Benchmarking Data Flow Systems for Scalable Machine Learning

Christoph Boden, Andrea Spina*, Tilmann Rabl, Volker Markl
TU Berlin
firstname.lastname@tu-berlin.de

ABSTRACT

Distributed data flow systems such as Apache Spark or Apache Flink are popular choices for scaling machine learning algorithms in production. Industry applications of large scale machine learning such as click-through rate prediction rely on models trained on billions of data points which are both highly sparse and high-dimensional. Existing Benchmarks attempt to assess the performance of data flow systems such as Apache Flink, Spark or Hadoop with non-representative workloads such as *WordCount, Grep* or *Sort*. They only evaluate scalability with respect to data set size and fail to address the crucial requirement of handling high dimensional data.

We introduce a representative set of distributed machine learning algorithms suitable for large scale distributed settings which have close resemblance to industry-relevant applications and provide generalizable insights into system performance. We implement mathematically equivalent versions of these algorithms in *Apache Flink* and *Apache Spark*, tune relevant system parameters and run a comprehensive set of experiments to assess their scalability with respect to both: data set size and dimensionality of the data. We evaluate the systems for data up to four billion data points and 100 million dimensions. Additionally we compare the performance to single-node implementations to put the scalability results into perspective.

Our results indicate that while being able to robustly scale with increasing data set sizes, current state of the art data flow systems are surprisingly inefficient at coping with high dimensional data, which is a crucial requirement for large scale machine learning algorithms.

CCS Concepts

•Theory of computation → Massively parallel algorithms; MapReduce algorithms; •Software and its engineering → Data flow architectures;

*contributed while at TU Berlin, now at Radicalbit S.r.l.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org

BeyondMR'17 May 19, 2017, Chicago, IL, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-5019-8/17/05.
DOI: http://dx.doi.org/10.1145/3070607.3070612

1. INTRODUCTION AND MOTIVATION

In Big Data Analytics, the *MapReduce* [11] paradigm, popularized by its open-source implementation *Hadoop* [3] has been widely adopted as a solution to robustly scale data-intensive applications to very large data sets on clusters of commodity hardware. However it has also been established that Hadoop MapReduce is inefficient at executing iterative workloads such as distributed machine learning algorithms. [25, 15] This sparked the development of a multitude of novel approaches and systems aiming to improve the performance and ease of implementation of more complex workloads such as distributed machine learning algorithms. We consider the two most prominent representative systems which managed to morph from research prototypes into production systems enjoying widespread adoption in industry:

Apache Spark [5] introduced the concept of data-parallel transformations on Resilient Distributed Datasets (RDDs) [29]: read-only collections of data partitioned across nodes, which can be cached and recomputed in case of node failures, to support the efficient execution of iterative algorithms.

Apache Flink [2, 10] (formerly *Stratosphere* [6]) introduced a general data flow engine supporting the flexible execution of a more rich set of operators such as *map, reduce and co-group* as well as a native operator for iterative computations. Flink jobs are compiled and optimized by a cost-based optimizer before being scheduled and executed by the distributed streaming data flow engine. This distributed runtime allows for pipelining of data.

While these second generation big data analytics systems have been shown to outperform *Hadoop* for canonical iterative workloads[29, 12], it remains an open question how they perform in executing large scale machine learning algorithms.

Consider the prominent problem of *click-through rate prediction* for online advertisements, a crucial building block in the multi-billion dollar online advertising industry, as an example for large scale machine learning. To maximize revenue, platforms serving online advertisements must accurately, quickly and reliably predict the expected user behaviour for each displayed advertisement. These prediction models are trained on hundreds of terabytes of data with hundreds of billions of training samples . The data tends to be very sparse (10-100 non-zero features) but at the same time very high dimensional (up to 100 billion unique features [9]. For this important problem, algorithms such as regularized logistic regression are still the method of choice [21, 24]).

In the context of scalable, distributed machine learning, there are thus multiple dimensions of scalability which are of particular interest:

1. **Scaling the Data:** scaling the training of (supervised) machine learning models to extremely large data sets (in terms of the number of observations contained) is probably the most well established notion of scalability in this context as it has been shown that even simple models can outperform more complex approaches when trained on sufficiently large data sets [13, 7].
2. **Scaling the Model Size:** many large scale machine learning problems exhibit very high dimensionality. For example, classification algorithms that draw on textual data can easily contain 100 million dimensions or more, models for click-through rate prediction for online advertisements can reach up to 100 billion dimensions [9]. For these use cases, being able to efficiently handle high dimensional models is a crucial requirement as well.
3. **Scaling the Number of Models:** To tune hyperparameters many models with sightly different parameters are usually trained in parallel.

Ideally a system suited for scalable machine learning should efficiently support all three of these dimensions. However since scaling the number of models to be trained in parallel is essentially an embarrassingly parallel problem[1], we focus on the first two aspects: *scaling the data* and *scaling the model dimensionality* in our experiments.

We introduce a representative set of distributed machine learning algorithms suitable for large scale distributed settings comprising logistic regression and k-means clustering, which have close resemblance to industry-relevant applications and provide generalizable insights into system performance. We implement mathematically equivalent versions of these algorithms in *Apache Flink* and *Apache Spark*, tune relevant system parameters and run a comprehensive set of experiments to assess their performance. Additionally we explore efficient single-node and single threaded implementations of these machine learning algorithms in order to investigate the overhead incurred due to the use of the JVM and Scala as well as the distributed setting and to put the scalability results into perspective as has been suggested by [22].

Contrary to existing Benchmarks, which assess the performance of Flink, Spark or Hadoop with non representative workloads such as *WordCount, Grep* or *Sort*, we evaluate the performance of these systems for scalable machine learning algorithms.

Contributions. In order to solve problem of how to objectively and robustly assess and compare the performance of distributed data processing platforms for machine learning workloads, we present the following major contributions:

1. We present a Distributed Machine Learning Benchmark for distributed data analytics systems, an in-depth description of the individual algorithms, metrics and experiments to assess the performance and scalability characteristics of the systems for representative machine learning workloads as well as a detailed analysis and discussion of a comprehensive experimental evaluations.
2. To ensure reproducibility we provide our benchmark algorithms on top of *Apache Flink* and *Apache Spark* as open-source software [2] implemented using the *Peel* framework [1] for defining and executing experiments for distributed systems and algorithms. By providing simple reference implementations of a small set of core algorithms, we want to make it easier for new software-frameworks to compare themselves to existing frameworks.
3. The results of our experimental evaluation indicate that while being able to robustly scale with increasing data set sizes, current state of the art data flow systems for distributed data processing such as Apache Spark or Apache Flink struggle with the efficient execution of machine learning algorithms on high dimensional data, an issue which clearly deserves further investigation.

[1]We do acknowledge that there exists significant optimization potential in this dimension as well, as has recently been pointed out by Kumar et al *Model Selection Management* [16]. However, this requires the adaption if not redesign of the data processing systems and is thus out of the scope of this paper

2. OUTLINE

The rest of this paper is structured as follows: in Section 3 we provide the necessary systems background about Apache Spark and Apache Flink as well as important parameters that have to be set and tuned in each system. in Section 4 we present a detailed discussion of the chosen machine learning workloads and their implementations in the data flow systems. Section 5 introduces the metrics and experiments that constitute the Benchmark and Section 6 provides concrete results and a discussion of the comprehensive experimental evaluation of the benchmark workloads and systems under evaluation. In Section 7 we discuss related work in the area of benchmarking distributed data processing systems before we conclude and summarize our findings in Section 8

3. SYSTEMS BACKGROUND

In this section we provide a brief overview of the different systems under evaluation as well as relevant parameter settings.

3.1 Apache Spark

Apache Spark is a distributed big data analytics framework centered around the concept of Resilient Distributed Datasets (RDDs)[29]. A RDD is a distributed memory abstraction in the form of a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost, thus providing fault tolerance.

RDDs provide an interface based on transformations (e.g., `map()`, `filter()` or `join()`) and actions. Transformations on RDDs are lazily evaluated, thus computed only when needed e.g. by an action and can be pipelined. RDD actions (e.g. `count()`, `reduce()`) trigger the computation and thus execution of transformations on RDDs. Fault tolerance is provided by logging the transformations used to build a dataset (its lineage) rather than the actual data. If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[2]`https://github.com/bodenc/ml-benchmark`

just that partition. Whenever a user runs an action on an RDD, the Spark scheduler examines that RDD's lineage graph to build a directed acyclic graph (DAG) of so called stages to execute.

Users can control two main aspects of RDDs: persistence and partitioning: Users can indicate which RDDs they will reuse and would thus like to persist in memory and choose a `StorageLevel` for them (e.g., `MemoryOnly`). Spark keeps persistent RDDs in memory by default, but it can spill them to disk if there is not enough RAM. Users can also force a custom partitioning to be applied to an RDD, based on a key in each record.

3.2 Apache Flink

Apache Flink, formerly known as Stratosphere [6, 10], is essentially a streaming data flow engine designed to process both stream and batch workloads. The batch processing part is centered around the concept of a `DataSet` - a distributed collection comprising the elements of the data set to be processed. Users can specify functional transformations on these `DataSets` e.g. `map()`, `fiter()`, `reduce()`.

Flink programs are also executed lazily: the data loading and transformations do not happen immediately. Rather, each operation is created and added to the program's plan. The operations are only executed when one of the `execute()` methods is invoked on the `ExecutionEnvironment` object.

Analogous to query optimization in databases, the program is transformed to a logical plan and then compiled and optimized by a cost-based optimizer, which automatically chooses an execution strategy for the program based on various parameters such as data size or number of machines in the cluster. The final physical plan is then scheduled and executed by the distributed streaming data flow engine, which is capable of pipelining the data.

Apache Flink does not allow the user to specify `DataSets` to be cached in memory, but it does provide its very own native iterations operator, for specifying iterative algorithms. The Flink optimizer detects this and adds caching operators to the physical plan, ensuring that loop-invariant data is not re-read from the distributed file system in each iteration. In contrast, Spark implements iterations as regular for-loops and executes them by loop unrolling.

3.3 Parameter Configuration

While Apache Flink and Spark are both data flow systems, the architecture and configuration settings that have to be set and potentially tuned by the user differ quite substantially between the two systems.

Parallelism. In a *Flink* cluster, each node runs a `TaskManager` with a fixed number of processing slots, generally proportional to the number of available CPUs per node. Flink executes a program in parallel by splitting it into subtasks and scheduling these subtasks to individual processing slots. Once set, the number of slots serves as the maximum of possible parallel tasks and is used as the default parallelism of all operators. We follow the Flink recommendation[3] and set the number of task slots equal to the number of cores available in the cluster. This generally triggers an initial *repartitioning* phase in a job, as the number of HDFS blocks is rarely equivalent to the desired number of subtasks.

[3]https://ci.apache.org/projects/flink/flink-docs-release-1.0/setup/config.html#configuring-taskmanager-processing-slots

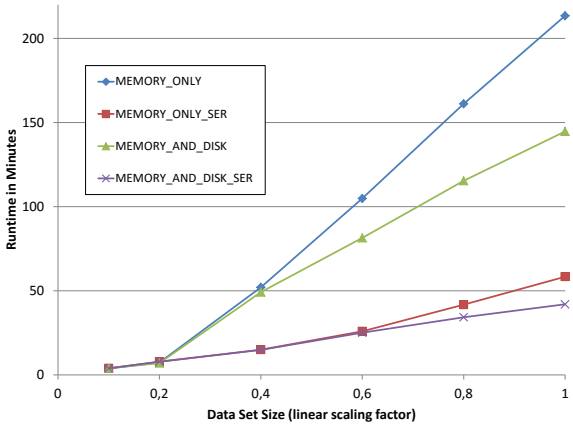

Figure 1: L2 regularized logistic regression training in Apache Spark with increasing data set size for a fixed number of nodes and different RDD StorageLevels.

In *Spark*, each worker node runs `Executors` with the ability to run `executor.cores` number of tasks concurrently. The actual degree of parallelism (number of tasks per stage) is furthermore determined by the number of partitions of the RDD (number of HDFS blocks the input data set by default), where the resulting parallelism is given by:

$$min\,(numExecutors \times coresPerExecutor, numPartitions)$$

Following the Spark recommendation[4] we set `executor.cores` equal to the number of cpu cores available in the cluster and set the parallelism (number of RDD partitions) to 3 times the number of CPU cores available in the cluster.

Caching. Contrary to Flink, Spark allows for the explicit caching of RDDs in Memory. For this, the user can choose one of four different `Storage Levels`:

`MEMORY_ONLY` stores the RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they are needed.

`MEMORY_AND_DISK` stores the RDD as deserialized Java objects in the JVM. However, if the RDD does not fit in memory, partitions that do not fit are stored on disk, and read from there when ever they are needed.

`MEMORY_ONLY_SERIALIZED`: the RDD is stored as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects but more CPU-intensive to read.

`MEMORY_AND_DISK_SERIALIZED`: the RDD is stored as serialized Java objects (one byte array per partition), but partitions that do not fit into memory are spilled to disk instead of recomputing them on the fly each time they are needed. Note that since the partitions which are spilled to disk are also written out in serialized form, the disk footprint is smaller than in the `MEMORY_AND_DISK` case.

In order to understand the impact of the different `Storage Levels` for a typical machine learning workload, we run ten iterations of gradient descent training of a l2 regularized logistic regression model (details in Section 4) on the criteo

[4]https://spark.apache.org/docs/latest/tuning.html#level-of-parallelism

data set (details in Section 5.4) for different `StorageLevel` settings on 30 compute nodes (details in Section 5.3).

Figure 1 shows the runtime results of our evaluation for increasing input data set sizes. It is apparent that the RDDs no longer fit into the combined memory of the cluster for the two non-serialized `StorageLevels` above a data set size of *0.2*. Performance significantly degrades, as partitions that do not fit into memory have to be re-read from disk or re-computed, where re-computation (`MEMORY_ONLY`) seems to be more expensive than re-reading partitions from disk (`MEMORY_AND_DISK`). The two serialized strategies show significantly better performance after a data set size of *0.2*, as the serialized RDD partitions are more compact and still fit into the combined memory up until a data set size of *0.6*. Beyond this point, partitions have to be re-read from disk or re-computed as well, where once again the `StorageLevel` relying on re-reading partitions from disk performs slightly better than the one that recomputes partitions that do not fit into memory. Based on these results we chose (`MEMORY_AND_DISK_SERIALIZED`) as the `StorageLevel` for all subsequent benchmark experiments. It consistently outperforms all other ones, except for very small data set sizes (data set size *0.1 - 0.2*) where it still shows comparable performance to the non-serialized `StorageLevels`.

Apache Flink does not allow the user to cache `DataSets` explicitly, but provides a native iteration operator which prompts the optimizer to cache the data. We thus implemented all benchmark algorithms with this operator.

Buffers. Network buffers are a critical resource of the communication. They are used to buffer records before transmission over a network, and to buffer incoming data before dissecting it into records and handing them to the application. In *Flink* the user can adjust both the number and size of the buffers. While Flink suggests[5] to use the approximately

$$numCores^2 \times numMachines \times 4$$

buffers, we encountered that a higher setting is advisably for machine learning workloads.

Serialization By default, Spark serializes objects using the java serialization framework, however Spark can also use the `Kryo` library to serialize objects more quickly when classes are registered. Flink on the other hand comes with its own custom serialization framework which attempts to assess the data type of user objects with help of the scala compiler and represent it via `TypeInformation`. Each `TypeInformation` provides a serializer for the data type it represents. For any data type that cannot be identified as another type, Flink returns a serializer that delegates serialization to `Kryo`. In order to ensure a fair assessment of the Systems under test, decided to force both systems to use `Kryo` as a serializer and provided custom serialization routines for the data points in both Spark and Flink.

4. BENCHMARK WORKLOADS

In this section we outline the main algorithms that constitute the benchmark workloads. As was laid out in the introduction, our goal is to provided a fair and insightful Benchmark which reflects the requirements of real-world machine learning applications that are deployed in production and generates meaningful results.

[5]https://ci.apache.org/projects/flink/flink-docs-release-1.0/setup/config.html#configuring-the-network-buffers

4.1 Supervised learning

The goal in supervised learning is to learn a function f_w which can accurately predict the labels $y \in Y$ for data points $x \in X$ given a set of labeled training examples (x_i, y_i). The actual task of learning a model is to fit the parameters w of the function f_w based on the training data and a loss function $l(f_w(x), y)$. To avoid overfitting, a regularization term $\Omega(w)$ that captures the model complexity is added to the objective. Different parametrizations of the components f_w, $l(f_w(x), y)$ and $\Omega(w)$ yield quite a variety of different supervised learning algorithms including SVMs, LASSO and RIDGE regression as well as logistic regression. For the important problem of *click-through rate prediction* for online advertisements, algorithms such as regularized logistic regression are still the method of choice [21, 24].

Solvers. The most commonly used loss functions happen to be both convex and differentiable, which guarantees the existence of a minimizer $\hat{w}$. It also enables the application of batch gradient-descent (BGD) as a solver. This algorithm performs the following step using the gradient of the loss until convergence:

$$w' = w - \left(\lambda \frac{\partial}{\partial w} \Omega(w) + \sum_{(x,y) \in (X,Y)} \frac{\partial}{\partial w} l(f_w(x), y) \right)$$

We choose and implemented this solver, because it actually represents the data flow and I/O footprint exhibited by a wide variety of (potentially more complex) optimization algorithms such as *L-BFGS*[19] or *TRON*[18].[6]

Implementation Rather than depending on existing machine learning library implementations, we implement all learning algorithms from scratch, in order to ensure that we analyze the performance of the underlying systems and not implementation details. As a common linear algebra abstraction we use the Breeze library [7] for numerical processing.

In *Flink* we implement batch gradient descent as `MapPartition` functions, which compute the individual BGD updates and pre-aggregate partial sums of gradients, which are ultimately summed up in a global reduce step. This turns out to be the more performant alternative to using a `map()` to compute the gradients and summing them up in a subsequent `reduce()` step during experimental evaluation (See Figure 2). To efficiently iterate over the training data set, we utilize Flink's batch `iterate()` operator, which feeds data back from the last operator in the iterative part of the data flow to the first operator in the iterative part of the data flow and thus attempts keep loop-invariant data in memory. The model vector is distributed to the individual tasks a broadcast variable.

In *Spark* we leverage the `TreeAggregate()` to perform the batch gradient descent computation and update, aggregating the partial updates in a multi-level tree pattern. The model vector is also distributed to the individual tasks as a broadcast variable. This turnes out to be more robust for

[6]The recently proposed algorithm *HOGWILD!* [23] suggests asynchronous stochastic gradient descent (SGD) solvers implemented without any locking but rather permitting conflicting model updates still converge and thus provide a more performant alternative to batch-type solvers. However neither Apache Spark nor Apache Flink are able to train models asynchronously, thus we do not consider this approach.

[7]`https://github.com/scalanlp/breeze`

higher dimensionalities than a `MapPartition` implementation and more performant than a `map() - reduce()` implementation (See Figure 2).

4.2 Unsupervised learning

For unsupervised learning we choose to implement the popular *k-means* clustering algorithm, which solves the following objective:

$$min \sum_{j=1}^{k} \sum_{i \in C} ||x_i - \mu_j||^2$$

with the heuristic where k cluster centers are sampled from the data set, the distance to each of these centroids, where μ_j is the centroid of the j-th cluster, is computed for each data point, every data point is assigned to its closest centroids, and the centroids subsequently updated. While also exhibiting the iterative nature like the supervised learning workload, *k-means* evaluates the effectiveness of the `reduceByKey()` operator in Flink and the `groupBy()` and `reduce()` operator in Spark. Furthermore *k-means* is part of most related work.

5. BENCHMARK DIMENSIONS AND SETTINGS

In this Section, we present the data generation strategies, data sets, experiments and measurements that constitute the Benchmark. Furthermore we provide the specification of the hardware we relied upon for our experimental evaluation.

5.1 Scalability

Traditionally, in the context of high performance computing (HPC), scalability is evaluated in two different notions:

Strong Scaling: is defined as how the runtime of an algorithm varies with the number of nodes for a fixed total problem size.

Weak Scaling: is defined as how the runtime of an algorithm varies with the number of nodes for a fixed problem size per node, thus a data size proportional to the number of nodes.

While these metrics have their merit in the evaluation of scalability of distributed algorithms on distributed systems, when it comes to scaling machine learning algorithms on distributed systems for real world uses cases, two other aspects become the primary concern, namely:

Scaling the Data: How does the algorithm runtime behave when the size of the data (number of data points) increases?

Scaling the Model: How does the algorithm runtime behave when the size of the model (number of dimensions) increases?

The main motivation for introducing distributed processing systems into production environments is usually the ability to robustly scale an application with a growing production workload (e.g. growing user base), by simply adding more hardware nodes. However in the short run, the hardware setup is usually fixed (assuming an on-premise solution). We thus need to introduce two new experiments to adequately capture the desired scaling dimensions *data* and *model*:

Experiment 1: Production Scaling: Measure the runtime for training a model while varying the size of the training data set for a fixed cluster setup (model size fixed)

criteo part	num data points	raw size in GB
day0	195,841,983	46.35
day1	199,563,535	47.22
day2	196,792,019	46.56
day3	181,115,208	42.79
day5	172,548,507	40.71
day6	204,846,845	48.50
total	**1,150,708,097**	**272.14**

Table 1: Subset of the criteo data set used in the experiments.

Experiment 2: Model Dimensionality Scaling: Measure the runtime for training a model on a fixed size cluster setup (training data set size fixed)

In practice the ability to scale the **number of models** i.e. to evaluate different hyperparameter settings is also a relevant dimension, however since this is essentially an embarrassingly parallel task, we consider it outside the scope of this benchmark.

5.2 Absolute Runtime and COST

Next to analyzing the scalability properties of the systems under test, we also measure and report the absolute runtimes for a fixed data set size and compare these to the runtime of single machine and single threaded implementations. McSherry et.al. [22] introduced a new metric, called *COST (the Configuration that Outperforms a Single Thread)*, that describes the point when a distributed solution outperforms a (competent) single threaded implementation. [8] Motivated by this example, we also consider efficient single threaded implementations of supervised machine learning algorithms as s COST baseline, thereby providing a COST metric for machine learning algorithms. We choose the *LibLinear*[9] solver as an efficient C++ single threaded implementation.

Experiment 3: Measure the runtime for training a model while varying the number of machines and model size (keeping the size of the training data set and) as well as the runtime of a *competent single-threaded implementation*

Model Quality. As we focus on the training phase to compare the performance, we validate that the prediction accuracy as well as the resulting model weights of the implementations is identical across systems in a separate test. However we do consider this to be a prerequisite for the above mentioned experiments and not an actual part of the Benchmark.

5.3 Cluster Hardware

We run our *supervised* and *unsupervised learning* benchmark experiments on the following homogeneous cluster nodes:

Quadcore Intel Xeon CPU E3-1230 V2 3.30GHz CPU with 8 hyperthreads, 16 GB RAM, 3x1TB hard disks (linux software RAID0) which are connected via 1 GBit Ethernet NIC via a HP 5412-92G-PoE+-4G v2 zl switch.

5.4 Data Sets

[8] The authors show on the example of several graph algorithms, that a single threaded implementation can compete and, when optimized, often outperform distributed frameworks. They therefore motivate to provide an single threaded implementation when benchmarking distributed systems.

[9] `https://www.csie.ntu.edu.tw/~cjlin/liblinear/`

We rely on generated data for the *unsupervised learning* experiments. We sample 100 dimensional data from k Gaussian distributions and add uniform random noise to the data, similar to the data generation for k-means in Mahout[4] and *HiBench*[14].

For the *supervised learning* experiments, we use parts of the *Criteo Click Logs*[10] data set. This dataset contains feature values and click feedback for millions of display ads drawn from a portion of Criteo's traffic over a period of 24 days. Its purpose is to benchmark algorithms for click-through rate (CTR) prediction. It consists of 13 numeric and 26 categorical features. In its entirety, the data set spawns about 4 billion data points, has a size of 1.5 TB . Our experiments are based on *days 0,1,2,3,5* and *6* of the data set.

As a pre-processing step we expand the categorical features in the data set using the hashing trick. The hashing trick vectorizes the categorical variables by applying a hash function to the feature values and using the hash values as indices. Potential collisions do not significantly reduce accuracy in practice, they certainly do not alter the computational footprint of the training algorithm. This allows us to control the dimensionality of the training data set via the size of the length of the vector to be hashed into. Experiments with fixed dimensionality were executed for $d = 1000$. The subset based on days 0,1,2,3,5 and 6 results in a data set of roughly **530 GB** in size, when hashed to 1000 dimensions. As collisions become less likely with higher dimensional hash vectors, the data set sizes increases slightly with higher dimensionality. However since the data set size is always identical for all systems, this effect does not perturb our findings. Different data set sizes have been generated by sub- und super-sampling the data. A *scaling factor* of `1.0` refers to the criteo subset as presented in Table 1 which contains about **1.15 billion data points**.

6. BENCHMARK RESULTS: EXPERIMENTS AND EVALUATION

In this section we present the results of our experimental evaluation of the presented systems for the different benchmark workloads. We ran all experiments using *Flink 1.0.3* and *Spark 1.6.2* in stand-alone mode.

6.1 Supervised Learning

6.1.1 Production Scaling

Figure 2 shows the runtimes for 5 iterations of batch gradient descent learning of a l2 regularized logistic regression model. We evaluate different implementation strategies (MapReduce, MapPartition and TreeAggregate) as introduced in Section 4.1 in both Spark and Flink. We measure the runtime for different data set sizes by scaling the criteo data set, which was hashed to 1000 dimensions.

While Flink strives to be declarative and to delegate the choice of physical execution plans to the optimizer, this experiment clearly shows that even for simple workloads such as batch gradient descent, the choice of implementation strategy matters and has a noticeable effect on performance for both Spark and Flink. Users must thus still be aware of the performance implications of implementation

[10] `http://labs.criteo.com/downloads/download-terabyte-click-logs/`

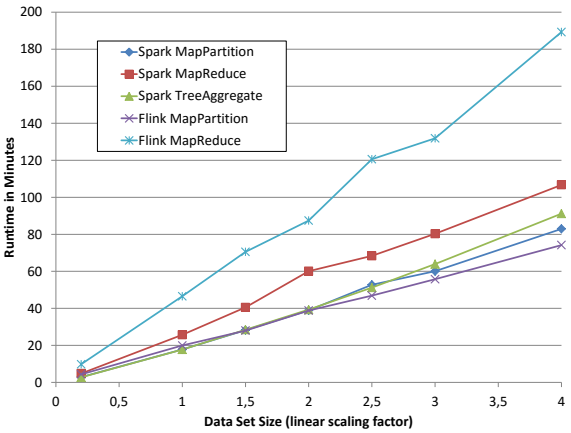

Figure 2: Production Scaling Experiment: We measure the runtime of different implementation strategies for l2 regularized logistic regression on a fixed set of 23 nodes for linearly growing data set sizes with 1000 dimensions.

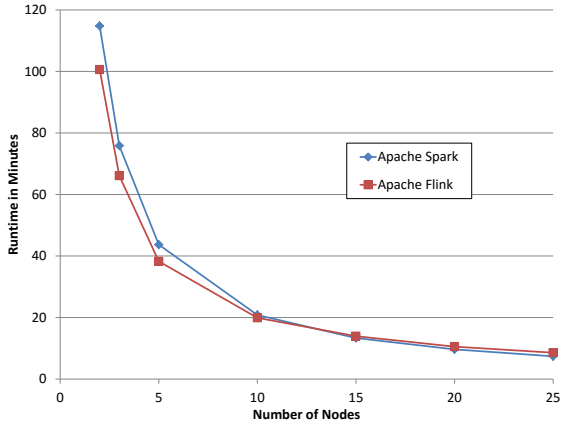

Figure 3: Strong Scaling for different implementations of l2 regularized logistic regression in Spark and Flink for 1000 dimensions and 530 GB.

choices in order to efficiently implement scalable machine learning algorithms on these data flow systems. It can be seen that the `MapPartition` based implementations, which pre-aggregate the partial gradient sums in the user code, as well as the `TreeAggregate` implementation in Spark outperform the `MapReduce` based implementation which rely on the system to place combiners on map outputs to efficiently aggregate the individual gradients. The slightly worse performance of Flink is due to unfortunate use of a newer version of the `Kryo` library, leading to constant re-building of *cached fields* for the Breeze SparseVectors, which are aggregated in the reduce phase. Overall however, all implementations on all systems show the desired scaling behaviour and exhibit linearly increasing runtime with increasing data set sizes. It is also noteworthy that both Spark and Flink show seamless out-of-core performance as the data set is scaled from moderate 230 million up to about 4.6 billion data points. We observe no performance degradation as the data set grows beyond the size of the combined main memory of the 23

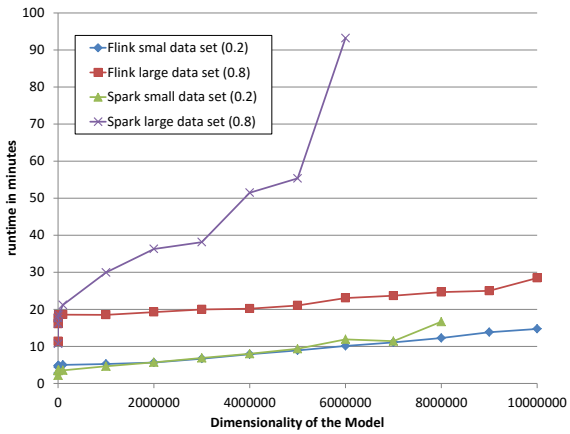

Figure 4: Scaling the Model: Runtimes for training a l2 regularized logistic regression model of different dimensions on 20 nodes. Results shown for two different data set sizes: small (about the size of main memory) and large (significantly larger than main memory).

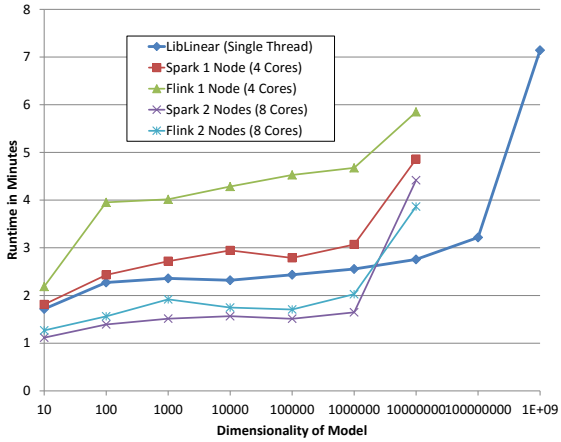

Figure 5: COST: Runtimes for training a l2 regularized logistic regression model of different dimensions different amounts of nodes for a small sub-sample (approx. 4GB) of the criteo data set compared to a single threaded implementation (LibLinear).

compute nodes (which would be the case beyond a scaling factor of 0.5).

6.1.2 *Strong Scaling*

Figure 3 shows the results of our strong scaling experiments for the batch gradient descent workload for the criteo data set hashed to 1000 dimensions. Figures 6 and 7 show the performance details[11] for a run with 25 nodes and Figures 8 and 9 for a run with three nodes. It is evident that while Flink tends to run faster on smaller cluster configurations, Spark has a slight edge on settings with many machines. The resource consumption shows that on three nodes, both system have to re-read significant portions of the data set in each iteration. However starting at about ten nodes, the amount of data read from disk per iteration continuously decreases in Spark, while it remains more or less constant in Flink. In the run with with 25 nodes depicted in Figures 6 and 7, Spark reads almost no data from disk at all, allowing for much higher CPU utilization compared to Flink, which is still practically I/O bound. This is most likely due to the different architecture with respect to memory management. While Spark schedules the tasks for each iteration separately, Flink actually instantiates the entire pipeline of operators in the plan a-priori and spreads the available memory equally amongst all memory consuming operators (reducers), leaving significantly less of the physically available main memory for the iterative computation than in Spark. In the highly resource-constrained setting of 2 or 3 nodes Flink's memory management and robust out of core performance actually leads to superior performance compared to Spark though. In general, both systems show the desired scaling behaviour which ensures that growing production workloads can be handled by adding more compute nodes, if the need arises.

6.1.3 *Scaling Model Dimensionality.*

As was described in the introduction, supervised learning models in production are not only trained on very sparse data set of massive size, but also tend to have a very high dimensionality. As it is computed from the sum of the sparse gradients of all data points, the model is always a `DenseVector` whose size is directly proportional to its dimensionality. In order to evaluate how well the systems can handle high dimensional `DenseVectors`, which have to be broadcasted after each iteration, we generate data sets of different dimensionality via adjusting the feature hashing in the preprocessing step. Figure 4 show the result of these experiments for two different data set sizes (a scaling factor of 0.2 and 0.8 of the criteo data set) for both Spark and Flink on 20 nodes. While the smaller data set has a total size comparable to the combined main memory of the 20 nodes, the larger version is significantly larger than main memory thus forcing the system to go out of core.

For the smaller data set (lower curves) both systems tend to exhibit rather similar performance for lower model dimensionalities, however Spark runs become more and more unstable, frequently failing starting at 5 million dimensions. We did not manage to successfully run Spark jobs for models with more than 8 million dimensions at all, since Spark fails due to a lack of memory. Flink on the other hand robustly scales to 10 million dimensions.

The situation becomes significantly worse for the larger data set: Spark runtimes are severely longer that Flink's, and Spark does not manage to train models beyond 6 million dimensions at all. Given the importance of being able to train models with at least 100 million if not billions of dimensions [9, 21, 24], this is a dissatisfying result. It seems the *Broadcast Variable* feature was simply not designed or intended to handle truly large objects.

6.1.4 *Comparison to single threaded implementation (COST)*

In order to gain an understanding of the performance of the distributed systems Spark and Flink compares to state of the art single core implementations we run experiments with the LibLinear solver, which provides a highly optimized single threaded C++ implementation. As this solver is limited

[11] Plots generated using `https://github.com/spi-x-i/shee`

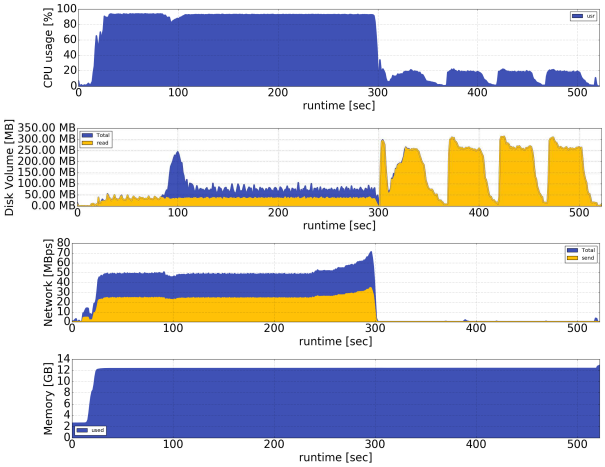

Figure 6: Performance Details for Flink on 25 nodes (logistic regression) (blue = total, yellow = read/sent)

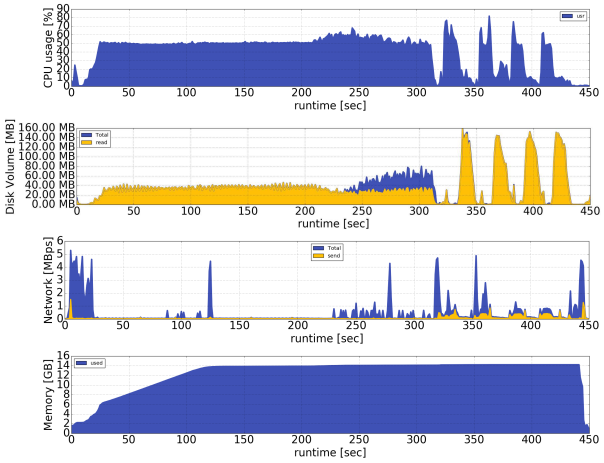

Figure 7: Performance Details for Spark on 25 nodes (logistic regression)

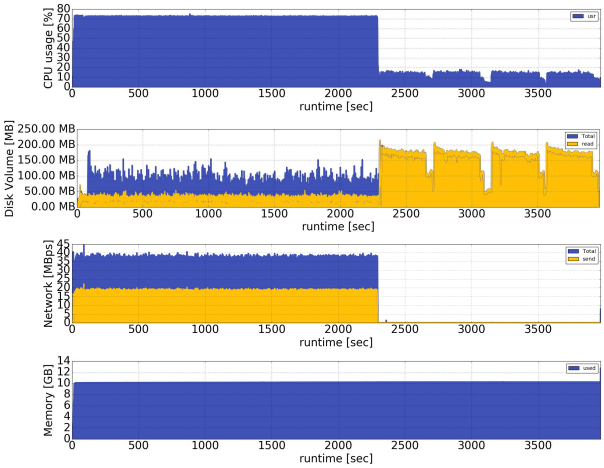

Figure 8: Performance Details for Flink on 3 nodes (logistic regression)

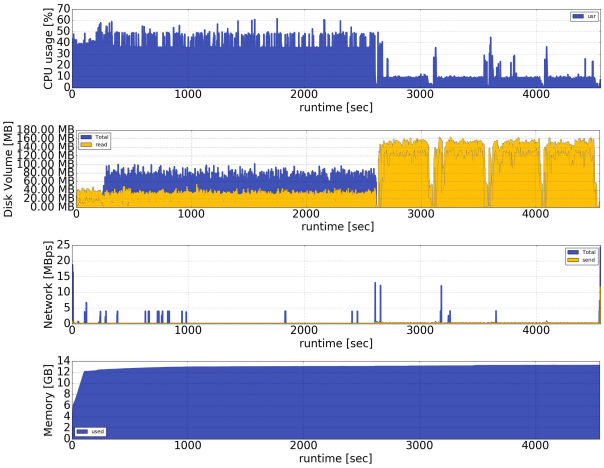

Figure 9: Performance Details for Spark on 3 nodes (logistic regression)

to data sets which fit into the main memory of a machine, we generate a smaller version of the criteo data set containing almost 10 million data points with dimensionalities ranging from 10 to 1,000,000,000. Figure 5 shows the runtime for 10 iterations of LibLinear training. To compare, we also run the Spark and Flink Solvers on one and two cores on these smaller data sets. It is apparent that while the runtimes for Spark and Flink are larger on one node (which has four cores), both systems run faster than LibLinear with two nodes (or 8 cores). It can thus be assessed that the hardware configuration required before the systems outperform a competent single-threaded implementation (COST) is between 4 and 8 cores. That is significantly less than observed for graph mining workloads by McSherry et al. [22]. A possible explanation could be that the ratio of computation to communication is much higher in ML workloads compared to graph processing workloads which can exhibit both: computation-intensive and communication-intensive phases.

However, we were not able to successfully train models with 100 million dimensions in both Flink and Spark, even though the data set is significantly smaller than the main memory of even one node. Furthermore, we observe a strong increase in runtime for 1 and 10 million dimensions for both Spark and Flink. This reemphasizes the observation of the dimensionality scaling experiment, that both data flow systems struggle to train truly large models due to apparent limitations in their support for large broadcast variables.

6.2 Unsupervised Learning

In order to evaluate the effectiveness of the `reduceByKey()` operator in Flink and the `groupBy()` and `reduce()` operator in Spark, we conduct both strong scaling and production scaling experiments with the unsupervised learning workload k-means. The production scaling experiments in Figure 11 show that the runtime of both Spark and Flink linearly increases with the data set size. Furthermore both Systems show no performance degradation once the data set does not fit into main memory anymore, but rather gracefully scale out-of-core. The strong scaling experiments in

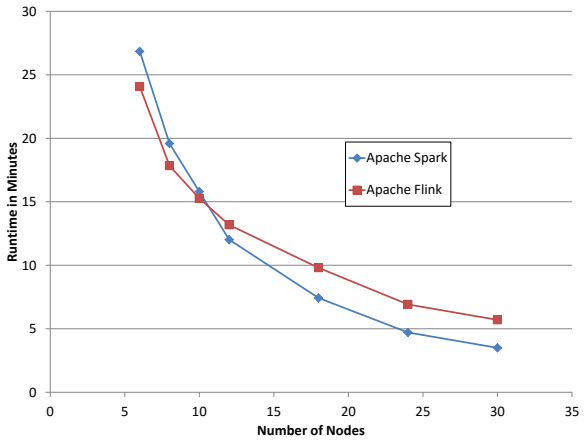

Figure 10: k-means Strong Scaling experiments for Spark and Flink in 200 GB of generated data with 100 dimensions and *k=10* clusters

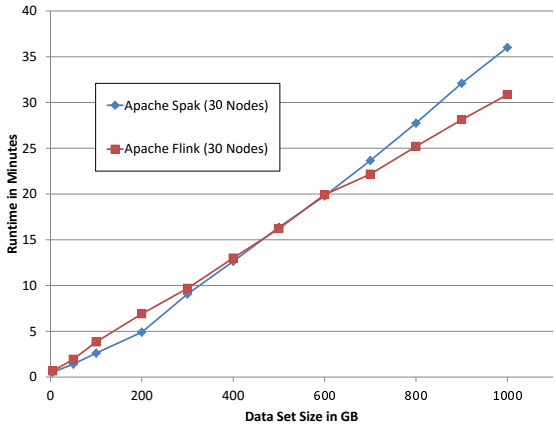

Figure 11: k-means Production Scaling experiments for Spark and Flink on 30 nodes with *k=30*

Figure 10 appear to confirm the observation already apparent for supervised learning workloads: Flink performs better for the resource-constrained setting with a few nodes, while Spark performs better once enough main memory is available due to the addition of compute nodes. This aspect is also reflected in the production scaling experiment. Apache Flink's approach to memory management: instantiating the entire pipeline a priori and distributing the available memory amongst the memory-consuming operators of the pipeline seems to be able to cope better with limited main memory than Spark's approach of separately scheduling each task.

7. RELATED WORK IN BENCHMARKING

In the last years, several papers have been published on the comparison and evaluation of distributed systems for very specific workloads:

Cai et al. [8] present a benchmark of statistical machine learning algorithms such as GMMs, HMMS and LDA on Spark, GraphLab, Giraph and SimSQL. They focus on users who want to implement their own ML algorithms, and thus evaluate the ease of implementation and the absolute runtime of the systems. However, they do not focus on providing comprehensive scalability evaluations but rather detailed discussions of the implementation details of the hierarchical statistical models on the different systems. The results are mixed, since no system clearly outperforms the others in all evaluated dimensions. However, since they utilized the *Python API* of Spark and noticed in the end of the paper, that it provides substantially slower performance than the *JAVA Api*, the runtime results are not directly comparable to our experimental evaluation.

More closely related is the work by Shi et a. [27], which presents an experimental evaluation of Apache Spark and MapReduce for Large Scale Data Analytics. They consider the workloads *Word Count, Sort, K-Means* and *PageRank*. Contrary to our approach, the authors rely on third party implementations in libraries (Mahout for MapReduce and Spark MLlib), so that it remains unclear if performance differences are due to algorithmic variations or the systems themselves. Furthermore, experiments were carried out on a cluster consisting of only four nodes, which is hardly the targeted setup for deployments of both systems, thus insights gained from these experiments may not be applicable for common setups.

In a very similar manner, Marcu et. al. [20] present a performance analysis of the big data analytics frameworks Apache Spark and Apache Flink, for the workloads *Word Count, Grep, Terra Sort, K-Means, PageRank* and *Connected Components*. Their results show, that while Spark slightly outperforms Flink at *Word Count* and *Grep*, Flink slightly outperforms Spark at *k-means*, *Terra Sort* and the graph algorithms *PageRank* and *Connected Components*. Contrary to our approach, the authors only consider simple workloads and do not evaluate distributed machine learning algorithms with respect to the crucial aspect of model dimensionality.

In a very similar manner, Marcu et. al. [28] present yet another performance evaluation of the systems Apache Flink, Spark and Hadoop. Contrary to our work they purely rely on existing libraries and example implementations for the workloads *Word Count, Grep, Terra Sort, K-Means, Page Rank* and *Connected Components*. Their results confirm that while Spark slightly outperforms Flink at *Word Count* and *Grep*, Flink outperforms Spark at the graph algorithms *PageRank* and *Connected Components*. However, contrary to the findings of [20], Spark outperforms Flink for the *k-means* workload. An observation which our findings confirm which are most likely due to improvements in *Spark 1.6.1.* (e.g. project Tungsten)which were not present in the Spark version used in [20].

8. CONCLUSIONS

In this paper we presented a comprehensive Benchmark to evaluate and assess data flow systems for distributed machine learning applications. The Benchmark comprises distributed optimization algorithms for supervised learning as well as algorithms for unsupervised learning. We motivated and described different experiments for evaluating the scalability of distributed data processing systems for all the aspects that arise when executing large scale machine learning algorithms. Next to *Strong Scaling* and *Production Scaling* experiments which asses the systems ability for scaling the data set size, we also introduced *Model Dimensionality Scal-*

ing and *COST experiments* to evaluate the ability to scale with growing model dimensionality.

Our comprehensive experimental evaluation of different implementations in Spark and Flink on up to 4.6 billion data points revealed both systems scale robustly with growing data set sizes. However, the choice of implementation strategy has a noticeable impact on performance, requiring users to carefully choose physical execution strategies when implementing machine learning algorithms on these data flow systems.

When it comes to scaling the model dimensionality however, Spark fails to train models beyond a size of 6 million dimensions. Both systems did not manage to train a model with 100 million dimensions even on a small data set. Finally, experiments with a state of the art single threaded implementation showed, that two nodes (8 cores) are a sufficient hardware configuration to outperform a competent single-threaded implementation.

Since being able to train models with hundreds of millions if not billions of dimensions is a crucial requirement in practice, these results are unsatisfactory. *ParameterServer* architectures [17] may pose a viable alternative, as they have been shown to scale to very high dimensionalities. However, they require asynchronous algorithms, which usually only approximate optimal solutions. Furthermore the significant communication cost associated with this approach is also a challenge [26]. It thus remains an open challenge to provide an adequate solution to the problem of robustly and efficiently scaling distributed machine learning algorithms both: with respect to data set size and model dimensionality at the same time.

Acknowledgments

This work has been supported through grants by German Ministry for Education and Research as Berlin Big Data Center BBDC (funding mark 01IS14013A).

We would like to acknowledge the valuable contributions of Marcus Leich in performance debugging the Flink implementations, the fruitful recommendations of Sebastian Schelter and Martin Jaggi as well as Andreas Kunft, Maximilan Alber and Seven Dähne.

9. REFERENCES

[1] http://peel-framework.org/.
[2] https://flink.apache.org/.
[3] https://hadoop.apache.org/.
[4] https://mahout.apache.org/.
[5] https://spark.apache.org/.
[6] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6), Dec. 2014.
[7] T. Brants, A. C. Popat, P. Xu, F. J. Och, J. Dean, and G. Inc. Large language models in machine translation. In *EMNLP*, pages 858–867, 2007.
[8] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1371–1382, 2014.
[9] k. Caninil. Sibyl: A system for large scale supervised machine learning.
[10] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.
[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.
[12] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *Proc. VLDB Endow.*, 2012.
[13] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2), Mar.
[14] HiBench. https://github.com/intel-hadoop/HiBench.
[15] L. Jimmy and A. Kolcz. Large-scale machine learning at twitter. *SIGMOD 2012*, 2012.
[16] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Records*, 44(4), May 2016.
[17] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pages 583–598, 2014.
[18] C.-J. Lin and J. J. Moré. Newton's method for large bound-constrained optimization problems. *SIAM J. on Optimization*, 9(4), Apr. 1999.
[19] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Math. Program.*, 1989.
[20] O. C. Marcu, A. Costan, G. Antoniu, and M. S. Pérez-Hernéndez. Spark versus flink: Understanding performance in big data analytics frameworks. In *IEEE CLUSTER 2016*, pages 433–442, Sept 2016.
[21] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica. Ad click prediction: A view from the trenches. In *KDD '13*. ACM, 2013.
[22] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *USENIX HOTOS'15*. USENIX Association, 2015.
[23] F. Niu, B. Recht, C. Re, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS 2011*, USA.
[24] M. Richardson, E. Dominowska, and R. Ragno. Predicting clicks: Estimating the click-through rate for new ads. In *WWW '07*. ACM, 2007.
[25] S. Schelter, C. Boden, M. Schenck, A. Alexandrov, and V. Markl. Distributed matrix factorization with mapreduce using a series of broadcast-joins. *ACM RecSys 2013*, 2013.
[26] S. Schelter, V. Satuluri, and R. Zadeh. Factorbird - a Parameter Server Approach to Distributed Matrix Factorization. *Distributed Machine Learning and Matrix Computations workshop at NIPS 2014*, 2014.
[27] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow.*, 8(13), Sept. 2015.
[28] J. Veiga, R. R. Expósito, X. C. Pardo, G. L. Taboada, and J. Tourifio. Performance evaluation of big data frameworks for large-scale data analytics. In *IEEE BigData 2016*, pages 424–431, Dec 2016.
[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI'12, 2012.