Benchmarking Dataflow Systems for Scalable Machine Learning

vorgelegt von
Dipl.-Ing. Christoph Boden
geb. in Berlin

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
- Dr. Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Tilmann Rabl
Gutachter:	Prof. Dr. Volker Markl
Gutachter:	Prof. Dr. Klaus-Robert Müller
Gutachter:	Prof. Dr. Asterios Katsifodimos

Tag der wissenschaftlichen Aussprache: 21. September 2018

Berlin 2018

Acknowledgements

First and foremost, I would like to thank my advisor Prof. Volker Markl for introducing me to the academic world of database systems research and for providing me with the opportunity to carry out this research. This thesis would not have been possible without the great research environment he created and his continuing support. I would also like to thank Prof. Tilmann Rabl for introducing me to the world of Benchmarking and for his relentless support and fruitful advice. Furthermore, I want to express my gratitude to Prof. Klaus-Robert Müller and Prof. Asterios Katsifodimos for agreeing to review this thesis.

I'd like to thank my colleagues, first and foremost Dr. Sebastian Schelter whose supportive and insightful advice was invaluable in the pursuit of my research as well as Dr. Alan Akbik for sharing his academic curiosity and enthusiasm. I would like to thank my collaborators and co-authors Alexander Alexandrov and Andreas Kunft as well as all of my current and former colleagues at the Database Systems and Information Management Group at TU Berlin who made this an inspiring place to pursue my research, including Max Heimel and Marcus Leich who helped me advance my research through advice and discussions. I am also grateful for having had the opportunity to advise the DIMA students André Hacker and Andrea Spina who carried out excellent thesis work.

Finally I would like to thank my proof readers Dr. Alan Akbik, Steve Aurin, Dr. Conrad Friedrich, Felix Neutatz and Prof. Tilmann Rabl.

Abstract

The popularity of the world wide web and its ubiquitous global online services have led to unprecedented amounts of available data. Novel distributed data processing systems have been developed in order to scale out computations and analysis to such massive data set sizes. These "Big Data Analytics" systems are also popular choices to scale out the execution of machine learning algorithms. However, it remains an open question how efficient they perform at this task and how to adequately evaluate and benchmark these systems for scalable machine learning workloads in general. In this thesis, we present work on all crucial building blocks for a benchmark of distributed data processing systems for scalable machine learning including extensive experimental evaluations of distributed data flow systems.

First, 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 specify data sets, workloads, experiments and metrics that address all relevant aspects of scalability, including the important aspect of model dimensionality. We provide results of a comprehensive experimental evaluation of popular distributed dataflow systems, which highlight shortcomings in these systems. Our results show, 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.

Second, we propose methods and experiments to explore the trade-off space between the runtime for training a machine learning model and the model quality. We make the case for state of the art, single machine algorithms as baselines when evaluating distributed data processing systems for scalable machine learning workloads and present such an experimental evaluation for two popular and representative machine learning algorithms with distributed data flow systems and single machine libraries. Our results show, that even latest generation distributed data flow systems require substantial hardware resources to provide comparable prediction quality to a state of the art single machine library within the same time frame. This insight is a valuable addition for future systems papers as well as for scientists and practitioners considering distributed data processing systems for applying machine learning algorithms to their problem domain.

Third, we present work on reducing the operational complexity of carrying out benchmark experiments. We introduce a framework for defining, executing, analyzing and sharing experiments on distributed data processing systems. On the one hand, this framework automatically orchestrates experiments, on the other hand, it introduces a unified and transparent way of specifying experiments, including the actual application code, system configuration, and experiment setup description enabling the sharing of end-to-end experiment artifacts. With this, our framework fosters reproducibility and portability of benchmark experiments and significantly reduces the "entry barrier" to running benchmarks of distributed data processing systems.

Zusammenfassung

Die Popularität des World Wide Web und seiner allgegenwärtigen global verfügbaren Online-Dienste haben zu beispiellosen Mengen an verfügbaren Daten geführt. Im Lichte dieser Entwicklung wurden neuartige, verteilte Datenverarbeitungssysteme (sogenannte "Big Data Analytics"-Systeme) entwickelt, um Berechnungen und Analysen auf solch massive Datengrößen skalieren zu können. Diese Systeme sind ebenfalls beliebte Ausführungsumgebungen für das Skalieren von Algorithmen des maschinellen Lernens. Es ist jedoch eine offene Frage, wie effizient diese "Big Data Analytics"-Systeme bei der Ausführung von skalierbaren Verfahren des maschinellen Lernens sind und wie man solche Systeme adäquat evaluieren und benchmarken kann. In dieser Doktorarbeit stellen wir Arbeiten für alle essenziellen Bausteine einer solchen Evaluierung von verteilten Datenverarbeitungssystemen für skalierbare Methoden des Maschinellen Lernens, inklusive einer umfassenden experimentellen Evaluierung von verteilten Datenflusssystemen, vor.

Zunächst stellen wir einen repräsentativen Satz verteilter maschineller Lernalgorithmen vor, welche für den Einsatz in massiv verteilten Umgebungen passend sind. Diese Lernalgorithmen besitzen substanzielle Ähnlichkeit zu einer breiten Palette von industrierelevanten Verfahren und bieten daher verallgemeinerbare Einblicke in die Systemleistung. Wir definieren Datensätze, Algorithmen, Experimente und Metriken, die alle relevanten Aspekte von Skalierbarkeit, einschließlich des wichtigen Aspekts der Modelldimensionalität abdecken. Wir präsentieren und diskutieren die Ergebnisse unserer umfassenden experimentellen Evaluierung gängiger verteilter Datenflusssysteme. Unsere Ergebnisse zeigen, dass die untersuchten aktuellen Datenflusssysteme zwar robust bzgl. der Anzahl der Rechner sowie der Datengröße skalieren können, jedoch bei der Skalierung der Modelldimensionalität substanzielle Schwächen aufweisen. Diese Ineffizienz überrascht, da die Bewältigung hochdimensionaler Daten eine Kernanforderung für das Ausführen skalierbarer Maschineller Lernverfahren darstellt.

Zweitens, schlagen wir Methoden und Experimente vor, um den zwischen Laufzeit des Trainings eines Modells des maschinellen Lernens und der Vorhersagequalität dieses Modells aufgespannten Raum zu erkunden. Wir argumentieren, dass effiziente, dem Stand der Technik entsprechende, Einzelmaschinenbibliotheken als Basis in vergleichenden Experimenten herangezogen werden sollen. Wir präsentieren Ergebnisse einer solchen Evaluierung für zwei populäre und repräsentative Algorithmen des maschinellen Lernens auf verteilten Datenflusssystemen und mit Einzelmaschinenbibliotheken. Die Ergebnisse unserer Experimente zeigen, dass selbst die neuesten verteilten Datenflusssysteme substanzielle Hardwareressourcen benötigen, um eine vergleichbare Vorhersagequalität zu Einzelmaschinenbibliotheken innerhalb vergleichbarer Trainingszeiträume zu erreichen. Dies ist eine wichtige Erkenntnis, welche für zukünftige Forschung und Entwicklung im Bereich der Datenverarbeitungssysteme zur Kenntnis genommen werden muss, aber auch eine relevante Information für Wissenschaftler und Anwender dieser Systeme, welche die Anwendung von verteilten Datenflusssystemen für Algorithmen des maschinellen Lernens in ihrer Domäne in Betracht ziehen.

Drittens, präsentieren wir Arbeiten zur Reduzierung der operativen Komplexität bei der Durchführung von Benchmark-Experimenten. Wir stellen ein Framework für

die Definition, Ausführung und Analyse von Experimenten auf verteilten Datenverarbeitungssystemen vor. Auf der einen Seite orchestriert unser Framework automatisch Experimente, auf der anderen Seite führt es eine einheitliche und transparente Art und Weise, Experimente zu spezifizieren, ein. Hierbei werden neben der eigentlichen Implementierung der Benchmarkalgorithmen auch sämtliche Parameter der Systemkonfiguration und die Beschreibung des Experimentaufbaus und der beteiligten Systeme und Komponenten inkludiert. Somit wird eine transparente Verfügbarmachung und das Teilen von kompletten “End-to-End” Experimentartefakten ermöglicht. Hierdurch fördert unser Framework die Reproduzierbarkeit und Portabilität von Benchmark-Experimenten und reduziert die "Eintrittsbarriere" bzgl. der Durchführung von Benchmarks für verteilte Datenverarbeitungssysteme signifikant.

Contents

1 Introduction

The advent of the World Wide Web has led to a massive increase of available data. In light of rapidly decreasing storage costs, the ubiquity of global online services and smart mobile phones, text, audio, and video data as well as user interaction logs are being collected at an unprecedented scale. This data has successfully been leveraged to build and tremendously improve data-driven applications [70]. The availability of this data has also revolutionized scientific research as it became possible to test hypotheses on samples several orders of magnitude larger and significantly more diverse than before. In computational social sciences for example, the increased availability of online social interaction data has offered new opportunities to map out the network structure of diffusion processes, i.e., the adoption of products or ideas through interpersonal networks of influence. Empirical analysis showed that, contrary to a longstanding hypothesis, adoptions resulting from long chains of referrals, as the analogy to the spreading of diseases would suggest, are extremely rare. The vast majority of cascades are small, and are described by a handful of simple tree structures that terminate within one degree of an initial adopting seed node [62]. Next to web and user interaction data, scientists are also collecting, releasing and aggregating massive amounts of observational and simulation data that enable anyone to perform their own analysis. For example, in Material Science, the NOMAD repository [93] makes available millions of total-energy calculations of materials data that have been produced by by CPU-intensive computations. The Sloan Digital Sky Survey [107] transformed astronomy by making available hundreds of terabytes of data including deep multi-color images of one third of the sky, and spectra for more than three million astronomical objects. In genomics, next-generation sequencing (NGS), which can produce billions of short DNA or RNA fragments in excess of a few terabytes of data in a single run, leads to massive data-sets driving personalized cancer treatment and precision medicine research [102]. All of these phenomena have been subsumed under the term *Big*

Data or *Big Data Analytics*.

In order to enable the processing and analysis of web scale data, the large web companies, foremost Google, developed novel storage and processing systems based on the use of large shared-nothing clusters of commodity hardware. Google's *MapReduce* [47], based on the *Google File System (GFS)* [60], was built to simplify large-scale special-purpose computations like the construction of inverted indices from a raw Web crawl for Web search [49] through automatic parallelization and distribution of computation. *MapReduce* abstracts away the complexity of scheduling a program's execution on large clusters, managing the inter-machine communication as well as coping with machine failures by exposing a simple functional programming API to users. It handles semi-structured data and does not require a pre-defined schema, enabling it to process a large variety of input data. Its open-source implementation *Hadoop* [8] and the *Hadoop Distributed File System (HDFS)* [105] have been widely adopted as a solution to robustly scale data-intensive applications to very large data sets on clusters of commodity hardware and are commonly referred to as *Big Data Analytics Systems*.

The availability of massive data sets and large scale data processing systems combined with machine learning algorithms have enabled remarkable breakthroughs in a number of core tasks including ranking web search results [31, 48], personalized content recommendation [46, 73], statistical language models for speech recognition and machine translation [64], click-through rate prediction for online advertisements [84, 98], credit scoring, fraud detection and many other applications [52]. It became apparent that for several problem settings, comparatively simple algorithms could attain superior performance to more complex and mathematically sophisticated approaches when being trained with enough data [64] . This effect, sometimes referred to as "the unreasonable effectiveness of data" became most obvious in the domain of statistical natural language processing [27]. Thus, in consequence of the sheer size of available data sets and the remarkable successes of machine learning algorithms for a variety of tasks, an unprecedented demand to efficiently scale the execution of machine learning algorithms materialized.

As focus shifted from rather simple "extract-transform-load" and aggregation jobs to the scalable execution of more complex workflows such as inferring statistical models and machine learning algorithms, it quickly became apparent that Hadoop MapReduce was inherently inefficient at executing such workloads [71, 100]. While many popular

machine learning algorithms can easily be formulated in the functional *MapReduce* programming model [43] on a logical level, the acyclic data flow model underlying Hadoop's implementation and the intricacies of its distributed implementation lead to unsatisfactory performance. Particularly the fixed *Map-Shuffe-Reduce* pipeline, the inability to efficiently execute *iterative computations* as well as *ad-hoc analysis* turned out to be major drawbacks of Hadoop [71, 100].

This shortcoming sparked the development of a multitude of novel approaches and systems aiming to improve the performance and ease of implementation of more complex iterative workloads such as distributed machine learning algorithms in the distributed systems and database systems research communities [33, 53, 55, 81, 113].

However, while these *Second Generation Big Data Analytics Systems* have been shown to outperform Hadoop MapReduce for canonical iterative workloads [18, 80, 113], it remains an open question how effectively they perform for actual large scale machine learning problems due to at least two major factors. First, the learning algorithms chosen in the corresponding systems papers are those that fit well onto the system's paradigm (e.g., batch gradient descent solvers for linear models) rather than state of the art methods, which would be chosen to solve a supervised learning problem in the absence of these systems' constraints and provide state of the art performance with respect to prediction quality.

Second, the experiments in scientific publications associated with these systems also generally fail to address aspects of scalability that are inherent to machine learning algorithms such as scaling the data and model dimensionality. However, understanding the systems' behaviour is crucial for machine learning practitioners, who need to assess the systems suitability to their problem setting. For this, machine learning researchers and other potential users who want to design and implement machine learning or analysis algorithms on top of these systems, need to understand the trade-offs these systems possess.

Finally, different workloads and implementations, usage of libraries, data sets and hardware configurations make it hard if not impossible to leverage the published experiments for an objective comparison of the performance of each system. It is a challenge to assess how much of reported performance gain is due to a superior paradigm or design and how much is due to a more efficient implementation of algorithms, which ultimately

impairs the scientific process due to a lack of verifiability. These shortcomings limit progress in the research and development of novel (distributed) machine learning systems. For this reason, it is crucial to enable and establish benchmarks for big data analytics systems with respect to scaling machine learning algorithms in order to steer systems research into a fruitful direction.

1.1 Thesis Statement

In order to objectively assess how the systems resting on the paradigm of distributed dataflow perform at scaling machine learning algorithms and to guide future systems research for distributed machine learning, it is imperative to benchmark and evaluate these systems for relevant and representative end-to-end machine learning workloads in a reproducible manner and to address all aspects of scalability. In this thesis, we present work in all of these areas. Our work identifies significant shortcomings in current distributed data processing systems with respect to efficiently executing machine learning algorithms.

1.2 Main Contributions

Benchmarking Scalability. 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 the comprehensive experimental evaluations. To ensure reproducibility we provide our benchmark algorithms on top of *Apache Flink* and *Apache Spark* as open-source software defining and executing experiments for distributed systems and algorithms. We provide a comprehensive set of experiments to assess their scalability with respect to both: data set size and dimensionality of the data based on mathematically equivalent versions of these algorithms. 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 that train models

on high dimensional data. Being able to assess the limitations and trade-offs inherent to these systems is crucial for machine learning researchers as well as other scientists who intend to apply distributed data flow systems to their problem domain.

Benchmarking Performance and Model Quality. We argue that systems should consider state of the art, single machine algorithms as baselines. Solely evaluating scalability and comparing with other distributed systems is not sufficient and provides insufficient insights. Distributed data processing systems for large scale machine learning should be benchmarked against sophisticated single machine libraries that practitioners would choose to solve an actual machine learning problem and evaluated with respect to both: runtime as well as prediction quality metrics. We present such a Benchmark and provide an experimental evaluation of state of the art machine learning libraries *XGBoost*, *LightGBM* and *Vowpal Wabbit* for supervised learning and compare them to *Apache Spark MLlib*, one of the most widely used distributed data processing system for machine learning workloads. Results indicate that while distributed data flow systems such as *Apache Spark* do provide robust scalability, it takes a substantial amount of extra compute resources to reach the performance of a single threaded or single machine implementation. The current lack of such experiments and evaluations is most likely due to the division of the communities conducting research in data processing systems on the one hand and machine learning systems on the other hand. While runtime of algorithms tends to be at best a secondary concern in machine learning research, the out-of-sample prediction performance of trained machine learning models is of little to no concern to systems researchers who frequently only report per-iteration runtimes without any information on model quality at all. With this work we intend to connect these research communities since understanding the trade-offs with respect to runtime and model quality in the context of distributed data processing systems is essential to both: machine learning researchers aiming to develop novel algorithms on these systems as well as systems researchers aiming to improve the systems for machine learning use cases.

Reproducibility of Benchmarks. In order to foster reproducible and portable experiments, our benchmarks are grounded in a framework to define, execute, analyze, and share experiments, enabling the transparent specification of benchmarking workloads and system configuration parameters. On the one hand, this framework automatically

orchestrates experiments and handles the systems' setup, configuration, deployment, teardown and cleanup as well as automatic log collection. On the other hand, it introduces a unified and transparent way of specifying experiments, including the actual application code, system configuration, and experiment setup description enabling the sharing of end-to-end experiment artifacts, thus fostering reproducibility and portability of benchmark experiments. It also allows for the hardware independent specification of these parameters, therefore enabling portability of experiments across different hardware setups. As a significant step towards adequate benchmarking of distributed data processing systems, this approach and framework contributes to guiding future systems research into a fruitful direction.

The core parts of this thesis have been published as follows:

- **Christoph Boden, Andrea Spina, Tilmann Rabl, Volker Markl:** *Benchmarking Data Flow Systems for Scalable Machine Learning* Algorithms and Systems for MapReduce and Beyond (BeyondMR) Workshop at SIGMOD 2017
- **Christoph Boden, Alexander Alexandrov, Andreas Kunft, Tilmann Rabl and Volker Markl:** *PEEL: A Framework for benchmarking distributed systems and algorithms* 9th TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC) at VLDB 2017
- **Christoph Boden, Tilmann Rabl and Volker Markl:** *Distributed Machine Learning - but at what COST?* ML Systems Workshop @ NIPS 2017

In preliminary research that we contributed to, we identified the shortcomings of the MapReduce paradigm and its Hadoop implementation with respect to scaling machine learning algorithms for recommendation systems:

- **Sebastian Schelter, Christoph Boden, Martin Schenck, Alexander Alexandrov and Volker Markl** *Distributed Matrix Factorization with MapReduce using a series of Broadcast-Joins* ACM Recommender Systems conference, RecSys 2013
- **Sebastian Schelter, Christoph Boden, Volker Markl** *Scalable Similarity-Based Neighborhood Methods with MapReduce* ACM Recommender Systems 2012

The insights obtained were a crucial prerequisite for developing and conducting the benchmark experiments we present in this thesis.

Impact

The benchmarking framework as well as code for all of our experiments are available as open source software on `GitHub` and have been used by other researchers, students and practitioners, for example by researchers of the ZUSE Institute Berlin (ZIB). The papers we published on the topics presented in this thesis have been published and presented to an international academic audience and have been cited multiple times by state of the art research papers. Next to the academic venues, we also presented our research to interested audiences, for example the SPEC Research Big Data Group.

1.3 Thesis Outline

The remainder of this thesis is structured as follows:

In **Chapter 2** we provide the background necessary for the later chapters of this thesis. We retrace the evolvement of distributed data processing systems from the perspective of a database researcher and emphasize for which requirements these systems were originally conceived. After reviewing fundamental concepts in scalability and parallelism we introduce the relevant massively parallel data processing systems and models. Next, we provide a concise introduction to machine learning on these data processing systems and also provide an overview of benchmarking efforts in the context of database systems.

In **Chapter 3** we present our work on the important aspect of scalability in the context of benchmarking distributed dataflow systems for machine learning workloads including the important one of feature space and model dimensionality. We define and characterize data sets, workloads, experiments and metrics and report experimental results for all of the defined experiments for the state of the art data flow systems Apache Spark and Apache Flink. Our results show that while both systems robustly cope with scaling the number of compute nodes or the data set size, they severely struggle with increasing model dimensionality.

In **Chapter 4** we present our work on exploring the model quality and runtime performance trade-off for distributed and single machine implementations of machine learning algorithms. Given the main memory sizes generally available today, we deem

it necessary to evaluate strong single machine baselines to assess the performance and potential benefit of scaling out machine learning algorithms on distributed data flow systems. We present data sets, selected algorithms and a methodology to explore the model quality vs. runtime trade-off for single machine and distributed approaches. Our results show that even latest generation distributed data flow systems such as Apache Spark require substantial hardware resources to provide comparable performance to a state of the art single machine library.

In **Chapter 5** we present our work on reducing the operational complexity of benchmark experiments. With PEEL, we introduce a framework for defining, executing, analyzing and sharing experiments on distributed data processing systems. We present a domain model and the core abstractions to specify the configurations and system environments of all systems involved, as well as the experiments and parameters to be varied. The PEEL framework has been successfully applied for all benchmark evaluations presented in this thesis and can easily be extended to include other systems.

Finally in **Chapter 6** we summarize our findings and discuss their implications for the systems research and benchmarking community and beyond. We examine limitations of our approach and outline interesting research questions that could be the subject of future work.

2 Background

This chapter introduces the major concepts and the background necessary for the the remainder of this thesis. First, we provide a brief historical overview of the development of large scale data processing systems from the perspective of the database research community in Section 2.1, emphasizing the use-cases and requirements each system was originally built to fulfill. We introduce fundamental concepts and terminology in parallelism and scalability in Section 2.2 and present the massively parallel data processing abstractions and systems relevant to this thesis in sufficient detail in Section 2.3. This overview provides the basis to understand benchmark results as well as inherent limitations of the different systems in later chapters. After having established both the historical development as well as the technological foundation of the relevant systems, we introduce essential terminology and concepts of machine learning in Section 2.4, including relevant models and algorithms. We succinctly discuss fundamental aspects of implementing machine learning algorithms in the context of distributed data processing systems. Finally, in Section 2.5 we provide a concise overview of benchmarking efforts in the field of database systems.

2.1 A Brief History of Big Data Analytics Systems

While the enormous hype surrounding the term "Big Data" in the last years may suggest this is a recent phenomena, the database research community and database vendors have actually been proposing and building solutions to handle "very large" data sets for decades [25]. Based on the relational model invented by Ted Codd in 1970 [44], relational database systems gained commercial adoption as enterprise data management systems in the early 1980's. Eventually, the focus shifted from merely processing and storing day-to-day transactional data to analyzing historical business data with large relational queries in

data warehouses to drive reporting and business analytics. This led to the development of software-based parallel database systems based on a *shared-nothing* architecture [50]. The *shared-nothing* architecture implies a set of interconnected but independent machines, each equipped with their own processors, main memory and storage as well as operating system and software. The machines can only communicate via message passing, necessitating that all inter-machine coordination and data exchange take place over the network. Based on this shared-nothing architecture, parallel database systems introduced the usage of divide-and-conquer parallelism based on hash partitioning the data for both storage as well as the execution of relational operators for query processing. Examples of such parallel database systems include GAMMA [51], GRACE, [58], Teradata [103] (the first successful commercial parallel database system [25]) as well as IBM DB2 Parallel Edition [20].

The advent of the world wide web and the unprecedented growth of available content and data that needed to be indexed and queried at the end of the 1990's led to new requirements and challenges. Efforts aiming to built large scale search engines for this massive amount of web data considered the aforementioned (parallel) database technology, but deemed it inapropriate [30] due two the following reasons. On a practical level, the existing database systems were simply too slow as they did not contain explicit support for (full) text search and thus left significant room for improvement to be exploited by specialized systems. On a conceptual level, the ACID[1] properties guaranteed for transactions by relational database systems turned out to be a miss-match for web search engines, as they are faced with read only queries that never cause updates and tend to value high availability and freshness over consistency. Next to serving read only web search queries, the web search scenario also led to other kinds of unusual workloads as the companies involved started to exploit the graph structure of the web to improve the ranking of web search results [31] and to personalize the advertising to be displayed next to search results based on users interaction histories with machine learning algorithms.

The response to these web-scale data management problems was the in-house development of custom pieces of technology tailored for these requirements instead of applying established database systems technology. Most prominently, Google built and later published the Google File System (GFS) [60] - a distributed file system for extremely large files (e.g., web crawls) and MapReduce [47] - a framework for large scale data processing

[1] acronym for atomicity, consistency isolation, and durability

tasks based on a functional programming model resting on shared-nothing clusters of commodity hardware. In this surprisingly simple framework, users only have to implement the two functions: *map* and *reduce*. The computations expressed in these functions are automatically executed in parallel on partitioned data. The scientific publications of both GFS and MapReduce have been adopted to built the open-source variants *Hadoop* [8] and the *Hadoop Distributed File System (HDFS)* [105] which have been widely adopted to scale data-intensive computations such as Web indexing or clickstream and log analysis to very large data sets. A rich ecosystem of open-source tools, including higher-level declarative languages that are compiled down to MapReduce, like Pig [12] or Hive [9], soon evolved. In retrospect, from the viewpoint of a database systems researcher, the GFS and MapReduce publications together with the Hadoop and HDFS open source implementations started the Big Data hype.

The popularity of the MapReduce paradigm and its open-source implementation Hadoop also evoked critics, which led to a fierce debate regarding the advantages and disadvantages of MapReduce and parallel database management systems which is best summarized in [49, 106]. The debate commenced, when Pavlo et al. [95] presented a benchmark evaluating MapReduce as well as two parallel database management systems. Their experiments showed that although the process to load data into and the tuning of parameters of the parallel database management systems took much longer than for MapReduce, the measured performance of the parallel database systems was significantly better for the evaluated data processing tasks.

It became apperant that several advantages of MapReduce other than plain performance probably contributed to the success and widespread adoption of it for large scale data processing. On the one hand, the absence of licensing fees for the Hadoop open source implementation as well as the choice to operate on inexpensive commodity hardware by providing robust fault tolerance rather than operating on more reliable, but costly high-end servers made MapReduce the more economical choice. On the other hand, the storage and schema independence of MapReduce allow for the processing of arbitrarily structured data that does not have to be cleaned and curated up front and can be read from a huge variety of data sources. Furthermore, MapReduce is quite effective for the "read-once" tasks it was purpose-build for, such as constructing an inverted index of crawled web documents since it does not require loading the data into a database before

processing it. And finally, MapReduce allows users to express complex transformations (e.g., information extraction tasks) that are too complicated to be expressed in a SQL query. It was also acknowledged that the "out-of-the-box" performance of the evaluated parallel database management systems is surprisingly sub-optimal and requires extensive tuning of various parameters compared to Hadoop MapReduce, which happens to perform quite well on its default settings [106].

As use cases shifted from rather simple "extract-transform-load" and aggregation jobs to executing machine learning algorithms and other, usually highly iterative computations, it quickly became clear that Hadoop MapReduce was inherently inefficient at executing such workloads [71, 100].

Early on, it has been shown that a large variety of popular machine learning algorithms, including k-means clustering, supervised methods such as logistic regression, naive Bayes, support vector machines, dimensionality reduction methods such as principal component analysis and back-propagation for neural networks, can be written in “summation form”, which can easily be formulated in the functional *MapReduce* programming model on a logical level [43]. The academic and open source community engaged to reformulate and redesign further machine learning algorithms, for example item-based collaborative filtering [99], such that they could be scaled using MapReduce and created Apache Mahout [10], an open source machine learning library implemented on top of Hadoop MapReduce.

However, early excitement vanished, as it became apparent that the acyclic data flow model underlying Hadoop’s implementation and the intricacies of its distributed implementation lead to unsatisfactory performance. Particularly the fixed *Map-Shuffe-Reduce* pipeline which persists data to disk to provide fault tolerance is a poor fit for *iterative* algorithms, where each iteration has to be scheduled as a single MapReduce job with a high start-up cost (potentially up to tens of seconds). Further, the system creates a lot of unnecessary I/O and network traffic as all static, iteration-invariant data has to be re-read from disk and re-processed during each iteration and the intermediary result of each iteration has to be materialized in the distributed file system. This inability to efficiently execute *iterative computations* as well as a lack of support for asynchronous computations turned out to be major drawbacks of Hadoop MapReduce when it comes to efficiently executing machine learning algorithms. [71, 100]

While some suggested workaround solutions that simply avoid iterative algorithms as much as possible [77], these shortcomings of Hadoop MapReduce also sparked the development of a multitude of novel approaches and systems aiming to improve the performance and ease of implementation of more complex iterative workloads such as distributed machine learning algorithms. While HaLoop [33] and Twister [53] extended the MapReduce runtime with broadcast and cache to support iterative computations efficiently, other systems rest on entirely new paradigms.

Apache Spark [13] introduced the concept of data-parallel transformations on Resilient Distributed Datasets (RDDs) [113]: 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 [7, 36] (formerly *Stratosphere* [18]) 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 [55]. 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. With this, Stratosphere (re-)introduced concepts from (parallel) database systems into the large scale distributed data processing systems field.

While both Apache Spark and Flink essentially stuck to the data flow model, other approaches such as **GraphLab** [81] introduced an asynchronous graph-based execution model which was subsequently distributed [80].

All of these so-called *Second Generation Big Data Analytics Systems* have been shown to outperform Hadoop MapReduce for canonical iterative workloads [18, 80, 113] in the research papers presenting the systems. However, it remains an open question how effectively they perform for actual large scale machine learning problems in general. We present the two most prominent representative systems, which managed to morph from research prototypes into production systems enjoying widespread adoption in industry, Apache Spark and Apache Flink in detail in Section 2.3 and use them throughout the experiments in this thesis.

2.2 Key Concepts in Scalability and Parallelism

In this section we will introduce the concept of scalability and notions of parallelism to achieve scalability. While there is no universally agreed upon definition of scalability [66], there is a certain consensus in the distributed systems, high performance computing and database community. As described in [50] in the context of parallel database systems, there are two notions of scalability which are of interest for a parallel system:

- **strong scaling:** The change in runtime for a varying number of processors and a fixed problem size. Let $t_{p,l}$ denote the runtime for a particular problem of input size l (e.g., lines of text) on p parallel processors. The *speed-up* is then defined as:

$$SpeedUp := \frac{t_{1,l}}{t_{p,l}}$$

  In the ideal setting, speed-up is linear, adding p processors will accelerate the processing time p-times

- **weak scaling:** The change in runtime for a fixed problem size per processor. Ideally an N-times larger system can process an N-times larger problem in the same elapsed time as the smaller system. This property is commonly referred to as *scale-up*

$$ScaleUp := \frac{t_{p_{small},l_{small}}}{t_{p_{big},l_{big}}}$$

  In the ideal setting this equation evaluates to one, which translates to *linear* ScaleUp.

2.2.1 Amdahl's Law.

Already in 1967, Gene M. Amdahl published thoughts on the theoretical limits of speedup and scalability in a parallel setting [19] known today colloquially as *Amdahls Law*. It assumes a fixed workload with a runtime that decomposes to a non-parallelizable sequential fraction s, and a parallelizable fraction π whose execution can be sped up by running in parallel on the available resources that sum to one such that $s + \pi = 1$ and thus $s = 1 - \pi$. The maximum possible speed-up under these assumptions is given by:

$$SpeedUp_{Amdahl}(n) = \frac{t_1}{t_n} = \frac{s+\pi}{s+\frac{\pi}{n}} = \frac{1}{s+\frac{\pi}{n}}$$

Assuming infinite available resources, we can evaluate the limit case where the paralellizable fraction of the runtime collapses to zero and thus arrive at the theoretical maximum speedup according to Amdahl:

$$SpeedUp_{Amdahl,max} = \lim_{n\to\infty} SpeedUp_{Amdahl}(n) = \lim_{n\to\infty} \frac{1}{s+\frac{\pi}{n}} = \frac{1}{s+0} = \frac{1}{s}$$

So, according to Amdahl's Law, even if only 10% of the sequential runtime is non-paralellizable, a plausible assumption for real world problems, the maximum possible speedup is 10, regardless how large a cluster of compute nodes is used for the problem. This is a somewhat surprising insight, in particular in light of the huge popularity of massively parallel processing systems and large compute clusters e.g., Hadoop installations with tens of thousands of machines.

2.2.2 Gustafson's Law.

However, as Gustafson [63] pointed out already in 1988, Amdahl made the assumption that the non-parallelizable runtime fraction s remains constant for growing problem sizes, implying that the absolute sequential runtime grows linearly with the problem size, ultimately leading to very large sequential runtimes for problems running on massive data sets. This assumption seems implausible, as non-paralellizable sequential runtime is often due to system overheads, start-up and caching cost, quantities which are unlikely to grow linearly with data set size. Also, the assumption of a fixed problem set size seems unrealistic. To quote Gustafson: *"One does not take a fixed-sized problem and run it on various numbers of processors except when doing academic research"* [63, pp. 532-533]. In practical applications, the problem size much more likely scales with the number of processors, however the maximum runtime is usually fixed, e.g., daily reports have to be available at a fixed deadline. Gustafson found empirically that the amount of work that can be done in parallel varies linearly with the number of processors and achieved speedup factors very close to the number of processors (scale-up of 1021 on a

1024-processor system). Gustafson thus reformulated the problem such that the execution time on a single node which has n times "less parallelism" than the parallel setup is given by $s + (\pi \cdot n)$, and the so-called "scaled speed-up", also referred to as *Gustafson's Law* is given by:

$$scaledSpeedUp(n) := \frac{s + (\pi \cdot n)}{s + \pi} = s + (\pi \cdot n)$$

Thus, the main difference between Amdahl's and Gustafson's approach to parallelism lies in their assumptions. The latter captures the core idea that also underlies the concept of Big Data as well: scale out the computations on data sets of rapidly increasing size in order to obtain feasible runtimes.

2.2.3 Scalable Algorithms and Scalable Systems

In the context of Big Data Analytics an ideal *scalable algorithm* has at worst linear runtime complexity, i.e., $O(n)$, and exhibits scalability behaviour in accordance with Gustafsons law. With this property, applications can be scaled out by merely adding machines in proportion to growing data set sizes (i.e., due to an increasing user base). In the context of cloud computing, this can be automated by elastically adding or removing virtual machines via auto-scaling, which makes the scalable execution of workloads cost effective.

A *scalable algorithm* only contains a small non-parallelizable sequential fraction of runtime that does not increase with growing input size. Thus with twice the amount of input data, an ideal scalable algorithm should take at most twice the runtime, and given a cluster twice the size, the same algorithm should take no more than half as long to run. [76, pp. 13-14] Ideally, a scalable algorithm maintains these properties for various size and distribution of input data as well as for different execution clusters. However, for many real world problems there are no known algorithms exhibiting this ideal behaviour, since the coordination and communication cost tend to grow with increasing parallelism and most algorithms contain a non-parallelizable part.

Scalable systems A *scalable system* is one which, given a *scalable algorithm*, does preserve its scalability characteristics and thus guarantee overall linear scalability and speedup.

2.2.4 Parallelism

A popular means to achieve performance gains and thus to provide scalability is to parallelize and potentially distribute the execution of algorithms. In the context of parallel database systems [50], we can distinguish the following canonical forms of parallelism:

- **Data Parallelism** refers to the simultaneous execution of analogous tasks on multiple parts of the inputs, e.g., data partitions on large data sets that have been split using methods such as hash, range or round-robin partitioning. For so-called *embarrassingly parallel* tasks, data parallelism is straightforward, as computation can be split into concurrent sub-tasks which require no inter-communication that can run independently on separate data partitions.
- **Task Parallelism** refers to segmenting the overall algorithm into potentially parallelizable parts. Contrary to data parallelism where the same task is run on many data partitions in parallel, task parallelism is characterized by running many different tasks on the same data simultaneously.

2.3 Massively Parallel Data Processing Models and Systems

In this section we will discuss the individual Big Data Analytics Systems that will be the subjects of our benchmark evaluations in later chapters in detail. First, we will introduce the Google Distributed Filesystem (GFS) whose main architecture is still of relevance, as its open source variant Hadoop Distributed File System (HDFS) constitutes the storage layer of many systems deployed in production today. Second, we will present the original MapReduce model, the PACT Programming Model making up Stratosphere (now called Apache Flink) and Resilient Distributed Data Sets (RDD) that constitute Apache Spark.

2.3.1 Distributed File Systems and HDFS

Most Big Data Analytics systems, certainly Hadoop MapReduce, Spark and Flink, run on the Hadoop Distributed File System (HDFS) [105], an open source implementation inspired by GFS: the Google File System. In 2003, Google developed GFS [60] as a scalable distributed file system for large distributed data-intensive applications to satisfy

the storage needs and to service the application workloads they faced at that time while adhering to the requirements set out by the technological environment present. Even back then, GFS was already deployed on clusters providing hundreds of terabytes of storage across thousands of disks on over a thousand machines that were concurrently accessed by hundreds of clients. Since Google's main task was the crawling of web data and the construction of inverted indices for their Web Search Engine on large clusters of inexpensive shared-nothing commodity hardware, GFS was built according to the following assumptions and requirements: as component failures are the norm rather than the exception for the utilized commodity hardware, fault tolerance was a major design goal and the system must tolerate and rapidly recover from frequent component failures. The web content stored by Google lead to unusually large files ranging from hundreds of megabytes to gigabytes. The common access patterns to this data include large, sequential writes that append data to files (e.g., crawling) and large streaming reads. For these, high sustained bandwidth is more important than low latency. Updates to files once written are extremely rare as practically all applications actually mutate files by appending rather than overwriting. The distributed file system comprises three types of components: a single *master* server, several *chunkservers* that hold the data and *clients* that run the data-intensive applications. The master only keeps a record of file and chunk locations by regularly polling the chunkservers at start-up and by monitoring the status of the chunkservers with regular so-called *HeartBeat* messages. When a client application requests access to a particular file, the master responds with the corresponding chunk handle and location. The subsequent data transfer is only occurring directly between clients and chunkservers, which may be running on the same machine, preserving data locality and never via the master. The chunks (called *blocks* in HDFS) are generally large, 64MB by default, and are replicated across multiple chunkservers to provide fault-tolerance and maintain high availability. The master server maintains the meta-data for all files and chunks and always persists a replicated write-ahead log of critical meta-data changes before responding to a client operation. The master furthermore executes all namespace operations, manages chunk replicas throughout the system and coordinates various system-wide activities to keep chunks fully replicated and to balance the load across all chunkservers.

Throughout our experiments in this thesis we use HDFS, the open source version

of GFS as storage layer for the distributed data processing systems. HDFS follows the major design decisions laid out in the GFS publication. However, there are also some notable differences. In GFS the master is replicated and in case its machine or disks fail, monitoring infrastructure outside GFS starts a new master process elsewhere with the replicated operation log. HDFS only has a single master called *NameNode* that fullfills similar functions as the master in GFS and a *secondary NameNode* which regularly backs up the directory information of the NameNode, however it cannot take over in case the NameNode fails. Also, HDFS is append only and does not permit update to files.

2.3.2 MapReduce and Hadoop

MapReduce is a programming model and an associated distributed data processing system build by Google to simplify their data-intensive computations that process large amounts of raw data, such as crawled documents or web request logs and to abstract away the complexities introduced by parallelizing the computation, distributing the data and managing machine failures [47].

In *MapReduce*, the user has to specify algorithms as first-order functions supplied to the second-order functions *map* and *reduce* The map function takes as input a set of key/value tuples of type $(k1, v1)$, transforms them according to the user specified code and emits a different set of intermediate key/value tuples of type $(k2, v2)$. The MapReduce system groups all of these tuples such that all values associated with a particular key of type $k2$ are delivered to the same reduce function. The reducers function takes as input a key of type $k2$ and the set of values of type $v2$ for this key supplied via an iterator. The reduce function itself is once again specified by the user and generally computes some aggregate, merging together the input values and producing a most likely smaller set of output values of type $v2$.

$$map : (k1, v1) \rightarrow list\,(k2, v2)$$

$$reduce : (k2, list\,((v2)) \rightarrow list\,(v2)$$

For the *map* function, the input key/value tuples $(k1, v1)$ are thus potentially drawn from a different domain than the output key/value tuples $(k2, v2)$. Contrary to the model

described above which is based on [47], Hadoop MapReduce emits $list\,(k3, v3)$ at the end of the reduce phase.

The MapReduce framework automatically parallelizes the program and takes care of details such as scheduling the program's execution on the cluster, managing the inter-machine communication as well as coping with machine failures. It has a similar design as the distributed files system GFS introduced above where one master server coordinates the execution and several workers carry out the computations. MapReduce actually runs on top of such a distributed file system. At the beginning of a MapReudce computation, the input file is divided into logical file input splits, which generally correspond to GFS chunks. In the first phase workers are assigned map tasks for input splits. The MapReduce framework attempts to schedule map task on machines that contain a replica of the corresponding input spit preserving data locality. The map tasks read the input key/value tuples of type $(k1, v1)$ and execute the user defined *map* function on it. The resulting intermediate key/value tuples of type $(k2, v2)$ are buffered in memory on the worker machine and periodically persisted to the local disk. The location of these buffered pairs are then passed back to the master who notifies reduce workers of the appropriate locations. Once a worker successfully processed all its input splits, reduce workers read the buffered data from these locations via remote reads. Next, the reduce worker sorts all intermediate tuples of type $(k2, v2)$ to group all values for a particular key together. Next, the reduce worker iterates over this sorted data. For each unique key of type $k2$ encountered, it passes the key and the corresponding set values of type $list(v2)$ to the user provided *reduce* function which appends its output to a final output file for this reduce partition. During a MapReduce execution, the master pings every worker periodically with HeartBeats. If no response is received, the worker is marked as failed and its map tasks are re-assigned to other available workers, making MapReduce resilient to large-scale worker failures. The master may also schedule backup executions of very long running map-tasks, so-called stragglers, in order to speed-up overall execution time, marking the task as complete as soon as either the original task or the backup finishes the computation. Apache Hadoop is a popular and widely deployed open source implementation of this MapReduce paradigm.

2.3.3 PACTs, Stratosphere and Apache Flink

Apache Flink, formerly known as Stratosphere [18], is essentially a streaming data flow engine designed to process both stream and batch workloads [36]. Originally, the Startosphere system was centered around the Parallelization Contracts (PACTS) programming model [22], which is a superset of MapReduce. Next to the `map()` and `reduce()` functions it introduces an extended set of operators and allows arbitrary chaining of those into a directed acyclic graph - in contrast to the fixed Map-Shuffle-Reduce execution pipeline in Hadoop MapReduce. Just as in MapReduce, PACTs are also second order functions that takes a user defined first order function and executes them in accordance with the semantics of the individual operators. The PACT programming model comprises the following operators: `map()`, `reduce()`, `cross()` (a cartesian product of two inputs) `match()` (essentially a join of two inputs) and `cogroup()` which groups together tuples with a similar key from two input relations.

The main API for batch processing today is centered around the concept of a `DataSet` - a distributed typed collection comprising the elements of the data set to be processed. Users can specify functional transformations on these `DataSets`, e.g., `map()`, `filter()`, `reduce()` essentially resembling the parallelization contracts.

Flink programs are 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 a physical execution strategy for each operator in 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 directly, but it does provide its very own native iterations operator, for specifying iterative algorithms by providing a user defined step function f. This step function is repeatedly executed until some termination criterion is reached, e.g., the computations converged to

a fix point s where $s = f^k(s)$ or after a fixed number of iterations as provided by the user.

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 does not provide an own operator for iterative computations, but implements iterations as regular for-loops and executes them by loop unrolling.

2.3.4 Resilient Distributed Data Sets and Apache Spark

Apache Spark is a distributed big data analytics framework centered around the concept of Resilient Distributed Datasets (RDDs) [113]. 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. The main motivation for introducing RDDs was the inefficiency of predominant distributed data processing frameworks at the time for applications that reuse intermediate results across multiple computations, e.g., iterative machine learning and graph mining algorithms like PageRank, K-means clustering, or logistic regression. RDDs provide an interface based on coarse grained transformations (e.g., `map()`, `filter()` or `join()`) and actions (e.g., `count()`, `reduce()`). They can only be created through transformations on other RDDs or data that is read from disk. Transformations on RDDs are lazily evaluated, thus computed only when needed, e.g., by an action and can be pipelined. RDD actions trigger the computation and thus execution of transformations on RDDs. Fault tolerance is provided by logging the transformations applied on an RDD to build a data set (its so-called lineage) rather than checkpointing the actual data. If a partition of an RDD is lost due to failure, Spark has enough information how it was derived to recompute the lost partition.

Users can control two main aspects of RDDs: its persistence and its partitioning: Users can indicate which RDDs they intend to reuse in future operations 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.

As illustrated in Figure 2.1, Spark's execution model is centered around a directed acyclic graph of so called stages: whenever a user runs an action on an RDD, the Spark

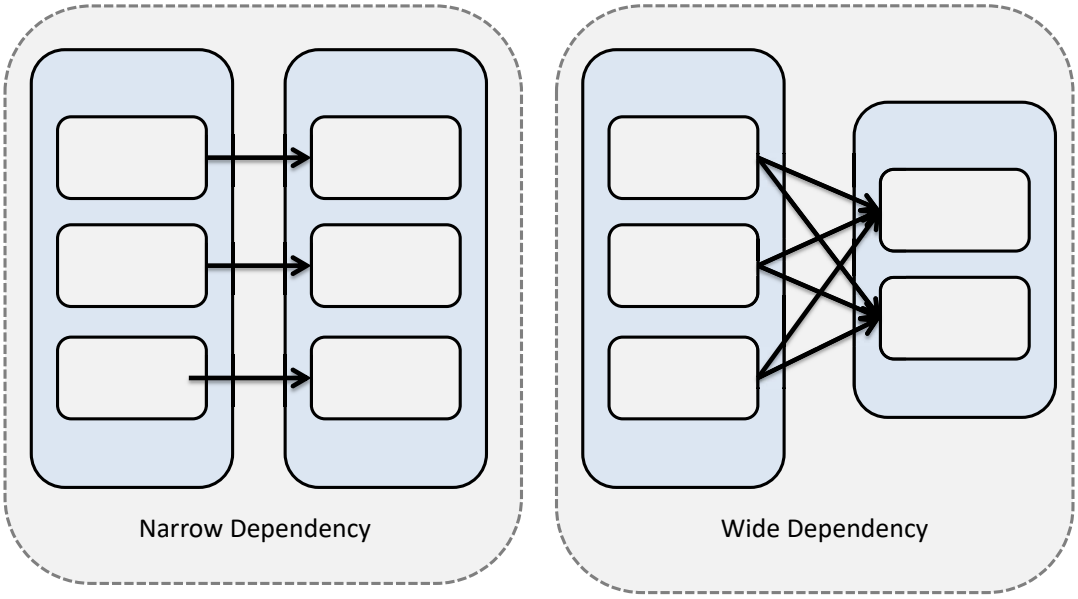

FIGURE 2.1: Two Spark stages containing an example of narrow and wide dependencies respectively. (The rectangles represent RDDs and their partitions.)

scheduler examines that RDD's lineage graph to build such a directed acyclic graph (DAG) to execute. Spark differentiates two different kinds of sets of dependencies between and RDD and its parents: *narrow dependencies* where each partition of the parent RDD is used by at most one partition of the child RDD and *wide dependencies*, where multiple child partitions may depend on a parent RDD partition.

Narrow dependencies allow for pipelined execution of multiple RDD transformations on one cluster node. This allows for very efficient recovery in case of failure, since only one parent RDD partition has to be recomputed in case a child RDD partition is lost. In contrast, in a lineage graph with wide dependencies, a single failed node might

cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution. Therefore, Spark generally materializes intermediate records on the nodes holding parent partitions with wide dependencies to simplify fault recovery in a similar manner to MapReduce, which materializes map outputs. Spark also offers the ability to checkpoint RDDs to stable storage, which is particularly helpful for applications with very long lineage graphs such as iterative computations.

2.4 Machine Learning

Machine Learning (ML) can be defined as *"a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty."* [87]. Machine Learning is a currently enormously popular field that comprises many different types of methods and settings. In this thesis, we restrict ourselves to the methods that have shown to be effective and are thus of relevance to the Big Data Analytics and distributed data processing setting.

The two canonical problems in Machine Learning are *unsupervised learning*, where the task is to discover "interesting patterns" in unlabeled data and *supervised learning* where we leverage labeled training data to learn a mapping from the inputs to the output labels to be used for prediction on unseen data. In order to apply methods of machine learning to real world data, we first have to find an appropriate (ideally numerical) representation of the real world objects in question, called *features*, through a process known as *feature extraction*. In the web setting, this may include the integration and parsing of massive amounts of log files from user-facing web applications or raw textual content from the web and subsequent extraction and transformation of various signals into numerical features in the form of feature vectors $x = (x_1, \dots x_n)^T$. The entire training data set is usually represented as a data matrix $X \in \mathbb{R}^{(n \times d)}$ that contains all n training data samples where d is the dimensionality of the feature space. In the context of distributed data processing systems this matrix has to be partitioned across different machines. The most common representation for this is a *RowMatrix*, where each row (ergo each sample vector x) of the matrix is stored as an element of the distributed data set such as an *Resilient Distributed Data Set* in Spark or a *DataSet* in Flink. This assumes that each row is small enough to fit into the main memory of a single machine of the distributed data processing system, a

fair assumption in particular since most data sets tend to be highly sparse and can thus be efficiently encoded. For the supervised learning setting, the label is usually stored in the same element of the distributed data set as part of a `labeled point` data structure that comprises both the sparse row vector and the associated label. In the remainder of the section we will briefly introduce the notation and algorithmic concepts necessary for the discussions in later chapters.

2.4.1 Unsupervised Learning

In *unsupervised learning*, we are faced with just a data matrix $X \in \mathbb{R}^{(n \times d)}$ without any associated label or class information. The task is then to discover interesting structures, patterns or classes in the data that may be used as input to subsequent learning tasks or to interpret the data. The most common unsupervised learning task is clustering. Clustering partitions the data into subsets (or *clusters*) such that elements within one cluster are similar to each other yet as dissimilar as possible to other clusters according to some particular similarity function. Popular large scale applications of clustering methods include: clustering of web text documents into categories, clustering of web search queries into semantically similar groups or clustering of gene expressions into functionally-similar clusters. A popular algorithm for clustering is *k-means*, which minimizes the intra-cluster distances between the data points x_i in a cluster j and it's center (or *centroid*) μ_j: by solving the following objective:

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

over the training data set X. Note that $i \in C_j$ represents the indices of the data points x_i that are currently assigned to Cluster C_j. It assumes a Euclidean space and that the number of clusters k be known beforehand. In *k-means*, the optimization problem is solved with the heuristic laid out in Algorithm 1 where k cluster centers are initially sampled from the data set, the euclidean distance to each of these so-called *centroids*, where c_j is the centroid of the j-th cluster, is computed for each data point and every data point is assigned to its closest centroid, and the centroids subsequently updated for each cluster that resulted.

Algorithm 1: k-means algorithm

choose k data points as the initial centroids (cluster centers) C
while k *clusters did not converge* **do**
  forall $x_i \in X$ **do**
    forall $c_j \in C$ **do**
      compute the distance from x_i to centroid c_j
    assign x_i to the closest centroid c_{min}
  re-compute the centroid using the current cluster memberships

Even for this simple yet quite common unsupervised learning algorithm we can see the the algorithm is *iterative* in nature, and we will have to access the data set multiple times, a circumstance that is problematic in the Hadoop MapReduce system.

2.4.2 Supervised Learning

In supervised learning, the canonical problem is to find a function $f : X \to Y$ that accurately predicts a label $y \in Y$ for unseen data based on a set of training samples $(x_i, y_i) \in X \times Y$ which are commonly assumed to have been generated from the joint distribution $\mathbb{P}_{X,Y}$. The objective of a supervised machine learning algorithm is to learn a function $f : \mathbb{R}^N \to \mathbb{R}$ in the case of regression, or $f : \mathbb{R}^N \to \{0, 1\}$ in the case of classification, that accurately predicts the labels y on previously unseen data.

The actual task of learning is to fit the parameters (also called *model weights*) w of the function $f_w : X \to Y$ based on the training data and a *loss function* $l : Y \times Y \to \mathbb{R}$ which encodes the fit between the known label y and the prediction $f_w(x)$. In order to avoid that the model w captures idiosyncrasies of the input data rather than generalize well to unseen data, a so-called regularization term $\Omega(w)$ that captures the model complexity is often added to the objective (e.g., the L_1 or L_2 norm of w). With this, the generic optimization problem which serves as a template of a supervised learning problem is given by:

$$\hat{w} = \arg\min_{w} \left(\sum_{(x,y)\in(X,Y)} l\left(f_w(x), y\right) + \lambda \cdot \Omega(w) \right)$$

Of course the optimization cannot be carried out on the actual data set we want to predict on, but rather on a training set that already has the corresponding labels y_i. We thus aim to learn a prediction function that generalizes well from the training data used for optimization to unseen data. Different instantiations of the function f, which may be selected from different function classes, and the loss function l yield different learning algorithms.

Solvers. The most commonly used loss functions have been designed 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:

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

This generalized formulation of a gradient-decent update encodes the solutions to a variety of data analytics tasks which be framed as convex optimization problems. However, this solver requires iterating over the entire training data set multiple times in a batch fashion. A more popular alternative is given by stochastic gradient descent (SGD), where each data point, or a small "mini-batch" of data, is used to update the model independently:

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

We can see that both Stochastic Gradient Descent as well as Batch Gradient Descent contain two so-called *hyperparameters* that are usually data set specific and have to be searched for using held-out validation data: the *learning rate* η and the *regularization weight* λ. Heuristics for the learning rate tend to use formulations that decay with increasing number of iterations τ, for example: $\frac{\eta}{\sqrt{\tau}}$ which we also rely on for some of our experiments. Before we consider implementations of these methods on MapReduce systems, we will first explore a very popular model class, namely logistic regression.

Logistic Regression

One of the most popular examples of supervised learning methods that fit this optimization based approach is *logistic regression*, which is a generalized linear model for binary classification leveraging the so-called *logistic*, *logit* or *sigmoid* function.

$$sigmoid\,(x) := \frac{1}{1 + e^{-x}}$$

Logistic regression has empirically been shown to provide competitive if not superior prediciton quality for large scale high-dimensional data [37] and has a long standing history as a statistical method used in a variety of academic fields such as medical research, econometrics or the social sciences in general. It is also a very popular choice for supervised machine learning [14] - or at least seen as a solid baseline.

In logistic regression, the prediction function f_w to be learned becomes:

$$f_w\,(x) := \frac{1}{1 + e^{-w^T x}}$$

For a binary classification setting $y \in \{0, 1\}$ we assume a Bernoulli distribution for y:

$$P(y|x, w) = Ber(y|\mu(x))$$

and define $\mu(x) := f_w(x)$ such that:

$$P(y|x, w) = (f_w(x)^y (1 - f_w(x))^{(1-y)}$$

Learning the model then translates to minimizing the log-likelihood of the data given the model:

$$l(w) = log(L(W)) = log(\prod_{i=1}^{n} P(y_i|x_i, w)) = log(\prod_{i=1}^{n} (f_w(x_i))^{y_i} (1 - f_w(x_i))^{(1-y_i)})$$

the optimization problem to find the model weights then becomes:

$$\arg\max_w -l(w) = \arg\max_w -\sum_{i=1}^{n} y_i \cdot log(f_w(x_i)) + (1 - y_i) \cdot log(1 - f_w(x_i)) + \lambda \cdot \Omega(x_i)$$

There is no closed form solution for this problem. One does thus rely on methods such as Gradient Descent or Newton-Raphson to solve it iteratively [92]. We will stick to gradient descent methods for now. The derivative evaluates to

$$\frac{\partial}{\partial w_j} l(w) = \sum_{i=1}^{n} (y_i - f_w(x_i)) x_i^{(j)} + \frac{\lambda}{n} w^{(j)}$$

which can be used for the Gradient Descent updates as laid out above. Different parametrizations of the components f_w, $l\left(f_w\left(x\right), y\right)$ and $\Omega\left(w\right)$ yield different supervised learning algorithms to which gradient descent can be applied analogously.

Gradient Descent Methods in MapReduce

While the implementation of Batch Gradient Descent (BGD) is rather straightforward in the MapRedce model, given the fact that the loss function and thus also the gradient decompose linearly, the implementation of Stochastic Gradient Descent is problematic. For BGD, the partial sums of gradient updates can be computed locally on data partitions in mappers, while a reducer sums up all gradient contributions or partial sums thereof and subsequently performs the model update. Figure 2.2 illustrates this. The updated model then has to be redistributed to the mappers by use of the *distributed cache*. For Stochastic Gradient Descent however, the updated model has to be communicated to all nodes after each update, so essentially after each single data point, which is practically in-feasible. However since the execution of multiple global iterations (over the entire data set as in BGD) is quite expensive in MapReduce, it has been proposed [71] to integrate Stochastic Gradient Descent into a solution. In this model the mappers perform one or more local, comparatively fast, SGD passes over their local data partitions to train one of M local model $f_i(x)$ and the reducer then assembles a so-called *ensemble model* of the data by averaging the individual model weights from the mappers

$$\hat{y} = \arg\max_y \sum_{k=1}^{M} \alpha_k f_k(x)$$

where $\alpha_k = 1 \forall k$ by default. Alternatively, the ensemble prediction can be computed by taking a majority vote of all trained models when classifying unseen data [83]. It has

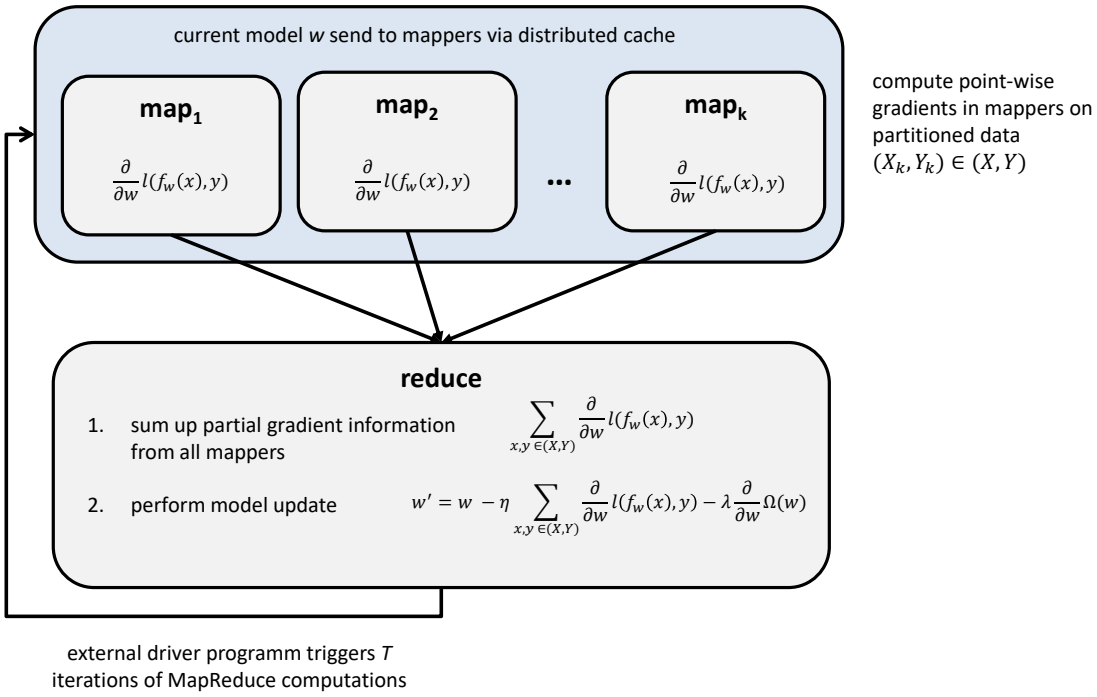

FIGURE 2.2: Batch Gradient Descent computation in MapReduce: the mappers compute partial sums of gradient information. A reduce operator aggregates all partial sums and performs the model update. This computation is repeated for T iterations which have to be triggered by an external driver iteration.

been shown empirically, that training an ensemble model on multiple partitions of the data actually tends to deliver superior test set accuracy than running training one single model using SGD on the same data [71]. A not entirely surprising effect since averaging models characterized by similar inductive biases in an ensemble leads to a reduction of the variance component of the ensemble when compared to an individual model [15].

Another approach is to use an averaged ensemble from all the mappers as a "warm-start" initialization for a couple of global optimization steps (involving iterations of all

data partitions), potentially using a more sophisticated solver e.g., quasi-Newton methods such as L-BFGS [17]. Such a *hybrid model* has been shown to significantly boost the improvement of prediction quality per iteration of the warm-started global model.

As second generation data analytics systems like Spark and Flink all support the efficient execution of iterative algorithms, one does not have to rely on such "algorithmic workarounds" designed to circumvent the deficiencies of MapReduce, but instead implement global solvers. An interesting alternative is proposed with the *HOGWILD!* algorithm [91]. The authors suggest 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. Interesting alternative system architectures called *Parameter Server* have been proposed for this setting [74] that are beyond the scope of this thesis.

2.5 Benchmarking

Benchmarking of data processing systems has a long standing history in the database community and is largely driven by industry-controlled consortia. One of them, the *Transaction Processing Performance Council (TPC)* grew out of an effort to provide a suitable benchmark and benchmarking process for the evaluation of online transaction processing (OLTP) systems [54] that were rapidly gaining popularity in the 1980s. Its goal is to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry. On the other hand, the *Standard Performance Evaluation Corporation (SPEC)* has a somewhat broader focus as it establishes, maintains and endorses standardized benchmarks and tools to evaluate hardware and software components. Both exist in the form of a non-profit corporation.

The main objective of standardized benchmarks is to enable a fair, reproducible and relevant comparison of data processing systems and products. Kistowski et al. define a benchmark as a *"Standard tool for the competitive evaluation and comparison of competing systems or components according to specific characteristics, such as performance, dependability, or security"* [108]. This is a challenging problem given that vendors compete

fiercely in the market for data processing systems and may fall for the temptation of indulging in *benchmarketing*. The main objective of the aforementioned industry driven consortia is thus to create a level playing field and to ensure that the benchmark workloads do in fact represent popular customer applications and that the measurements and metrics are relevant and results repeatable. Additionally all systems, products, and technologies used for these benchmarks have to be generally available to users. The process of developing and accepting Benchmark carried out by consortia such as SPEC or TPC are usually carried out under consortia confidentiality agreements. [108] In this context there exist two different kinds of benchmarks: specification based benchmarks where only the workload is specified leaving the implementation of that specification up to the user of the benchmark. Lately, *express benchmarks* that come with a "kit" that already includes the workload implementations have also been introduced [68].

The benchmarks most relevant to the topics covered in the thesis are:

- **TPC Express™ Big Bench (TPCx-BB)** is an application benchmark for Big Data targeting Big Data Analytics Systems (BDAS) with simple ETL and analytical workloads [21, 59]. It does not cover machine learning workloads.
- **TPC Express Benchmark™ HS (TPCx-HS)** is a benchmark developed to evaluate commercial Apache Hadoop File System API compatible software distributions such as Apache Hadoop itself or Apache Spark. It aims to provide verifiable performance, price-performance and availability metrics and comprises an extended version of the TeraSort [88].
- **TPC Benchmark™ DS (TPC-DS)** is a benchmark targeting general purpose decision support systems [89]. The associated workload contains both business intelligence queries as well as data maintenance aspects.

While these benchmarks are widely accepted in industry, the motivation when developing and carrying out benchmark experiments in academia is usually different. While the resulting workloads, metrics and experiments might very well be adopted by industry consortia for future benchmarks, the primary goal in an academic setting is to "stress" the systems in order to discover potential shortcomings that can serve as a starting point

for further investigation and development, rather than running a competitive evaluation of different products.

In [56] the authors provide a summary of the requirements a "good benchmark" ought to fulfill: First, the benchmark should be targeting a substantial target audience and measure relevant, typical operations in the targeted problem domain while remaining be simple, understandable and affordable to run. Second, the implementation of the benchmark should be fair and portable, repeatable, configurable and comprehensive enough to cover all major features of the systems under test. Finally, a good benchmark should provide a flexible performance analysis framework which allows the users to configure and customize the workloads of the benchmark. The workloads themselves should be representative, scalable and be measured with meaningful and understandable metrics.

3 Benchmarking Scalability

3.1 Problem Statement

In this chapter, we present one of the most important aspects in the context of benchmarking data processing systems for scalable machine learning workloads: addressing all relevant dimensions of scalability. In Section 2.1, we have stressed the fact that the major design goals and requirements that distributed data processing systems were built to adhere to are: to robustly scale data-intensive computations to very large data sets on clusters of commodity hardware. The second generation data processing systems presented in Section 2.3 essentially stuck to the paradigm of distributed dataflow when trying to address the identified shortcomings of the first generation systems, namely efficiently executing iterative computations. However, these second generation systems were still designed as general purpose data processing systems that also excel at scaling up the execution of machine learning algorithms - and not as systems that solely and efficiently support the scalable execution of machine learning systems.

We consider the two most prominent representative systems that managed to morph from research prototypes into production systems enjoying widespread adoption in industry: *Apache Spark* and *Apache Flink* as introduced in Section 2.3. However the experiments outlined are of relevance to a broader set of systems that target similar workloads. While these second generation big data analytics systems have been shown to outperform *Hadoop* for canonical iterative workloads [55, 113], it remains an open question how they perform in executing large scale machine learning algorithms. In this context, the motivations to scale up computations tend to be much broader than facing a high volume of data [23].

Consider the prominent problem of *click-through rate prediction* for online advertisements, a crucial building block in the multi-billion dollar online advertisement 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 [35]). For this important problem, algorithms such as regularized logistic regression that we introduced in Section 2.4.2 are still the method of choice [39,65,78,84]). Generalized linear methods, such as logistic regression, are a popular choice for general supervised learning settings [14] with very large data sets [71]. Since they cannot handle feature dependencies directly, combinations of features ("crossings") have to be added manually. This feature crossing leads to very high-dimensional training data sets after expansion even if the original data dimensionality was modest.

In the context of scalable, distributed machine learning, there are thus multiple dimensions of scalability that 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 [28, 64]. The widespread dissemination of global web applications that generate tremendous amounts of log data pushed the relevance of this dimension of scalabiliy early on.

2. **Scaling the Model Size:** many large-scale machine learning problems exhibit very high-dimensionality. For example, classification algorithms that draw on textual data based on individual words or *n-grams* easily contain 100 million dimensions or more in particular in light of combinations of features, models for click-through rate prediction for online advertisements can reach up to 100 billion dimensions [35]. 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 optimize hyper-parameters, many models with sightly different parameters are usually trained in parallel to perform grid search for these parameters.

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 an embarrassingly parallel problem, we focus on the first two aspects: *scaling the data* and *scaling the model dimensionality*.

3.2 Contributions

We introduce a representative set of distributed machine learning algorithms that are suitable for large scale distributed settings comprising logistic regression and k-means clustering, which have close resemblance to industry-relevant applications and provide potentially 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 to investigate the overhead that is 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 in [85].

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.

In order to solve the 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 processing 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 comprehensive experimental evaluations of distributed dataflow systems.

2. To ensure reproducibility, we provide our benchmark algorithms on top of *Apache Flink* and *Apache Spark* as open-source software. 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 that deserves further investigation.

3.3 Overview

The rest of this chapter is structured as follows: in Section 3.4 we present a detailed discussion of the chosen machine learning workloads and their implementations in the data flow systems. In Section 3.6 we provide important parameters that have to be set and tuned in each system under test. Section 3.5 introduces the metrics and experiments that constitute the benchmark and Section 3.7 provides concrete results and a discussion of the comprehensive experimental evaluation of the benchmark workloads and systems under evaluation. In Section 3.8 we discuss related work in the area of benchmarking distributed data processing systems before we conclude and summarize our findings in Section 3.9.

3.4 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 provide a fair and insightful benchmark, which is in line with the desiderata outlined in Section 2.5 and reflects the requirements of real-world machine learning applications that are deployed in production and generates meaningful results. To achieve this, we propose algorithms that are successfully applied

in the context of very large data sets and strive for truly equivalent and thus fair implementations of these algorithms on the various systems under test. We do *not* rely on existing library implementations, but use the very same data structures and abstractions for vectors and mathematical operations.

3.4.1 Supervised Learning

As was introduced in Section 2.4.2, 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 $l\left(f_w\left(x\right), y\right)$. To avoid overfitting, a regularization term $\Omega\left(w\right)$ that captures the model complexity is added to the objective. Different parametrizations of the components f_w, $l\left(f_w\left(x\right), y\right)$ and $\Omega\left(w\right)$ yield 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 [39, 65, 78, 84]. We choose regularized logistic regression for this workload, however, as long as we ensure that we always measure the exact same methods on all systems, the results of our experiments provide generlizable insights into the systems performance.

Solvers

The most commonly used loss functions were designed to be both convex and differentiable, which guarantees the existence of a minimizer $\hat{w}$. This circumstance enables the application of batch gradient descent (BGD) as a solver, as we introduced in Section 2.4.2. This algorithm performs the following step using the gradient of the loss until convergence:

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

We choose and implement this solver, because, while not being the method of choice, it actually represents the data flow and I/O footprint exhibited by a wide variety of (potentially more complex) optimization algorithms [92], such as *L-BFGS* [79] or *TRON*

[75]. While different parametrizations of loss function $l(f_w(x), y)$, regularizer $\Omega(w)$ and prediction function f_w may lead to different algorithms. They can all be learned with batch gradient descent solvers or similar algorithms that express a comparable "computational footprint". All of these gradient descent solvers can be captured by the *Iterative Map-Reduce-Update* pattern [32]. In this pattern, an iterative algorithm is subdivided into a `map` to compute the gradient components, a `reduce` to globally aggregate all components and an `update` step to revise the machine learning model vector.

Implementation

Rather than depending on existing machine learning library implementations, we implement all learning algorithms from scratch, 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[1] for numerical processing. Data points are represented as `SparseVectors` that maintain two arrays: one containing the indices of non-zero dimensions and one containing the values for these non-zero dimensions. The labeled data points are represented by the class `LabeledPoint` that included the `SparseVectors` and ensured that they could properly serialized by all systems under test. As we discussed in Section 2.4.2, the straightforward implementation strategy for batch gradient descent is as follows (depicted in Figure 3.1): each mapper reads the local partitions of training data from the distributed file system and processes it each data point at a time. Inside the mapper, we compute the gradient per point and emit one summand for the global update which is performed inside a reducer.

Alternatively, we can leverage the `MapPartition` operator in both Spark and Flink, which provides us with an iterator of the entire data partition, rather than just a data point per function call. This enables the pre-aggregation of partial sums inside the user-defined function as depicted in Figure 3.2. While both Spark and Flink should automatically add a combiner to the aforementioned MapReduce implementation, which also pre-aggregates the gradient contributions on the mapper side, the MapPartition-based implementation ensures that we do not unnecessarily serialize and deserialze partial sum vectors before aggregating them. Additionally, Spark also provides an operator seemingly tailored

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

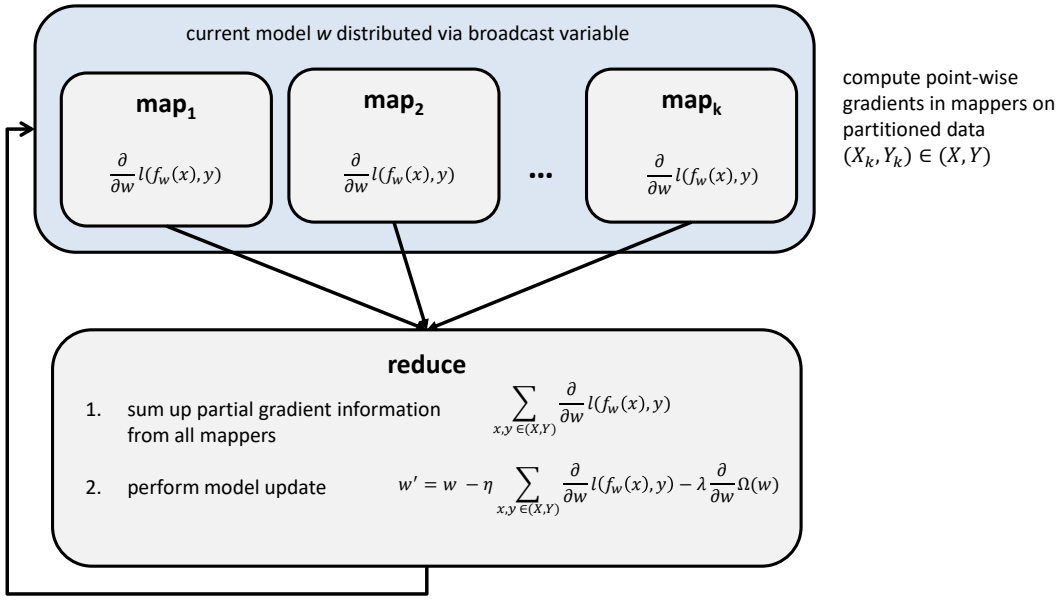

FIGURE 3.1: Batch Gradient Descent using Map and Reduce

towards aggregating model updates called `TreeAggregate`. It combines the partials sums in a tree-like multi-level aggregation on a small set of executors before sending the results to the driver. We evaluated all available implementation variants and their performance. The results are shown in Figure 3.5 and discussed in Section 3.7.1. Based on these findings, we choose the following implementations for the benchmark experiments: 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 is the more performant alternative to using a `map()` to compute the gradients and summing them up in a subsequent `reduce()` step during experimental evaluation as we later show in our experimental evaluation

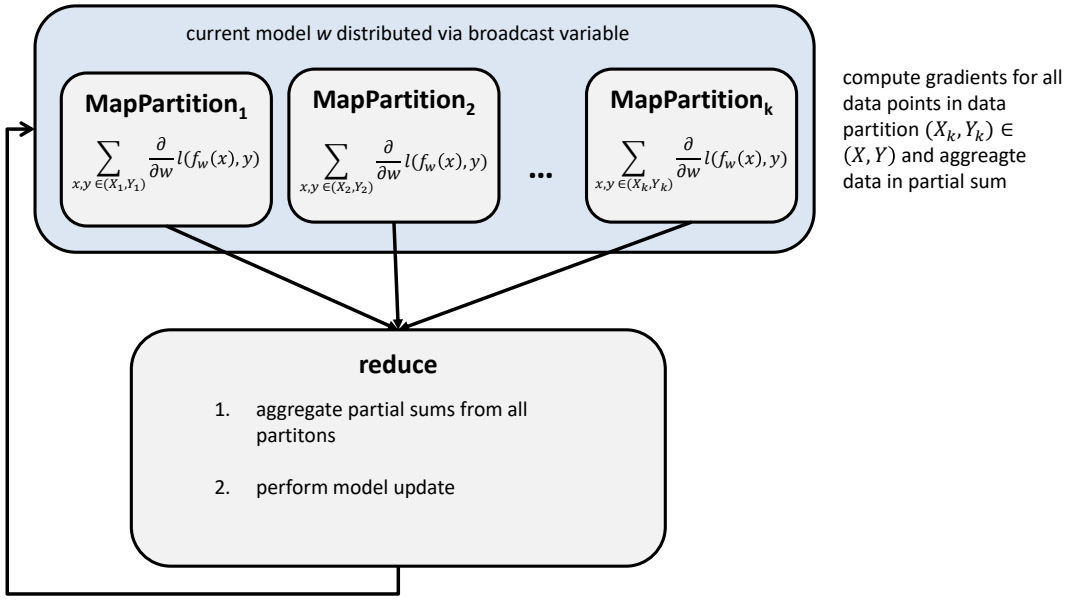

FIGURE 3.2: Batch Gradient Descent using MapPartition

(see Figure 3.5). 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 to keep loop-invariant data in memory. The weight vector of the model 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 weight vector of the model is also distributed to the individual tasks as a broadcast variable. The `TreeAggregate()` implementation is more robust (i.e. crashes less often with *OutOfMemory* exceptions) for higher dimensionalities than a `MapPartition` implementation and more performant than a `map() - reduce()` implementation as we later show (see Figure 3.5). The iterations are simply encoded as plain loops in Scala, thus no

specific operator for iterative computations is needed.

Listing 3.1: Flink implementation of the k-means algorithm

```
 def computeBreezeClustering(points: DataSet[BDVector[Double]], centroids:
DataSet[(Int, BDVector[Double])], iterations: Int): DataSet[(Int,
BDVector[Double])] = {
   val finalCentroids: DataSet[(Int, BDVector[Double])] =
centroids.iterate(iterations) { currentCentroids =>
    val newCentroids = points
      .map(new breezeSelectNearestCenter).withBroadcastSet(currentCentroids, "centroids")
      .groupBy(0)
      .reduce((p1, p2) => {(p1._1, p1._2 + p2._2, p1._3 + p2._3)}).withForwardedFields("_1")

    val avgNewCentroids = newCentroids
      .map(x => {
       val avgCenter = x._2 / x._3.toDouble
       (x._1, avgCenter)
      }).withForwardedFields("_1")

    avgNewCentroids
   }

   finalCentroids
 }}
```

3.4.2 Unsupervised learning

For unsupervised learning, we choose to implement the popular *k-means* clustering algorithm introduced in Section 2.4.1. As we introduced in Section 2.4.2, it 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. This is once again an iterative algorithm, which is implemented using a loop in Spark and the native iteration operator in Apache Flink. The centroids are sent to the worker / executor machines as *broadcast variables*. We use a `RichMapFunction()` in Flink and a `map()` in Spark to compute the nearest centers for all data points in the partition. Next, we aggregate the results by centroids, which boils down to the usage of the `groupBy()` and `reduce()` operator in Flink and the `reduceByKey()` operator in Spark. Listings 3.1

and 3.2 show the implementation in Flink and Spark respectively. In Listing 3.1, the `@ForwardedFields` function annotation provides a hint to the Flink optimizer to generate an efficient execution plan. Such annotations declare fields that are not manipulated by the annotated function and thus forwarded untouched at the same position to the output. Spark provides the `reduceByKey()` operator to be used instead of `groupByKey()` which would need to be shuffled around all the key-value pairs and a subsequent `map()` to aggregate. `reduceByKey()` indicates to Spark that it can combine output tuples with the same key on each partition before shuffling the data points.

LISTING 3.2: Spark implementation of the k-means algorithm

```
 def computeBreezeClustering(data: RDD[BDVector[Double]],
          centroids: Array[(Int, BDVector[Double])],
                      maxIterations: Int): Array[(Int,
BDVector[Double])] = {
   var iterations = 0
   var currentCentroids = centroids

   while(iterations < maxIterations) {

    val bcCentroids = data.context.broadcast(currentCentroids)

    val newCentroids: RDD[(Int, (BDVector[Double], Long))] = data.map (point => {
     var minDistance: Double = Double.MaxValue
     var closestCentroidId: Int = -1
     val centers = bcCentroids.value

     centers.foreach(c => { // c = (idx, centroid)
     val distance = squaredDistance(point, c._2)
      if (distance < minDistance) {
       minDistance = distance
       closestCentroidId = c._1
      }
     })

     (closestCentroidId, (point, 1L))
    }).reduceByKey(mergeContribs)

    currentCentroids = newCentroids
     .map(x => {
      val (center, count) = x._2
      val avgCenter = center / count.toDouble
      (x._1, avgCenter)
     }).collect()

    iterations += 1
  }
  currentCentroids
 }
```

3.5 Benchmark Dimensions and Settings

In this section, we present the data generation strategies, data sets, experiments as well as metrics and measurements that constitute the scalability benchmark. Furthermore, we provide the specification of the hardware that we relied upon for our experimental evaluation.

3.5.1 Scalability

As we discussed in Section 2.2, 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 processors 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 use 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 Dimensionality: 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 (e.g. 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)

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

We complete the scalability experiments by adding a the traditional scaling experiment:

Experiment 3: Strong Scaling: Measure the runtime for training a model for a varying number of cluster compute nodes while holding the data set size and dimensionality fixed.

Figure 3.3 illustrates these three experiments and the parameters that are varied within each of them. 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 thesis.

	# nodes	# data points	# dimensions
Production Scaling	const.	⟷	const.
Strong Scaling	⟷	const.	const.
Model Scaling	const.	const.	⟷

FIGURE 3.3: Overview of the different scalability experiments and associated parameters to be varied.

3.5.2 Absolute and Single Machine Runtimes

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 of the algorithms presented in Section 3.4.1 as suggested in [85]. We choose the *LibLinear*[2] solver as an efficient C++ single-threaded implementation and transformed the implementation of the solver to plain batch gradient descend (BGD) to foster a fair comparison.

Experiment 4: Evaluation of Single Machine Performance Measure the runtime for training a model while varying the number of machines and model dimensionality

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

(keeping the size of the training data set fixed) as well as the runtime of a single-threaded implementation

3.5.3 Model Quality

The main focus of the work presented in this chapter is on evaluating the scalability behavior of distributed data processing systems when executing machine learning workloads. We do not evaluate prediction quality directly as part of this component of the benchmark. Contrary to traditional benchmarks of database systems and distributed data processing systems we introduced in Section 2.5, it cannot be assumed that all queries will produce the same result set. In machine learning different algorithms - or even variants of the same algorithm - can produce different results. To ensure that potentially different convergence properties of the optimization algorithms have no impact on the measurements, we decided to rely on mathematically equivalent implementations for both supervised and unsupervised workloads across all systems. We validated that all implementations do converge to the same model weights given the same data and verified that the prediction accuracy is identical across systems as well as within the expected range. All experiments are then executed for a fixed number of iterations, which is the same across all systems.

Figure 3.3 provides an overview of all the proposed experiments and the parameters that are varied within each of them.

3.5.4 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.

3.5.5 Data Sets

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 [10] and *HiBench* [67].

For the *supervised learning* experiments, we use parts of the *Criteo Click Logs*[3] data set. This data set 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.

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 3.1: Subset of the Criteo data set used in the experiments.

As a pre-processing step, we expand the categorical features in the data set using the hashing trick [111]. 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 the accuracy in practice and 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 hash vector. Experiments with fixed dimensionality d 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

[3] http://labs.Criteo.com/downloads/download-terabyte-click-logs/

sub- and super-sampling the data. A *scaling factor* of `1.0` refers to the Criteo subset as presented in Table 3.1 which contains about **1.15 billion data points**.

3.6 System Parameter Configuration

In Section 2.3 we already introduced the systems background necessary. 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 substantially between the two systems. We will present and discuss the concepts behind the relevant parameters as well as the rationale for setting or tuning them in this section. In general, we rely on "reasonable best effort" for tuning, referring to best practices published in each system's documentation as well as the associated open source mailing lists and related work.

3.6.1 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 outlined in the configuration guide and set the number of task slots equal to the number of available CPU cores in the cluster. This generally triggers an initial *re-partitioning* phase in a job, as the number of HDFS blocks is rarely equivalent to the desired number of subtasks.

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:

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

Following the Spark recommendation outlined in the configuration guide, we set `executor.cores` equal to the number of cpu cores available in the cluster and set the parallelism (number of RDD partitions) to three times the number of CPU cores available in the cluster.

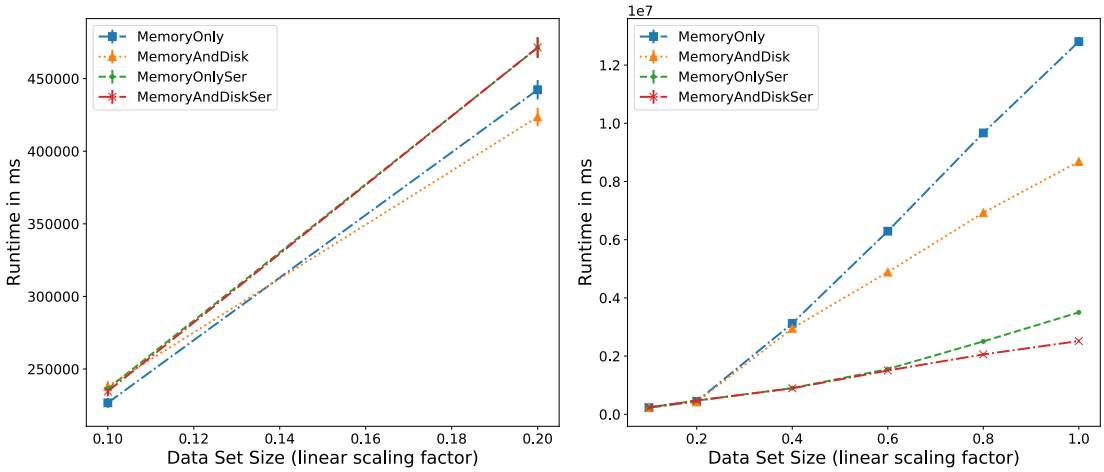

FIGURE 3.4: l_2-regularized logistic regression training in Apache Spark with increasing data set size for a fixed number of nodes and different RDD StorageLevels. The left-hand side contains the runtimes for small data set sizes.

3.6.2 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`[4]:

`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 that 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 whenever they are needed.

[4] https://spark.apache.org/docs/1.6.2/programming-guide.html#rdd-persistence

`MEMORY_ONLY_SERIALIZED`: the RDD is stored as serialized Java objects (one byte array per partition). This representation 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 that 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 l_2-regularized logistic regression model on the Criteo data set for different `StorageLevel` settings on 30 compute nodes (details in Section 3.5.4).

Figure 3.4 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 storage strategies, 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`. Interestingly, the overhead incurred for serializing and writing out to disk seems to be limited, such that `MEMORY_ONLY` is only the fastest caching strategy at comparatively small data set sizes (0.1) and quickly outperformed by `MEMORY_AND_DISK` at a data scaling factor of 0.2 as can be seen on the left-hand side of Figure 3.4.

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.

3.6.3 Buffers

Network buffers are a critical resource of communication. They are used to buffer records before transmission over a network, 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 to use approximately

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

buffers, we found that a higher setting is advantageous for machine learning workloads and thus set the buffers to: `numberOfBuffers = 16384` and `bufferSizeInBytes = 16384`.

3.6.4 Serialization

By default, Spark serializes objects using the Java serialization framework, however, Spark can also use the `Kryo`[5] 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 corresponding data type which 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.

3.6.5 Broadcast

Both Spark and Flink provide means to make available variables inside operators that run distributed across machines by declaring them as so-called *broadcast variables*. While Flink does not allow the user to control the broadcast method used, Spark actually

[5] https://github.com/EsotericSoftware/kryo

provides multiple alternative broadcast strategies that can be chosen. Next to the traditional `HttpBroadcast`, Spark also offers `TorrentBroadcast`. Here the driver machine that orchestrates the executers which carry out the actual computations, divides the broadcast variable into multiple chunks after it has been serialized and broadcasts the chunks to different executors. Subsequently, executors can fetch chunks individually from other executors that have fetched the chunks previously, rather than having to rely on communication to the driver alone, which can quickly become the bottleneck when broadcasting large broadcast variables such as weight vectors of machine learning models after an update computation. We thus set `spark.broadcast.factory` to `TorrentBroadcastFactory` for all of our experiments.

3.7 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.

3.7.1 Supervised Learning

Production Scaling

Figure 3.5 shows the runtimes for 5 iterations of batch gradient descent learning of a l_2-regularized logistic regression model. We evaluate different implementation strategies (MapReduce, MapPartition and TreeAggregate) as introduced in Section 3.4.1 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 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

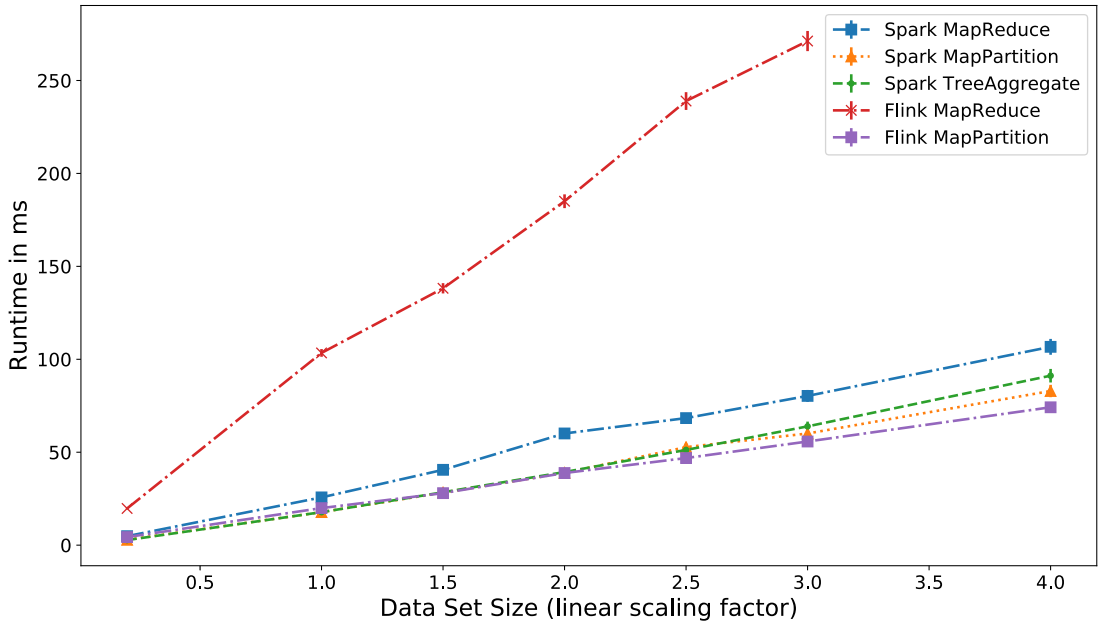

FIGURE 3.5: Production scaling experiment: We measure the runtime of different implementation strategies for l_2 regularized logistic regression on a fixed set of 23 nodes for linearly growing data set sizes with 1000 dimensions.

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

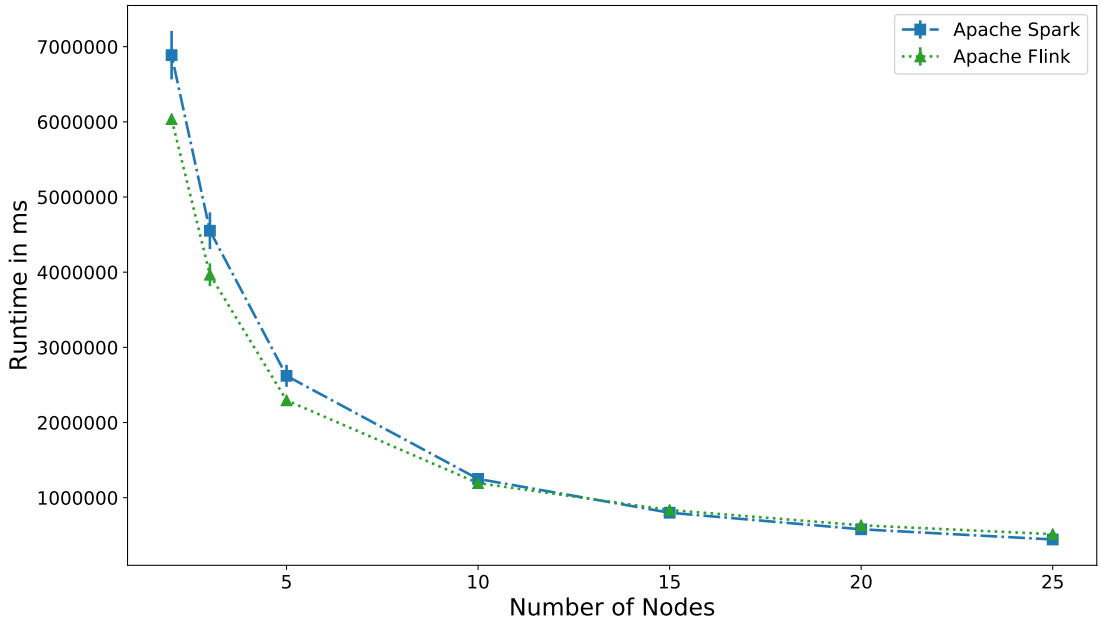

FIGURE 3.6: Strong scaling for different implementations of l_2 regularized logistic regression in Spark and Flink for 1000 dimensions and 530 GB.

Strong Scaling

Figure 3.6 shows the results of our strong scaling experiments for the batch gradient descent workload for the Criteo data set hashed to 1000 dimensions. Figures 3.7 and 3.8 show the performance details for a run with 25 nodes and Figures 3.9 and 3.10 for a run with three nodes. In this figures, it is evident that while Flink tends to run faster on smaller cluster configurations, Spark has runs faster 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 3.7 and 3.8, 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.

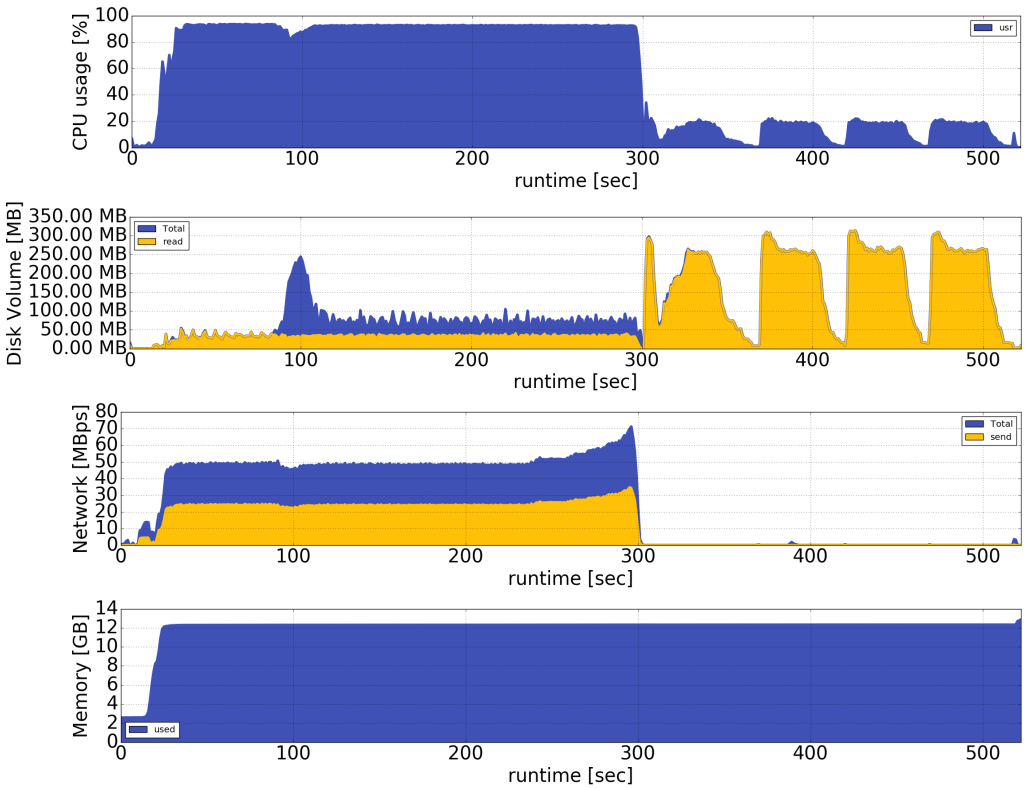

FIGURE 3.7: Performance details for training a l_2 regularized logistic regression model with Flink on 25 nodes (logistic regression) (blue = total, yellow = read/sent)

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 two or three nodes, Flink's memory management and robust out of core performance lead to superior performance compared to Spark. 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.

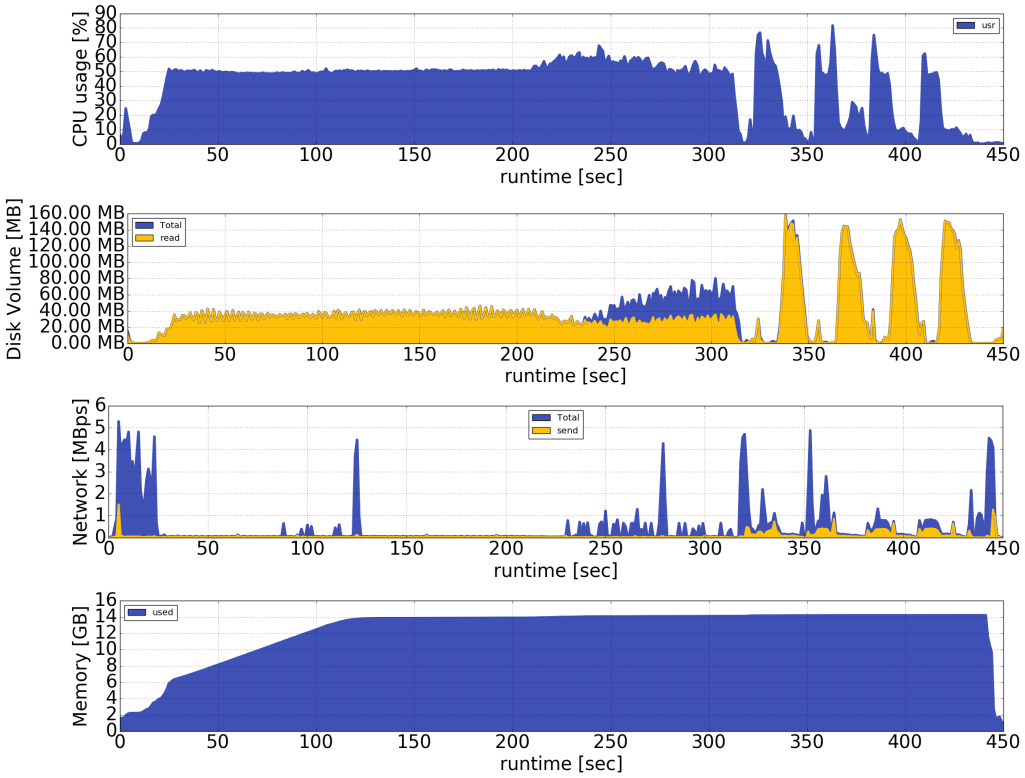

FIGURE 3.8: Performance details for training a l_2 regularized logistic regression model with Spark on 25 nodes (logistic regression)

Scaling Model Dimensionality.

As was described in the introduction, supervised learning models in production are not only trained on very sparse data sets 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 pre-processing step. Figure 3.11 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

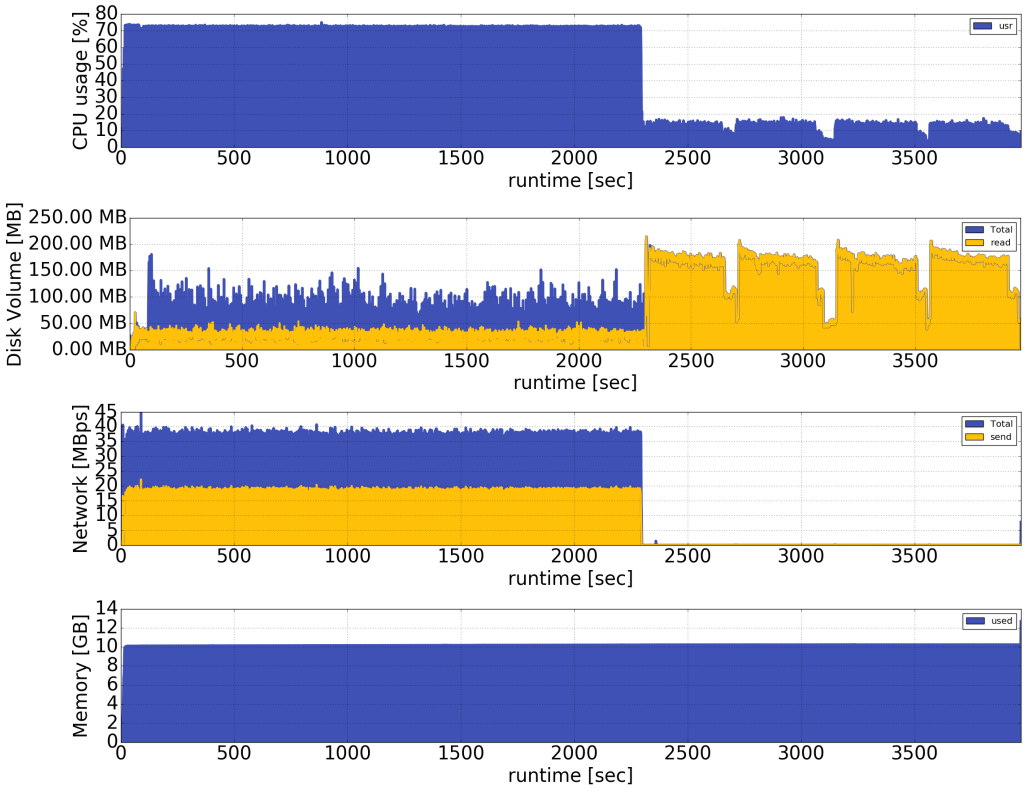

FIGURE 3.9: Performance details for training a l_2 regularized logistic regression model with Flink on 3 nodes (logistic regression)

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 with *OutOfMemory* exceptions 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

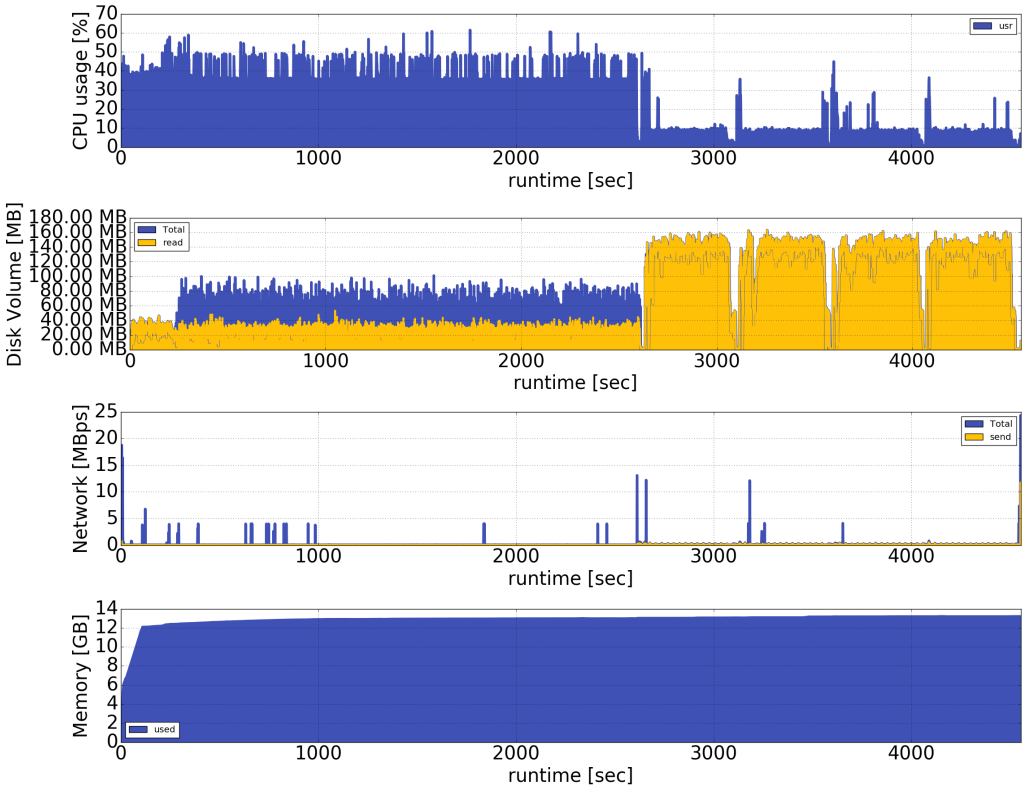

FIGURE 3.10: Performance details for training a l_2 regularized logistic regression model with Spark on 3 nodes (logistic regression)

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 [35, 78, 84], this is a dissatisfying result. It seems the *Broadcast Variable* feature was simply not designed or intended to handle truly large objects. The systems allot an insufficient amount of memory for the boadcast vector variable, a fact that cannot be alleviated by the user, as both Spark and Flink do not expose parameters to alter the amount of main memory reserved for broadcast variables.

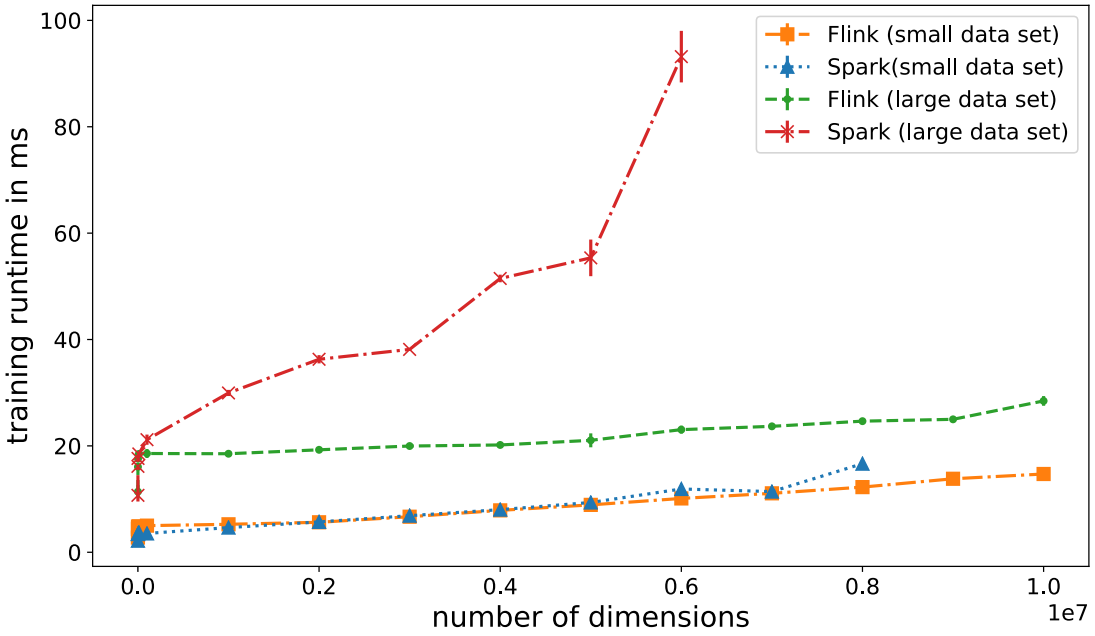

FIGURE 3.11: **Scaling the model:** Runtimes for training a l_2 regularized logistic regression model of different dimensions on 20 nodes. Results shown for two different data set sizes: small (about the size of main memory) and large (significantly larger than main memory).

Comparison to single-threaded implementation

In order to gain an understanding of how the performance of the distributed systems Spark and Flink compare to state of the art single core implementations we run experiments with the LibLinear solver, which provides a highly optimized single-threaded C++ implementation. However we did change the actual computation implementation to batch gradient descent similar to the Spark and Flink implementations. As this library is limited 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 3.12 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

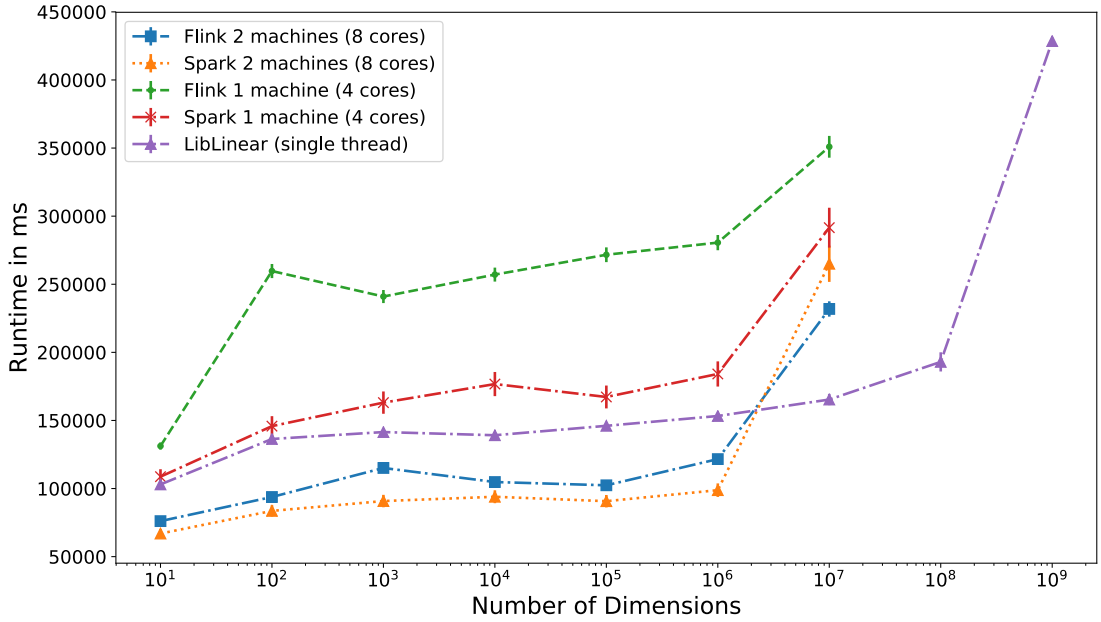

FIGURE 3.12: **Comparison to single-threaded implementation:** Runtimes for training a l_2 regularized logistic regression model of different dimensions different amounts of nodes for a small sub-sample (approx. 4GB) of the Criteo data set compared to a single-threaded implementation (modified LibLinear).

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. [85]. 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.

3.7.2 Unsupervised Learning

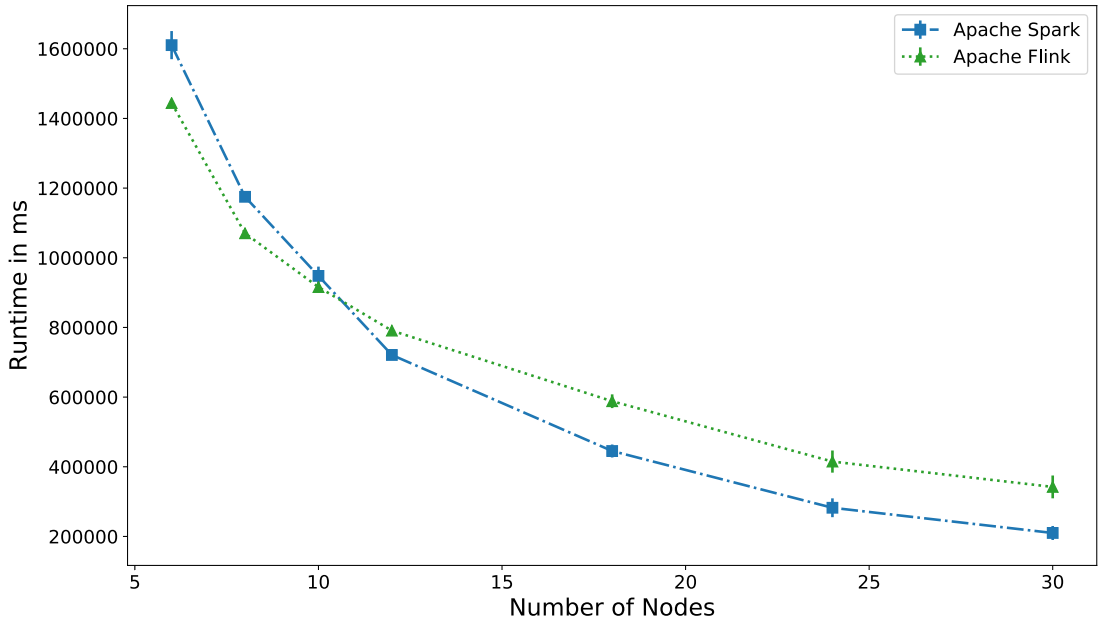

FIGURE 3.13: k-means strong scaling experiments for Spark and Flink in 200 GB of generated data with 100 dimensions and *k=10* clusters

In order to evaluate the effectiveness of the `reduceByKey()` operator in Spark and the `groupBy()` and `reduce()` operator in Flink, we conduct both strong scaling and production scaling experiments with the unsupervised learning workload k-means. The production scaling experiments in Figure 3.14 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 3.13 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, namely instantiating the entire pipeline a priori and distributing

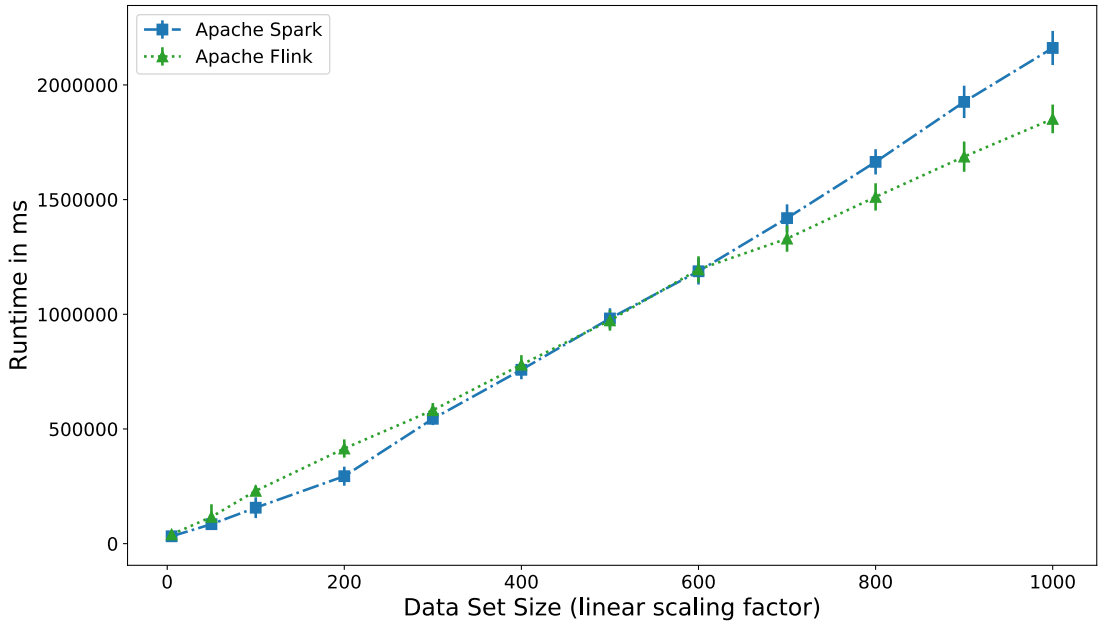

FIGURE 3.14: k-means production scaling experiments for Spark and Flink on 30 nodes with *k=30*

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.

3.8 Related Work

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

Cai et al. [34] present an evaluation 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. [104], 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. While the authors do investigate in detail where performance differences may stem from, the paper does not contain scale-out experiments and does not vary the dimensionality of the models.

In a very similar manner, Marcu et. al. [82] present a performance analysis of the big data analytics frameworks Apache Spark and Apache Flink, for the workloads *word count, grep, TeraSort, 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*, *TeraSort* 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 [110] presents a 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, TeraSort, k-means, PageRank* 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 [82], Spark outperforms Flink for the *k-means* workload. An observation, that our findings confirm.This is due to improvements in *Spark 1.6.1.* (e.g. project Tungsten) which were not present in the Spark version used in [82]. Once again, the work [110] only considers

simple workloads and does not evaluate distributed machine learning algorithms with respect to the crucial aspect of model dimensionality.

3.9 Discussion

In this chapter, we introduced crucial building blocks of a comprehensive benchmark to evaluate and assess distributed data flow systems for distributed machine learning applications. In order to understand the scalability of the systems and paradigms, we presented mathematically equivalent implementations of supervised and unsupervised learning methods that constitute the benchmark workloads. We motivated and described different experiments and measurements 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 assess the systems ability for scaling the data set size, we also introduced *Model Dimensionality Scaling* to evaluate the ability to scale with growing model dimensionality as well as a *comparison to a single-threaded implementation.*

Our comprehensive experimental evaluation of different implementations in Apache Spark and Apache Flink on up to 4.6 billion data points revealed that 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. This issue occurs because the systems allot insufficient amounts of memory to the broadcast variables, a circumstance that cannot be corrected by the user, as both systems do not expose parameters to alter the memory provisioning for broadcast variables. 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 in an experiment with a fixed number of iterations.

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 [74] 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 [101]. It thus remains an open challenge to provide an adequate solution to the problem of robustly and efficiently scaling distributed machine learning algorithms with respect to both data set size and model dimensionality at the same time.

Limitations. Potential *limitations* of our approach to benchmarking are due to the pursuit of fairness via equivalent implementations across systems. To achieve this goal, some system-specific features that are not available in others may not be leveraged in the implementations. Furthermore, beneficial adjustments of machine learning algorithms, e.g. to speed up the convergence, that may only be implementable in specific systems, cannot be applied in our approach. The use of one specific data set may also limit generalizability of the experimental results to some extend.

The scalability experiments introduced in this chapter, in particular *Model Dimensionality Scaling*, which has not been addressed previously, should be a core part of experimental evaluations of distributed data processing systems and an integral part of any benchmark to evaluate distributed data processing systems for scalable machine learning workloads. The shortcomings identified by our experiments can serve as starting point for future research and developments, for example in the area of adaptive memory management for distributed data processing systems.

4 Benchmarking Performance and Model Quality

4.1 Problem Statement

In this chapter we address another important aspect in the context of benchmarking distributed data processing systems for scalable machine learning workloads: the dimension of *model quality*. Contrary to traditional relational database queries that have an exact result set which will always be returned regardless of the physical execution plan chosen by the database optimizer, different machine learning algorithms are known to produce models of different prediction quality when applied to a supervised learning problem on the same data set [37, 38]. Even for one particular machine learning method, e.g., logistic regression introduced in Section 2.4.2, different solvers for the underlying optimization problem may possess different convergence properties and thus produce models of different prediction quality after a fixed runtime. Different machine learning methods and solvers possess different runtime complexity and thus scalability proprieties with respect to the number of data points in the training set. Given a fixed time budget, we are thus faced with a trade-off space spanned by runtime and model quality. More complex algorithms may ultimately lead to superior prediction quality, but take longer - potentially prohibitively long - to run until convergence.

When these machine learning algorithms are scaled out using second generation distributed data flow systems, additional complexity arises, as different algorithms may be more or less well suited for distribution using this paradigm. The learning algorithms chosen to evaluate the second generation distributed data flow systems in the associated research papers tend to be those that fit well onto the system's paradigm (e.g., gradient descent solvers for generalized linear models) rather than state of the art methods which

would be chosen to solve a supervised learning problem in the absence of the systems' constraints.

In order to accurately assess how well the systems resting on the paradigm of distributed data flow perform for the task of scaling out the training of machine learning methods, it is thus imperative to benchmark and evaluate them for relevant and representative machine learning algorithms and to explore the trade-off between runtime and model quality. Such an evaluation is also crucial for scientists and practitioners, we intend to apply machine learning algorithms to their problem domain and want to assess the applicability of distributed data processing systems to their setting.

4.2 Contributions

In this chapter we present the position that benchmarking distributed data processing systems for scalable machine learning workloads needs to take into account the dimension of model quality as well. We propose experiments and a method to explore the trade-off space between runtime and model quality. Furthermore, we argue, that benchmarks for distributed data processing systems should consider state of the art, single machine algorithms as baselines when evaluating scalable machine learning workloads. Solely evaluating scalability and comparing with other distributed systems is not sufficient and leads to inconclusive results. Distributed data processing systems for large scale machine learning should be benchmarked against sophisticated single machine libraries that practitioners would choose to solve an actual machine learning problem and evaluated with respect to both: runtime as well as prediction quality metrics. We introduce such experiments and present an experimental evaluation of state of the art machine learning libraries *XGBoost*, *LightGBM* and *Vowpal Wabbit* for supervised learning and compare them to *Apache Spark MLlib*, one of the most widely used distributed data processing system for machine learning workloads. Results (Section 4.7) indicate that while distributed data flow systems such as *Apache Spark* do provide robust scalability, it takes a substantial amount of extra compute resources to reach the performance of a single threaded or single machine implementation.

In summary, the contributions presented in this chapter are the following:

- We argue that benchmarking of large scale machine learning systems should consider state of the art, single machine algorithms as baselines, should be evaluated with respect to both: runtime and prediction quality and present a methodology and experiments for this task.
- We present an experimental evaluation of a representative problem with state of the art machine learning libraries *XGBoost*, *LightGBM* and *Vowpal Wabbit* for supervised learning and compare them to *Apache Spark MLlib*.
- In order to foster the reproducibility of benchmarks and experiments and in hope of benefiting future systems research we make the code (including config files and hyperparameters used) available as open source software.

4.3 Overview

The rest of this chapter is structured as follows: in Section 4.4, we present the rationale for single machine baseline experiments. Next, in Section 4.5 we present a detailed discussion of the chosen machine learning algorithms and implementations. In Section 4.6, we introduce the methodology for the experiments and Section 4.7 provides concrete results and a discussion of the experiments. Finally, in Section 4.8, we discuss related work before we conclude and summarize our findings in Section 4.9.

4.4 The Case for Single Machine Baselines

Analysis of industrial MapReduce workload traces over long duration [42] has revealed, that most jobs in big data analytics Systems have input, shuffle, and output sizes in the MB to GB range. Annual polls of data scientists regarding the largest data set analyzed conducted by KDnuggets[1] (Figure 4.1), a popular site for analytics and data science, while non-representative, do seem to suggest that this pattern holds for data analysis in general. While some respondents reported the raw size of their largest data sets analyzed to be in the PB range, the vast majority are actually in the GB range, in fact about 70% are smaller than 100 GB. This trend has been surprisingly stable for years. Since

[1] https://www.kdnuggets.com/2016/11/poll-results-largest-dataset-analyzed.html

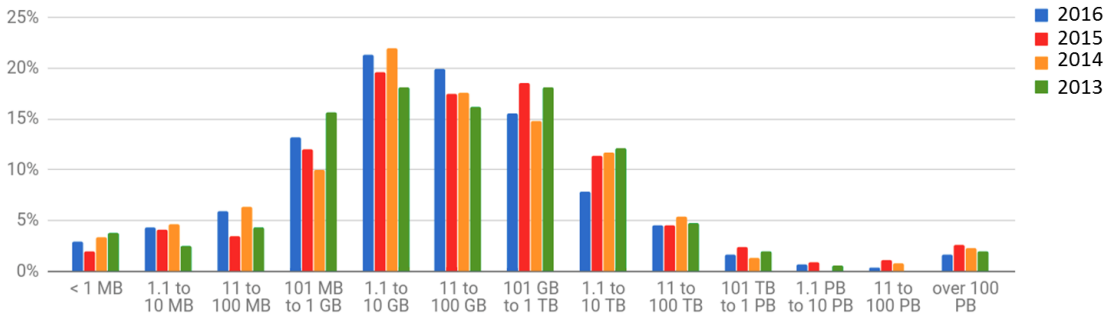

FIGURE 4.1: Poll of data scientists regarding the largest (raw) data set analyzed, 2013-2016: While some data sets that are analyzed are indeed in the PB range, the wast majority are in the GB range - a trend that shows surprising stability over the years. (Data provided by Gregory Piatetsky, KDnuggets, http://www.kdnuggets.com)

all major Infrastructure as a Service (IaaS) providers offer on-demand compute nodes with hundreds of GBs of main memory [2], the question arises whether distributed data flow systems really are the optimal design choice for large scale machine learning systems, in particular since the actual training data is often significantly smaller after feature extraction has been performed on the raw data. The yardstick to compare against should be the performance of state of the art single machine ML libraries, as large portions of the training data sets used, are within the order of magnitude of the available main memory systems and out-of core libraries can scale even beyond main memory on a single machine. In the next section, we will present the machine learning algorithms we deem suitable for such an evaluation.

4.5 Machine Learning Methods and Libraries

Logistic regression is one of the most popular algorithms for supervised learning on big data sets due to its simplicity and the straightforward parallelizability of its training

[2] At the time of writing of this thesis, the three major providers of cloud computing resources: Amazon AWS EC2, Google Compute Engine and Microsoft Azure all offer compute nodes with very large main memory configurations: Amazon AWS instances actually go up to 4191 GB of main memory (x1e.32xlarge) [3], Google Compute Engine to 3844 GB (n1-ultramem-160(Beta)) [5] and Microsoft Azure to 4080 GB (M128ms) [4].

algorithms [14, 71]. It has been implemented on nearly all big data analytics systems. However, from a implementation-agnostic point of view, it is not at all clear that logistic regression should be the method of choice. In fact, comprehensive empirical evaluations of several different supervised learning methods including Support Vector Machines, Neural Networks, Logistic Regression, Naive Bayes, Memory-Based Learning, Random Forests, Decision Trees, Bagged Trees, Boosted Trees, and Boosted Stumps concluded that *Boosted Trees* deliver superior performance with respect to prediction quality [38] and predict better probabilities than all other methods evaluated.

Among the machine learning methods used in practice, gradient tree boosting [57, 97] is one technique that shines in many applications, for example if data is not abundant or of limited dimensionality. In particular `XGboost` [40] is a very popular tree boosting algorithm that is also available as an open source library. It is a popular choice by data scientists and has been used quite successfully in machine learning competitions such as *Kaggle*[3].

4.5.1 Gradient Boosted Trees

In this section we will introduce the basics of tree boosting and the algorithm used by XGBoost in particular as outlined in [90], which should be consulted for an in-depth discussion and derivation of the topic. In general, the method of boosting fits so-called *adaptive basis-functions:*

$$f(x) = \theta_0 + \sum_{m=1}^{M} \theta_m \phi_m(x)$$

of *weak* or *base* learners $\phi_m(x)$ to the training data data in a greedy manner. The weak learner can be of any suitable model class, however it is common to use Classification and Regression Tree (CART) models [87].

[3] Among the 29 challenge winning solutions published at *Kaggle's* blog during 2015, 17 solutions used *XGBoost*. Among these solutions, eight solely used *XGBoost* to train the model, while most others combined *XGBoost* with neural nets in ensembles. For comparison, the second most popular method, deep neural nets, was used in 11 solutions [40].

The canonical optimization problem to be solved when learning a model in boosting is given by the following objective:

$$\min_f \sum_{i=1}^{N} l(y_i, f(x_i)),$$

where $l : Y \times Y \to \mathbb{R}$ is a differentiable and convex loss function, N is the total number of data points and $f(x)$ is an adaptive basis-function consisting of M individual models as defined above. It expands to finding the set of M optimal parameters θ and basis functions ϕ:

$$\{\hat{\theta_m}, \hat{\phi_m}\}_{m=1}^{M} = \underset{\{\theta_m, \phi_m\}_{m=1}^{M}}{\arg\min} \sum_{i=1}^{N} l(y_i, \theta_0 + \sum_{m=1}^{M} \theta_m \phi_m(x_i)).$$

To solve this, one generally applies *forward stage wise additive modeling* which iteratively fits:

$$\{\hat{\theta_m}, \hat{\phi_m}\} = \underset{\theta_m, \phi_m}{\arg\min} \sum_{i=1}^{N} l(y_i, f^{(m-1)} + \theta_m \phi_m(x_i))$$

at each iteration m with

$$f^m = f^{(m-1)} + \theta_m \phi_m(x_i)$$

approximately by so-called gradient boosting or second-order boosting leveraging newton methods. The boosting algorithm thus applies the weak learners sequentially to weighted versions of the data, where more weight is given to examples that were misclassified by earlier rounds.

Trees as weak learners

Gradient boosted trees are based on simple tree models, which partition the feature space X into a set of T rectangular and non-overlapping regions or *quadrants* $R_{i..T}$ and fit a (very) simple model to each region, namely a simple constant or *response value*. This can be expressed as an adaptive basis-functions model where the basis functions ϕ_j indicate whether a particular data point x is a member of the leaf node R_j using an indicator

function I and the weights θ_j encode the constant response value w_j for each region. A tree model is then formalized by:

$$f(x) = \sum_{j=1}^{T} w_j I\left(x \in R_j\right).$$

Because learning the optimal partitions $R_1, \ldots, R_T$ that represent the structure of the tree is NP-complete [69] the problem has to be simplified, instead computing an approximate solution. A popular algorithm for this is *Classification And Regression Trees (CART)* which greedily grows the decision tree top-down with binary splits starting from the root node [29].

When plugging in the tree formulation into the original boosting formulation, the optimization problem to be solved when learning a model for *tree boosting* becomes:

$$\{\{\hat{w}_{jm}, \hat{R}_{jm}\}_{j=1}^{T_m}\}_{m=1}^{M} = \underset{\{\{w_{jm}, R_{jm}\}_{j=1}^{T_m}\}_{m=1}^{M}}{\arg\min} \sum_{i=1}^{N} l(y_i, \sum_{m=1}^{M} \sum_{j=1}^{T_m} w_j I\left(x \in R_j\right)).$$

Which once again is transformed by applying forward stage wise additive modeling, which adds one tree at a time rather than jointly optimizing M trees and thus learns a new tree at each iteration m using:

$$\{\hat{w}_{jm}, \hat{R}_{jm}\}_{j=1}^{T_m} = \underset{\{w_{jm}, R_{jm}\}_{j=1}^{T_m}}{\arg\min} \sum_{i=1}^{N} l(y_i, \hat{f}^{(m-1)}(x_i) + \sum_{j=1}^{T_m} w_j I\left(x \in R_j\right)).$$

XGBoost adds a regularization term Ω to control the complexity of the learned model and to adjust the final learned weights via smoothing in order to avoid overfitting in the following manner:

$$\Omega(T, w) = \gamma T + \frac{1}{2}\lambda \|w\|^2.$$

Where T is the number of leaves in each tree and w are the leaf weights. XGBoost uses second-order approximation to solve the regularized objective, which translates to the

following approximation of the objective to find the minimizers $\{\hat{w}_{jm}, \hat{R}_{jm}\}$:

$$\min_{\{w_{jm},R_{jm}\}} \sum_{i=1}^{N} \left[\hat{g}_m(x_i) \sum_{j=1}^{T} w_{jm} I\left(x \in R_j\right)) + \frac{1}{2} \hat{h}_m \left(\sum_{j=1}^{T} w_{jm} I\left(x \in R_j\right) \right)^2 \right] + \Omega(T, w)$$

to be solved for each tree to be added at each iteration m. Here

$$\hat{g}_m(x_i) = \left[\frac{\partial l(y_i, f(x_i))}{\partial f(x_i)} \right]_{f(x)=\hat{f}^{(m-1)}(x_i)}$$

is the gradient and

$$\hat{h}_m(x_i) = \left[\frac{\partial^2 l(y_i, f(x_i))}{\partial^2 f(x_i)} \right]_{f(x)=\hat{f}^{(m-1)}(x_i)}$$

is the hessian. Let I_{jm} denote the set of indices of the data points x_i falling into the quadrant R_{jm}. The objective then simplifies to:

$$\{\hat{w}_{jm}, \hat{R}_{jm}\} = \underset{\{w_{jm},R_{jm}\}}{\arg\min} \sum_{j=1}^{T} \sum_{i \in I_{jm}} \left[\hat{g}_m(x_i) w_{jm} + \frac{1}{2} \hat{h}_m(x_i) w_{jm}^2 \right] + \Omega(T, w)$$

And can be further compacted by letting $G_{jm} = \sum_{i \in I_{jm}} \hat{g}_m(x_i)$ and $H_{jm} = \sum_{i \in I_{jm}} \hat{h}_m(x_i)$ to:

$$\{\hat{w}_{jm}, \hat{R}_{jm}\} = \underset{\{w_{jm},R_{jm}\}}{\arg\min} \sum_{j=1}^{T} \left[\hat{G}_{jm}(x_i) w_{jm} + \frac{1}{2} \hat{H}_{jm}(x_i) w_{jm}^2 \right] + \Omega(T, w)$$

Where G_L, G_R and H_L, H_R are defined over the instance set I_L of the left and I_R of the tight nodes after the split with $I_L \cup I_R = I$. With this, the algorithm to train a boosted

tree model used by XGBoost can be formulated as outlined in Algorithm 2 [40].

Algorithm 2: XGBoost second order boosting algorithm

Input: Data Set $\{x_i, y_i\} \in (X, Y)$, number of iterations M, learning rate η, number of leaf nodes T

Initialize $\hat{f}^{(0)}(x) = \theta_0 = \arg\min_\theta \sum_{i=1}^{N} l(y_i, \theta)$

for $m = 1, \ldots, M$ **do**

$\hat{g}_m(x_i) = \left[\frac{\partial l(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}^{(m-1)}(x_i)}$

$\hat{h}_m(x_i) = \left[\frac{\partial^2 l(y_i, f(x_i))}{\partial^2 f(x_i)}\right]_{f(x)=\hat{f}^{(m-1)}(x_i)}$

Determine the structure $\{\hat{R}_{jm}\}_{j=1}^{T}$ by selecting splits that maximize the gain:

$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G_{jm}^2}{H_{jm}+\lambda}\right] - \gamma$

Determine the leaf weights $\{\hat{w}_j\}_{j=1}^{T}$ for the learned tree structure with

$\hat{w}_{jm} = -\frac{G_{jm}}{H_{jm}+\lambda}$

$\hat{f}_m(x) = \eta \sum_{j=1}^{T} \hat{w}_{jm} I(x \in \hat{R}_{jm})$

$\hat{f}^{(m)}(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x)$

Output: $\hat{f}^{(M)}(x) = \sum_{m=0}^{M} \hat{f}_m(x)$

One of the key parts of this algorithm as well as the computationally most expensive one is finding the optimal splits according to:

$$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G_{jm}^2}{H_{jm} + \lambda}\right] - \gamma$$

A popular choice to solve this is to pre-sort the training data according to feature values and then enumerate all potential split points in sorted order to compute the *gain* outlined above. While this approach is simple and will indeed find the optimal split points, it is also inefficient in both training speed and memory consumption [72]. Instead, state of the art libraries such as XGBoost use *approximate algorithms* for this step. We use two state of the art libraries that implement different versions of approximate split finding methods, which we will outline below. We rely on the following three boosting libraries for our experiments:

XGBoost

XGBoost[4] is a scalable end-to-end tree boosting system which implements the algorithm outlined above in Algorithm 2. Next to the exact greedy method outlined above, it also provides a histogram-based algorithm, which proposes candidate points for splits based on percentiles of feature distributions based on a *Distributed Weighted Quantile Sketch* [40]. XGBoost then maps continuous feature values into discrete buckets based on the proposed candidate points and uses these bins to construct feature histograms during training which turns out to be more efficient with respect to both memory consumption and training speed.

LightGBM

LightGBM[5] is another fast, distributed, high performance gradient boosting framework based on decision tree algorithms. It addresses the problem of finding optimal splits by using *Gradient-based One-Side Sampling* and *Exclusive Feature Bundling* [72]. It has been shown to outperform XGBoost in terms of computational speed and memory consumption, we thus include it in our experiments.

Apache Spark MLLib

For the distributed experiments we use *Apache Spark* as introduced in Section 2.3. It is one of the most popular open source systems for big data analytics and ships with the scalable machine learning library *MLlib* [86]. The primary Machine Learning API for Spark since Version 2.0 is the DataFrame-based API, which leverages Tungsten's fast in-memory encoding and code generation. We use Spark and MLlib version 2.2.0 and the aforementioned API (`spark.ml`) for all of our Spark experiments in our experiments. The gradient boosted trees implementation in Apache Spark MLLib relies on the tree-training algorithm of its random forest implementation but calls it with only one tree. This implementation also uses quantiles for each feature as split candidates and discretizes the feature values into bins for the computation of sufficient statistics for the best split computation.

[4] `https://github.com/dmlc/xgboost`
[5] `https://github.com/Microsoft/LightGBM`

4.5.2 Logistic Regression

As a solid baseline we also use logistic regression in our experiments. As we established in Section 3.4.1, logistic regression is a popular choice by data scientists [14] for general supervised learning settings with very large data sets [71] and amenable to distributed execution. We use the following libraries for our experiments:

Vowpal Wabbit (VW)

As a popular and highly efficient single machine library, we use Vowpal Wabbit, which is a fast out-of-core library that trains (among other things) a logistic regression classifier with stochastic gradient descent as introduced in 2.4.2. By default, *Vowpal Wabbit* runs the parsing and learning in two separate threads and can thus essentially be seen as a single threaded implementation compared to the other libraries and systems that exploit all available cores.

Apache Spark MLLib

We once again rely on the MLLib machine learning library of Spark for logistic regression, which provides a *L-BFGS* [79] based implementation.

4.6 Methodology

In this section we outline the Methodology used to carry out the experiments, addressing the relevant steps of feature extraction, parameter tuning and how to actually measure model quality for the different libraries and systems evaluated.

4.6.1 Feature Extraction

For the gradient-boosted tree libraries we only transformed the categorical features from *hex* to *integer* values, for *Vowpal Wabbit* we left them unchanged. Since *Vowpal Wabbit* uses feature hashing as we described in Section 3.5.5 internally, we implemented the same feature hashing as a pre-processing step for the *Spark* version of Logistic regression with the $2^{18} = 262144$ features, which is the default value for *Vowpal Wabbit* and small enough

to avoid problems that high dimensional vectors have been shown to cause within Apache Spark in the previous chapter.

4.6.2 Parameter Tuning

The search for the optimal (hyper-) parameters is a crucial part of applying machine learning methods and can have a significant impact on an algorithms performance. In order to strive for a fair comparison in our experiments we allotted a fixed and identical time frame for tuning the parameters to all systems and libraries, honouring the fact that practitioners also face tight deadlines when performing hyperparameter tuning and may not have the time for exhaustive search of a global optimum [26]. We built all three libraries based on their latest available release on `GitHub`. For the benchmark experiments we relied on the command line versions of *XGBoost* and *LightGBM*, however for hyperparamter tuning we relied on the python interface and used the `hyperopt`[6] package to find the optimal parameters. For *Vowpal Wabbit* we used `vw-hypersearch` to tune the hyperparamters for learning-rate and regularization. Apache Spark MLlib provides a `CrossValidator` to find the best parameter for a model. Unfortunately we ran into continuous OutOfMemory issues even for data sets several orders of magnitude smaller than the available main memory and thus had to rely on rather coarse grained grid search for the Spark algorithms. The resulting hyperparameters that were used for all the experiments are listed in Table 4.1.

4.6.3 Measurements

In order to be able to explore the trade-off space between runtime and model quality and to illustrate prediction quality over time we propose the following methodology: for `VW's` Stochastic Gradient Descent we measured training time for different fractions of the data set, for *Apache Spark MLLib* Logistic regression and gradient boosted trees as well as *XGBoost* and *LightGBM* for different numbers of iterations. We ran all experiments with and without evaluation of model quality on held-out test data and only plotted the time elapsed in the no-evaluation runs. As *Apache Spark* does not allow intermediate evaluation of trained models across iterations, we re-run the training with

[6] https://github.com/hyperopt/hyperopt

System	(Hyper-)parameter	Value
XGBoost	`eta`	0.1
	`num_round`	500
	`nthread`	24
	`min_child_weight`	100
	`tree_method`	hist
	`grow_policy`	lossguide
	`max_depth`	0
	`max_leaves`	255
LightGBT	`learning_rate`	0.1
	`num_leaves`	255
	`num_iterations`	500
	`num_thread`	24
	`tree_learner`	serial
	`objective`	binary
Spark GBT	`MinInstancesPerNode`	3
	`MaxDepth`	10
	`MaxBins`	64
Spark LR	`RegParam`	0.01
Vowpal Wabbit	`loss_function`	logistic
	`b`	18
	`l`	0.3
	`initial_t`	1
	`decay_learning_rate`	0.5
	`power_t`	0.5
	`l1`	1e-15
	`l2`	0

TABLE 4.1: Hyperparamters used for the experiments

different numbers of iterations from scratch, measured the training time and subsequently evaluated model quality on a held out set of test data.

4.6.4 Data Set

In order to evaluate a relevant and representative standard problem, we choose to use the *Criteo Click Logs* [7] data set introduced in Section 3.5.5. This click-through rate (CTR) prediction data set 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. 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. For our experiments in this chapter we sub-sampled the data such that both classes have equal probability, resulting in roughly **270 million data points** and **200 GB of data** in tsv/text format. Given the insights from figure 4.1, this seems to be a reasonable size. We use the first 23 days (except day 4 due to a corrupted archive) as training data and day 24 as test data, ensuring a proper time split.

4.6.5 Cluster Hardware

We run our experiments on two different cluster setups available in the local university data center: a *small* one with very limited amounts of cores and memory but more powerful CPUs and a *big* setup with large amounts of main memory, cpu cores and storage.

Small node: Quadcore Intel Xeon CPU E3-1230 V2 3.30GHz CPU with 8 hyperthreads (4 cores), 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.

Big node: 2 x AMD Opteron 6238 CPU with 12 Cores @ 2,6 GHz (24 cores), 256 GB RAM, 8x 2 TB Disk, 6 × GE Ethernet via a Cisco C2969-48TD-L Rack Switch.

4.7 Experiments

In this section we introduce the experiments and measurements to explore the trade-off between runtime and model quality in the context of evaluating distributed data flow systems for the task of scaling out the training of machine learning methods.

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

data flow systems for scalable machine learning workloads. We measure training time (including loading the data and writing out the model) and (in a separate run) the AuC the trained models achieve on a held-out test set (day 24 of Criteo). The relevant hyperparameters have been tuned beforehand as described in Section 4.6. Please keep in mind that our motivation is to evaluate systems with relevant machine learning workloads and not machine learning algorithms themselves.

4.7.1 Experiment 1: Logistic Regression

In the first experiment we look at regularized logistic regression, a popular baseline method that can be solved using embarrassingly parallel algorithms such as batch gradient descent. Apache *Spark MLlib* implements Logistic Regression training using the Breeze[8] library's LBFGS solver where it locally computes partial gradient updates in parallel using all available cores and and aggregates them in the driver. As a single machine baseline we use *Vowpal Wabbit* which implements an online stochastic gradient descent using only two cores: one for parsing the input data and one to compute gradient updates.

Figure 4.2 shows the results for experiments on the *small* nodes (left) with only 4 cores and 16 GB RAM and the *big* nodes (right) with 24 cores and 256 GB RAM. For both hardware configurations, the *Spark MLlib* implementation needs significantly more time to achieve comparable AuC, even though it runs on substantially more resources. While *Vowpal Wabbit* can immediately start to update the model as data is read, Spark spends considerable time reading and caching the data, before it can run the first L-BFGS iteration. Once this is accomplished, additional iterations run very fast on the cached data. For the runs on *big* nodes (right), where the data set is actually smaller than the available main memory, it takes 6 nodes (144 cores) to get within reach of the *Vowpal Wabbit* performance using only one node and two threads/cores. For the runs on *small* nodes we observe that for 6 nodes, *Spark MLlib* is slower than *Vowpal Wabbit* on a single node. We increased the number of nodes to 15 *small* nodes (60 cores) and here (grey line on in the left plot) *Spark MLlib*gets within reach of *Vowpal Wabbit* 's performance on a single *small* node (4 cores).

[8] https://github.com/scalanlp/breeze

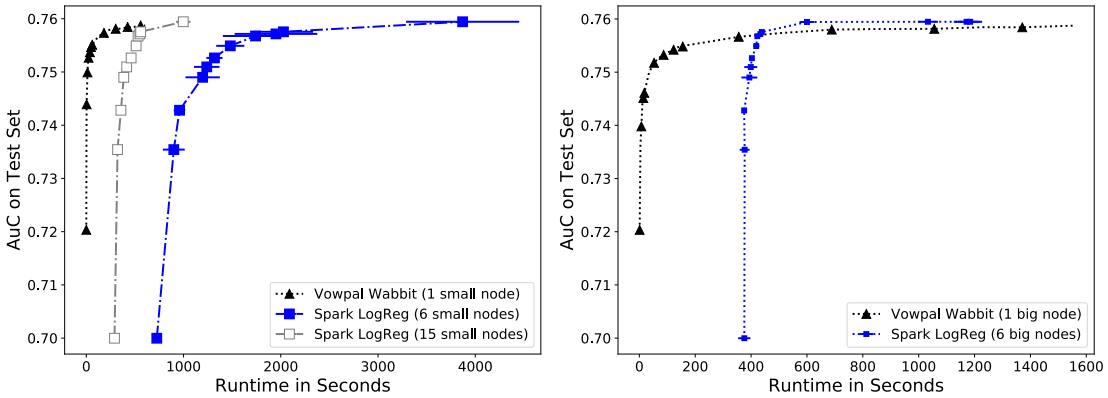

FIGURE 4.2: Logistic regression experiments using *Spark MLlib* and *Vowpal Wabbit* on *small* (4 cores, 16 GB RAM) and *big* (24 cores, 256 GB RAM) cluster nodes. The plots show AuC on a test set achieved after a certain amount of training time. Apache *Spark MLlib* needs substantially more hardware resources (15 *small* or 6 *big* nodes) to come within reach of *Vowpal Wabbit*'s performance, even though *Vowpal Wabbit* only utilizes two cores and Spark implements L-BFGS which is embarrassingly parallel.

These experiments show that even for an embarrassingly parallel learning algorithm, the latest generation distributed data flows systems such as *Spark 2.2.0*, which leverages explicit memory management and code-generation to exploit modern compilers and CPUs to obtain optimal performance on the JVM, need substantial hardware resources (factor 6-15) to obtain comparable prediction quality with a competent single machine implementation within the same time frame. (Note that since *Vowpal Wabbit* is an out-of-core library both *Spark MLLib* and *Vowpal Wabbit* could be scaled to even larger data set sizes.)

4.7.2 Experiment 2: Gradient Boosted Trees

In the second experiment we evaluate Gradient Boosted Tree classifiers, a very popular ensemble method on the *Big Node* cluster setup. We use *LightGBM*, *XGBoost* with its histogram-based algorithm as tree method and *Spark MLLib*'s Gradient Boosted Tree classifier. Figure 4.3 shows the results of our experiments. For this experiment all three evaluated libraries and systems are multi-core implementations that leverage all available

cores. Both *XGBoost* and *LightGBM* achieve about the same AuC which is better than the one achieved in the logistic regression experiments as expected. While training runs somewhat faster with*LightGBM* in our setup, there is no substantial difference to *XGBoost*. The *Spark* GBT training is substantially slower on 6 *big* nodes however. It takes almost an order of magnitude more time per iteration. It does not manage to achieve AuC comparable to *XGBoost* and *LightGBM* within the same time frame even though it uses six times the hardware resources. The overheads introduced by the distributed data processing system are substantial and the amount of extra resources needed to reach performance comparable to single machine libraries appears to lie beyond six times the single machine configuration (which is the maximum we had available for our experiments.)

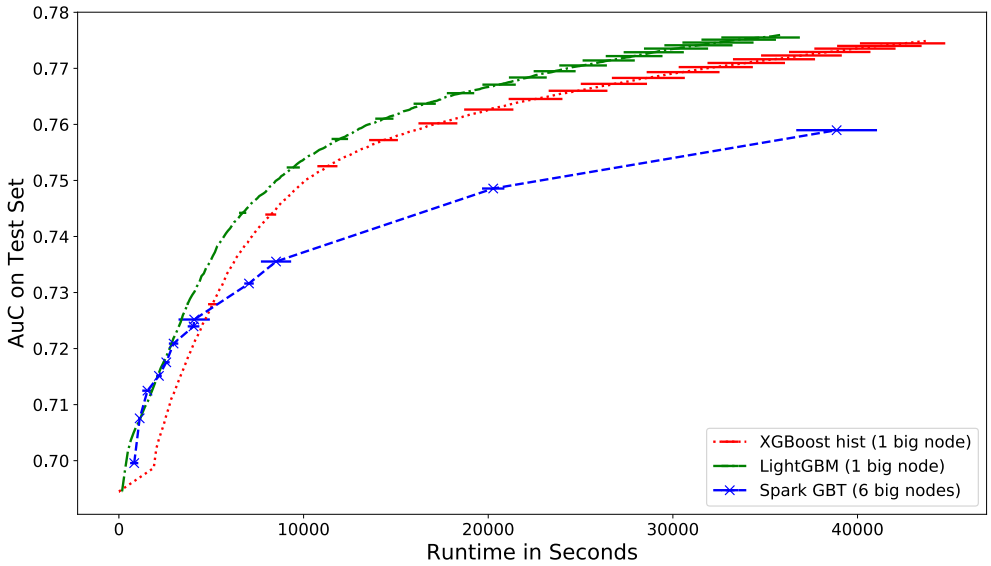

FIGURE 4.3: Gradient Boosted Trees classifier trained using *Apache Spark MLlib*, *XGBoost* and *LightGBM* on *big* (24 cores, 256 GB RAM) cluster nodes. The plots show AuC on a test set achieved after a certain amount of training time. The single machine (24 cores) implementations of *XGBoost* and *LightGBM* outperform *Spark*'s implementation on 6 nodes (144 cores).

In the absence of other reasons to execute the training of machine learning models on distributed data flow systems (e.g., raw data being already hosted in HDFS, integration

of pre-processing and feature extraction pipelines) single machine libraries would thus be the preferred method in the evaluated setting. It is thus crucial to assess if the original data pre-processing and feature extraction step necessitates a distributed data processing system in itself (e.g., because of massive raw input data set sizes) and how often this step will have to be executed and adjusted in the context of developing a machine learning model. It may very well result in the insight, that a distributed data processing system is the system of choice in a concrete application setting, in spite of the comparatively inefficient training performance.

4.8 Related Work

Benchmarking and performance analysis of data analytics frameworks have received some attention in the research community [82, 94, 104, 110]. However most of the papers focus on evaluating runtime and execution speed of non-representative workloads such as *WordCount, Grep* or *Sort*. The ones that do focus on machine learning workloads [24, 34] neglect quality metrics such as accuracy completely. Unfortunately, the systems papers introducing the second generation distributed data flow systems Apache Spark, Apache Flink and Graphlab [18, 80, 113] themselves do not provide meaningful experiments with respect to machine learning model quality. The paper presenting the *MLlib* Machine Learning of Apache Spark [86] actually only reports speed-up of the runtime relative to an older version of MLlib itself.

On the other hand there have been several endeavours in evaluating different machine learning algorithms empirically with respect to their prediction quality, e.g., [37, 38], however none of them in the light of distributed data processing systems. They actually do not taking into account the runtime of the different machine learning methods at all.

McSherry et. al [85] introduced *COST (the Configuration that Outperforms a Single Thread)* as a new metric distributed data processing systems should be evaluated against. This metric weighs a system's scalability against the overheads it introduces and reflects actual performance gains without rewarding systems that simply introduce substantial but parallelizable overheads. The authors showed, that for popular graph processing algorithms, none of the published systems managed to outperform a competent single-threaded implementation using a high-end 2014 laptop even though the distributed

systems leveraged substantial compute resources. It is thus imperative to compare these distributed data processing systems to competent single machine baselines. Contrary to this work, the authors only cover graph algorithms with a fixed result set and thus do not address the quality - runtime trade-off encountered with supervised machine learning workloads. Furthermore, they only collect published results from the system papers and do not report on any own experiments with the distributed data processing systems.

4.9 Discussion

In this Section we dealt with the important aspect of machine learning model prediction quality in the context of benchmarking distributed data processing systems. We present the position that when it comes to machine learning workloads these systems should be evaluated against sophisticated single machine libraries that practitioners would choose to solve an actual machine learning problem with respect to both: runtime as well as prediction quality metrics.

We introduced experiments, measurements and a tuning process for such a benchmark and performed such an evaluation comparing single machine libraries *Vowpal Wabbit*, *XGBoost* and *LightGBM* to *Spark MLlib*'s Logistic Regression and Gradient Boosted Trees implementation in order to explore the trade-off between runtime and model quality.

Our results indicate, that the distributed data flow system *Spark* needs substantially more resources to reach comparable performance to the single machine libraries (or even fails to do so with all nodes available to us). Given that in many cases, the size of data sets used to train machine learning models in practice appears to be in the range of available main memory of current compute nodes, it is not clear whether executing the training of machine learning models on distributed data flow systems is always the most efficient choice. However, there can be other valid reasons to rely on a distributed solution (e.g., raw data being already hosted in HDFS, integration of pre-processing and feature extraction pipelines) as we discussed in Section 4.7.

Of course this result is also not entirely surprising. The Spark implementation runs inside a JVM and introduces and enables a scale out the computations by merely adding more computed nodes. Due to this, Spark cannot exploit data locality and caches to the same degree as an optimized single machine library written in C++.

We also observed that the process of tuning hyperparamters for the libraries is still surprisingly challenging, as it is often not directly integrated into the libraries and quite time consuming to execute. However, it would appear that this problem is a perfect candidate for parallelization and distribution. Focusing on distributed grid search, cross validation and hyperparameter search thus may be a fruitful path to pursue in future work rather than focusing on distributed training of machine learning models.

Limitations. There are *limitations* to our proposed approach: First, every machine learning algorithm and its associated implementation comes with a significant set of hyperparameters that have to be tuned for each new data set. Exhaustively searching for the optimal parameter combinations is infeasible - not just for benchmarking comparisons but also when applying these algorithms in practice. The right choice of hyperparameters can have a significant impact on the prediction quality performance of a machine learning model. Our proposed approach to allot equal time frames for hyperparameter tuning is thus a limitation, but a necessary and reasonable one, since it is faced by practitioners applying these algorithms as well.

Second, the choice of algorithms and data set is strongly limited. While we are convinced that they provide valuable insights into system performance and also argued why we deem them relevant and representative, if another, special "breed" of machine learning algorithms becomes popular and thus the "method of choice", the results obtained in this thesis may not be as representative.

Third, we rely on library implementations for the prediction quality experiments, which is not a strict comparison of algorithmically and mathematically equivalent approaches. Superior performance may thus be due to a superior implementation or approach and not necessarily just due to systems constraints. However, these are the libraries that are available to scientists and practitioners that wish to apply these algorithms.

Furthermore, the level of sophistication and optimization that these libraries have achieved by significant development efforts by the scientific and open source communities is one that is hard if not impossible to obtain by re-implementing algorithms from scratch for benchmarking purposes.

Future systems research should include an evaluation like the one we presented in this chapter for representative use cases. These experiments which explore the trade-off between runtime and model quality and provide solid single machine baselines have to be

a core part of experimental evaluations of distributed data processing systems and an integral part of any benchmark to evaluate these systems for scalable machine learning workloads.

5 Reproducibility of Benchmarks

5.1 Problem Statement

In this chapter, we discuss the complexity of the process of carrying out benchmark experiments for distributed data processing systems and present a framework to reduce the operational complexity of executing such benchmarks and thus to foster repeatability and reproducibility of benchmark experiments. As we introduced in Section 2.5, a benchmark consists of data sets (or data generators), workload implementations and experiment definitions assembled with the purpose to evaluate one particular system, referred to as the *system under test (SUT)*. These experiment definitions outline specific values for system configuration and workload parameters that have to be varied in order to evaluate a particular characteristic of the system under test. In the context of evaluating distributed data processing systems for machine learning workloads, as we discussed in the previous two chapters, this requires: (1) the implementation of the application workload (e.g., learning algorithm) to be evaluated which takes specific input parameters (e.g., input and output path, loss function or regularizer to be used) specified in the experiment definitions. This workload implementation then (2) needs to be executed on top of the particular distributed data processing system under test with the exact configuration specified in the experiment definitions. (e.g., memory allocation settings, serializer to be used or degree of parallelism) while its key performance characteristics (e.g., runtime, throughput, accuracy) are measured and recorded.

Executing all these steps for benchmark experiments on modern data management systems such as distributed data processing systems is significantly more complex than evaluating relational database management system (RDBMS). Figure 5.1 illustrates that rather than having a single system under test running in isolation, novel data processing systems in fact require several components such as a distributed file system

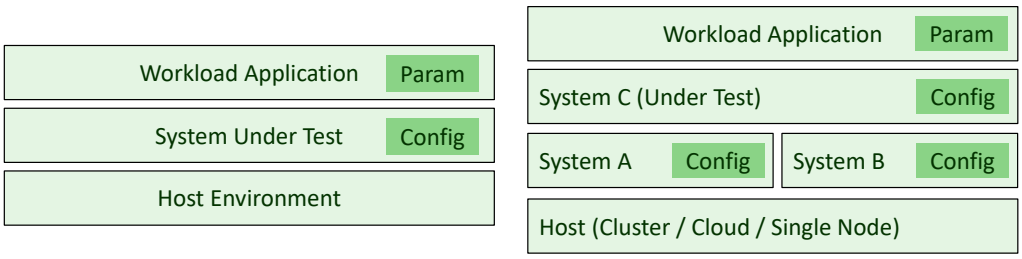

FIGURE 5.1: The general setup: contrary to the setup when benchmarking traditional relational database management systems (RDBMS) (left) where we evaluate only one system under test (SUT), the landscape is more complicated when evaluating novel distributed data processing frameworks (right), as they usually require the interplay between multiple independent systems. Each of these systems has its own configurations with sets of parameters that have to be set, and potentially tuned.

and a distributed data processing system to be executed jointly. Current trends actually advocate an architecture based on interconnected systems (e.g., HDFS, Yarn, Spark, Flink, Storm). Each of these systems has to be set up and launched with their own set of, potentially hardware-dependent, configurations that tend to span multiple configuration files.

Typically, one is not just interested in the insights obtained by a single experiment, but in trends highlighted by a suite of experiments where a certain system under test configuration or workload parameter value is varied while all other parameters remain fixed. When running a scale-out experiment with a varying number of nodes for example, the configuration of both the distributed file system as well as the distributed data processing system under test have to be changed, and the systems have to be appropriately orchestrated. This further complicates the benchmarking process. Additionally, hardware-dependent configurations hinder portability and thus reproducibility of benchmarks. When adjusting the systems is done manually, huge amounts of temporary files and generated data tend to clog up the disk space, as experiments may be run without proper tear-down of systems and cleaning of temporary directories. When such experiments are run on a shared cluster, as is often the case in an academic environment, this issue becomes even more severe.

However, these challenges are not only encountered in the context of benchmarking, they are also faced when running systems experiments that form an integral part of database and distributed systems research papers. As we discussed in Section 2.1, distributed data processing systems went through a rapid development in the light of *Big Data Analytics*. While nearly all these systems have been presented in scientific publications containing an experimental evaluation, it remains a challenge to objectively compare the performance of each system. Different workloads and implementations, usage of libraries, data sets and hardware configurations make it hard if not impossible to leverage the published experiments for such a comparison. Furthermore, it is a challenge to assess how much of the performance gain is due to a superior paradigm or design and how much is due to a more efficient implementation, which ultimately impairs the scientific process due to a lack of verifiability. For this reason, it is imperative to enable and establish benchmarks for big data analytics systems. However, even if workloads and data sets or data generators are fixed, orchestrating and executing benchmarks can be a major challenge.

Reproducible and repeatable systems experiments are a crucial building block to successful and fruitful systems research. This fact is also honored by the introduction of a so-called Reproducibility Track in ACM SIGMOD and Experiments & Analysis Papers in VLDB, two of the top tier academic conferences in database systems research. Next to systems researchers, scientists and practitioners who wish to assess the suitability of distributed data processing systems to their problem domain also face the significant operational complexity of executing benchmark experiments for distributed data processing systems.

5.2 Contribution

To address these problems and to enable and foster reproducible experiments and benchmarks of distributed data processing systems by lowering the operational complexity, we present *PEEL*, a framework to define, execute, analyze, and share experiments. On the one hand, PEEL automatically orchestrates experiments and handles the systems' setup, configuration, deployment, tear-down and cleanup as well as automatic log collection. On the other hand, PEEL introduces a unified and transparent way of specifying

experiments, including the actual application code, system configuration, and experiment setup description. With this transparent specification, PEEL enables the sharing of end-to-end experiment artifacts, thus fostering reproducibility and portability of benchmark experiments. PEEL also allows for the hardware independent specification of these parameters, therefore enabling portability of experiments across different hardware setups. Figure 5.2 illustrates the process enabled by our framework. Finally, PEEL is available as open-source software on `GitHub`.

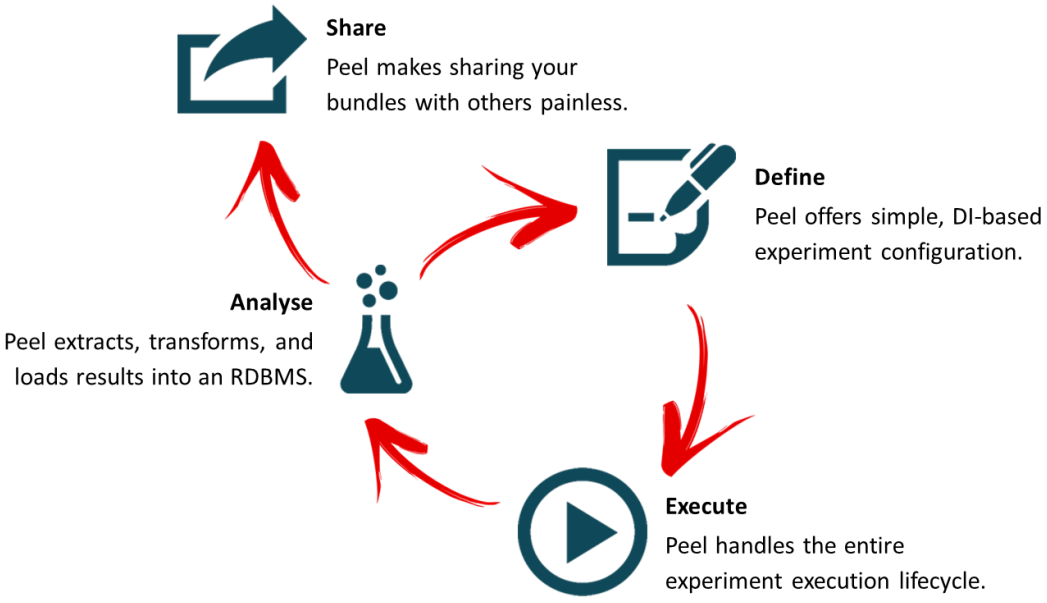

FIGURE 5.2: The PEEL process: PEEL enables the transparent specification of workloads, systems configurations, and parameters to be varied. It also automatically handles the distributed orchestration and execution as well as sharing of these experiment bundles. After successfully running all experiments in a bundle, PEEL automatically extracts, transforms and loads relevant measurements from collected log files and makes them available in an RDBMS.

5.3 Overview

The rest of this Chapter is structured as follows: In Section 5.4 we briefly introduce the running example of a supervised machine learning workload, which we will use to explain the details of defining an experiment. Section 5.5 introduces experiment definitions. Next, Section 5.6 describes the basics of a bundle. Section 5.7 discusses the approach of a unified, global experiment environment configuration and Section 5.8 illustrates how PEEL bundles can be deployed and executed on cluster environments, before Section 5.9 provides an overview, how results can be gathered and analyzed within the framework. Finally, we describe how PEEL can be extended with additional systems in Section 5.10.

5.4 Running Example: Benchmarking a Supervised Machine Learning Workload

As a running example, we consider a supervised machine learning workload as described in Section 3.5. The workload task is to train a logistic regression model for *click-through rate prediction* using a batch gradient descent solver. We are interested in evaluating the scaling behavior of the systems not just with respect to growing data set sizes and increasing numbers of compute nodes but also with respect to increasing model dimensionality as we motivated in 3.5. Since increasing the model dimensionality invariably increases the main memory required for the broadcast variable, we also wish to evaluate the workload on two different cluster hardware configurations with varying memory size.

Figure 5.3 illustrates the running example. We want to evaluate *batch gradient descent* training of a supervised learning model as a workload on Apache Spark and Apache Flink as a *system under test*. Only for this part do we want to time the execution and record the system performance characteristics of the individual compute nodes involved. In order to evaluate the scalability of the systems with respect to a varying number of compute nodes as well as a varying dimensionality of the training data (and thus also the model to be trained) the two parameters: *nodes* and *dimensions* have to be varied accordingly.

For each particular configuration of physical compute nodes, a new distributed file system (HDFS) instance has to be set up. Next the raw Criteo data, which is ingested from some external storage, has to be transformed to feature vectors of the desired

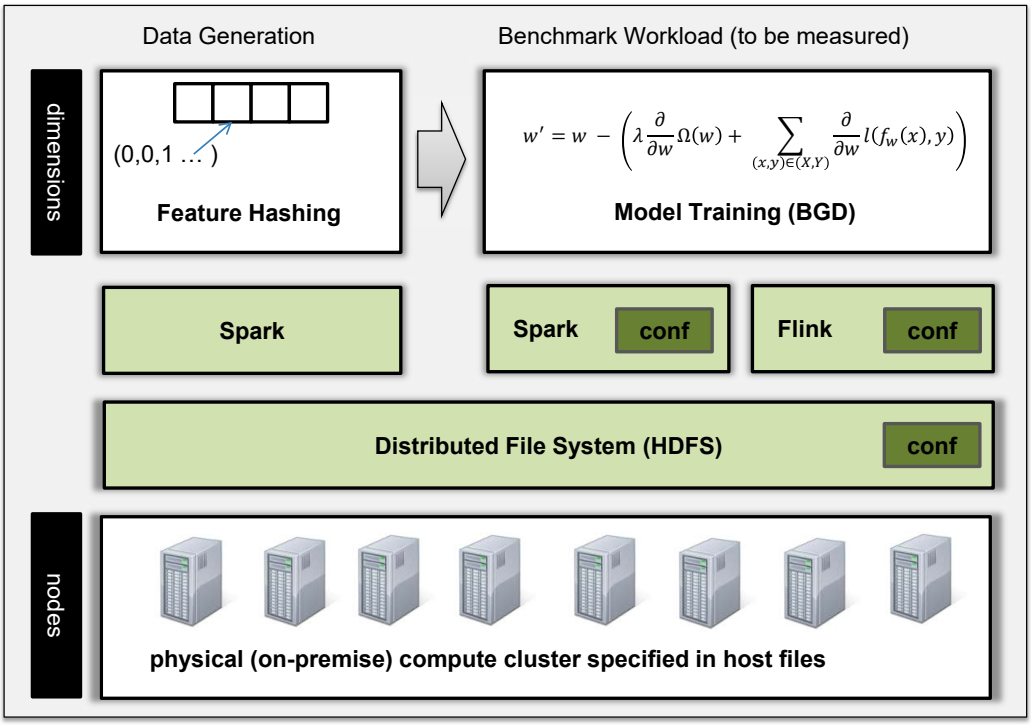

FIGURE 5.3: An illustration of the running example: we evaluate *batch gradient descent* training of a supervised learning model as a workload on Apache Spark and Apache Flink as *systems under test*. The individual experiments depend on two parameters: the number of physical compute *nodes* and the dimensionality of the training data set (*dimensions*), which specify how the benchmark workload shall be executed. The data generation job is executed on a system independent of the system under test.

dimensionality via feature hashing. Any system may be used for this step - independent of the actual system under test. In the example, we choose to run a Spark Job, which writes out the experimentation data into the temporary HDFS instantiated for the current (cluster) configuration. Next, the actual system under test (Spark or Flink in our example) will have to be set up and instantiated with its proper configuration. Once it is up and running, the benchmark workload can be submitted as a job to the system under test

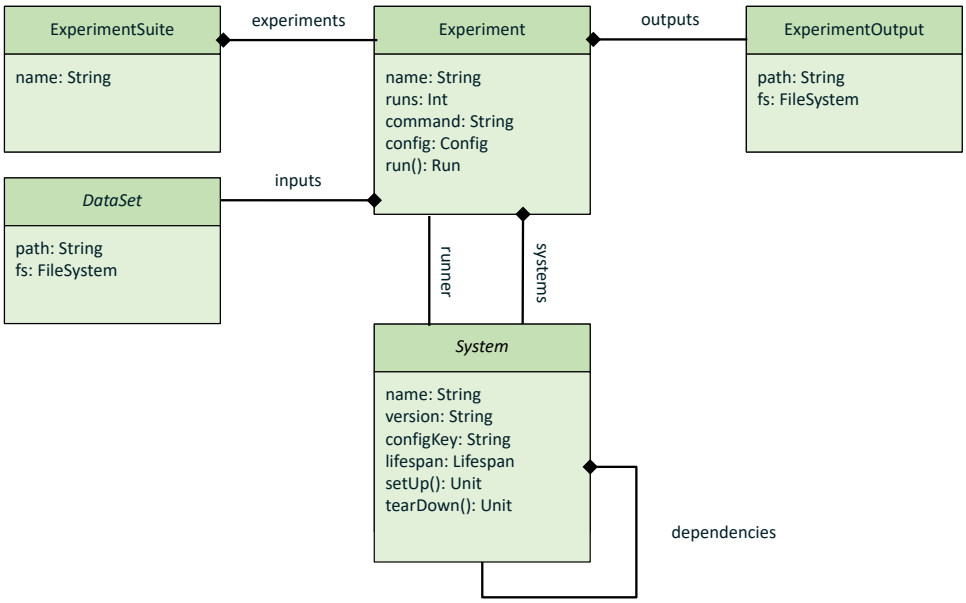

FIGURE 5.4: A domain model of the PEEL experiment definition elements.

and its execution is timed. In order to record the performance characteristics on the individual compute nodes, an additional monitoring system such as *dstat* will have to be started on all compute nodes.

After successful execution, the system under test will have to be shut down. In order to archive all aspects of the benchmark experiments, various logs of the different systems involved (e.g., *dstat* for performance statistics, system under test, HDFS) will have to be gathered from all the compute nodes. Next, all temporary directories have to be cleaned, and the next system has to be set up and instantiated. Once all systems have been evaluated for a concrete dimensionality, the data set has to be deleted from the distributed file system and the next one, with a new dimensionality, has to be created. When all parameter settings for a particular node configuration (i.e., all dimensionalities) have been evaluated, the distributed file system will have to be torn down and a new one, with a different node configuration, will have to be set up.

Manually administering all these steps is a tedious and error-prone process as jobs can run for long periods of time, and may fail. PEEL automatically takes care of all the steps outlined above and thus reduces the operational complexity of benchmarking distributed data processing systems significantly. In the following sections, we will explore how such a workload like the supervised learning example has to be specified in PEEL in order to benefit from this automation.

5.5 Experiments and ExperimentSuites

Figure 5.4 illustrates the main components of PEEL in a domain model. A central element of this model are *Experiment* definitions, which specify all the parameters necessary for experiment runs. These experiments are grouped together in *ExperimentSuites* that embody entire parameter sweeps that constitute a benchmark experiment. The PEEL *Experiments* are defined using a Spring dependency injection container as a set of interconnected beans. Definition of beans can be carried out in either XML or via annotated Scala classes. (Scala is used for all examples in this chapter.)

We introduce and discuss the individual bean types in light of our example of running *batch gradient descent* training of a logistic regression model for click-through rate prediction. We discuss the components of the definition in Listings 5.1 - 5.3 in the next subsections. Listing 5.1 registers an application context and introduces the datasets and the feature hashing pre-processing job used to generate data with the required dimensionality. Listing 5.2 shows the actual *Experiment* definition of the Spark and Flink jobs that are the subject of the benchmark experiments. Listing 5.3 shows the defintion of the *ExperimentSuite* including parameter ranges and experiments.

5.5.1 Data Sets

Experiments typically depend on some kind of input data, represented as abstract *DataSet* elements associated with a particular *FileSystem* in our model. The following types are currently supported:

- *CopiedDataSet* - used for static data copied into the target FileSystem;

- *GeneratedDataSet* - used for data generated by a Job into the target FileSystem.

In the example we rely on a `GeneratedDataSet` to trigger the Spark job for feature hashing (Lines 21-25 in Listing 5.1). In addition, each experiment bean is associated with an *ExperimentOutput* which describes the paths the data is written to by the experiment workload application (Lines 27-30 Listing 5.1). This meta-information is used to clean those paths upon execution.

Listing 5.1: First part of the defintion of the running example.

```
1  class ExperimentsDimensionScaling extends ApplicationContextAware {
2    var ctx: ApplicationContext = null
3    def setApplicationContext(ctx: ApplicationContext): Unit = {
4      this.ctx = ctx
5    }
6
7    def sparkFeatureHashing(i: String, numF: Int, perc: Double): SparkJob = new SparkJob(
8      timeout = 10000L,
9      runner = ctx.getBean("spark-1.6.2", classOf[Spark]),
10     command =
11       s"""
12         |--class dima.tu.berlin.generators.spark.SparkCriteoExtract \\
13         |${app.path.datagens}/peel-bundle-datagens-1.0-SNAPSHOT.jar \\
14         |--inputPath=$i \\
15         |--outputPath=${system.hadoop-2.path.input}/train/$numF/$perc \\
16         |--numFeatures=$numFeatures \\
17         |--percDataPoints=$perc \\
18       """.stripMargin.trim
19   )
20
21   def `bgd.input`(D: Int): DataSet = new GeneratedDataSet(
22     src = sparkFeatureHashing("/criteo/", numFeatures, perc),
23     dst = s"${system.hadoop-2.path.input}/train/" + numFeatures + "/" + perc,
24     fs = ctx.getBean("hdfs-2.7.1", classOf[HDFS2])
25   )
26
27   def `bgd.output`: ExperimentOutput = new ExperimentOutput(
28     path = "{system.hadoop-2.path.input}/benchmark/",
29     fs = ctx.getBean("hdfs-2.7.1", classOf[HDFS2])
30   )
```

LISTING 5.2: The main experiment definition of the running example batch gradient descent training of a supervised learning model

```
def ‘bgd.flink‘(D: Int, N: String) = new FlinkExperiment(
  name  = s"flink.train.$D",
  command =
    s"""
      |--class dima.tu.berlin.benchmark.flink.mlbench.NewLogReg \\
      |$${app.path.apps}/peel-bundle-flink-jobs-1.0-SNAPSHOT.jar \\
      |--trainDir=$${system.hadoop-2.path.input}/train/$D \\
      |--outputDir=$${system.hadoop-2.path.input}/benchmark/$N/$D/flink/ \\
      |--degOfParall=$${system.default.config.parallelism.total} \\
      |--dimensions=$D \\
    """.stripMargin.trim,
  config = ConfigFactory.parseString(
    s"""
      |system.default.config.slaves  = $${env.slaves.$N.hosts}
      |system.default.config.parallelism.total = $${env.slaves.$N.total.parallelism}
    """.stripMargin.trim),
  runs  = 3,
  runner = ctx.getBean("flink-1.0.3", classOf[Flink]),
  systems = Set(ctx.getBean("dstat-0.7.2", classOf[Dstat])),
  inputs = Set(‘bgd.input‘(D), classOf[DataSet])),
  outputs = Set(‘bgd.output‘)
)

def ‘bgd.spark‘(D: Int, N: String) = new SparkExperiment(
  name  = s"spark.train.$D",
  command =
    s"""
      |--class dima.tu.berlin.benchmark.spark.mlbench.RUN \\
      |$${app.path.apps}/peel-bundle-spark-jobs-1.0-SNAPSHOT.jar \\
      |--trainDir=$${system.hadoop-2.path.input}/train /$D \\
      |--outputDir=$${system.hadoop-2.path.input}/benchmark/$N/$D/spark \\
      |--numSplits=$${system.default.config.parallelism.total} \\
    """.stripMargin.trim,
  config = ConfigFactory.parseString(
    s"""
      |system.default.config.slaves  = $${env.slaves.$N.hosts}
      |system.default.config.parallelism.total = $${env.slaves.$N.total.parallelism}
    """.stripMargin.trim),
  runs  = 3,
  runner = ctx.getBean("spark-1.6.2", classOf[Spark]),
  systems = Set(ctx.getBean("dstat-0.7.2", classOf[Dstat])),
  inputs = Set(‘bgd.input‘(D), classOf[DataSet])),
  outputs = Set(‘bgd.output‘)
)
```

5.5.2 Experiment

The central class in the domain model shown in Figure 5.4 is *Experiment*. In our example definition in Listing 5.2 we specify two experiments: one for Flink (Lines 1-22 in Listing 5.2) and one for Spark (Lines 24-44 in Listing 5.2). Each experiment specifies the following properties: the experiment name, the command that executes the experiment's job, the number of runs (repetitions) the experiment is executed, the inputs required and outputs produced by each run, the runner system that carries the execution, other systems, upon which the execution of the experiment depends (e.g., *dstat* in Lines 19 and 41 in Listing 5.2 for monitoring the resource usage on the compute nodes) as well as the experiment-specific environment config, which is discussed in Section 5.7.

5.5.3 System

The second important class in the model is *System*. It specifies the following properties: the system name, usually fixed per System implementation, e.g., `flink` for the Flink system or `spark` for the Spark system, the system version (e.g., 1.0.3 for Flink or 1.6.2 for Spark), a *configKey* under which config parameters will be located in the environment configuration, usually the same as the system name, a Lifespan value (one of *Provided*, *Suite*, *Experiment*, or *Run*) which indicates when to start and stop the system and a list of systems upon which the current system depends.

5.5.4 ExperimentSuite

A series of related experiment beans are organized in an *ExperimentSuite*. In our example listing, we define an ExperimentSuite in Listing 5.3. Recall that our original motivation was to compare the scale-out characteristics of Spark and Flink with respect to both: scaling the nodes and scaling the model size. To accomplish this, we vary two parameters: *Dims* which specifies the dimensionality of the training data and *Nodes*, which refers to a list of hosts the experiment should run on. The for-comprehension creates a cartesian product of all parameter values and the two experiments. With this, we ensure that we only generate a new data set whenever either the node configuration or the desired dimensionality changes, but not for each experiment separately.

Listing 5.3: Definition of the ExperimentSuite

```
  def `bgd.dimensions.scaling`: ExperimentSuite = new ExperimentSuite(
    for {
      Dims <- Seq(10, 100, 1000, 10000, 100000, 1000000)
      Nodes <- Seq("top020", "top010", "top005")
      Exps <- Seq(`bgd.spark`(Dims, Nodes), `bgd.flink`(Dims, Nodes))
    } yield Exps
  )
}
@Bean(name = Array("bgd.dimensions.scaling"))
```

5.6 PEEL Bundles

As a central structure, PEEL *bundle* packages together the configuration data, datasets, and workload jobs as well as experiment definitions required for the execution of a particular set of experiments. Table 5.1 provides an overview of the top-level elements of such a bundle. It is self-contained and can be pushed to a remote cluster for execution as well as shared for reproducibility purposes. The main components of a bundle can be grouped as follows:

Default Path	Config Parameter	Fixed	Description
./apps	app.path.apps	Yes	Workload applications.
./config	app.path.config	Yes	Configurations and experiment definitions.
./datagens	app.path.datagens	No	Data generators.
./datasets	app.path.datasets	No	Static datasets.
./downloads	app.path.downloads	No	Archived system binaries.
./lib	app.path.log	Yes	PEEL libraries and dependencies.
./log	app.path.log	Yes	PEEL execution logs.
./results	app.path.results	No	State and log data from experiment runs.
./systems	app.path.systems	No	Contains all running systems.
./utils	app.path.utils	No	Utility scripts and files.
./peel.sh	app.path.cli	Yes	The PEEL command line interface.

Table 5.1: The top-level elements of a bundle.
(Non-fixed paths can be customized.)

At the center of a bundle is the *PEEL command line interface (PEEL CLI)*, which provides the basic functionality of PEEL. While running, the PEEL CLI spawns and executes operating system processes. It can be used to start and stop experiments, and to push and pull bundles to and from remote locations. The *log* folder contains the `stdout` and `stderr` output of these processes, as well as a copy of the actual console output produced by PEEL itself. The *config* folder contains the experiment definitions in scala as well as **.conf* files written in HOCON[1] syntax which define the environment configuration parameters we discuss in Section 5.7. The *apps* folder contains the binaries of the workloads to be executed in the experiments. The *datasets folder* can be used to store static datasets to be used for the experiments. (Larger Data sets can also be generated using Data Generators or read directly from static HDFS instances.) The *datagens* folder may contain applications for the dynamic generation of datasets for experiments. The *downloads* folder contains system binary archives for all the systems in the experiment environment which are extracted into the *systems* folder by default. The *results* folder unites the data of all experiments - attempted and successful - that has been collected. It is stored in folders following the hierarchy `$suite/$expName.run$NN` as naming convention. Finally, the *utils folder* contains utility scripts (e.g., SQL queries and gnuplot scripts) that can be used next to or in conjunction with PEEL CLI commands. A *PEEL bundle* is the unit to be shared when making available experiment suites or entire benchmarks, which utilize the framework.

5.7 Environment Configurations

In order to foster transparency and to enable automatic execution of the specified experiments, PEEL introduces a unified, global approach to the configuration of the environment and all systems involved in the experiments. Configurations are instantiated as *Environments* with a concrete set of configuration values for all systems involved and parameter values for the actual experiment workloads. Consider our example of running batch gradient descent with a varying number of nodes and data dimensionality depicted in Figure 5.3. A concrete *Environment* instantiation for this example contains

[1] https://github.com/typesafehub/config/blob/master/HOCON.md

specific configuration values for HDFS, Spark and Flink as well as a particular dimension parameter. PEEL's global approach allows proper management of the following issues:

Syntax Heterogeneity. Each system involved (e.g., HDFS, Spark and Flink) would have to be configured separately, relying on its own own special syntax when using a naïve approach of manual configuration for each system and experiment. This requires knowledge of the various configuration parameters of all systems in the stack. (e.g., the number of processing slots is called `spark.executor.cores` in Spark and `taskmanager.numberOfTaskSlots` in Flink) and thus should be avoided.

Variable Interdependence. The sets of configuration variables associated with each system are not mutually exclusive. Thus, care has to be taken such that the corresponding values are consistent for the overlapping fragment (e.g., the slaves list containing the compute nodes should be consistent across systems).

Value Tuning. For a series of related experiments, all but a very small set of configuration values remain fixed. These configuration values (e.g., memory allocation, degree of parallelism, temp paths for spilling) have to be set and tuned beforehand and chosen such that they maximize the performance of the corresponding systems on the host hardware used for the experiments.

With its global environment approach and by associating one global environment configuration to each experiment, PEEL can automate these steps and promotes configuration reuse through layering as well as configuration uniformity through a hierarchical syntax.

In our example of running batch gradient descent with a varying number of nodes and data dimensionality we vary the number of nodes between three different configurations (20, 10, and 5 nodes) as specified in the configuration Listing 5.2 in line 75. For each of the different dimensionalities to be evaluated (specified in line 74) this will result in six experiments ($3 \times$ SparkBGD $+ 3 \times$ FlinkBGD). Each of these will have an associated *config* property containing the configuration settings for the system under test (Spark or Flink), HDFS as well as the parameters of the workload (BGD) - as depicted in Figure 5.5. This config property is a hierarchical map of key-value pairs which constitute the configuration of all systems and jobs required for that particular experiment. They are constructed according to a layering scheme and conventions which we discuss next.

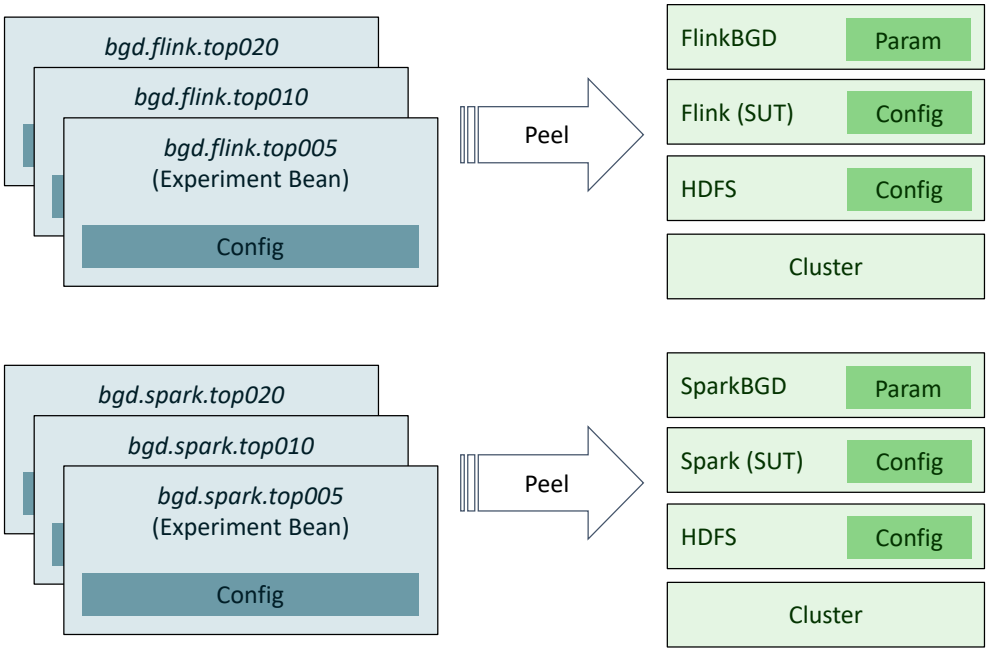

FIGURE 5.5: Mapping the environment configurations for the six Batch Gradient Descent experiments: for each different experiment host configuration (on the left) PEEL instantiates the relevant systems with the appropriate system configurations and job paramters (on the right).

5.7.1 Configuration Layers.

PEEL's configuration system is built upon the concept of layered construction and resolution based on the following three layers of configuration:

- **Default.** Default configuration values for PEEL itself and the supported systems. Packaged as resources in related jars located in the bundle's `app.path.lib` folder.
- **Bundle.** Bundle-specific configuration values. Located in `app.path.config` (the *config* sub-folder of the current bundle by default).
- **Host.** Host-specific configuration values. Located in the `$HOSTNAME` sub-folder of the `app.path.config` folder.

Path	**Description**
`reference.peel.conf`	Default PEEL config
`reference.$systemID.conf`	Default system config
`config/$systemID.conf`	Bundle-specific system config (opt)
`config/hosts/$hostname/$systemID.conf`	Host-specific system config (opt)
`config/application.conf`	Bundle-specific PEEL config (opt)
`config/hosts/$hostname/application.conf`	Host-specific PEEL config (opt)
Experiment bean *config* value	Experiment specific config (opt)
System	JVM system properties (constant)

TABLE 5.2: Hierarchy of configurations which are associated with an experiment bean (higher in the list represents lower priority).

PEEL constructs an associated configuration for each *Experiment* bean defined in an *ExperimentSuite* in accordance with the priority outlined in Table 5.2, where higher in the list represents lower priority.

At lowest priority comes the *default* configuration, located in the `peel-core.jar` package. Next, for each system upon which the experiment depends (with corresponding system bean identified by `systemID`), PEEL tries to load the default configuration for that system as well as bundle- or host-specific configurations.

Third, a bundle- and host-specific `application.conf`, which is a counterpart of and respectively overrides bundle-wide values defined in `reference.peel.conf` is considered

Next, the values defined in the `config` property of the current experiment bean are considered. These are typically used to vary one particular parameter in a sequence of experiments in a suite (e.g., varying the number of nodes in our example). Finally, a set of hardware configuration parameters derived from the host description (e.g., the number of CPUs or the total amount of available memory) are appended to the configuration. This allows for the sharing of configurations for a particular host environment (hardware setup). The required evaluation on two different hardware setups that we indicated in the example introduced in Section 5.4 can be achieved by adding another host configuration that lays out the hardware specification of the cluster nodes.

5.8 Execution Workflow

In the previous sections, we explained how to define experiments and the internals required to configure the environment in a PEEL bundle. In this section, we explain how to leverage the PEEL command line interface and to use the commands provided by the PEEL CLI to deploy and run the experiments in a bundle.

As a first step, the bundle has to be assembled from the sources with `mvn deploy`. For large-scale applications, the environment where the experiments need to be executed typically differs from the environment of the machine where the bundle binaries are assembled. In order to start the execution process, the user therefore needs to first deploy the bundle binaries from the local machine to the desired host environment. The PEEL CLI offers a special command for this. In order to push the PEEL-bundle to the remote cluster, one has to run: `./peel.sh rsync:push remote-cluster-name`. The command uses rsync to copy the contents of the enclosing PEEL bundle to the target environment. The connection options for the rsync calls are thereby taken from the environment configuration of the local environment. The remote environment has to be specified in the `application.conf`.

As explained above, PEEL organizes experiments in sequences called *experiment suites*. The easiest option is to start an entire suite via `./peel.sh suite:run` which automatically steps through the entire execution life cycle for each experiment. The following steps are considered by PEEL:

- **Setup Experiment.** Ensure that the required inputs are materialized (either generated or copied) in the respective file system. Check the configuration of associated descendant systems with *provided* or *suite* lifespan against the values defined in the current experiment config. If the values do not match, it reconfigures and restarts the system. Set up systems with *experiment* lifespan.

- **Execute Experiment** For each experiment run which has not been completed by a previous invocation of the same suite: Check and set up systems with *run* lifespan, execute experiment run, collect log data from the associated systems and clear the produced outputs.

- **Tear Down Experiment.** Tear down all systems with *experiment* lifespan.

Next to simply running a full *ExperimentSuite* which automatically executes all experiments specified, each of the above steps can be executed individually. This is particularly useful when developing and debugging a benchmark, as it allows to validate that each step is executed correctly.

Since PEEL also keeps track of failed experiments, one can simply re-run an entire suite in order to re-attempt the execution of the failed experiments. PEEL will automatically skip all experiments, which have already been successfully run.

5.9 Results Analysis

As we described in Section 5.6, the results of all experiments are stored in a folder structure which contains log file data collected from all the systems involved in the experiments. In order to enable the analysis and inspection of this data, PEEL provides an extensible ETL pipeline that extracts relevant data from the log files, transforms it into a relational schema, and loads it into a database where it can be analyzed and queried. The experiment suite defined by the running example in Section 5.4 will produce results similar to Table 5.3.

experiment	**nodes**	**dimensions**	**runtime in ms**
flink.train	top023	10	165612
flink.train	top023	100	265034
flink.train	top023	1000	289115
flink.train	top023	10000	291966
flink.train	top023	100000	300280
flink.train	top023	1000000	315500
spark.train	top023	10	128286
spark.train	top023	100	205061
spark.train	top023	1000	208647
spark.train	top023	10000	219103
spark.train	top023	100000	222236
spark.train	top023	1000000	298778
...	...	...	...

TABLE 5.3: Exemplary table listing the results of experiment runs.

5.10 Extending PEEL

In order to add support for a new system, one simply has to define the start-up and shutdown behavior of the system, the configuration files and their management, and the way log files are to be collected inside the system class. As was presented in the example definition in Listing 5.2, the experiment bean then defines how jobs for the system are started and which arguments are passed. For cluster configurations without a network file system, PEEL also provides utility functions to distribute the required system files among the cluster nodes, as well as the collection of log files.

5.11 Related Work

While not directly targeted at benchmarking systems, certainly related in light of the evaluation of machine learning workloads is OpenML [109], an online machine learning platform that enables machine learning practitioners to log and upload data sets, machine learning tasks, machine learning workflows (pipelines) and results of concrete runs of such workflows. Contrary to the work presented in this chapter, it is not a framework to run experiments, but rather an archive and reservoir of data sets, tasks, workflows and run results that can be leveraged by researchers and practitioners which also fosters reproducibility and transparency. As such, it also offeres so-called *benchmarking suites* [6], which are collections of data sets that can be used for benchmarking activities.

Another somewhat related initiative is ALOJA [96], a Big Data Benchmark Repository and platform for performance analysis. ALOJA features a database of thousands of benchmark experiment runs of Apache Hadoop. Contrary to the work presented in this chapter, it is not an extendable framework that can be used to automatically evaluate various big data processing systems, but rather a repository of experiment runs that was partially filled by automatically carrying out over 30,000 executions [1].

5.12 Discussion

Properly carrying out benchmark experiments of distributed data processing systems in a transparent and reproducible manner is a complex challenge but also a crucial building

block to successful and fruitful systems research. In this chapter, we introduced PEEL as a framework for benchmarking distributed systems and algorithms. PEEL significantly reduces the operational complexity of performing benchmarks of novel distributed data processing systems. It automatically orchestrates all systems involved, executes the experiments and collects all relevant log data. Through the central structure of a PEEL-bundle, a unified approach to system configurations and its experiment definitions, PEEL fosters the transparency, portability, and reproducibility of benchmarking experiments. Based on the running example of a supervised machine learning workload, we introduced all the major concepts of PEEL, including experiment definitions and its experimentation process. We have successfully used PEEL in practice to orchestrate all of the distributed experiments in this thesis and hope that it will be a useful tool for researchers and practitioners in the benchmarking and systems research community alike, as PEEL is freely available as open-source software.

Limitations. Current *limitations* of PEEL are its support of a limited amount of systems. PEEL supports various versions of the following systems out of the box: *Hadoop MapReduce, Spark, Flink, HDFS, dstat* and *Zookeeper*. However, the framework can easily be extended. Adding support for a new system is uncomplicated and only requires the definition of system specific sub-classes for the `System` and texttttExperiment base-classes that were discussed in Section 5.5. The communication between the framework and the systems is typically done by calling scripts via external processes with the abstractions provided in PEEL. Thus, the range of systems that can be supported is not strictly limited to JVM-based ones. However, the current set up is geared towards distributed data processing systems that run on top of the HDFS distributed file system.

Next to the important components of addressing all aspects of scalability discussed in Chapter 3 and properly exploring the trade-off space between algorithm runtime and model quality discussed in Chapter 4, this framework constitutes an essential building block for building benchmarks for distributed data flow systems in the context of scalable machine learning algorithms.

6 Conclusion

In this Section we summarize our findings and distill the key insights with regard to both how to design a benchmark for distributed data processing systems for scalable machine learning workloads as well as with regard to the results obtained by our experiments with state of the art distributed data flow systems and machine learning algorithms. We conclude by putting the results into context and sketching interesting research questions that could be the subject of future work.

6.1 Summary

Big Data Analytics frameworks that can robustly scale out computations on massive data sets to many compute nodes such as distributed data flow systems have been a fruitful research topic in academic systems research and have been widely adopted in industrial practice. Their importance as a stimulus for the database systems and distributed systems research community is for instance emphasized by the fact that Matei Zaharia received the ACM Doctoral Dissertation Award for his work on Resilient Distributed Datasets and Spark [2]. Both Apache Spark and Apache Flink have built a rich community of developers and committers, and the associated developer conferences Flink Forward and Spark Summit attract hundreds and thousands of attendees respectively. While most of these systems like Apache Spark or Apache Flink are general purpose data processing systems that target a broad variety of workloads, they also explicitly claim to be suitable for scaling out machine learning algorithms. Apache Spark and its associated machine learning library MLLib are popular choices for this particular use case.

Benchmarks for traditional database management systems that evaluate the performance of database systems for transactional workloads *(TPC-C)* and for decision support scenarios *(TPC-H)* have evolved and are widely accepted. However, in the context of

distributed data flow systems, in particular for scalable machine learning workloads, it remained an open question how to properly evaluate these systems for this use case. An objective set of workloads, experiments and metrics that adequately assess how well data processing systems achieve the objective to scale out machine learning algorithms is essential to steer future systems research in the distributed systems and database systems communities. It is also a useful tool for scientists and practitioners who want to apply scalable machine learning algorithms to their problem domains and to asses which system is suitable for their problem setting.

In this thesis, we presented work that establishes such a benchmark of distributed dataflow system for scalable machine learning workloads and provides valuable insights for systems researchers by highlighting shortcomings in current system architectures.

In Chapter 3 we discussed the need to address all dimensions of scalability, including the one of model dimensionality when performing such an evaluation. Experiments with the state of the art distributed data flow systems Apache Flink and Apache Spark revealed, that while both are able to robustly scale with increasing data set sizes, the systems are surprisingly inefficient at coping with high dimensional data due to the chosen approach to memory management and support of broadcast variables.

In Chapter 4 we discussed the relevance of exploring the trade-off between runtime and model quality when evaluating distributed data flow systems for scalable machine learning workloads. Since the main memory sizes of compute nodes generally available today are in the same range as the majority of data set sizes analyzed in practice, as we discussed in Section 4.4, it is also imperative to compare against sophisticated single machine implementations of machine learning algorithms as an absolute baseline. Taking into account scalability experiments is not sufficient. Our evaluation indicates that even latest generation distributed data flows systems such as Spark require substantial hardware resources to obtain prediction quality comparable to a competent single machine implementation within the same time-frame.

Finally in Chapter 5 we discussed the operational complexity of implementing and executing such benchmark experiments with distributed data processing systems. With PEEL, we presented a framework to define, execute, analyze, and share experiments. PEEL provides a unified and transparent way of specifying experiments, including the actual application code, systems configuration, and experiment setup description. It reduces the

operational complexity by automatically scheduling and executing the experiments.

With these three aspects we provided all crucial building blocks of a benchmark for assessing how well the systems resting on the paradigm of distributed dataflow perform for the task of scaling out the training of machine learning methods. Our experimental evaluations provided novel insights into the performance of current dataflow systems. The inability to scale up model training to high dimensions as discussed in Section 3.7.1 revealed opportunities for improvement in the memory management of broadcast variables. The comparison to sophisticated single machine libraries in Chapter 4 indicates that providing optimized single machine execution next to the ability to scale out computations on multiple compute nodes is also an important requirement for distributed data processing systems. Future systems research papers should include such evaluations to provide an adequate assessment of the system performance.

In particular they should include scale-out experiments that vary the dimensionality of the input data and thus model vector fur supervised learning workloads such as logistic regression as introduced in Section 2.2. This can be achieved by applying the feature hashing approach we presented in Section 3.5.5 based on the *Criteo* data set and is a crucial addition to "traditional" scalability experiments such as strong or weak scaling.

They should also include experiments that explore the prediction quality vs. training time trade-off and compare the distributed data processing systems against sophisticated single machine libraries for supervised learning methods as we described in Section 4.6. Evaluations of the absolute wall-clock training time needed to achieve comparable prediction quality for the distributed as well as the single machine solutions provide valuable insights into the overhead incurred and thus additional hardware resources necessary to run the training of machine learning models on distributed data processing systems. Scalability experiments alone are not sufficient for this.

The PEEL Framework we introduced in Chapter 5 provides clear abstractions for specifying both proposed experiment types and significantly reduces the operational complexity of carrying out such evaluations. Since this framework as well as all of our benchmark experiments are available as open source software, we hope it will have a positive impact on the distributed systems, database systems and machine learning research communities and beyond. Our work also benefits practitioners and scientists who wish to assess the suitability of distributed dataflow systems to their problem setting.

6.2 Outlook

More recently specialized machine learning systems that can potentially also distribute computations, have been proposed and developed. While the distributed data flow systems that were subject to the experiments we discussed in this thesis are essentially general purpose data processing systems catering to a broad range of workloads, these novel specialized machine learning systems have been purposefully designed for machine learning workloads, in particular the training of "deep" artificial neural networks (DNNs). While *SystemML* [61] targets data-intensive linear algebra operations on matrices, *TensorFlow* [16] and *MXNet* [41] provide a tensor abstractions as central data type and can automatically carry out mathematical operations (e.g., automatic differentiation). Next to efficiently executing computations on a single node, all of these systems can also distribute the execution of computations. While SystemML relies on Spark for the distributed execution, TensorFlow and MXNet handle this task by themselves.

The training of artificial neural networks is almost exclusively carried out via back-propagation and gradient descend algorithms, e.g., mini-batch stochastic gradient descend. The requirements for this are different compared to those of the more general distributed machine learning algorithms popular with distributed data processing systems we discussed in this thesis. Contrary to the latter, where I/O and network communication are the primary bottlenecks, training deep artificial neural networks is generally constrained by computing power. With having both the algorithm (back propagation) and data model (tensors) fixed, systems like TensorFlow, CNTK [112] or MXNet could be built and optimized for this particular use case to a degree that was not possible for general purpose distributed data flow systems. One important reason for the recent popularity and successes of deep neural networks can be seen in the application of GPUs, which provide at least an order magnitude more floating point operations per second while being more power and cost-efficient than a CPU. With that the rather computation-intensive training of artificial neural networks with "deep" architectures, which often translates to solving a non-convex optimization problem, became feasible and given the results achieved in tasks such as object recognition also quite popular.

Due to the importance of deep neural networks and their computation-intensive training, purpose built acceleration hardware, for example Tensor Processing Units (TPUs)

by Google, have been introduced. In order to guide the development of such hardware, representative benchmark experiments are evermore important. Given the variety of modern hardware available to train machine learning models, adequate assessment of end-to-end performance as we introduced in this thesis is crucial.

This need has also been realized by researchers in the DAWN lab at Stanford, who have introduced DAWNBench [45] an End-to-End Deep Learning Benchmark Competition that invites submissions of runtimes for specified tasks. The MLPerf [11] initiative by researchers from Harvard, Stanford, University of California, Berkeley and others aims to do the same for a more broad set of machine learning tasks. Namely to evaluate the performance of ML software frameworks, purpose-built machine learning hardware accelerators and machine learning cloud platforms for the following tasks: Image Classification, Object Identification, Translation, Speech-to-Text, Recommendation, Sentiment Analysis and Reinforcement Learning. These initiatives underline the relevance of our work. However in contrast to the experiments presented in this thesis the authors of both DAWNBench and MLPerf did not run any benchmark experiments themselves, but rather propose data sets and papers of algorithms, for which the community is invited to submit runtime statistics. While this is certainly an important endeavor, relying on runtimes submitted by third parties does not allow the in-depth profiling of resource consumption we presented for example in Chapter 3. The broad support of MLPerf by AMD, Baidu, Google, Intel and others suggests that this topic is not just of academic relevance, but also addresses important questions faced in industry.

There still exist a lot of interesting open questions that need to be addressed in this space. While global competition based approaches like the one proposed as MLPerf [11] can be useful yardsticks to identify "best practices" for certain machine learning tasks, it would also be interesting to explore - on the algorithmic level - which machine learning methods and associated solvers perform particularly well for certain systems. For example, asynchronous stochastic gradient descend (e.g., HOGWILD!) may be a good fit for parameter server architectures, but tends to create significant communication overhead.

One of the major benefits of distributed data flow systems is the ability to robustly scale data intensive transformations such as pre-processing and aggregation of unstructured or semi-structured data sets. While machine learning systems like TensorFlow or MXNet have been heavily optimized for the training of deep neural networks, they are sub-optimal

choices for data pre-processing, a step that is still important even in light of deep learning and likely to be carried out on potentially very large (raw) data sets. In order to gain a decent understanding of the true end-to-end performance of systems for the scalable execution of machine learning pipelines, it would thus be interesting to explore benchmarks that unite pre-processing and training of end-to-end pipelines.

The insights obtained from such benchmarks and the ones presented in this thesis are not just of use to the database and distributed systems researchers but also to practitioners and scientists. Scalable data processing ("Big Data") systems and more recently machine learning systems for training deep artificial neural networks enable researchers from diverse disciplines, e.g., computational social science, digital humanities, bio-medicine or material science to apply machine learning and data analysis methods to their problem domains to gain new insights that otherwise would not have been obtainable. However, given the level of hype and pace of development of new systems and approaches, it is important to assess whether the problem at hand truly necessitates a distributed data flow system like Apache Spark or may be better served with single machine solutions like we proposed in Chapter 4. The benchmark methodologies and experiments developed and presented in this thesis can serve as useful tools for exactly this problem and thus contribute to scientific progress in domains beyond computer science alone.

List of Figures

List of Tables

Listings

Bibliography

[1] https://aloja.bsc.es/ accessed 24/10/2018.

[2] https://awards.acm.org/award_winners/zaharia_7692208 accessed 24/10/2018.

[3] https://aws.amazon.com/ec2/instance-types/ accessed 24/10/2018.

[4] https://azure.microsoft.com/de-de/pricing/details/virtual-machines/linux/ accessed 24/10/2018.

[5] https://cloud.google.com/compute/pricing#processors accessed 24/10/2018.

[6] https://docs.openml.org/benchmark/ accessed 24/10/2018.

[7] https://flink.apache.org/ accessed 24/10/2018.

[8] https://hadoop.apache.org/ accessed 24/10/2018.

[9] https://hive.apache.org/ accessed 24/10/2018.

[10] https://mahout.apache.org/ accessed 24/10/2018.

[11] https://mlperf.org/ accessed 24/10/2018.

[12] https://pig.apache.org/ accessed 24/10/2018.

[13] https://spark.apache.org/ accessed 24/10/2018.

[14] https://www.kaggle.com/surveys/2017 accessed 24/10/2018.

[15] *Combining Pattern Classifiers: Methods and Algorithms*. Wiley Publishing, 2nd edition, 2014.

[16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283. USENIX Association, 2016.

[17] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *J. Mach. Learn. Res.*, 15(1):1111–1133, January 2014.

[18] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6), December 2014.

[19] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[20] C. K. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G. P. Copeland, and W. G. Wilson. Db2 parallel edition. *IBM Systems Journal*, 34(2):292–322, 1995.

[21] Chaitanya Baru, Milind Bhandarkar, Carlo Curino, Manuel Danisch, Michael Frank, Bhaskar Gowda, Hans-Arno Jacobsen, Huang Jie, Dileep Kumar, Raghunath Nambiar, Meikel Poess, Francois Raab, Tilmann Rabl, Nishkam Ravi, Kai Sachs, Saptak Sen, Lan Yi, and Choonhan Youn. Discussion of BigBench: A Proposed Industry Standard Performance Benchmark for Big Data. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking. Traditional to Big Data*, page 44–63, Cham, 2015. Springer International Publishing.

[22] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *Symposium on Cloud Computing*, 2010.

[23] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling Up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, New York, NY, USA, 2011.

[24] Christoph Boden, Andrea Spina, Tilmann Rabl, and Volker Markl. Benchmarking data flow systems for scalable machine learning. In *Proceedings of the 4th Algorithms and Systems on MapReduce and Beyond*, BeyondMR'17, pages 5:1–5:10, New York, NY, USA, 2017. ACM.

[25] Vinayak Borkar, Michael J. Carey, and Chen Li. Inside "big data management": Ogres, onions, or parfaits? In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 3–14, New York, NY, USA, 2012. ACM.

[26] Joos-Hendrik Böse, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, Dustin Lange, David Salinas, Sebastian Schelter, Matthias Seeger, and Yuyang Wang. Probabilistic demand forecasting at scale. *Proc. VLDB Endow.*, 10(12):1694–1705, August 2017.

[27] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, 2007.

[28] Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, Jeffrey Dean, and Google Inc. Large language models in machine translation. In *EMNLP*, pages 858–867, 2007.

[29] Leo Breiman, Jerome Friedman, Charles J. Stone, and R. A. Olshen. *Classification and Regression Trees (Wadsworth Statistics/Probability)*. Chapman and Hall/CRC, 1 edition, January 1984.

[30] Eric A Brewer. Combining systems and databases: A search engine retrospective. *Readings in Database Systems*, 4, 2005.

[31] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998. Proceedings of the Seventh International World Wide Web Conference.

[32] Yingyi Bu, Vinayak R. Borkar, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.

[33] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[34] Zhuhua Cai, Zekai J. Gao, Shangyu Luo, Luis L. Perez, Zografoula Vagena, and Christopher 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.

[35] kevin Caninil. Sibyl: A system for large scale supervised machine learning.

[36] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[37] Rich Caruana, Nikos Karampatziakis, and Ainur Yessenalina. An empirical evaluation of supervised learning in high dimensions. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 96–103, New York, NY, USA, 2008. ACM.

[38] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 161–168, New York, NY, USA, 2006. ACM.

[39] Olivier Chapelle, Eren Manavoglu, and Romer Rosales. Simple and scalable response prediction for display advertising. *ACM Trans. Intell. Syst. Technol.*, 5(4):61:1–61:34, December 2014.

[40] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[41] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[42] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, August 2012.

[43] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *Proceedings of the 19th International Conference on Neural Information Processing Systems*, NIPS'06, pages 281–288, Cambridge, MA, USA, 2006. MIT Press.

[44] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[45] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. *ML Systems Workshop @ NIPS 2017*, 100(101):102, 2017.

[46] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: Scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 271–280, New York, NY, USA, 2007. ACM.

[47] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[48] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[49] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: A flexible data processing tool. *Commun. ACM*, 53(1):72–77, January 2010.

[50] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, June 1992.

[51] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 228–237, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[52] Pedro Domingos. A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87, October 2012.

[53] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[54] Anon et al, Dina Bitton, Mark Brown, Rick Catell, Stefano Ceri, Tim Chou, Dave DeWitt, Dieter Gawlick, Hector Garcia-Molina, Bob Good, Jim Gray, Pete Homan, Bob Jolls, Tony Lukes, Ed Lazowska, John Nauman, Mike Pong, Alfred Spector, Kent Trieber, Harald Sammer, Omri Serlin, Mike Stonebraker, Andreas Reuter, and Peter Weinberger. A measure of transaction processing power. *Datamation*, 31(7):112–118, April 1985.

[55] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *Proc. VLDB Endow.*, 2012.

[56] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. Benchmarking in the Cloud: What It Should, Can, and Cannot Be. In Raghunath Nambiar and Meikel Poess, editors, *Selected Topics in Performance Evaluation and Benchmarking*, page 173–188, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[57] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[58] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. An overview of the system software of a parallel relational database machine grace. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 209–219, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[59] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1197–1208, New York, NY, USA, 2013. ACM.

[60] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[61] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 231–242. IEEE, 2011.

[62] Sharad Goel, Duncan J. Watts, and Daniel G. Goldstein. The structure of online diffusion networks. In *Proceedings of the 13th ACM Conference on Electronic Commerce*, EC '12, pages 623–638, New York, NY, USA, 2012. ACM.

[63] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[64] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2), March.

[65] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, ADKDD'14, pages 5:1–5:9, New York, NY, USA, 2014. ACM.

[66] Mark D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, December 1990.

[67] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. *The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis*, pages 209–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[68] Karl Huppler and Douglas Johnson. TPC Express – A New Path for TPC Benchmarks. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking*, page 48–60, Cham, 2014. Springer International Publishing.

[69] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.

[70] H. V. Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi. Big data and its technical challenges. *Commun. ACM*, 57(7):86–94, July 2014.

[71] Lin Jimmy and Alek Kolcz. Large-scale machine learning at twitter. *SIGMOD 2012*, 2012.

[72] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, page 3146–3154. Curran Associates, Inc., 2017.

[73] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, August 2009.

[74] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pages 583–598, 2014.

[75] Chih-Jen Lin and Jorge J. Moré. Newton's method for large bound-constrained optimization problems. *SIAM J. on Optimization*, 9(4), April 1999.

[76] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.

[77] Jimmy J. Lin. Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail! *CoRR*, abs/1209.2191, 2012.

[78] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, and Feng Sun. Model ensemble for click prediction in bing search ads. In *Proceedings of the 26th International Conference on World Wide Web Companion*, WWW '17 Companion, pages 689–698, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[79] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Math. Program.*, 1989.

[80] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[81] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[82] 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.

[83] Ryan Mcdonald, Mehryar Mohri, Nathan Silberman, Dan Walker, and Gideon S. Mann. Efficient Large-Scale Distributed Training of Conditional Maximum Entropy Models. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, page 1231–1239. Curran Associates, Inc., 2009.

[84] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. Ad click prediction: A view from the trenches. In *KDD '13*. ACM, 2013.

[85] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *USENIX HOTOS'15*. USENIX Association, 2015.

[86] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, January 2016.

[87] Kevin P Murphy. *Machine learning: a probabilistic perspective*. Cambridge, MA, 2012.

[88] Raghunath Nambiar, Meikel Poess, Akon Dey, Paul Cao, Tariq Magdon-Ismail, Da Qi Ren, and Andrew Bond. Introducing TPCx-HS: The First Industry Standard for Benchmarking Big Data Systems. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking. Traditional to Big Data*, page 1–12, Cham, 2015. Springer International Publishing.

[89] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1049–1058. VLDB Endowment, 2006.

[90] Didrik Nielsen. *Tree Boosting With XGBoost - Why Does XGBoost Win "Every" Machine Learning Competition?* NTNU (Masters Thesis), 2016.

[91] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS 2011*, USA.

[92] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[93] NOMAD. https://repository.nomad-coe.eu/ accessed 24/10/2018.

[94] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association.

[95] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[96] Nicolas Poggi, David Carrera, Aaron Call, Sergio Mendoza, Yolanda Becerra, Jordi Torres, Eduard Ayguadé, Fabrizio Gagliardi, Jesús Labarta, Rob Reinauer, et al. Aloja: A systematic study of hadoop deployment variables to enable automated characterization of cost-effectiveness. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 905–913. IEEE, 2014.

[97] G. Rätsch, T. Onoda, and K.-R. Müller. Soft margins for adaboost. *Mach. Learn.*, 42(3):287–320, March 2001.

[98] Matthew Richardson, Ewa Dominowska, and Robert Ragno. Predicting clicks: Estimating the click-through rate for new ads. In *WWW '07*. ACM, 2007.

[99] Sebastian Schelter, Christoph Boden, and Volker Markl. Scalable similarity-based neighborhood methods with mapreduce. In *Proceedings of the Sixth ACM Conference on Recommender Systems*, RecSys '12, pages 163–170, New York, NY, USA, 2012. ACM.

[100] Sebastian Schelter, Christoph Boden, Martin Schenck, Alexander Alexandrov, and Volker Markl. Distributed matrix factorization with mapreduce using a series of broadcast-joins. *ACM RecSys 2013*, 2013.

[101] Sebastian Schelter, Venu Satuluri, and Reza Zadeh. Factorbird - a Parameter Server Approach to Distributed Matrix Factorization. *Distributed Machine Learning and Matrix Computations workshop at NIPS 2014*, 2014.

[102] Bertil Schmidt and Andreas Hildebrandt. Next-generation sequencing: big data meets high performance computing. *Drug Discovery Today*, 22(4):712–717, 2017.

[103] J. Shemer and P. Neches. The genesis of a database computer. *Computer*, 17(11):42–56, November 1984.

[104] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow.*, 8(13), September 2015.

[105] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[106] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: Friends or foes? *Commun. ACM*, 53(1):64–71, January 2010.

[107] Sloan Digital Sky Survey. http://www.sdss.org/ accessed 24/10/2018.

[108] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. How to build a benchmark. In *Proceedings of the 6th*

ACM/SPEC International Conference on Performance Engineering, ICPE '15, pages 333–336, New York, NY, USA, 2015. ACM.

[109] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[110] 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.

[111] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1113–1120, New York, NY, USA, 2009. ACM.

[112] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014–112*, 2014.

[113] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI'12, 2012.