# General-Purpose vs. Specialized Data Analytics Systems: A Game of ML & SQL Thrones

Evdokia Kassela*, Nikodimos Provatas*, Ioannis Konstantinou*, Avrilia Floratou†, Nectarios Koziris*
*CSLab, National Technical University of Athens, Greece
{evie, nprov, ikons, nkoziris}@cslab.ece.ntua.gr
†Cloud and Information Services Lab, Microsoft US
avflor@microsoft.com

***Abstract*—Over the past decade, a plethora of systems have emerged to support data analytics in various domains such as SQL and machine learning, among others. In each of the data analysis domains, there are now many different *specialized* systems that leverage domain-specific optimizations to efficiently execute their workloads. An alternative approach is to build a *general-purpose* data analytics system that uses a common execution engine and programming model to support workloads in different domains. In this work, we choose representative systems of each class (Spark, TensorFlow, Presto and Hive) and benchmark their performance on a wide variety of machine learning and SQL workloads. We perform an extensive comparative analysis on the strengths and limitations of each system and highlight major areas for improvement for all systems. We believe that the major insights gained from this study will be useful for developers to improve the performance of these systems.**

***Index Terms*—benchmarking, ML, SQL, Spark, TensorFlow, Presto, Hive, TPC-H, general-purpose system, specialized system**

## I. Introduction

As industry in nearly every sector of the economy has moved to a data-driven world [1], there has been an explosion in the volume of data that needs to processed and analyzed almost as soon as it is generated. Deriving value from data is typically a multi-stage process that involves data analysis workloads from various domains, such as SQL, machine learning (ML), and graph analytics, among others. The need for efficiently supporting such complex analytics has never been higher than the current level as gaining actionable insights from the data has become a key service differentiator.

Researchers and practitioners in our community have risen to this challenge by building a wide variety of data analytics systems. In each of the data analysis domains, there are now many different *specialized* systems that leverage domain-specific optimizations to efficiently execute their workloads. Prominent examples in this category are Apache Hive [2], Impala [3], and Presto [4] for SQL processing on top of Hadoop data, Google's TensorFlow [5], GraphLab [6], and Mahout [7] for machine learning applications and Giraph [8] for graph analytics.

An alternative approach is to build a *general-purpose* data analytics system that uses a common execution engine and programming model to support workloads in different domains using domain-specific libraries and modules. The prototypical example in this class is the widely used Apache Spark [9] framework with its Spark SQL [10] module for relational workloads, MLlib [11] for machine learning, Spark Streaming [12] for streaming workloads and GraphX [13] for graph analytics. Apache Flink is another system in this class that supports both batch and streaming analytics [14].

In this paper, we present a detailed experimental study to compare and contrast popular, representative systems of each class. We benchmark their performance on a wide variety of workloads from the ML and SQL domains. In particular, we compare Spark's MLlib module with TensorFlow, which follows a *parameter server* architecture specialized for ML workloads. We also compare Spark SQL's general-purpose architecture with Presto, a specialized SQL engine that employs a *massively parallel processing* (MPP) architecture. Moreover, we perform experiments with Hive, a standalone state-of-the-art SQL engine that relies on an underlying general-purpose processing framework.

Our main contributions are the following:

- We quantify the performance gap between rerpresentative generalized and specialized data analytics systems on both ML and SQL workloads.
- We perform experiments with both real and synthetic datasets using up to 140 nodes. To the best of our knowledge, we are the first to perform an evaluation of these systems in such large-scale.
- We perform a thorough analysis of the experimental results focusing on both the architectural and the implementation differences of the systems and present our major insights.

We believe that this study will be useful for developers to improve the performance of these systems. Our key results include:

- In the context of ML, we find that TensorFlow is more computationally efficient than Spark even when TensorFlow operates in synchronous training mode which resembles Spark's execution model. However, Spark has better reading throughput. These differences are mainly attributed to the different architectures of the two systems.
- In the context of SQL workloads, we find that both Hive and Spark SQL outperform Presto although the latter employs a MPP architecture. The performance differences can be attributed to the sub-optimal query optimizer and memory manager that Presto employs.

978-1-7281-0858-2/19/$31.00 ©2019 IEEE

Admittedly, comparing these approaches solely based on their performance ignores other relevant aspects including ease of application integration, expressiveness of the programming model, manageability, and fault-tolerance, among others. We leave a more thorough comparison including these aspects and other analysis domains out of the scope of this work.

The rest of the paper is organized as follows: in Section II, we discuss the architecture of TensorFlow and Spark MLlib and present an experimental evaluation using three popular ML algorithms. In Section III, we present the basic features of Hive, Spark SQL and Presto and perform a performance analysis using the TPC-H benchmark [15]. We present related work in Section IV and conclude the paper in Section V.

## II. Machine Learning

In this section, we examine the performance characteristics of **Spark MLlib** [11] and **Google TensorFlow** [5]. We compare the two systems using three popular ML algorithms (logistic regression, linear regression, and multi-layer perceptron classifier). These algorithms are supported out-of-the-box by both systems. Unfortunately, we are not able to compare the systems on deep learning algorithms as Spark does not currently provide this functionality. However, the perceptron algorithm can be considered a simple deep learning representative that is supported in Spark. Our analysis shows that there is a large performance gap between the two systems which can mainly be attributed to their different architectures.

### A. Background

Many machine learning algorithms can be cast as convex optimization problems by defining a *loss function* of the model parameters which quantifies the prediction error of the parameterized model. The most widely used approach for solving this optimization problem is *Gradient Descent* (GD) and its stochastic variant, *Stochastic Gradient Descent* (SGD). Both these algorithms are used to optimize an objective function by efficiently exploring its surface using iterative gradient computation and movement in the direction of steepest gradient. The difference between the two algorithms lies in how much of the training data is used to compute the gradient each time: GD makes a full pass over the training data whereas SGD looks at a minibatch (subset) of the training data, possibly as small as a single training example. Typically, the optimization process terminates when the model's prediction error is below a certain threshold (convergence). SGD typically converges faster than GD particularly for large datasets [16].

### B. Systems Architecture

In this section, we present an overview of the architectures of TensorFlow and Spark MLlib.

*1) Google TensorFlow:* TensorFlow [5] is an open-source, scalable machine learning platform developed by Google.

**Abstract Programming Model.** TensorFlow models machine learning algorithms as dataflow graphs and data as *tensors*. A graph edge represents a tensor transfer between nodes. Graph nodes represent units of local computation on the input tensors, called *operations*, such as matrix multiplication.

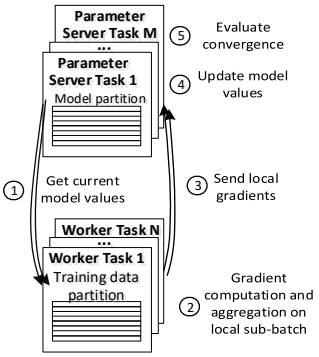



Fig. 1: SGD in TensorFlow

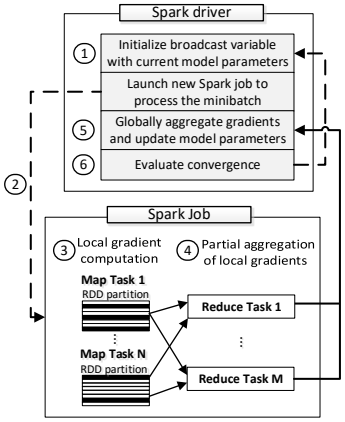



Fig. 2: SGD in Spark MLlib

**Execution Model.** TensorFlow is deployed as a set of *tasks* which are processes that can communicate over a network. The tasks are of two types, namely *parameter server tasks* and *worker tasks* following the parameter server architecture [17]–[19]. The parameter server tasks maintain the current version of the globally shared model parameters. The worker tasks, on the other hand, are responsible for performing the bulk of the computation on top of their local training data.

For example, in the context of SGD with a minibatch of $T$ training examples and $N$ worker tasks, each worker task computes gradients based on a *local sub-batch* consisting of $T/N$ local training examples, and then updates the shared parameters hosted by M parameter server tasks.The process is depicted in Figure 1. Steps 1-3 and 4-5 are executed in parallel on the worker and the parameter server tasks respectively. Each worker gets the latest model parameters from the parameter server (Step 1), performs the local gradient computation (Step 2) and then sends the updated gradient to the parameter server tasks (Step 3). Sequentially, parameter servers update the model parameters with the received gradients (Step 4) and evaluate convergence (Step 5).

**Training Modes.** TensorFlow supports both *asynchronous* and *synchronous training*. The former is performed by allowing Step 4 in Figure 1 to be executed by the parameter server tasks whenever a worker task sends a new gradient update (Step 3). On the other hand, in the Synchronous mode, Step 4 is executed only when the parameter server has received one gradient update from each worker, which it aggregates into one using a *sum* operator. For this case study, we configured TensorFlow in an appropriate way to perform updates only after it receives one local update from each worker.

In our experiments with real datasets, we found that Asynchronous Training needed less time to converge than synchronous at most cases. However, it needed more iterations in some cases, due to workers overriding each other's work.

**Data Access.** In order to fetch data from disk, TensorFlow comes with an API, called Dataset API which provides a set of functions that can be sequentially used to create a pipeline that fetches, decodes and creates data batches, which are subsets of the data. These batches will be consumed from each worker to compute the next gradient updates.

The pipeline operates as follows: First, data is loaded from disk as raw bytes. Afterwards, the data rows are *mapped* to their decoded Tensor format in parallel. Finally, a *batch* is created. During these operations, *caching*, *batch prefetching* and *data shuffling* is performed. We discuss these operations and their performance implications in the following sections.

*2) Spark MLlib:* Spark MLlib [11] is Spark's scalable machine learning library. We now describe its basic features.

**Abstract Programming Model.** Since MLlib is part of the Spark framework, it employs Spark's computation model. More specifically, Spark performs a series of computations following the Map Reduce Model upon Resilient Distributed Datasets (RDDs). The RDD is a basic abstraction in Spark that represents an immutable distributed collection of data.

**Execution Model.** As opposed to TensorFlow that employs long-running processes, the machine learning algorithms are executed in MLlib as a sequence of Spark jobs that consist of *map* and *reduce* stages whose corresponding tasks are launched by *Spark executors*. State is maintained and transferred across jobs using *broadcast variables* through the *Spark driver* and training data are represented as a partitioned RDD.

In the context of the SGD algorithm, a sequence of Spark jobs are launched, each one processing a minibatch. The process is depicted in Figure 2, where dashed and solid arrows denote the control flow and data flow respectively. First, the Spark driver initializes the model parameters as a broadcast variable (Step 1) and launches a Spark job (Step 2). The map tasks of the job generate a *local sub-batch* randomly from their local partition of the RDD and compute the gradients on top of their local sub-batch (Step 3). Then, the reduce tasks perform a partial aggregation of the computed gradients (Step 4). Finally, the Spark driver performs the final global aggregation, updates the model parameters (Step 5) and evaluates the convergence criteria (Step 6) before launching another job.

**Training Modes.** MLlib supports only synchronous training in contrast with TensorFlow, because of its MapReduce-based execution model.

**Data Access.** Since Spark employs lazy evaluation, each local RDD partition that points to the training data will be fully read from disk only when the map tasks perform their computations. Note that local RDD partitions are also cached in memory to avoid repeatedly fetching from the disk.

### C. Experimental Evaluation

We now present a detailed experimental comparison of TensorFlow and MLlib using both real and synthetic datasets. Our key findings are the following:

1) TensorFlow is more computationally efficient than Spark MLlib regardless of the training mode (see Figures 3 and 6a), but Spark has better read throughput (see Table III).
2) Spark's short-running tasks introduce scheduling and initialization overheads, especially for small minibatches (see Section II-C2). TensorFlow's long-running processes avoid such overheads.
3) TensorFlow needs less time to converge than Spark MLlib in almost all the machine learning models that we examined (see Figure 7).
4) Spark meets its best performance when the minibatch size is such that the initialization overheads of Spark are amortized. (see Figures 7a and 7b).

In the following sections, we describe our experimental setup and further analyze our findings.

*1) Experimental Setup:* **Hardware and Software Configuration.** Our experiments are performed on a cluster of 141 virtual machines ("nodes") in Okeanos public cloud [20], [21]. Each virtual machine has 2 virtual CPUs @ 2.1GHz, 8 GB of RAM and 20 GB of hard disk storage. We use one of these nodes as *master node* and the remaining ones as *worker nodes*. Thus, the worker nodes provide a total of 280 virtual CPUs, 1 TB of RAM and 2.8 TB of hard disk storage.

The operating system used is Debian Jessie 8.10. We use TensorFlow version 1.13 and Spark version 2.4. In the case of TensorFlow, we deploy one worker task on each of the 140 worker nodes. We use one parameter server task deployed at the master node. In the case of MLlib, we deploy one Spark executor on each of the 140 worker nodes. We host the Spark driver at the master node.

**Machine Learning Models.** We use three models for predictive analysis, namely *linear regression* [22], *binary logistic regression* [23], and the *multilayer perceptron (MLPC)* classifier [24]. The perceptron classifier is chosen, since it is a representative from the deep learning subdomain and is implemented in both systems. Specifically, the corresponding artificial neural network consists of 4 layers, two of which are the hidden ones, with 28, 15, 15 and 2 neurons respectively. We used the GD and SGD optimizers in the training, since they are widely used and supported by both systems. However, for perceptron, only GD is used, since SGD is not supported by MLlib for this algorithm.

**Methodology.** Our experimental evaluation consists of two parts. First, we generate synthetic data and perform a series of experiments in a 141-node cluster in order to examine how both systems operate at large scale. Since evaluating convergence on top of synthetic data is not recommended, we fix the number of minibatches that the algorithms process before terminating. We further conduct experiments using real datasets in a 5-node cluster in order to study convergence. Note that a small cluster is used, since real datasets are not large enough to fully utilize a larger one. For TensorFlow, we examine both the synchronous and the asynchronous training mode in each of our experimental sections.

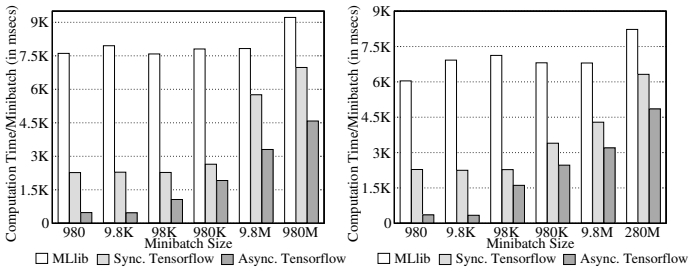
**Datasets.** For the experiments with real data, we use

TABLE I: Real datasets

| Dataset | #Examples | #Features | Size |
|---|---|---|---|
| HIGGS | 10,500,000 | 28 | 2.7 GB |
| Year_Prediction_MSD | 470,000 | 90 | 0.391 GB |

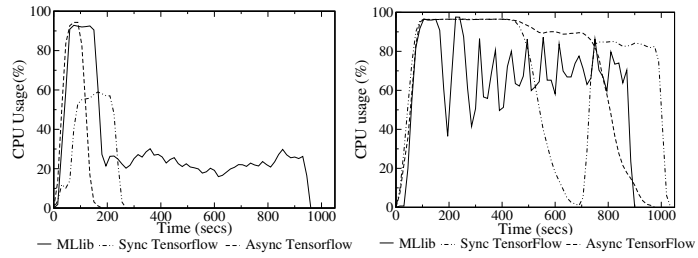
TABLE II: Synthetic datasets

| ML Model | #Examples | #Features | Size |
|---|---|---|---|
| Logistic Regression | 980,000,000 | 28 | 252 GB |
| Linear Regression | 280,000,000 | 90 | 238 GB |
| Perceptron | 441,000,000 | 28 | 110 GB |



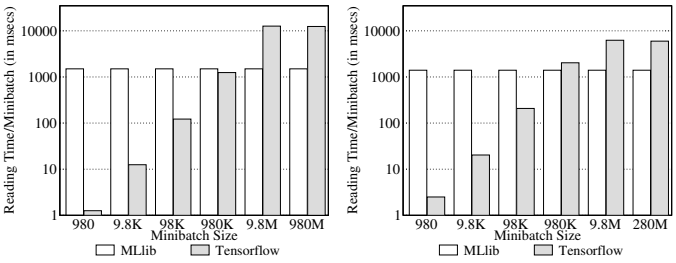

(a) Logistic Regression (b) Linear Regression

Fig. 3: Computation time per minibatch in MLlib and TensorFlow (synchronous and asynchronous) on a 141-node cluster.



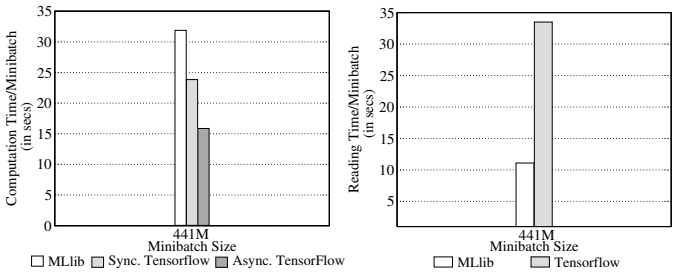

(a) Log. Regression SGD (b) Perceptron

Fig. 4: Cluster CPU usage for Logistic Regression (minibatch=980K) and Perceptron




(a) Logistic Regression (b) Linear Regression

Fig. 5: Reading time per minibatch in MLlib and TensorFlow on a 141-node cluster




(a) Comp. Time per Minibatch (b) Reading Time per Minibatch

Fig. 6: Times per minibatch spent for Perceptron in MLlib and TensorFlow on a 141-node cluster

datasets from the UCI Machine Learning Repository [25] stored as comma separated text files. More specifically, we use the `HIGGS` [26] dataset for the *logistic regression* and *perceptron* models and the `Year_Prediction_MSD` [27] dataset for the *linear regression* model, which were the largest real ones without missing values in the repository. Table I presents the characteristics of these datasets.

To generate the synthetic data, we replicated each real dataset multiple times so that the resulting dataset barely fits in the memory of the cluster (see Table II). This is because, for the GD algorithm, TensorFlow requires the whole dataset to fit in the aggregate memory of the cluster in contrast with Spark.

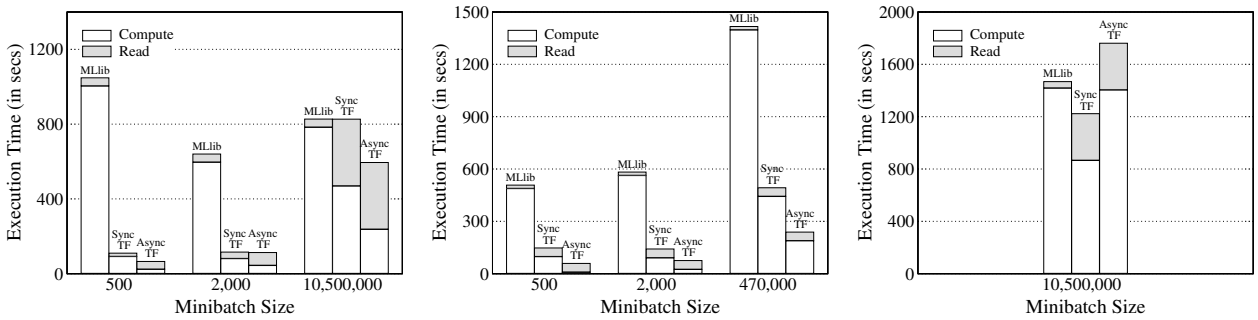
The datasets are stored in HDFS when MLlib is used and are split into equal parts that are uniformly distributed across all the cluster nodes when TensorFlow is used.

*2) Experiments with Synthetic Data:* In this section, we present experiments on the 141 node cluster using synthetic data, as presented in Table II, to identify the performance bottlenecks of each system by carefully profiling them on a large-scale. As described in Section II-C1, we terminate the training process after the algorithms process a fixed number of minibatches, which are set to 100 for the *logistic* and *linear regression* models and to 20 for the *perceptron* model.

Figures 3 and 5 show the average time spent by each system in performing gradient computations and reading data (fetch, deserialize and decode raw bytes) when processing a minibatch using *logistic* and *linear regression*. For the computation time, three bars are presented for each minibatch size, referring to Spark MLlib, synchronous and asynchronous Tensorflow. Reading time is presented for MLlib and TensorFlow irrespective of its training modes, since it does not depend on them.

As shown in Figure 3, TensorFlow spends always less time for computing gradients regardless of the training mode. As minibatch size decreases, TensorFlow is more computationally efficient than Spark MLlib. Moreover, when decreasing the minibatch size, TensorFlow achieves up to $14X$ speedup, whereas Spark becomes faster up to $1.36X$. Compared to TensorFlow's long-running processes, Spark launches a new Spark job every time a new minibatch is processed. This introduces task scheduling and initialization overheads (especially when the minibatch decreases and map tasks become much shorter (≈100msecs)), and low CPU utilization, as confirmed in Figure 4a. The same figure suggests that synchronous TensorFlow also suffers from some overheads but these are mostly synchronization overheads. Despite these overheads, it still has better CPU utilization than Spark.

Regarding the average reading time per minibatch, as we see in Figure 5, Spark MLlib spends the same amount of time reading data irrespective of the minibatch size, as it reads the whole RDD partition in all cases. TensorFlow on the other hand, fetches only the data needed for the current gradient computation. As a result, in TensorFlow the average reading time drops when the minibatch sizes decreases. However, in the *linear regression case*, Tensorflow spends almost the same time reading each minibatch for both GD and SGD (minibatch size 9.8M) as shown in Figure 5b. Since each algorithm runs until the system processes 100 minibatches, the full dataset is read by both algorithms (see dataset size in Table II). In SGD, each row is cached after it has been decoded and before it is used for minibatch creation. On the contrary, we found that in the case of GD, it is optimal to cache the whole minibatch instead of a row-at-a-time, resulting in $1.62X$ speedup in




(a) Logistic Regression (b) Linear Regression (c) Perceptron

Fig. 7: Performance of Spark MLlib, synchronous (Sync TF) and asynchronous TensorFlow (Async TF) on a 5-node cluster

TABLE III: Read Throughput of Spark MLlib and TensorFlow

| ML Model | Read Throughput (#training examples/sec) | |
|---|---|---|
| | Spark MLlib | TensorFlow |
| Logistic Regression | 6,533,333 | 785,323 |
| Linear Regression | 2,000,000 | 456,774 |
| Perceptron | 1,986,486 | 788,048 |

reading time and achieving a similar performance as SGD.

Figure 6 presents the computation and reading time spent per minibatch for *perceptron*. Since the full dataset is used as a minibatch (GD), TensorFlow spends more time reading data than Spark. As shown in Figure 4b, TensorFlow, especially in synchronous mode, suffers from reading the whole dataset before the first gradient computation: unlike asynchronous mode, the system is blocked until all their workers finish reading their local data depicted by the CPU usage drop between 500 and 700 sec.

To better understand the reading performance, we computed the read throughput of each system and present it in Table III. As shown in the table, the throughput varies across different algorithms depending on the number of features and the transformations required for each dataset. However, Spark always reads data faster and has up to $8X$ better reading throughput than TensorFlow.

*3) Experiments with Real Data:* In this section, we present experiments using the real datasets presented in Table I on a 5-node cluster. As noted before, we could not run experiments on a larger cluser as the real datasets are generally small. All the algorithms are executed until they have converged.

**Linear and Logistic Regression.** Figures 7a and 7b present the total execution time of TensorFlow and Spark MLlib for the *logistic regression* and *linear regression* algorithms respectively, including the portion of the time spent in reading data and performing computations in each system. The various minibatch sizes are selected to be realistic [16].

As shown in the figures, TensorFlow is faster than Spark MLlib by up to $16X$ when the SGD algorithm is used (Figure 7a, minibatch size 500). This behaviour is mainly attributed to architectural differences of the two systems, as we explained in Section II-C2. Regardless of the training mode, TensorFlow presents at worst the same performance as MLlib, with the asynchronous mode being faster.

Table IV shows the number of minibatches that each system

TABLE IV: Number of minibatches processed per system

| ML Model | Minibatch Size | #Minibatches processed | | |
|---|---|---|---|---|
| | | MLlib | Sync TF | Async TF |
| Logistic Regression | 10,500,000 | 317 | 317 | 137 |
| | 2000 | 774 | 737 | 1006 |
| | 500 | 1375 | 1644 | 2449 |
| Linear Regression | 470,000 | 3104 | 3104 | 1328 |
| | 2000 | 2678 | 2773 | 1042 |
| | 500 | 2438 | 2540 | 1121 |
| Perceptron | 10,500,000 | 72 | 72 | 118 |

processes before the algorithm converges. While the number of minibatches is the same between synchronous TensorFlow and MLlib in the case of GD as expected, a small devitation is observed in the SGD case. This is due to the fact that each system performs batch selection differently as explained in Section II-B, resulting to dissimilar data per minibatch. Another interesting observation is that the number of minibatches processed by GD and SGD is different for the same machine learning model, as SGD might need to process more minibatches until convergence [16] (see *logistic regression*).

As we explained in section II-C2, TensorFlow has different behaviour in the two training modes when the minibatch size decreases. For instance, as shown in Figure 7a in the context of *logistic regression*, asynchronous training is $1.72X$ faster when reducing the minibatch size from 2000 to 500, since the system processes $1.65X$ fewer rows in total. On the contrary, synchronous training needs almost the same time to converge with the two batch sizes. Similar trends are noticed in the case of Spark MLlib. For example, in the case of *linear regression*, Spark needs almost the same time to process $4.02X$ more rows until convergence, when increasing the minibatch size from 500 to 2000. This behavior is attributed to reading the whole dataset at every gradient computation irrespective of the minibatch size and to various other computational overheads as discussed in Section II-C2. However, Figures 7a and 7b show that SGD with a minibatch size of 2000 is $1.29X$ and $2.43X$ faster than when GD is used. Thus, Spark meets its best performance when the batch size is such that the overheads mentioned above do not dominate the execution time.

In the context of *linear regression*, as we observe in Figure 7b, TensorFlow's reading time is approximately 50 sec irrespective of the batch size and the training mode. A closer look in Table IV, indicates that TensorFlow processes at least 1000 minibatches of 500, $2K$ and $470K$ rows respectively. In

all cases, more rows are processed in total than the dataset size (Table I). Since the full dataset is loaded in memory, TensorFlow's caching minimizes the data reading overheads (see Section II-C2).

**Perceptron.** Figure 7c presents the total execution time of the systems on the *perceptron* classifier. Note that only the GD optimization algorithm is used as discussed in Section II-C1. As we can see in the Figure, Spark MLlib is $1.2X$ faster than asynchronous TensorFlow for this algorithm. However, as shown in Table IV, TensorFlow processed almost twice the number of minibatches that Spark MLlib processed before converging. This can be attributed to workers overwritting each others' work introducing more computational steps before converging. In synchronous mode, TensorFlow needed $1.2X$ less time to converge with the same number of minibatches with MLlib, while its reading phase is $7.2X$ slower, which confirms its computational efficiency.

## III. Relational Query Processing

In this section, we present a detailed experimental analysis of **Apache Hive** [2], **Presto** [4] and **Spark SQL** [10] using the TPC-H [15] workload. Our analysis shows that Hive and Spark SQL have similar performance, and they both outperform Presto, although the latter employs a MPP architecture. The overall measured performance is mainly affected by the efficiency of the plans produced by the query optimizers of the systems and by their internal memory management.

### A. Systems Architecture

In this section, we provide an overview of Apache Hive, Presto, and Spark SQL and discuss their basic features.

*1) Apache Hive:* Apache Hive [2] is a popular SQL-on-Hadoop engine which provides a SQL-like query language called HiveQL.

**Execution Model.** The HiveQL statements submitted to Hive are parsed, compiled and optimized to produce a query execution plan. The plan is represented as a Directed Acyclic Graph (DAG) consisting of map and reduce stages which are executed through an underlying data processing framework such as MapReduce, Spark or Tez [28]. Although Spark is a general purpose processing engine, Tez is a YARN-based [29] framework which is developed specifically for purpose-built tools such as Hive. Also, as shown in previous work [30], Hive on Tez is more efficient than Hive on MapReduce as it avoids the scheduling, initialization and materialization overheads of MapReduce. It also offers dynamic run-time task scheduling and plan reconfiguration. For these reasons, in this work we perform all our experiments on top of the Tez framework.

**Query Optimization.** Hive employs cost-based optimization through the Apache Calcite query optimizer [31]. Calcite supports filter push down, column pruning, partition pruning, physical operator selection (e.g., join implementation) and join reordering. Calcite relies on the available statistics to cost the query plans. These statistics along with other table metadata (e.g, schema) are accessed through the Hive Metastore.

*2) Spark SQL:* Spark SQL [10] is a module in Apache Spark [9] that supports relational processing on top of structured data.

**Execution Model.** Spark SQL operates on top of *DataFrames*. A DataFrame is an internal data structure that is conceptually equivalent to a table in a relational database. DataFrames can be constructed from various sources such as HDFS or Spark's RDDs and are serialized in off-heap storage in binary format. Spark SQL users can submit their queries on top of DataFrames using SQL or the *DataFrame API*. These queries are parsed, optimized and the resulting physical plan is executed through the Spark data processing framework. The plan is represented as a DAG which consists of multiple stages and stages are separated by shuffle operations, meaning that consecutive map tasks can be pipelined within a single stage. Each stage's tasks are scheduled before it is launched.

**Query Optimization.** Spark SQL optimizes incoming queries using the Catalyst query optimizer. Catalyst uses the statistics of the Hive Metastore too and employs cost-based optimization techniques such as predicate pushdown, join reordering, and star schema detection, among others.

*3) Presto:* Presto [4] is an open-source distributed query engine for interactive analytics, originally developped at Facebook.

**Execution Model.** Unlike Hive and Spark SQL that use an underlying data processing framework to process queries, Presto follows a Massively Parallel Processing (MPP) architecture that consists of a coordinator and multiple long-running worker processes. When a SQL query is received by the coordinator, it is translated into a DAG of stages. A series of stages is translated into inter-connected tasks which are then scheduled and executed by the workers using multiple threads in a fully pipelined way.

**Query Optimization.** Presto employs a cost-based optimizer that supports predicate pushdown and automatic selection of the build and probe tables when executing shuffle joins. Recently, cost-based join reordering and cost-based selection of the join implementation have been introduced, based on the Hive Metastore statistics too.

### B. Experimental Evaluation

We now present a detailed experimental comparison of Apache Hive, Spark SQL and Presto utilizing the widely used TPC-H [15] benchmark. Our key results are shown in Table V and below:

1) Spark SQL has similar performance with Hive over the entire workload (see Figure 8). Hive is faster than Spark SQL in six queries and has comparable performance with Spark SQL in six others. However, both Presto and Spark SQL are able to execute a query that fails in Hive.
2) Hive and Spark SQL are on average $1.2X$ faster than Presto over the entire TPC-H workload (see Figure 8) although Presto employs a MPP architecture. Presto is, however, similar or faster than Spark SQL and Hive in eight and ten queries respectively.

TABLE V: Summary of results

| | | Hive | Spark SQL | Presto |
|---|---|---|---|---|
| **Query Optimization** | **Join ordering** | V.Good | Good | Good |
| | **Join selection** | V.Good | Good | Poor |
| | **Projection** | V.Good | V.Good | V.Good |
| | **Predicate pushdown** | V.Good | V.Good | V.Good |
| **Operator Optimization** | **Scan implementation** | Good | V.Good | V.Good |
| | **Join implementation** | Good | V.Good | V.Good |

3) Spark SQL benefits from Spark's non-SQL-specific optimizations such as efficient internal data representation mechanisms that reduce the memory footprint and SQL-specific optimizations such as efficient implementations for certain join operators that provide high performance.
4) Although the Catalyst optimizer in Spark SQL occasionally produces better plans than Hive's Calcite, it presents suboptimal behavior with some complex queries as it lacks some optimizations that Hive supports. Presto's optimizer produces suboptimal plans in multiple queries.
5) Presto introduces performance overheads in large queries mostly due to suboptimal memory management.

In the following sections, we present in more detail our experiments and corresponding analysis.

*1) Experimental Setup:* **Hardware and Software Configuration.** For our experiments, we use the 141 node cluster described in Section II-C1. We deploy Hive version 2.3.2 on top of Tez 0.9.0, Spark version 2.4.0 and Presto version 0.216. The systems are deployed on top of Hadoop version 2.7.3. In the case of Spark SQL, we deploy one Spark executor on each of the 140 worker nodes and the Spark driver on the master node. Hive is deployed on top of the YARN resource manager [29], allowing two YARN containers on each worker node. In the case of Presto, we deploy one Presto worker on each of the 140 worker nodes and the coordinator on the master node.

**TPC-H like Workload.** For our experimental evaluation, we use a TPC-H database with a scale factor of 1000 GB. We use the 22 read-only queries of the TPC-H benchmark, but not the refresh streams. Some of the SQL constructs in the TPC-H queries are not supported directly in Hive. For this reason, we rewrote seven queries (Q2, Q11, Q15, Q17, Q20, Q21 and Q22) to use common table expressions (CTEs) instead of nested sub-queries in Hive. In Spark SQL, we also rewrote the CTE of Query 15 to a table for the query to work.

**System Configuration.** In Hive, we enabled cost-based optimization, optimization of correlated queries, predicate push-down and index filtering, map-side join and aggregation, and the vectorized execution engine, among others. We also configured the number of reduce tasks appropriately so that the cluster is fully utilized. In Tez, we enabled the compression of intermediate results with the LZ4 compression codec. In Spark SQL, we enabled cost-based optimization and appropriately tuned the number of reduce tasks. In Presto, we enabled cost-based optimization, phased execution, spilling to disk, intermediate aggregations and compression of intermediate results with the default LZ4 codec. We also increased the buffer size for intermediate data that will be pulled by workers and set the number of hash partitions equal to the number of workers. Finally, we carefully tuned the maximum number of running threads per worker so that the optimal performance is observed on our setup. We use the G1 garbage collector and carefully tuned the JVM settings until all the queries run successfully. Finally, we tuned the broadcast join configurations and collected table and column statistics in all the systems.

We used columnar file formats as they are more suitable for data warehousing queries that typically access a small number of columns. Following prior work [30] and recommendations from system developers [32], we use the ORC file format in Hive and Presto and the Parquet file format in Spark SQL. After experimenting with various compression codecs, we selected the ZLIB and Snappy codecs in ORC and Parquet respectively as they provide the best performance on our setup.

*2) Overall Performance:* We run the 22 TPC-H queries, one after the other and measure the execution time for each query. We do not consider JVM initialization overheads in our evaluation. Before running each query, we flush the file cache at all the compute nodes. We performed three full runs and report the average execution time across the three runs.

Our results are shown in Figure 8. Note that Query 9 failed in Hive because of out of disk space errors, but the remaining queries were completed in all systems. Thus, to evaluate the overall performance of each system, we use the arithmetic and geometric mean of the execution times of all the queries but Query 9. Our results show that Hive has similar overall performance with Spark SQL. We also find that Spark SQL and Hive are on average $18\%$ faster than Presto over the entire workload when the arithmetic mean is the metric used to compare the systems. They are however $26\%$ faster than Presto when the geometric mean is used. It is worth noting, that if we do not include Query 16 in our metrics (as it is significantly slower in Spark SQL than the remaining queries), Spark SQL is on average $27\%$ faster than the other systems using a geometric mean.

Our analysis shows that the observed performance differences are due to the following reasons:

**Impact of the Query Optimizer.** As shown in Table V, Hive generally has a more sophisticated optimizer than Spark SQL and Presto. Although the Catalyst optimizer in Spark SQL occasionally produces better plans than Hive's Calcite, it is inefficient in certain complex queries and does not employ some optimizations that Hive supports. Presto's optimizer, on the other hand, often produces suboptimal plans and requires many improvements in its cost estimator [33].

In particular, Presto's optimizer often produces sub-optimal plans when combining features such as cost-based join reordering and cost-based join selection (Table V). In some cases, such as Queries 8, 11 and 20, we disabled the cost-based join reordering as it led to a much worse performance. For example, in Query 11, although Presto and Hive picked the same join order, Presto executed two shuffle joins when using cost-based optimization whereas Hive executed only

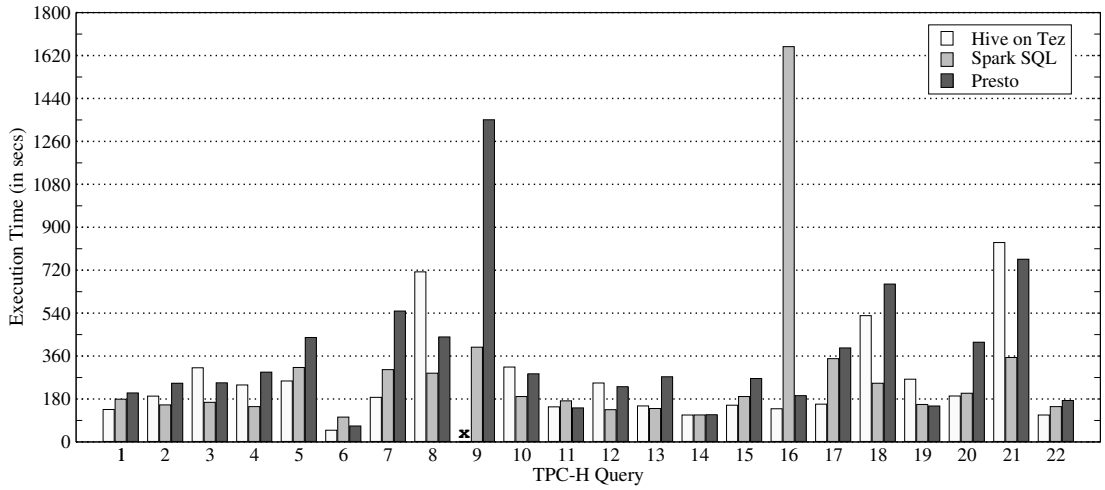



Fig. 8: TPC-H performance of Hive, Spark SQL and Presto

broadcast joins. By disabling the cost-based join reordering Presto selected a broadcast join and was able to run this query $50\%$ faster than before. In multiple other queries (e.g. Q2, Q7, Q8, Q9, Q18, Q20, etc.) Presto picks a plan with either a suboptimal join type or a suboptimal join order irrespective of whether the cost-based join reordering is used. To quantify the impact of Presto's optimizer, we manually split certain Presto queries with suboptimal plans (the 6 aforementioned queries) into multiple sub-queries so that the resulting plans are similar to the optimal ones generated by either Spark SQL or Hive. We observed that Presto's performance in these queries improved by $50\%$.

As previously noted, Spark SQL occassionally produces better plans than Hive (e.g., Query 8) but in general lacks some features of Hive's optimizer. As a result, Hive is much faster than Spark SQL in five queries (Q6, Q7, Q16, Q17, Q22). We now analyze some of these queries focusing on the join selection and join re-ordering capabilities of the optimizers.

Spark SQL is $2.5X$ faster than Hive when executing Query 8. This query joins the `Lineitem` table with the `Part` table. We observed that Hive is $5X$ slower than Spark SQL when executing this join operation. Hive's optimizer picks a sort-merge join implementation to execute this join which requires a data shuffle of the large `Lineitem` table. Spark SQL, on the other hand, avoids data shuffling by performing a broadcast hash join implementation. Thus, in this case Spark's Catalyst optimizer is able to generate a much more efficient query plan than the Calcite optimizer.

Spark SQL is approximately $12X$ slower than Hive when executing Query 16. This query contains a nested sub-query in a NOT IN clause which returns the keys of the `Partsupp` table that do not match with a specific set of keys in the `Supplier` table. Spark SQL executes a left anti join for this operation and uses a broadcast nested loop join implementation. We noticed that this operation takes $1508$ secs to complete which constitutes $91\%$ of the total query time in Spark SQL. Hive, on the other hand, selects a different logical plan and uses a broadcast hash join implementation which completes in about 60 secs.

The TPC-H Query 7 is about $1.6X$ faster in Hive than Spark SQL due to Hive's better join reordering mechanisms. This query contains multiple join operations across six tables, including the `Lineitem` and `Orders` tables which are the largest tables of the TPC-H benchmark. Figure 9 shows the query plans produced by Hive and Spark for this query. As shown in the figure, Hive first performs joins with the two small `Nation` tables, which produce relatively small outputs and thus prune a large amount of data. Spark SQL on the other hand, first performs a join between the large `Lineitem` and `Orders` (*Inner Join 1*) tables, which requires a shuffle of two large tables and produces a large number of intermediate results that are given as input to the subsequent join operators in the query plan. On the other hand, Hive's join reordering strategy is more effective and thus, it ends up using more map-side joins (e.g. *Inner Join 3*, *Inner Join 4*).

Finally, TPC-H Query 17 is another interesting query, where Spark SQL is about $2.2X$ slower than Hive. In this query Hive generates a plan that applies dynamic min-max filtering on the *l_partkey* column of the `Lineitem` table before performing an aggregation on it, allowing it to skip entire stripes/rows using the ORC file indices. We omit a more thorough analysis in the interest of space.

**Impact of Memory Management.** A major advantage of Spark SQL over Hive and Presto is its efficient implementation that avoids object creations. Spark SQL exploits its serialization mechanisms and serializes the data into off-heap storage in binary format. It then performs transformations directly on this format. Thus, it avoids object creation overheads while reducing the memory footprint. This results in better join and scan performance in some queries (see Table V). Hive, on the other hand, performs extensive object creation which affects performance and increases the amount of memory used during scans and join operations. As a result, Hive is approximately $75\%$ slower than Spark SQL in four TPC-H queries (Q3, Q4, Q12, and Q19) although both systems produce similar query plans. Queries 8, 10, 18, and 21 are also affected, irrespective of the fact that Hive has better join ordering in the produced plans due to the large amount of data that is processed.

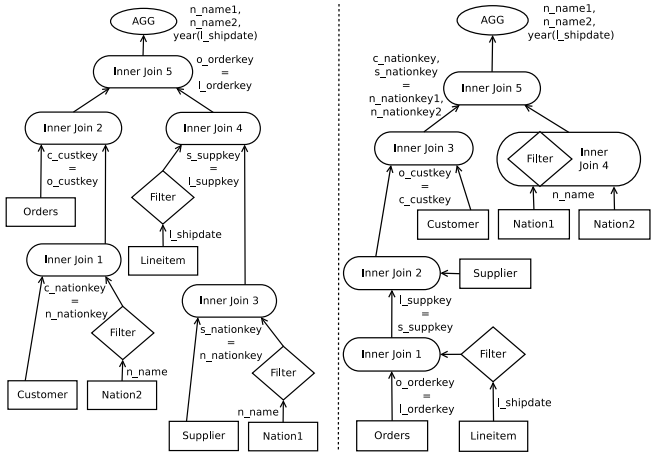
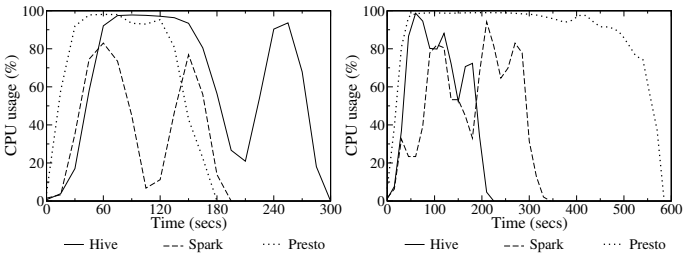



Fig. 9: Hive (left) and Spark SQL (right) query plan for Q7

The impact of such optimizations in Hive can be seen in Query 9. This query performs multiple join operations that generate a large number of intermediate results. During query execution, Hive quickly runs out of memory and starts spilling data to disk. Hive eventually fails because of out of disk space errors, whereas Spark SQL and Presto succesfully complete the query with less data spilling.

To better understand the effect of object creation in Hive we also analyze Query 3, where all of the systems produced the same plan (Queries 4, 12, and 19 have similar behavior). Query 3 joins the `Lineitem` table with an intermediate table that is produced by joining the `Orders` and `Customer` tables. All the systems use a sort-merge join. We observe that Hive is $2X$ slower than Spark SQL during the join operation. Our analysis shows that the performance of the sort operator at the map-side is similar in both systems. However, Hive spends significantly more time processing the sorted data that received over the network and feeding them into the merge operator at the reduce phase. In particular, approximately $90\%$ of the the reducer's time is spent creating new objects that are fed into the merge operator. Spark SQL on the other hand, does not have such overheads. Another insteresting observation from Query 3 is that Spark SQL is $2X$ faster when scanning the large `Lineitem` and `Orders` tables. By profiling the query, we observe that Hive has deserialization overheads due to object creations that Spark SQL avoids.

Finally, Presto uses efficient flat in-memory representations of columns and processes data by generating unrolled loops over columns. Although we would expect it to be as efficient as Spark SQL in memory management, we instead experienced low memory usage and increased execution time in queries that read and process a large amount of data. Many queries are affected, including Q4, Q8, Q9, Q10, Q12, Q18 and Q21 irrespective of the plan used. The reason behind this suboptimal performance is that Presto has a reserved memory pool which is useful when handling concurrent workloads but leads to unnecessary memory limitations for sequential workloads [34]. Unfortunately, we could not disable the reserved pool and we were also unable to configure it appropriately with the available memory configurations. Nevertheless, in other




(a) TPC-H Query 19 (b) TPC-H Query 7

Fig. 10: Cluster CPU usage for TPC-H Queries 19 and 7

smaller queries, Presto was similar of better than Spark SQL and Hive in eight and ten queries respectively.

**Impact of the Execution Model.** As mentioned in Section III-A, Presto follows a generic MPP architecture where all tasks are fully pipelined in contrast with Spark and Hive which use a MR-based execution model and consecutive stages must be scheduled and launched one after the other. The impact can be seen in Figure 10 where the cluster CPU utilization is presented for two different queries i.e. Q19 and Q7. In both queries, Hive and Spark present some CPU under-utilization between consecutive stages while Presto appears to be fully utilizing the cluster's CPU during query execution. In particular, in Query 19 all the systems produce the same query plan and Presto is faster than both Hive and Spark as we can see in Figure 10a. However, in Query 7 all systems have different plans (as we previously analyzed), with Hive's being the better one and Presto's the worst which is also depicted in the execution time in Fig. 10b. In general, the benefits of Presto's MPP architecture with respect to CPU utilization, cannot yet be fully quantified as there still exist other factors that affect its overall performance such as the query plans and the memory limitations.

Moreover, our analysis shows that Presto can often introduce performance overheads if the number of threads assigned to the operators is not configured appropriately. For our experiments, we configured several parameters so that the number of threads used by each worker is small given that our worker nodes have two physical cores. With the default configuration, Presto generated a very large number of threads which introduced context switching overheads that negatively impacted performance. In particular, when using the default parameters Presto becomes $2X$ slower across the entire workload. We believe that automatically determining the number of threads based on the available hardware and data size will not only improve performance but will lead to better user experience as well.

## IV. Related Work

Over the last decade, numerous studies have evaluated the performance of data management systems. The work in [35] compares the MapReduce system with traditional relational database systems (RDBMSs). Follow-up work [36] shows that with appropriate configuration, the performance of MapReduce can reach that of parallel RDBMSs on certain workloads.

The work in [30] experimentally compares the shared-nothing database architecture adopted by systems like Im-

pala [3] with Hive's MapReduce-based architecture. The work in [37] compares the performance and scalability of SQL-on-Hadoop engines with that of parallel RDBMSs. The work in [38] shows that it is possible to design a highly-efficient SQL-on-Hadoop engine by building upon a mature RDBMs.

Driven by the increased popularity of the Spark framework, the authors of [39] present an experimental evaluation of Spark and MapReduce focusing on their shuffling and caching mechanisms as well as their data pipelining capabilities. The work in [40] offers a more limited analysis on a smaller scale of Hive, Spark SQL and AsterixDB under the TPC-H workload. Finally, the work in [41] presents a detailed experimental evaluation of various Big Data Systems (including Spark and TensorFlow) focusing on image analytics workloads.

## V. Conclusions

In this work, we present a thorough performance study of general-purpose and specialized data analytics systems. In particular, we select representative systems from each class, namely Spark, TensorFlow, Presto, and Hive and present a detailed overview of their architecture and functionality. We then benchmark them on workloads from the machine learning and SQL domains in a large-scale cloud setting and highlight the strengths and limitations of each system.

Overall, the general-purpose engine (Spark) is superior during data reading (loading raw bytes, decoding and deserializing). In the ML domain, the specialized engine (TensorFlow) outperforms Spark's MLlib during gradient computation, mainly due to the fundamental architectural differences of the two systems. However, in the SQL domain it is not straightforward to announce a clear winner, as Hive has the best query optimizer and Spark the best memory management.

As part of future work, we would like to extend this comparison to include other aspects of the systems and different analysis domains such as graph and streaming analytics.

## VI. Acknowledgment

This paper is supported by European Union's Horizon 2020 Framework Programme - Grant Agreement Number: 780245, project E2Data (European Extreme Performing Big Data Stacks). The computational and storage resources were granted by the National Infrastructures for Research and Technology (GRNET) under project with id bf760e86-2e06-4c51-bd7c-332e587eb98a.

## References

[1] T. Hey, S. Tansley, K. M. Tolle *et al.*, *The fourth paradigm: data-intensive scientific discovery*. Microsoft research, WA, 2009, vol. 1.

[2] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, March 2010, pp. 996–1005.

[3] "Apache Impala." https://impala.apache.org/.

[4] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner, "Presto: Sql on everything," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, April 2019, pp. 1802–1813.

[5] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX, 2016, pp. 265–283.

[6] "GraphLab." https://en.wikipedia.org/wiki/GraphLab.

[7] "Apache Mahout." http://mahout.apache.org/.

[8] "Apache Giraph." https://giraph.apache.org/.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010.

[10] "Spark SQL." http://spark.apache.org/sql/.

[11] "MLlib." https://spark.apache.org/mllib/.

[12] "Spark Streaming." https://spark.apache.org/streaming/.

[13] "GraphX." https://spark.apache.org/graphx/.

[14] "Apache Flink," https://flink.apache.org/.

[15] "TPC-H." http://www.tpc.org/tpch/.

[16] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," *CoRR*, vol. abs/1206.5533, 2012. [Online]. Available: http://arxiv.org/abs/1206.5533

[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 *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI '14)*. ACM, 2014, pp. 583–598.

[18] A. Smola and S. Narayanamurthy, "An architecture for parallel topic models," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 703–710, 2010.

[19] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, 2010, pp. 2595–2603.

[20] "Okeanos," https://okeanos.grnet.gr/home/.

[21] V. Koukis, C. Venetsanopoulos, and N. Koziris, "˜ okeanos: Building a cloud, cluster by cluster," *IEEE internet computing*, vol. 17, no. 3, pp. 67–71, 2013.

[22] "Linear Regression," https://en.wikipedia.org/wiki/Linear_regression.

[23] "Logistic Regression," https://en.wikipedia.org/wiki/Logistic_regression.

[24] "Multilayer Perceptron," https://en.wikipedia.org/wiki/Multilayer_perceptron.

[25] "The UCI Machine Learning Repository," http://archive.ics.uci.edu/ml/index.php.

[26] "The HIGGS Dataset," https://archive.ics.uci.edu/ml/datasets/HIGGS.

[27] "The YearPredictionMSD Dataset," https://archive.ics.uci.edu/ml/datasets/yearpredictionmsd.

[28] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *SIGMOD*, 2015, pp. 1357–1369.

[29] "Apache Hadoop YARN." https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[30] A. Floratou, U. F. Minhas, and F. Özcan, "Sql-on-hadoop: Full circle back to shared-nothing database architectures," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1295–1306, 2014.

[31] "Apache Calcite," https://calcite.apache.org/.

[32] "Reading Parquet Files in Spark," https://docs.databricks.com/spark/latest/data-sources/read-parquet.html.

[33] https://github.com/prestodb/presto/issues/11669.

[34] https://github.com/prestodb/presto/issues/10512.

[35] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A Comparison of Approaches to Large-scale Data Analysis," in *SIGMOD*, 2009, pp. 165–178.

[36] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of MapReduce: an in-depth study," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 472–483, 2010.

[37] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang, "Can the Elephants Handle the NoSQL Onslaught?" *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1712–1723, Aug. 2012.

[38] A. Costea *et al.*, "VectorH: Taking SQL-on-Hadoop to the Next Level," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. ACM, 2016, pp. 1105–1117.

[39] 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," *VLDB*, vol. 8, no. 13, pp. 2110–2121, 2015.

[40] P. Pirzadeh, M. Carey, and T. Westmann, "A performance study of big data analytics platforms," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec 2017, pp. 2911–2920.

[41] P. Mehta, S. Dorkenwald, D. Zhao, T. Kaftan, A. Cheung, M. Balazinska, A. Rokem, A. Connolly, J. Vanderplas, and Y. AlSayyad, "Comparative evaluation of big-data systems on scientific image analytics workloads," *VLDB*, vol. 10, no. 11, pp. 1226–1237, 2017.