$. It computes the matrix $\hat{\Sigma_j} = (\mathbf{x}_j - \mu_{1(\mathbf{c}_j)})(\mathbf{x}_j - \mu_{1(\mathbf{c}_j)})^T$. (Again, $1(\mathbf{c}_j)$ returns the identity of the entry in $\mathbf{c}_j$ having the value "1".) It then sends the double $\langle \mathbf{x}_j, \hat{\Sigma_j} \rangle$ to the cluster vertex indicated by $\mathbf{c}_j$. Each cluster aggregate those messages, and resamples its posterior covariance Σ_k and mean μ_k. Each cluster counts the number of points assigned to it, and sends this count in a message to the cluster-membership vertex, which updates π.

We attempted to optimize the computation as much as possible. To save the memory, we do not record the edges between the data vertices and the cluster vertices explicitly; we instead use broadcast to communicate from the cluster vertices to data vertices, and we use a naming scheme allowing each data vertex to send a message to the appropriate cluster vertex without the system recording the edge. Giraph's combiner functionality is used to reduce communication and increase load balancing during aggregation.

One significant issue is the difficulty of finding an appropriate Java-language statistical/numerical software package. Even finding

a Dirichlet sampler written in Java is difficult. In the end, we went with the Mallet library [13] due to its completeness. We found Mallet to be slow for problems requiring higher-dimensional linear algebra. But without writing a lot of code (perhaps using JBLAS) there did not appear to be a better solution.

5.5 Experiments and Results

We tuned each code and platform in order to obtain the fastest per-iteration running time possible. We then created a synthetic data set having ten dimensions, generated using a mixture of ten Gaussians, and used each platform to learn these Gaussians. For each platform, we ran the GMM inference code on three compute clusters of different sizes: five machines, 20 machines, and 100 machines. The amount of data per machine was kept constant at ten million data points, so (for example) the 100 machine cluster was used to perform GMM inference over one billion data points.

In addition, we created a second data set with 100 dimensions. We then ran an experiment where we kept the amount of data constant at one million data points per machine. We performed inference for this data set only on the five-machine compute cluster.

Note that each platform is running exactly the same MCMC simulation (with only minor differences in the initialization and order of the updates in each iteration). Thus, it is not informative to examine the actual models learned.

The results are shown in Figure 1 (a). In this figure we show, for each experiment, the average per-iteration running time, computed over the first five iterations of the MCMC simulation. In parens, we give the time required to initialize the simulation (this typically includes choosing the initial cluster parameters and perhaps doing the initial assignment of data points to clusters). This is a one-time cost. "Fail" means that while the code could be executed on a small problem size, it could not be made to run at the scale required by the experiment via any reasonable amount of tuning.

5.6 Discussion

SimSQL, Spark and Giraph: No Significant Differences. Putting aside for a moment the failure of Giraph to scale to 100 machines or handle the 100-dimensional problem, there is not a significant difference in the non-GraphLab runtimes depicted in Figure 1 (a). The exception is SimSQL at 100 dimensions, which has a per-iteration time that is twice that of Spark. The reason for this is that the GMM simulation must aggregate one $(\mathbf{x}_j - \mu)(\mathbf{x}_j - \mu)^T$ matrix (which is a 10,000 entry matrix for 100-dimensional data) for each data point. In SimSQL, this is performed using a costly `GROUP BY`, which is slower than the Spark matrix/vector operations.

Java vs. Python. We were curious as to whether there was a performance hit using Python with Spark as opposed to Java with Spark. We chose Python because while Python is commonly used for statistical/numerical computing (NumPy and PyGSL, which we used, are quite popular, for example), there does not really exist any good support for this sort of computing within the JVM. However, Spark + Python uses Py4J, which uses sockets to send data back and forth between a Python interpreter and the JVM. This may be slow.

Thus, we re-implemented the Spark GMM MCMC code in Java with Mallet, and ran the experiments once again. The results are shown in Figure 1 (b). For the ten-dimensional inference problem, Java takes around 50% of the Spark time. But for the 100-dimensional problem, it is more than eight times slower. Granted, we might have been able to do better by writing a lot of code on top of JBLAS. But in light of these results—and considering the beauty of our Python codes—we decided to stick with Python.

GraphLab and Super Vertex Codes. Our initial GraphLab implementation failed every attempt to run it at the scale required by the experiments. Why? In a typical "hand-coded" implementation of the GMM inference algorithm, a program would store the current model in RAM, and then cycle through all of the data points in sequence, updating the cluster membership for each. In GraphLab, there is no notion of pinning a single copy of the model in RAM and cycling through the data points. Instead, the data vertex associated with each node must gather a copy of the model via the links in the graph, and the user cannot control this process. In practice, GraphLab seems to simultaneously materialize one 50KB copy of the model for each data point, which quickly exhausts the available memory and the computation fails.

How can one get around this problem? One method we considered was to move the model out of the graph, eschew the gather-apply-scatter mechanism, and perform the aggregation needed to update the model with one of GraphLab's distributed aggregation facilities. The problem is that it was unclear to us if such an implementation would be correct—with no synchronization and hence no global state, it was difficult for us to understand what "aggregation" would actually mean. Further, such an implementation would not use GraphLab's graph abstraction in any meaningful way.

Instead, we settled on the idea of combining a large numbers (hundreds of thousands) of data points together to form "super vertices" (we use 8000 super vertices on our 100 machine cluster). The gather phase in this implementation changes in that there is only *one* copy of the model obtained for the *entire* super vertex. Further, the view of the super vertex exported to its neighbors becomes an array of $\langle k, n_k, \mu_k, \Sigma_k \rangle$ tuples, with one tuple for each value of k in $1...K$. Here, n_k is the number of points in the super vertex assigned to cluster k, μ_k is the sum over $\mathbf{x}_j$ where $\mathbf{x}_j$ has been assigned to cluster k, and Σ_k is computed similarly from a sum over $(\mathbf{x}_j - \mu_k^{(i)})(\mathbf{x}_j - \mu_k^{(i)})^T$ for those points assigned to cluster k. A cluster vertex in the gather phase collects only the one triple assigned to it by each super vertex. In the apply phase, the super vertex then runs through each of its data points, in sequence.

In addition to radically reducing the memory requirements, this has the additional benefit of distributing most of the heavy-duty aggregation to the super vertices, instead of at the cluster vertices.

The result is a very fast GMM implementation, as shown in Figure 1 (b). A similar "super vertex" construction was a necessary part of each one of the GraphLab implementations described in the paper; without it, none of our GraphLab codes would run.

Super Vertex Codes for the Other Platforms. Naturally, GraphLab is not the only platform that can benefit from the super vertex construction. By grouping data points together and handling them as a single unit in "hand-coded" C++ (SimSQL/GraphLab), Java (in Giraph), or Python (Spark) we can often realize significant speedups. In Figure 1 (c) which shows the running times obtained through GMM super vertex codes on each of the platforms. SimSQL in particular can be made to run extremely fast using a super vertex construction; the 100-dimensional GMM implementation ran in a time that was only 20% of its nearest competitor (GraphLab).

6. THE BAYESIAN LASSO

The Bayesian Lasso [16] is a well-known regularized Bayesian linear regression formulation and associated MCMC sampler. Let the data set D consist of a set of n $\langle \mathbf{x}, y \rangle$ pairs. The model consists of a p-dimensional vector of regression coefficients β, where the (centered) response $\tilde{y}$ associated with the data point $\mathbf{x}$ is assumed to be generated as $\tilde{y} \sim \text{Normal}(\beta \cdot \mathbf{x}, \sigma^2)$, and σ^2 has an inverse-gamma prior.

To simplify the MCMC simulation used to learn the model, a set

(a) GMM: Initial Implementations					
		10 dimensions			100 dimensions
	lines of code	5 machines	20 machines	100 machines	5 machines
SimSQL	197	27:55 (13:55)	28:55 (14:38)	35:54 (18:58)	1:51:12 (36:08)
GraphLab	661	Fail	Fail	Fail	Fail
Spark (Python)	236	26:04 (4:10)	37:34 (2:27)	38:09 (2:00)	47:40 (0:52)
Giraph	2131	25:21 (0:18)	30:26 (0:15)	Fail	Fail

(b) GMM: Alternative Implementations					
		10 dimensions			100 dimensions
	lines of code	5 machines	20 machines	100 machines	5 machines
Spark (Java)	737	12:30 (2:01)	12:25 (2:03)	18:11 (2:26)	6:25:04 (36:08)
GraphLab (Super Vertex)	681	6:13 (1:13)	4:36 (2:47)	6:09 (1:21)*	33:32 (0:42)

(c) GMM: Super Vertex Implementations				
	10 dimensions, 5 machines		100 dimensions, 5 machines	
	w/o super vertex	with super vertex	w/o super vertex	with super vertex
SimSQL	27:55 (13:55)	6:20 (12:33)	1:51:12 (36:08)	7:22 (14:07)
GraphLab	Fail	6:13 (1:13)	Fail	33:32 (0:42)
Spark (Python)	26:04 (4:10)	29:12 (4:01)	47:40 (0:52)	47:03 (2:17)
Giraph	25:21 (0:18)	13:48 (0:03)	Fail	6:17:32 (0:03)

Figure 1: GMM MCMC implementation; lines of code and average time per iteration. Time in parens is for the initialization/setup. Format is HH:MM:SS or MM:SS. *We were actually unable to run GraphLab at 100 machines. Past 40 machines, GraphLab would not boot up at many cluster sizes. The closest to 100 machines that we were able to get was 96 machines.

of p auxiliary variables $\tau_1^2, \tau_2^2, ...$ are introduced that control the variance of the various regression coefficients as they are updated. Let $\tilde{\mathbf{y}}$ be the centered response vector and $\mathbf{X}$ denote the matrix of regressors constructed from D. Then the MCMC simulation for learning the model is as follows:

$$1/(\tau_j^{(i)})^2 \sim \text{InvGaussian}(\sqrt{\frac{\lambda^2(\sigma^{(i-1)})^2}{(\beta_j^{(i-1)})^2}}, \lambda^2)$$

$$\beta^{(i)} \sim \text{Normal}\left((\mathbf{A}^{(i)})^{-1}\mathbf{X}^T\tilde{\mathbf{y}}, \sigma^2(\mathbf{A}^{(i)})^{-1}\right), \text{ where}$$

$$\mathbf{A}^{(i)} = \mathbf{X}^T\mathbf{X} + (\mathbf{D}_\tau^{(i)})^{-1} \text{ and}$$

$$\mathbf{D}_\tau^{(i)} = diag\left((\tau_1^{(i)})^2, (\tau_2^{(i)})^2, ...\right)$$

$$(\sigma^{(i)})^2 \sim \text{InvGamma}(\frac{1+n+p}{2}, \frac{2 + \sum_{\langle \mathbf{x}, y\rangle \in D}\left(\tilde{y} - \beta^{(i)} \cdot \mathbf{x}\right)^2 + \sum_j (\beta_j^{(i)})^2/(\tau_j^{(i)})^2}{2})$$

6.1 Spark Implementation

In the Spark implementation, we first create a RDD `data` by reading the input files from HDFS, and use the RDD to compute the centered response $\tilde{\mathbf{y}}$:

```
data = lines.map(parseData).cache()
y_sum = data.map(lambda (id, (x, y)): y).sum()
y_avg = y_sum/data.count()
data = data.map(lambda (id,(x, y)): (id, (x, y-y_avg)))
```

Next, the Gram matrix and the quantity vector are computed:

```
XX = data.flatMap(lambda (id, (x, y)):
      computePairSum(x, y)).reduceByKey(add)
XY = data.flatMap(lambda (id, (x, y)):
      computeXYSum(x, y)).reduceByKey(add)
```

Most of the code of the main loop of the Gibbs sampler is run locally, since most of the computation required by the loop is quite modest for data of low to medium dimensionality (up to a few thousand dimensions). Only one actual MapReduce job is needed per iteration. Specifically, we must compute $\sum_{\langle \mathbf{x}, y\rangle \in D}\left(\tilde{y} - \beta \cdot \mathbf{x}\right)^2$ in a distributed fashion in order to parameterize the inverse-gamma distribution needed to compute $(\sigma^{(i)})^2$. The code is:

```
remain_sum1 = data.map(lambda (id,(x, y)):
       computeRemainSquare(x, y, beta)).sum()
```

6.2 SimSQL Implementation

The SimSQL implementation begins by creating three materialized views that will be used repeatedly, in each iteration of the MCMC simulation. The three materialized views include (1) the Gram matrix computed over $\mathbf{X}$, (2) the centered response vector $\tilde{\mathbf{y}}$, and (3) the quantity $||y_i \times \mathbf{x}_i||_1$. The most expensive of these views to actually compute is the Gram matrix $\mathbf{X}^T\mathbf{X}$.

Aside from those three materialized views, the SimSQL implementation of the Bayesian Lasso utilizes three random tables: (1) the `beta[i]` table which stores the $\beta^{(i)}$ vector as a set of tuples, (2) the `sigma[i]` table (with one tuple) that stores the ith value of $(\sigma^{(i)})^2$, and (3) the `tau[i]` that stores one tuple for each $1/(\tau_j^{(i)})^2$ value. These `CREATE TABLE` statements for these three tables have a very close correspondence with the three different MCMC update steps described above. Consider `tau[i]`:

```
CREATE TABLE tau[i](rigid, tau Value) AS
  FOR EACH r IN regressor IDs
    WITH IG AS InvGaussian(
      (SELECT sqrt((pr1.lambda * pr1.lambda * s.sigma)
                   / (b.bet * b.bet))
       FROM prior pr1, sigma[i] s, beta[i-1] b
       WHERE b.rigid = r.grid),
      (SELECT (pr2.lambda * pr2.lambda)
       FROM prior pr2))
  SELECT r.grid, (1.0 / IG.out) FROM IG;
```

Here, the table `prior` has a single tuple that gives all of the prior hyperparameters.

6.3 GraphLab Implementation

Again, we use a super vertex-based implementation. The ith data vertex in the graph is made up of a large number of data points, forming a matrix $\mathbf{X}_i$ and vector $\mathbf{y}_i$. The jth model vertex consists of the inverse of the auxiliary variable $1/\tau_j^2$. We use one more vertex to store the regression coefficient vector β and the variance σ^2. This vertex sits in the center of the graph structure.

We begin using GraphLab's `map_reduce_vertices` method to obtain a couple of invariant statistics. While using an operation that measures the (non-existent!) global state of an asynchronous system might be problematic during the actual Markov chain simulation, it is a nice way to collect statistics before the simulation begins. We use one MapReduce to calculate the Gram matrix $\mathbf{X}^T\mathbf{X}$ and to center the response. Based on $\overline{y}$, the second MapReduce calculates $\mathbf{X}^T\tilde{\mathbf{y}}$ and stores it as a global variable.

Next we use GraphLab's gather-apply-scatter abstraction to simulate the required Markov Chain. The gather phase collects $\langle \beta_j, \sigma^2 \rangle$ for the j-th model vertex from the center vertex. The center vertex collects the vector of auxiliary variables $1/\tau^2$ from the model vertices, as well as $\sum_{\langle \mathbf{x}, y \rangle \in D} (\tilde{y} - \beta \cdot \mathbf{x})^2$ from the data vertices.

The apply phase then updates the center vertex by sampling new β and σ^2 values, and updates each model vertex by sampling $1/(\tau_j)^2$ from the inverse Gaussian distribution based on σ^2 and β_j.

6.4 Giraph Implementation

Our implementation uses three types of graph vertices: data vertices, dimensional vertices and a model vertex, which are used to store the dataset, collect required statistics such as the Gram matrix $\mathbf{X}^T\mathbf{X}$, and update the model (β, σ^2 and each τ_j^2), respectively. A dimensional vertex is associated with each data dimension.

The actual learning computation begins with the computation of $\mathbf{X}^T\mathbf{X}$ and $\mathbf{X}^T\tilde{\mathbf{y}}$. First, each data vertex computes $\mathbf{x}^T\mathbf{x}$ and sends this quantity along with $\mathbf{y}$ these to the other two types of vertices. The jth dimensional vertex collects the statistics for the jth row of the Gram matrix, while the model vertex computes $\tilde{\mathbf{y}}$. The Gram matrix is saved to the model vertex based on the messages passed from the dimensional vertices, while each data vertex computes $\mathbf{x}^T\tilde{\mathbf{y}}$, and sends it to the model vertex. The model vertex first initializes each $1/\tau_j^2$, σ^2, and β. The model vertex then broadcasts β to the data vertices so that they can compute $(\tilde{y} - \beta \cdot \mathbf{x})^2$. The model vertex obtains the sum of these values over all data vertices, and then updates each $1/\tau_j^2$, σ^2, and β. The updated β is then re-broadcast to all of the data vertices and the process is repeated. We make use of Giraph's combiner and aggregator facilities wherever possible to speed the computation and reduce communication.

6.5 Experiments and Results

To test these four implementations, we created a synthetic data set having 10^3 regressor dimensions and a one-dimensional response. As in GMM experiments, we held the number of data points per machine constant at 10^5 and tested compute clusters consisting of five, 20, and 100 machines. The results are shown in Figure 2. Giraph was unable to run without implementing the simulation using the super vertex construction.

6.6 Discussion

SimSQL, Spark, and Long Initialization Times. Note the substantial performance gap between SimSQL and Spark on one hand and GraphLab and Giraph on the other, considering the time to run the initialization code. Both of the former systems require between one and three hours for initialization/startup, while GraphLab takes under one minute. Normally, MCMC initialization time is not particularly important, since initialization is run only once. But in the case of the Bayesian Lasso, the simulation converges very quickly, magnifying the importance of the initialization.

SimSQL's relatively slow performance can be explained by its lack of support for vector and matrix operations. Consider the initialization phase of the simulation, which requires computing the Gram matrix $\mathbf{X}^T\mathbf{X}$ over the data set. In the case of SimSQL, the computation is performed as an aggregate-`GROUP BY` query, with one group for every one of the one million entries in the Gram matrix. For 100 machines, this means that SimSQL is computing one million aggregates over 10 million data points, meaning that it must aggregate 10^{13} tuples in all. All of those 10^{13} tuples are pushed through the SimSQL system itself, rather than being manipulated by a hand-built C++, Python, or Java code.

When one contrasts SimSQL's tuple-oriented implementation with GraphLab's super vertex code, it is easy to see why GraphLab is so fast. The ith super vertex in the GraphLab implementation has several thousand data points, which are stored in C++ as a matrix $\mathbf{X}_i$. This super vertex locally computes $\mathbf{X}_i^T\mathbf{X}_i$ using a fast matrix multiplication. At the same time, a similar computation is taking place all over the compute cluster. The resulting few thousand, million-entry matrices are then sent to a central location and added together.

SimSQL's Relatively High Per-Iteration Times. SimSQL is also slow on a per-iteration bases. SimSQL takes about ten times as long as Spark, 20 times as long as GraphLab, and five times as long as Giraph, per iteration. This means that SimSQL would spend around five hours to initialize and run twelve iterations, as opposed to two hours 15 minutes for Spark. Here again, SimSQL seems to suffer greatly from the fact that each $\mathbf{x}_i$ is stored as one thousand tuples rather than a single vector, and $\mathbf{A}^{-1}\mathbf{X}^T\tilde{\mathbf{y}}$ must be computed using set-oriented aggregates.

7. HIDDEN MARKOV MODEL

The next model we consider is a hidden Markov model (HMM) for text. This model and the associated MCMC simulation are somewhat more intricate than the previous two.

Assume that $\mathbf{x}_j$ is the ordered list or vector of words that make up the jth document. Each word $x_{j,k}$ in $\mathbf{x}_j$ is assumed to be produced by one of K hidden states. Hidden state s has an associated probability vector Ψ_s, where $\Psi_{s,w}$ is the probability that state s would produce word w. Since the model is sequential, there is also a state-to-state transition probability vector δ_s associated with state s. Thus, $\delta_{s,s'}$ is the probability of transitioning from state s to state s' when one moves to the next word in the document.

Let $\mathbf{y}_j$ be the vector of n state assignments associated with the jth document. To generate $\mathbf{y}_j$ using the HMM, we imagine that the start state $y_{j,1}$ for document j is first selected by sampling from a Categorical(δ_0) distribution (since there does not exist a zeroth state, δ_0 is used to control the start state of each document). Then, for each k in $2...n$, $\Pr[y_{j,k+1}|y_{j,k}] = \delta_{y_{j,k},y_{j,k+1}}$. Finally, we generate the ith document by sampling each $x_{j,k}$ from a Categorical($\Psi_{y_{j,k}}$) distribution. We put a Dirichlet(α) prior on each δ_s, and a Dirichlet(β) prior on each Ψ_s.

To learn this model, we utilize a simulation that updates every other state assignment in a given step. Specifically, if the current step i of the simulation is even and k is even, or if the current step ith is odd and k is odd, we have:

$$\begin{aligned}\Pr[y_{j,k}^{(i)} = s] &\propto \delta_{0,s}^{(i-1)} \times \Psi_{s,x_{j,k}}^{(i-1)} \times \delta_{s,y_{j,k+1}^{(i-1)}}^{(i-1)} \text{ if } k = 1 \\ &\propto \delta_{y_{j,k-1}^{(i-1)},s}^{(i-1)} \times \Psi_{s,x_{j,k}}^{(i-1)} \text{ if } k \text{ ends the document} \\ &\propto \delta_{y_{j,k-1}^{(i-1)},s}^{(i-1)} \times \Psi_{s,x_{j,k}}^{(i-1)} \times \delta_{s,y_{j,k+1}^{(i-1)}}^{(i-1)} \text{ otherwise.}\end{aligned}$$

Bayesian Lasso				
	lines of code	5 machines	20 machines	100 machines
SimSQL	100	7:09 (2:40:06)	8:04 (2:45:28)	12:24 (2:54:45)
GraphLab (Super Vertex)	572	0:36 (0:37)	0:26 (0:35)	0:31 (0:50)
Spark (Python)	168	0:55 (1:26:59)	0:59 (1:33:13)	1:12 (2:06:30)
Giraph	1871	Fail	Fail	Fail
Giraph (Super Vertex)	1953	0:58 (1:14)	1:03 (1:14)	2:08 (6:31)

Figure 2: Bayesian Lasso MCMC implementation; lines of code and average time per iteration. Format is HH:MM:SS or MM:SS.

Otherwise, the state assignment does not change in iteration i:

$$y_{j,k}^{(i)} = y_{j,k}^{(i-i)}$$

The other updates are as follows. Let $f(w,s) = \sum_{j,k} one(x_{j,k} = w \text{ and } y_{j,k}^{(i)} = s)$, where $one()$ returns 1 if the boolean argument is true and 0 otherwise. Similarly, $g(s) = \sum_j one(y_{j,0}^{(i)} = s)$ and $h(s,s') = \sum_{j,k} one(y_{j,k}^{(i)} = s \text{ and } y_{j,k+1}^{(i)} = s')$. Then:

$$\Psi_s^{(i)} \sim \text{Dirichlet}(\beta + \langle f(1,s), f(2,s), f(3,s)...\rangle)$$
$$\delta_0^{(i)} \sim \text{Dirichlet}(\alpha + \langle g(1), g(2), g(3)...\rangle)$$
$$\delta_s^{(i)} \sim \text{Dirichlet}(\alpha + \langle h(s,1), h(s,2), h(s,3)...\rangle)$$

7.1 Spark Implementation

We describe our document-based Spark implementation, where Spark is asked to manage and aggregate data at the document, rather than the word level. This Spark implementation begins by creating a RDD called `d_w_seq`. This RDD stores, for each document, the document identifier and its associated list of words. Then, a transformation is applied to randomly initialize the states, so the list of words in each document in `d_w_seq` is replaced by a list of (word, state) pairs. In the following code snippets, we use `d_id` and `s_id` to refer to the the document identifier and state identifier, and `state_size` to store the total number of states:

```
d_w_seq = lines.map(parseDoc).cache()
d_w_s_seq = d_w_seq.mapValues(lambda w_seq:
                init_state(w_seq, state_size))
```

Now it comes the main loop of Gibbs sampler. In each iteration, we first sample the state transition matrix δ by using two jobs. The first job computes, for each state s (used as the key), the total number of state transitions (s, s') for each target state s' that occurred in the previous iteration. These aggregates are then used to parameterize a Map-only job that updates the δ values:

```
h = d_w_s_seq.flatMap(lambda (d_id, w_s_seq):
     comp_h(w_s_seq, state_size)).reduceByKey
     lambda p1,p2: addStateCount(p1, p2, state_size))
delta = h.mapValues(lambda h_s: sample_delta(h_s,
     state_size))
```

Similarly, we use two jobs to sample Ψ. One more MapReduce job applies a self-transformation of `d_w_s_seq` to the states:

```
d_w_s_seq = d_w_s_seq.mapValues(lambda w_s_seq:
                update_state(w_s_seq, delta, psi))
```

Note that `update_state(.)` is a user-defined function, that alternatively updates the states for the even words and the odd words.

7.2 SimSQL Implementation

The word-based SimSQL HMM relies on four random tables: `states[i]`, `starts[i]`, `trans[i]`, and `emits[i]`. These four tables store all of the $\mathbf{y}_j^{(i)}$ vectors, the δ_0 vector, all of the $\delta_s^{(i)}$ vectors, and all of the $\Psi_s^{(i)}$ vectors, respectively.

At the heart of the SimSQL implementation is the `states[i]` table, with schema `(docID, prevPos, curPos, nextPos, prevStateID, curStateID)`. This table stores the current set of state assignments. If i is even, then for each (j,k) pair where k is even, there will be one tuple in this table that stores:

$$(j, k-1, k, k+1, y_{j,k}^{(i)}, y_{j,k+1}^{(i)})$$

If i is odd, then there will be one such tuple for each odd k.

To update each tuple to create the $(i+1)$th version of the `states` table, we would update the above tuple to produce:

$$(j, k-2, k-1, k, y_{j,k-1}^{(i+1)}, y_{j,k}^{(i+1)})$$

All of the new tuples produced in this way from `states[i]` then constitute the new contents of `states[i+1]`.

Performing this update requires (a) setting $y_{j,k}^{(i+1)}$ in the new tuple to be equal to $y_{j,k}^{(i)}$ in the old tuple, and (b) sampling a new value of $y_{j,k-1}^{(i+1)}$ using the `Categorical()` VG function. Performing this sampling in turn requires having access to three types of data, in addition to the tuple being updated: (1) the value for $y_{j,k-2}^{(i)}$, (2) all of transition probabilities out of state $y_{j,k-2}^{(i)}$, and (3) the probability that each and every possible state would emit word $x_{j,k-1}$. Gathering all of these values together actually requires a six-table join to parameterize the `Categorical` VG function.

The reader may wonder why we included all three of `prevPos`, `curPos`, and `nextPos` in each tuple in `states[i]`. Why not simply include `curPos`? The reason is a quirk in the SimSQL implementation. The six-table join required to parameterize the `Categorical` VG function requires join predicates of the form `t1.curPos = t2.curPos + 1`. The SimSQL optimizer has problems with this sort of equality predicate, implementing it inefficiently as a cross-product. Our way around this was to explicitly store `nextPos`, so the join could be written as `t1.curPos = t2.nextPos`, which is handled efficiently as an equi-join.

7.3 GraphLab Implementation

The graph we use for the HMM simulation contains only two types of vertices. Each data (super) vertex contains the $\mathbf{x}_j$ vectors for a large number of documents as well as the corresponding state assignment vectors. In addition, there is one state vertex associated with each of the K hidden states. The state vertex for state s stores the word emission probability vector Ψ_s and the state-to-state transition probability vector δ_s. This graph is complete and bipartite.

The GraphLab computation begins by initializing the state assignment for each word by performing a `transform_vertices` operation over all the data vertices. Then the gather-apply-scatter computation begins. In the gather phase, the data vertices collect Ψ_s and δ_s from all the state vertices, while each state vertex s computes $f(w,s)$, $g(s)$, $h(s,s')$ based on the words whose state as-

signments are also s from the data vertices. The *apply* phase simply updates $\mathbf{y}$, Ψ_s and δ_s according to the rules described above.

7.4 Giraph Implementation

Similarly, two types of vertices are used in the Giraph implementation. Since (unlike for GraphLab) we did not implement only a super vertex version of the Giraph simulation, depending upon the level of granularity, a data vertex could correspond to a word, a document, or a set of documents. Just as in the GraphLab implementation, a state vertex maintains Ψ_s and δ_s.

Consider the simple, word-based Giraph implementation, where each data vertex stores only a word. In this case, there is an edge between the vertex associated with each $x_{j,k}$ and the vertices associated with both $x_{j,k-1}$ and $x_{j,k+1}$. To begin the computation, the data vertices randomly initialize each $y_{j,k}$, and then send $y_{j,k}$ to their neighbors. All the data vertices then record their neighbor's states $y_{j,k-1}$ and $y_{j,k+1}$, and then send a word-count pair $\langle x_{j,k}, 1\rangle$ and a state-count pair $\langle y_{j,k+1}, 1\rangle$ to the $(y_{j,k})$th state vertex, where they are aggregated. To facilitate fast aggregation, we use the Giraph combiner mechanism. Next, the state vertices update their emission and transition probability vectors based on the collected statistics. Both probability vectors are broadcast back to the data vertices for updating their states.

7.5 Experiments and Results

To test the four implementations, we created a synthetic document database. To create each "document", we choose two newsgroup postings from the ubiquitous 20 newsgroups data set and concatenate the postings end-on-end. Since there are 20,000 posts in this data set, it was possible to create up to 400 million different synthetic documents in this way. We used a dictionary size of 10 thousand words and used $K = 20$ different states. The average document in the database was 210 words in length. As before, we tested the ability of each implementation to run on various compute cluster sizes, and we kept the amount of data on each machine constant. We used 2.5 million documents on each machine.

In our first experiment, we tested "word-based" HMM implementations built on top of Giraph, SimSQL, and Spark. Here "word-based" corresponds to the simulation where each word (with its associated hidden state) is individually pushed through the system. Since only SimSQL was able to handle this simulation (and SimSQL took more than eight hours per iteration), we only ran this simulation on five machines. The results are shown at the left in Figure 3 (a). Note the "NA" for the number of lines of code for Spark. We could not get Spark to perform the required self-join of the the set state assignments with itself without failing. Since this join is a prerequisite for running a"word-based" HMM, we did not implement the rest of the algorithm.

We also tested a "document-based" HMM implementation built on Giraph, SimSQL, and Spark; the experiment was run using five machines. In this implementation, the re-sampling of all of the hidden states for an entire document is handled as a group, and hand-coded in Java (Giraph), Python (Spark), or C++ (SimSQL). For all of the platforms, this has three main benefits. First, the state of the model (all of the $\Psi_s^{(i)}$ and $\delta_s^{(i)}$ vectors) need only be associated with each *document*, and not with each word individually. Second, the platform itself need not link up all of the adjacent words and states in order to re-sample each $y_{j,k}^{(i)}$. This linking can be done internally, within the user-supplied code. On all platforms except for SimSQL, the statistics necessary to update all of the $\Psi_s^{(i)}$ and $\delta_s^{(i)}$ vectors can be output at once on a per-document basis. The results are shown at the right in Figure 3 (a).

Finally, we wrote a "super vertex" HMM implementation for all four platforms where a large number of documents are grouped together, and the hidden states associated with all of the words for all of the documents are updated together. The results obtained by running the HMM super vertex implementation for each of the four platforms is shown in Figure 3 (b).

7.6 Discussion

The Giraph HMM Simulation Is Really Fast. This much is obvious; we consider some of the reasons now.

Giraph's Speed Relative to SimSQL. Even though the SimSQL super vertex implementation groups a large number of documents together, and then generates all of the $y_{j,k}^{(i)}$ vales associated with each word in each document in the group via a single C++ VG function invocation, all of those generated values must be output by the VG function as tuples. SimSQL must then aggregate all of the output tuples to compute the $f(.)$, $g(.)$, and $h(.)$ functions required by the simulation. This is very time-consuming. True, the process could likely be sped up considerably by doing pre-aggregation within the VG function (a similar tactic was used to make the SimSQL GMM super vertex simulation the fastest of all of the platforms) but this is a bit awkward because it requires encoding all of the output $y_{j,k}^{(i)}$ values plus all of the aggregates as a single output table. Giraph, on the other hand, need not output all of the $y_{j,k}^{(i)}$ values; they are stored internally, within the super vertex.

Comparing GraphLab and Giraph. Why was Giraph so much faster than GraphLab, and why does it scale better?

We tackle the second question first. All failures were memory-related. It is not that the data set itself is too large for GraphLab to handle. Our compute cluster has 7× the RAM needed to store the corpus. In practice, however, we were constantly struggling to overcome memory problems with GraphLab. It appears that this is related to the lack of control that a GraphLab programmer has over the way that data are materialized and moved around. Consider the problem of aggregating all of the data required to compute $f(w, s)$, $g(s)$, and $h(s, s')$ for a particular s—this aggregation is going to happen at each of the 20 graph vertices that correspond to the 20 hidden HMM states. Each super vertex will produce around 10MB of data that counts the number of times that each particular state-to-state transition happens as well as the number of times that each word is associated to each state. If this set of counts arrives at a state vertex from each of the 10,000 super vertices, we could easily end up having to materialize 100GB of data, killing the computation.

Giraph provides a richer programming interface that allowed us to sidestep some of the more serious computational and memory-related problems that are associated with mapping the simulation to a graph. For example, consider the aggregate computation described above. Giraph features graph-based aggregation functionality (with combiners) that offers a far faster (and safer) mechanism for gathering the required statistics.

8. LATENT DIRICHLET ALLOCATION

Latent Dirichlet Allocation (LDA) is a model for text mining. It views each word in a document as being produced by a latent *topic*, so that if a topic t produces the kth word in document j (denoted as $w_{j,k}$), then $\Pr[w_{j,k} = \omega] = \phi_{t,\omega}$, where ϕ_t represents the word distribution for topic t. Further, each document has a topic distribution, where the distribution for document j is a vector θ_j. Let $z_{j,k}$ indicate which topic produced word k in document j; then $\Pr[z_{j,k} = t] = \theta_{j,t}$. A Dirichlet($\alpha$) prior is put on each θ_j vector, and a Dirichlet(β) parameter is put on each ϕ_t vector.

Gibbs sampling is the typical way that an LDA model is learned.

(a) HMM: Word-based and document-based implementations				
	Word-based, 5 machines		Document-based, 5 machines	
	lines of code	running time	lines of code	running time
SimSQL	131	8:17:07 (10:51:32)	123	3:42:40 (20:44)
Spark (Python)	NA	Fail	214	4:21:36 (27:36)
Giraph	1717	Fail	1470	11:02 (7:03)

(b) HMM: Super Vertex Implementations				
	lines of code	5 machines	20 machines	100 machines
Giraph	1735	2:27 (1:12)	2:44 (1:52)	3:12 (2:56)
GraphLab	681	20:39 (16:28)	Fail	Fail
Spark (Python)	215	3:45:58 (11:02)	4:01:02 (13:04)	Fail
SimSQL	136	2:05:12 (1:44:45)	2:05:31 (1:44:36)	2:19:10 (2:04:40)

Figure 3: HMM results. Time in parens is for the initialization/setup. Format is HH:MM:SS or MM:SS.

The "collapsed" LDA Gibbs sampler is standard. Here "collapsed" means that one or more variables have been integrated out in the derivation of the Gibbs sampler. We choose, however, to include the non-collapsed sampler in our benchmark for two reasons. First, it is more interesting as a benchmark because it is a bit more complicated, including more parameters that must be computed. Second, there is the issue of correctness. It is very challenging to parallelize the collapsed LDA Gibbs sampler correctly because of the complex correlation structure that the collapsing induces among the updates to the various $\mathbf{z}_j$ vectors. Most parallel/distributed LDA Gibbs samplers ignore these correlations and update the vectors in parallel, disregarding the effect of the concurrent updates of the $\mathbf{z}_j$ vectors. We are uncomfortable with benchmarking an aggressive (and somewhat questionable) computational trick.

This simulation corresponding to the non-collapsed sampler is then defined as follows. Let $f(j,t) = \sum_k one(z_{j,k}^{(i)} = t)$ and let $g(t,\omega) = \sum_{j,k} one(w_{j,k} = \omega \text{ and } z_{j,k}^{(i)} = t)$. Then:

$$\Pr[z_{j,k}^{(i)} = t] \propto \theta_{j,t}^{(i-1)} \times \phi_{t,w_{j,k}}^{(i-1)}$$

$$\theta_j^{(i)} \sim \text{Dirichlet}(\alpha + \langle f(j,1), f(j,2), f(j,3), ...\rangle)$$

$$\phi_t^{(i)} \sim \text{Dirichlet}(\beta + \langle g(t,1), g(t,2), g(t,3), ...\rangle)$$

Lacking space, we do not describe the LDA implementations.

8.1 Experiments and Results

We used the document database used in the HMM experiments, a dictionary size of 10,000 words, and a model size of 100 topics.

The first thing that we wanted to do was to implement a "pure", word-based LDA. In this implementation, the $w_{j,k}$ and $z_{j,k}^{(i)}$ values are managed as individual elements by the underlying compute platform. However, since neither Spark nor Giraph were able to handle the word-based HMM simulation (and GraphLab was unable to handle a similar simulation for the relatively simple GMM inference problem), it seemed unlikely that we would be able to develop a word-based LDA for any of those platforms. Thus, we only implemented this version of the LDA simulation for SimSQL. The result of running this simulation on five machines (again with 2.5 million documents per machine) is shown in Figure 4 (a).

We next implemented a document-based LDA, where all of the $z_{j,k}^{(i)}$ values are re-sampled as a group, on a per-document basis. In SimSQL, these values still need to be output (and aggregated) as individual tuples, but the VG function for re-sampling all of the $z_{j,k}^{(i)}$ values for a single document need be parameterized only once, as a group. In Spark, it is necessary to output only a single $\mathbf{z}_j^{(i)}$ vector for a document, which should result in an even more efficient sampler than the one written in SimSQL. Giraph should be more efficient still, since they never need to output any $z_{j,k}^{(i)}$ values; these an be maintained internally, in memory, within a graph vertex. The results are also shown in Figure 4 (a).

Finally we implemented a super vertex version of the code four each of the four platforms. The results are shown in Figure 4 (b).

8.2 Discussion

Everyone Fails Except for SimSQL. The LDA simulation is quite similar to the HMM simulation, particularly the document-based and super vertex versions. The big difference is that since there are 100 topics, the size of the model that must be learned (and the size of the set of statistics computed) is around five times as large as in the case of the HMM. This appears to makes the task a bit more difficult, especially for Giraph. Our Giraph LDA implementation ran about ten times longer than the HMM implementation, and, perhaps most significantly, failed to run at all on 100 machines. SimSQL was the only platform that was able to run the LDA simulation on 100 machines and 250 million documents.

9. GAUSSIAN IMPUTATION

The last model we consider is the most complicated of the five models, though not the most difficult model to handle computationally: a GMM modified to impute missing values in the data. The MCMC simulation to learn the GMM and simultaneously use it to impute the missing null values is similar to the GMM simulation, but with one additional step where we must estimate the censored values. For the jth data point, let $\mathbf{x}_j^1$ denote the vector of censored values from $\mathbf{x}_j$ and let $\mathbf{x}_j^2$ denote the vector of un-censored values. Then $\mathbf{x}_j^1 \bullet \mathbf{x}_j^2$ is equivalent to $\mathbf{x}_j$, with the dimensions "scrambled" so that the first few dimensions were the ones that were censored. Let μ_j denote mean vector of the cluster that (according to $\mathbf{c}_j^{(i-1)}$) produced the jth data point, and let Σ_j denote the corresponding covariance matrix. Re-arrange and partition the dimensions of μ_j and Σ_j so that it is the case that:

$$\mathbf{x}_j^1 \bullet \mathbf{x}_j^2 \sim \text{Normal}\left(\begin{bmatrix}\mu_j^1\\ \mu_j^2\end{bmatrix}, \begin{bmatrix}\Sigma_j^{11}\Sigma_j^{12}\\ \Sigma_j^{21}\Sigma_j^{22}\end{bmatrix}\right)$$

We add an additional step to the GMM simulation. For each $\mathbf{x}_j^1$:

$$(\mathbf{x}_j^1)^{(i)} \sim \text{Normal}\left(\mu_j^1 + \Sigma_j^{12}(\Sigma_j^{22})^{-1}(\mathbf{x}_j^2 - \mu_j^2), \Sigma_j^{11} - \Sigma_j^{12}(\Sigma_j^{22})^{-1}\Sigma_j^{21}\right)$$

As was the case with LDA, space precludes us from describing each of our Gaussian imputation implementations in detail.

(a) LDA: Word-based and document-based implementations				
	Word-based, 5 machines		Document-based, 5 machines	
	lines of code	running time	lines of code	running time
SimSQL	126	16:34:39 (11:23:22)	129	4:52:06 (4:34:27)
Spark (Python)	NA	NA	188	≈15:45:00 (≈2:30:00)
Giraph	NA	NA	1358	22:22 (5:46)

(b) LDA: Super Vertex Implementations				
	lines of code	5 machines	20 machines	100 machines
Giraph	1406	18:49 (2:35)	20:02 (2:46)	Fail
GraphLab	517	39:27 (32:14)	Fail	Fail
Spark (Python)	220	≈3:56:00 (≈2:15:00)	≈3:57:00 (≈2:15:00)	Fail
SimSQL	117	1:00:17 (3:09)	1:06:59 (3:34)	1:13:58 (4:28)

Figure 4: LDA results. Time in parens is for the initialization/setup. Format is HH:MM:SS or MM:SS.

9.1 Experiments and Results

We used the ten-dimensional data from the GMM experiments, but performed a bit of additional post-processing. For each data point, we took a sample $p \sim \text{Beta}(1, 1)$, which gave us a probability p of between 0 and 1, with a 0.5 average. Each of the ten attribute values within the data point were then censored by flipping a synthesized coin which came up heads with probability p. If a heads was observed, the value was replaced with a $\varnothing$. In this way, 50% of the attribute values in the data set were censored.

As before, we kept the amount of data constant at 10 million data points per machine, and tested the per-iteration running times at five, 20, and 100 machines. The results are shown in Figure 5.

9.2 Discussion

Spark's Increase in Running Time. What struck us as remarkable here was how these results look almost exactly the same to the GMM results, with the exception of a very significant running time increase for Spark. The reason may be that in the case of the GMM, we can store all of the data in memory using the `cache()` function. However, in the imputation model, the actual data set changes constantly as imputation is being performed, which might affect Spark's performance.

10. SUMMARY OF FINDINGS

We end the paper by sorting through the various results.

Giraph and GraphLab. We begin by discussing the two graph-based platforms: Giraph and GraphLab.

On the positive side, they are both quite fast—typically faster than Spark and SimSQL. However, both Giraph and GraphLab suffer greatly from memory-related difficulties, and could not be made to run on the largest, most complicated problems. The data sets were not too large to fit into RAM. Our largest data set was around 1TB in size, which fits comfortably in the 7TB of aggregate RAM of our 100 machine compute cluster. In fact, our experiments demonstrate how the constant mantra that "memory is the new disk" must be applied carefully. True, there are few problems (even "big data" problems) where the raw data cannot fit into RAM, but ML inference problems typically require the computation of large sets of complicated statistics, and the computation can temporarily blow up the size of the data by orders of magnitude. All of that data has to be managed.

GraphLab's asynchronous, pull-based programming model proved to be a problematic mechanism for collecting such statistics (see the detailed discussion in the section on HMM inference), and GraphLab was generally unsuitable for large problems. Giraph was typically

LDA Spark Java Implementation			
lines of code	5 machines	20 machines	100 machines
377	9:47 (0:53)	19:36 (1:15)	Fail

Figure 6: Average time per iteration (and startup time).

faster and more widely applicable than GraphLab, seemingly because Giraph's synchronous programming model permits a much richer API than GraphLab's, including mechanisms (such as combiners for performing distributed aggregation) that can be used to overcome some of the memory-related problems that plagued the graph-based platforms. The downside is that Giraph had large and complex codes. In contrast, GraphLab codes were small and elegant, especially considering that the programming language is C++. It will be a challenge to maintain this elegance and yet make GraphLab more suitable for large and complex problems.

SimSQL. SimSQL was often—though not always—slower than Giraph and GraphLab. SimSQL's purely tuple-oriented approach can hurt its performance, because (for example) a 1,000 by 1,000 matrix is pushed through the system as a set of one million tuples. This can result in some very long compute times—see the section on the Bayesian Lasso (though, interestingly, SimSQL's Gram matrix computation times were only a bit slower than Spark's). It seems that this is a serious issue with SimSQL that needs to be addressed.

On the positive side, SimSQL was consistently able to perform computations that none of the other three platforms could run, and was the only platform that never failed to run any of the computations tested. All of the other platforms were clearly at or beyond their limit on the largest problems, whereas SimSQL appeared ready to scale up further. This robustness is due to the fact that under the hood, SimSQL looks a lot like a parallel database system—it is well-understood how to scale such a system.

Spark. On the positive side, Spark codes (particularly those written in Python) are incredibly short and beautiful. Spark's succinctness rivals that of SimSQL's SQL codes, though many users will find Spark codes to be preferable. They are imperative/functional compared to SimSQL's SQL-based approach, which many will be uncomfortable with.

On the negative side, Spark was slower than the two graph-based platforms, and it was the slowest platform of the four on the largest and most complicated models. We were a bit worried these results were related to our choice of Python instead of Scala or Java. We had already tested both Python and Java GMM implementations, but just to be sure, we tested a Java Spark LDA implementation

Gaussian Imputation				
	lines of code	5 machines	20 machines	100 machines
Giraph	2274	28:43 (0:19)	31:23 (0:18)	Fail
GraphLab (Super vertex)	1197	6:59 (3:41)	6:12 (8:40)	6:08 (3:03)
Spark (Python)	294	1:22:48 (3:52)	1:27:39 (4:03)	1:29:27 (4:27)
SimSQL	182	28:53 (14:29)	30:41 (15:30)	39:33 (22:15)

Figure 5: Gaussian imputation results. Time in parens is for the initialization/setup. Format is HH:MM:SS or MM:SS.

as well. The results are shown in Figure 6. The speed is much better than the Python implementation, but we could still not get Spark to run the LDA inference algorithm on 100 machines. The implementation failed on 20 machines after 18 iterations as well.

We debated whether our difficulties with Spark were the result of simple engineering problems that could easily be fixed, or whether the issue is more fundamental. For example, Spark appears to rely greatly upon "lazy evaluation" for speed and job scheduling, which (in the context of large-scale ML inference) looks a lot like pipelining in a database system. However, a database system uses complex models and statistical information describing the data to decide when and how to pipeline. Without this, it is easy to make bad decisions, pipelining operations that together consume far more than the resources available. We also observed a very large variance in running times across MCMC iterations (and sometimes running times that increased from iteration to iteration), and worried that Spark was making poor choices in terms of which RDDs to persist and discard—in effect discarding RDDs that needed to be recomputed a few minutes later. We were able to alleviate some of these problems through careful tuning and forcing writes of RDDs to disk. Perhaps the ultimate solution is to make Spark work more like a database system, carefully planning computational choices such as RDD materialization and pipelining using cost models.

Final Thoughts. Forced to distill our findings into one, succinct statement, we give the following: The best platform depends upon the model and the data set; no single platform dominated, and each was, under certain circumstances, an abysmal choice. We can confidently say that developing the ultimate platform for large-scale ML is not a solved problem. While the platforms are all useful, it was never as easy as writing some code and pressing return. Tuning and trial and error are required, and it may be necessary to try more than one platform to get things to work at scale.

11. REFERENCES

[1] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. *arXiv preprint arXiv:1110.4198*, 2011.

[2] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[3] D. M. Blei. Probabilistic topic models. *Communications of the ACM*, 55(4):77–84, 2012.

[4] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued markov chains using simsql. In *SIGMOD*, pages 637–648, 2013.

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

[6] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[7] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter. *Markov chain Monte Carlo in practice*, volume 2. CRC press, 1996.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[9] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *PVLDB*, 5(12):1700–1711, 2012.

[10] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[11] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[12] 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*, pages 135–146, 2010.

[13] A. McCallum. Mallet: A machine learning for language toolkit. *http://mallet.cs.umass.edu/*, 2002.

[14] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455, 2013.

[15] S. Owen, R. Anil, T. Dunning, and E. Friedman. *Mahout in action*. Manning, 2011.

[16] T. Park and G. Casella. The bayesian lasso. *JASA*, 103(482):681–686, 2008.

[17] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *CloudCom*, pages 721–726, 2010.

[18] J. A. Snyman. *Practical mathematical optimization*, volume 97. Springer, 2005.

[19] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun. Optiml: an implicitly parallel domain-specific language for machine learning. In *ICML*, pages 609–616, 2011.

[20] M. Weimer, T. Condie, R. Ramakrishnan, et al. Machine learning in scalops, a higher order cloud computing language. In *BigLearn*, volume 9, pages 389–396, 2011.

[21] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX*, pages 2–2, 2012.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX*, pages 10–10, 2010.