Evaluating Parallel Data Processing Systems for Scalable Feature Selection

Bachelorarbeit

von

André Hacker

zur Erlangung des akademischen Grades
Bachelor of Science
im Studiengang Informatik

Prüfer:
Prof. Dr.-Ing. Robert Tolksdorf (FU Berlin)
Prof. Dr. rer. nat. Volker Markl (TU Berlin)

Betreuer:
MSc. Alexander Alexandrov (TU Berlin)
Dipl.-Ing. Christoph Boden (TU Berlin)

Bearbeitungszeitraum: 14.08.2013 – 06.11.2013

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben. Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommision vorgelegt und auch nicht veröffentlicht.

Datum, Unterschrift

Contents

Abstract

Parallel Data Processing Systems such as Apache Hadoop and Stratosphere are designed to run complex analysis tasks on data of any type and scale. While Hadoop and the MapReduce programming model are criticised for lacking expressiveness and being inefficient, Stratosphere claims to overcome this with a more expressive programming model and automatic optimizations. However, no independent comparisons of Hadoop and Stratosphere have been made so far, and for some fields such as predictive analytics only few algorithms have been implemented in Stratosphere. This thesis evaluates Stratosphere in comparison to Hadoop for a scalable feature selection algorithm (SFS). The objective of SFS is to evaluate a large number of features regarding their usefulness for predictive analytics. First, we contribute an implementation of SFS for Stratosphere that will be compared to an existing Hadoop implementation. Second, we evaluate both programming models based on an ideal programming model. Third, we conduct a series of experiments to compare the performance. The evaluation emphasizes the potential of Stratosphere but also reveals several limitations, most notably the limited support for closures and the finding that expressiveness does often not imply better performance.

Zusammenfassung

Parallele Verarbeitungssysteme wie Apache Hadoop und Stratosphere ermöglichen komplexe Analysen von Daten beliebigen Typs und Ausmaßes. Während Hadoop und das dazugehörige MapReduce Programmiermodell dafür kritisiert werden einen Mangel an Ausdrucksfähigkeit und Effizienz zu haben, behauptet Stratosphere diese Probleme durch ein ausdrucksstärkeres Programmiermodell und automatische Optimierungen zu überwinden. Es wurde bisher jedoch kein unabhängiger Vergleich der beiden Systeme durchgeführt, und für einige Bereiche, wie z.B. Vorhersageanalysen, wurden kaum Algorithmen in Stratosphere implementiert. Diese Arbeit evaluiert Stratosphere im Vergleich zu Hadoop für einen Algorithmus zur skalierbaren Merkmalsauswahl (SFS). Das Ziel von SFS ist es eine Vielzahl an Merkmalen im Hinblick auf ihren Nutzen für eine Vorhersageanalyse zu evaluieren. Als ersten Beitrag implementieren wir SFS in Stratosphere, für den späteren Vergleich mit einer bestehenden Hadoop Implementierung. Zweitens evaluieren wir die Programmiermodelle basierend auf einem idealen Programmiermodel. Drittens führen wir eine Reihe von Tests für die Performance-Analyse durch. Unsere Evaluierung unterstreicht das Potential von Stratosphere, deckt aber auch Schwächen auf, insbesondere die eingeschränkte Unterstützung von Closures und die Erkenntnis, dass Ausdrucksstärke oft nicht in besserer Performance resultiert.

1 Introduction and Motivation

Producing large amounts of data has never been easier. And increasingly often this data is analysed and used to drive decisions. The sensors embedded in mobile devices, which have become constant companions in our daily lives, produce enormous amounts of audio, video, image, and text data. Software is increasingly often Internet based and fosters social interaction, yielding an immense number of data to be handled. Behind the scenes, companies process and analyse these data to make services such as search, social based recommendations and fraud detection possible, but also to improve their efficiency and competitiveness. The explosion of data, however, is not limited to the Internet. Nearly all fields of science draw conclusions from ever larger amounts of data. Also other industry sectors such as health care or traffic and energy management begin to make sense of this new amount and variety of data, which is captured by the term "Big Data" [31].

Parallel Data Processing Systems. These trends produced a strong need for new cost-efficient technologies capable of storing and analysing huge datasets with support for arbitrary data types and complex analysis tasks. The family of Parallel Data Processing Systems emerged to address this need, with systems such as Apache Hadoop [1] and Stratosphere [4]. They borrow the principles from parallel databases to exploit parallelism in clusters of commodity hardware computers and offer programming models for custom data analysis tasks. MapReduce [16], the programming model underlying Hadoop, was criticized for lacking expressiveness and being a bad choice for general purpose analysis tasks since many tasks do not map naturally to MapReduce or have poor performance [34, 2]. Stratosphere is one of the systems addressing this by generalizing MapReduce to a more expressive programming model, called PACT, adding support for complex data flows, iterations, and joins. In this thesis we evaluate Stratosphere in comparison to Hadoop for the task of feature selection, a use case from machine learning that will be introduced next.

Feature Selection. There is an increasing interest in predictive analytics [31], a field from machine learning and statistics aiming to learn from data to forecast or to make predictions about unseen events. For instance we could predict the category of a news article using the words of the document as features. As a more interesting example, an massive open online course system might want to predict how many of the students of a currently running course will pass the exam, based on the homework questions they answered so far and based on the data collected from the last time the course was held. Features capture all we know about the input, for example occurrences of words or interactions with an online learning system, and the design and selection of relevant features is essential for good predictions. Feature selection has also the benefit that it gives insights, since it answers questions like the following:

"Which words, or combination of words, are the best indicators that an article belongs to a certain category?"

"What are the homework questions a student has to answer correctly to perform good in the exam, i.e. which subjects are important for the learning process?"

While feature selection is a well studied field [19], it is an ongoing effort in research and industry to rewrite algorithms to support the parallelism inherent in systems such as Hadoop or Stratosphere. Singh et al. proposed a scalable feature selection algorithm based on approximate models and showed how to parallelize it using MapReduce [38]. This algorithm, hereinafter referred to as Scalable Feature Selection (SFS), serves as the use case in this thesis. This use case is of particular interest for Stratosphere as it is the first use case from the area of predictive analytics implemented and evaluated.

Goals. Stratosphere claims that the improvements over Hadoop and MapReduce such as the more expressive programming model significantly ease the implementation of many complex data processing tasks and yield better performance [2]. This thesis aims to test this claim for the use case of SFS, since no independent comparisons have been made so far for Stratosphere[1]. For this task we will implement SFS for Stratosphere and compare it to an existing implementation in Hadoop. The evaluation and comparison will be done in two dimensions. First, we will evaluate the programming models qualitatively to see how well SFS can be expressed in each. Second, we will run a series of experiments on a cluster for both systems to evaluate the performance of both systems. The performance will be measured in terms of speedup and scaleout behaviour: we test how the runtime varies when changing the cluster size, the problem size or both. The methodology for evaluation of both dimensions and the definitions for speedup and scaleout will be developed in the following chapters.

Structure of this thesis. The following chapter introduces the background necessary for this work. We introduce the fundamentals underlying Parallel Data Processing Systems such as types of parallelism, scaleout and speedup and introduce Hadoop and Stratosphere. The second part of the Chapter 2 introduces the SFS algorithm and required background including the basic machine learning terminology, feature selection and logistic regression. We close the background chapter with a short survey of related work. Chapter 3 presents the qualitative part of the evaluation by comparing the programming models for the two implementations of SFS. Chapter 4 presents the quantitative evaluation of both systems, reporting the results of a series of experiments. At the end of Chapter 4 we shortly evaluate our SFS implementation. The conclusion in Chapter 5 summarizes our findings.

[1] We are not aware of any comparisons or experiments for Stratosphere other than those from the authors of the system. The author of this thesis joined the Stratosphere group recently but started to work with Stratosphere only shortly before the thesis and tried to be as independent as possible.

2 Background & Related Work

This thesis is about the marriage of two very complementary fields: Parallel Data Processing and machine learning, more specifically feature selection. In this chapter we introduce both topics independently from each other. First, we discuss the fundamentals of Parallel Data Processing Systems and introduce Hadoop and Stratosphere. Afterwards we introduce the basic machine learning terminology and the SFS use case. Since the focus of our work is on the system evaluation side, we keep the machine learning part more compact. For some involved topics such as mathematical optimization we can only explain the motivation and the parts that are relevant for our algorithm. For both fields we will give a short survey of related work at the end of this Chapter.

2.1 Parallel Data Processing Systems

In this section we will describe Parallel Data Processing Systems[2], a family of systems that arose in the mid 2000s to address the emerging requirements of Big Data. We will start with an informal definition of Big Data and see that it is more than just big in terms of size in bytes. Afterwards we describe how Parallel Data Processing Systems solve the requirements for Big Data, explaining the different forms of parallelism, the limits and other fundamental concepts used by all systems. Then we will describe the systems we compare in this thesis, Hadoop and Stratosphere, in more detail.

Big Data - more than just "big" data. The challenge to process large amounts of data exists since the early days of computing, it is just the definition of large that is constantly, and rapidly, changing. An often cited[3] definition of Gartner, also known as the "three Vs", describes Big Data in a broader sense, as a variety of new requirements and challenges arising in our current information landscape:

> „Big Data is high-volume, high-velocity and high-variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making." [6]

Our definition is mostly in line with this definition of Gartner that was first coined in 2001 and updated in 2012 as cited above. The following list gives an overview of the requirements for Parallel Data Processing Systems resulting from our understanding of Big Data.

[2]There is no unique name or definition for this family of systems. Other frequently used names include Large Scale Data Processing Systems [37], Big Data Analytics Systems, Big Data Platforms or simply Cluster Computing Systems [45].

[3]Ward and Barker recently published a survey of definitions for Big Data in [41] stating that the three Vs of Gartner are among the most cited definitions.

- **High volume and high scalability**: The system shall enable storage and processing of large scale data. It shall be easy to scale by adding machines, and tolerate a certain amount of failure without loosing data.
- **High variety**: The system shall support arbitrary data types, ranging from structured data to less structured data such as text, JSON, logfiles, graph data or even binary data such as images or movies.
- **High velocity**: The system shall support a high rate of incoming data, e.g. from data streams, but also allow quick or near real-time analyses to take fast decisions based on new data. For long this was a less emphasized point for Parallel Data Processing Systems since they were originally designed for long-running batch queries.
- **"Big" Analytics**: The system shall enable custom and complex analytical analysis tasks, ranging from SQL-like operations to advanced machine learning and data mining tasks. Ideally the system offers direct support to execute established languages for data analyses such as R [35] or Apache Pig [33] and SQL.
- **Efficiency and new hardware architectures**: The system shall make use of all available hardware resources efficiently, including new hardware architectures. This comprises increasing number of cores, heterogeneous hardware environments, e.g. with graphical processing units, large main memory and solid state disks. This efficiency becomes substantial for some systems if they want to overcome the critique of being inefficient brute-force approaches [34].

While this list gives a good impression of the wide variety of requirements, the discussion and evaluation of all these dimensions is beyond the scope of this thesis. Instead we take the approach to start with a specific use case and evaluate the system based on this.

There is a ongoing controversy about the high volume aspect of Big Data that is relevant to discuss shortly. First of all there is a political controversy about the potential miss-use of Big Data technologies as "Big Brother" technologies[4]. Second, there is a controversy about whether the assumption is true that large scale batch-workloads are the predominant use case or how much the systems should also support other types of workloads. The inventors of many Big Data technologies, Google, argues for a data-driven approach and describes that „invariably, simple models and a lot of data trump more elaborate models based on less data" [20] and other companies follow this approach [30]. On the other hand, an empirical analysis of Hadoop work-

[4]This is a important debate in the opinion of the author, considering a developer has a responsibility for the software he develops and the special historical background in Germany. The actual discussion, however, is out of scope of this thesis.

loads from various companies revealed that 90% of jobs accesses files of less than a few GBs [44]. They and others [36] argue that many smaller interactive ad-hoc queries are equally important and should be supported in addition to long-running batch queries. For this work we will stay with a broader understanding of Big Data, including the need to efficiently handle relatively small datasets.

2.1.1 Fundamental Concepts

This section introduces the fundamental concepts underlying Parallel Data Processing Systems: Parallelism, speedup, and scalability. It also discusses the laws and the limits of these. In the course of this introduction we will identify many requirements and desirable properties for systems and algorithms and we will use these requirements in our evaluation. Furthermore, explaining these fundamentals in detail enables a more intuitive understanding of the systems.

The most important concept is parallelism. We can distinguish at least 3 forms of parallelism [12] and will see that most systems actually use a mix of those three.

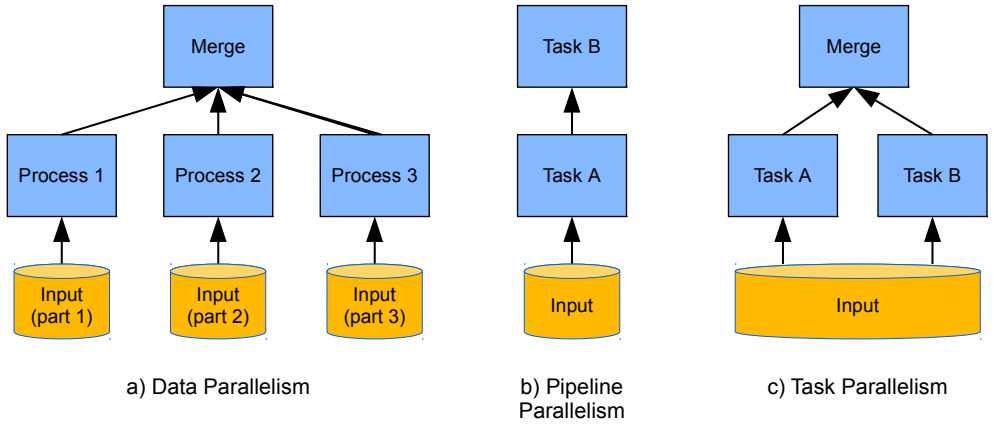

Figure 1: Visualization of the three types of parallelism, inspired by [12].

Data Parallelism comprises to split the data into multiple parts (partitions) and run the same task in parallel and independently on the different partitions of data.

Pipeline Parallelism exists if the output of one task, e.g. a single line of text, is directly forwarded to and processed by a subsequent task without waiting for the first task to compute the complete output. One can think of this as a stream of data flowing through a pipeline of concurrently running tasks, which is also a form of task parallelism.

Task Parallelism comprises to divide different tasks among different parallel computing nodes or processes. A big cluster for instance executes many different workloads

at the same time and distributes the various tasks among the nodes. Task parallelism is also possible on a lower level, if a workload consists of multiple tasks and no dependencies exist between them.

We will use the abbreviation *dop* to describe the degree of parallelism, i.e. how many instances of a task work in parallel. In our context dop equals to the number of nodes in a cluster unless otherwise specified. Dop could however also describe the number of tasks, processes or threads executed in parallel on a single machine, and in this case we call it *intra-node dop*.

Let us look closer on how data parallelism can be achieved in our context. Parallel Data Processing systems rely on a „Shared Nothing“ architecture [39] where multiple independent machines are interconnected by a high speed local area network and every node usually stores only a part of the overall data. In such an environment the best way to achieve data parallelism is to move the computation to the data. An ideal example is to compute the global sum of numeric records: Most of the work can be done locally, i.e. so called *data locality* is given, where each machine computes a partial sum of his records. Each machine just sends a single number over the network to a single machine computing the final sum. Let us contrast this with a more realistic example: If we want to make a SQL-Join of two very large tables that are randomly distributed among the nodes we have to sent nearly all data over the network to do a so called *repartitioning*: Every key of the join-key is mapped to a single machine and all records with this key are sent to this machine. This way the data are *partitioned* by the join-key and the system can do the joining completely locally. The notion of partitioning is very important because if there is a partitioning but the system does not know about it, it would repartition the data again.

The applicability of the different types of parallelism highly depends on the problem and the availability of parallel algorithms for it, but the system has an equally important responsibility. First it should optimize the data flow in a way that maximizes data locality and minimizes the costs, especially for network transfer. To give an example, if we want to compute a cross product between one very large and one very small input in a distributed way, the system should decide to broadcast the small input to all nodes. Furthermore it can also store the data in a way that is optimized for the typical use cases, e.g. by using columnar storage for aggregations and indices. A system can increase its degree of freedom by exposing a declarative and expressive programming model, where the developer states what he wants to see or what parts can be computed in parallel and the system cares about how to execute it. SQL is such a declarative language and its success gives evidence that declarativity is a desirable property. These considerations will be important in our subsequent evaluation of the systems.

Speedup and Scalability. The main motivation behind the usage of parallelism is speedup and scalability which will be introduced next.

Speedup describes the ability to solve a given problem with fixed input size faster, in our case by adding more computing nodes and using any form of parallelism.

(Horizontal) scalability describes the ability to efficiently solve problems growing in size by adding more nodes. In this thesis we use scalability as a synonym for horizontal scalability (scaleout). It contrasts to vertical scaling (scaleup), where we would not add more nodes but instead use more powerful hardware. We found that scaleout and scaleup are complementary since powerful hardware such as large memory, multi-core and multiple disks allow us to increase the intra-node dop in our experiments which leads to significant performance improvements.

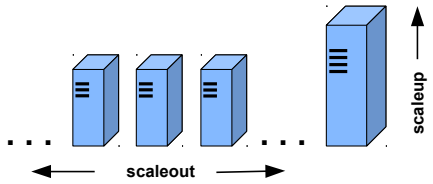

Figure 2: Scaleout and scaleup as complementary concepts.

We use the following two metrics to express both desired properties[5].

$$Speedup = \frac{\text{elapsed time using a small system}}{\text{elapsed time using a big system}}$$

$$Scaleout = horizontal\ scalability = \frac{\text{elapsed time small system small problem}}{\text{elapsed time big system big problem}}$$

The goal is to get as close as possible to *linear speedup* and *linear scaleout*. Linear speedup means that a problem of fixed size can be solved twice as fast by doubling the computing resources. Linear scaleout means that a problem doubled in size can be solved in the same time by using twice as many computing resources.

Amdahl's law. How close can we get to linear speedup and scalability? As early as in 1967 Gene Amdahl described the maximum possible speedup, now known as *Amdahl's law*. It assumes a problem of constant size and that there is a fraction p of the runtime that can be executed in parallel (with linear speedup or worse) and a serial fraction s of the program that cannot be parallelized and thus limits the speedup. It is important to note that s and p are defined as fractions, or relative time, and so $s+p=1$ and $s=1-p$. Under these assumptions the maximum speedup is defined as:

[5]We use the terminology as in [29]. In the past, the term scaleup was used for what we call scaleout, e.g. by DeWitt and Grey [12].

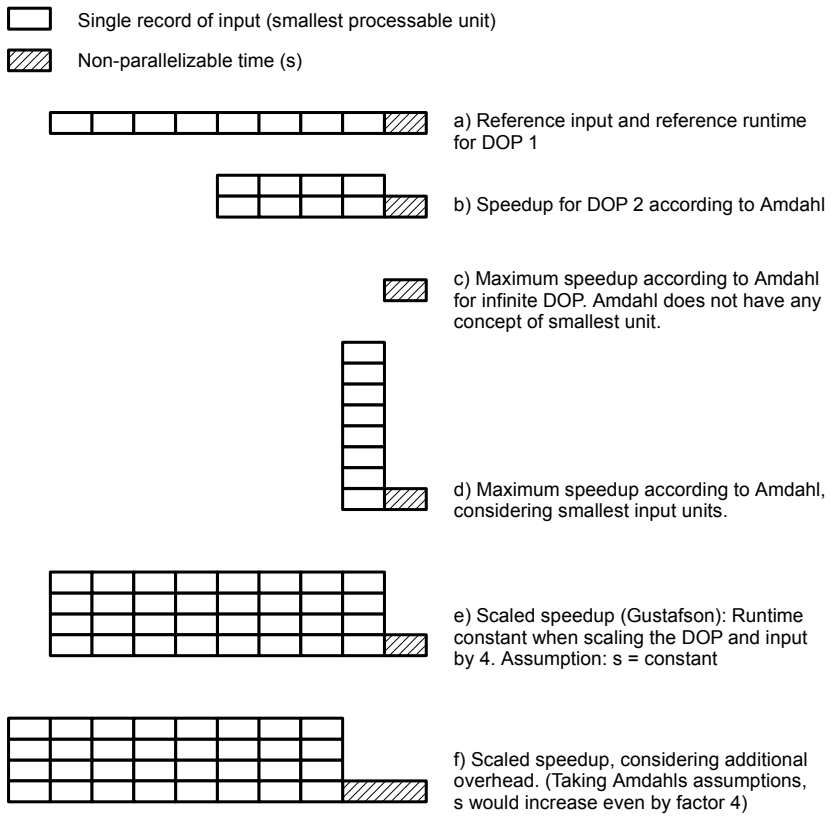

Figure 3: Visualization of different aspects of parallelism. Starting with a reference problem and runtime we analyse the speedup and scaleout behaviour using the assumptions of Amdahl's and Gustafson's law. The horizontal axis denotes the total runtime, the vertical axis the degree of parallelism.

$$S(N)_{s,p} = \frac{\text{time without parallelism}}{\text{time with parallelism N}} = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{p}{N}}.$$

Assuming unlimited computing resources we can compute the parallel part in zero time and receive the maximum possible speedup for a given problem:

$$\lim_{N\to\infty} S(N)_{s,p} = \frac{1}{s+0} = \frac{1}{s}$$

Reevaluating Amdahl Let us assume a problem with 10% serial time, according to Amdahl's assumptions, which might seam realistic for some real world problems at a first glance. Amdahl's law proofs that even large industry clusters with tens of thousands nodes could solve such problem only 10 times faster than a single computer. In 1988 Gustafson argued [18] that Amdahl's law incorrectly implies that only

very few algorithms with very low s can experience a high speedup. He found that the assumption that the absolute serial time s_{absolut} is linearly growing with bigger problems is "virtually never the case". Neither it is the case that we want to solve a problem of fixed size faster, but more often we want to solve the largest possible problem in a defined time. He changed the assumptions: Given a problem that takes s serial and p parallel time to be executed on a parallel system with a given dop N, where $s+p$ is again normed to 1. Then the time for the execution on a single system, which has N times less parallelism, is defined to be $s + (N \cdot p)$. This is equivalent to saying that s is constant and p can experiences linear speedup. He called the new metric *scaled speedup* which is better known as Gustafson's law:

$$S(N)_{s_{\text{absolut}},p} = \frac{\text{time without parallelism}}{\text{time with dop N}} = \frac{s + (p \cdot N)}{\underbrace{s+p}_{=1}} = s + (p \cdot N)$$

We can see that the serial fraction is getting smaller and smaller the bigger the problem becomes - and linear speedup is finally feasible. Gustafson describes his law to be closely related to the scaleout behaviour and that this is what usually matters more than the speedup: We add more nodes and want to solve bigger problems in the same ammount of time. The algebraic differences of the two laws can however be reduced to the different definitions of s and p[6].

Figure 3 illustrates the speedup and scaleout behaviour according to Gustafson's and Amdahl's law using a simplified problem consisting of 8 records that can not be further divided and an additional serial fraction.

Now as we defined scalability and speedup, we can use them to evaluate 1) the systems, here Stratosphere and Hadoop, and 2) the algorithm, here SFS.

Scalable algorithms. A *scalable algorithm* has linear or better asymptotic runtime, i.e. $O(n)$, and behaves as assumed in Gustafson's law, with a small serial runtime that does not change with the dop. Lin and Dyer [29] describe such ideal algorithm as follows: Doubling the size of the input at most doubles the runtime and doubling the dop, the algorithm runs in half the time or better. Unfortunately for many real world problems there are no known algorithms exposing such ideal behaviour, since the coordination and communication effort usually grows with parallelism and most algorithms have a non-parallelizable part.

Scalable systems. A *scalable system* is a system giving the following guaranty:

"If you express your scalable algorithm in my programming model, it is guaranteed that you get overall linear scalability and speedup."

[6]Literature sometimes calls both formulas "speedup" without distinguishing the different meanings of s and thereby ignores the different assumptions.

This informal definition highlights that the developer must find a parallel algorithm for his problem, using the specific Programming Model exposed by the system. We conclude that a system can be evaluated in two dimensions:

1. How well the Programming Model fits the problems it is designed for.
2. How well the system ensures scalability and speedup, for programs written in the programming model.

The task of this thesis is to answer these two questions for Stratosphere in comparison to Hadoop for SFS. For both questions we have individual Chapters 3 and 4. We intentionally stay with this simplistic view. Additional relevant aspects will come on the stage when we discuss the concrete systems in the next two sections.

2.1.2 Hadoop & MapReduce

Apache describes their Hadoop project as „open-source software for reliable, scalable, distributed computing“ [1]. Indeed one could say that it evolved to a Data Operating System[7] and stands for many things: the distributed file system HDFS, the programming model MapReduce, a huge ecosystem for Big Data and a generic framework for job scheduling and resource management on a cluster called YARN. Hadoop is currently in a transition from version 1.0 to version 2.0 with a major change in the architecture. We focus on Hadoop 2.0 and will now introduce Hadoop bottom up.

HDFS. The Hadoop Distributed Filesystem (HDFS) is a distributed filesystem inspired by the Google File System [16]. It offers a hierarchical folder-based API and stores its file in a shared nothing cluster using commodity hardware. Each file copied to the HDFS is divided into blocks, the default block size being 128MB, and each block is stored on a few nodes. The replication factor, typically 3, defines to how many nodes it is sent - a high value increases fault tolerance and sometimes data locality but slows down data loading and requires more space. HDFS offers an API for higher level systems to see where blocks are stored, allowing them to move the computation to the data.

YARN. In Hadoop 2.0, HDFS and YARN[8] can be seen as the two supporting pillars, as illustrated in Figure 4. YARN is the service owning all resources (nodes) in a cluster and overseas and manages all applications running on the cluster. YARN consists of a central ResourceManager and a NodeManager for every node. Applications written

[7]This analogy is used multiple times in recent articles, e.g. http://de.slideshare.net/awadallah/introducing-apache-hadoop-the-modern-data-operating-system-stanford-ee380, accessed 2013, Okt. 16th

[8]YARN is used as a synonym for Hadoop 2.0 but also for the resource management component, which is a part of Hadoop 2.0. It should be clear from context what we refer to.

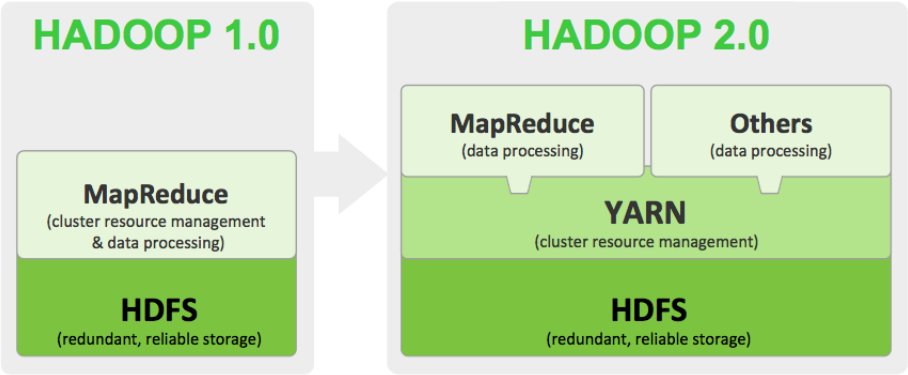

Figure 4: Hadoop 1.0 versus Hadoop 2.0, taken from http://hortonworks.com/hadoop/yarn/

for YARN themselves have their own master and worker tasks, but YARN will receive and schedule the job submissions and grant the resources.

MapReduce. Hadoop MapReduce is a programming model inspired by Googles MapReduce [11] and a runtime for parallel execution of programs written in MapReduce. When Hadoop was developed in 2005 at Yahoo it consisted of HDFS and MapReduce and both were tightly coupled. Today, MapReduce is the most important out of a growing number of YARN applications[9].

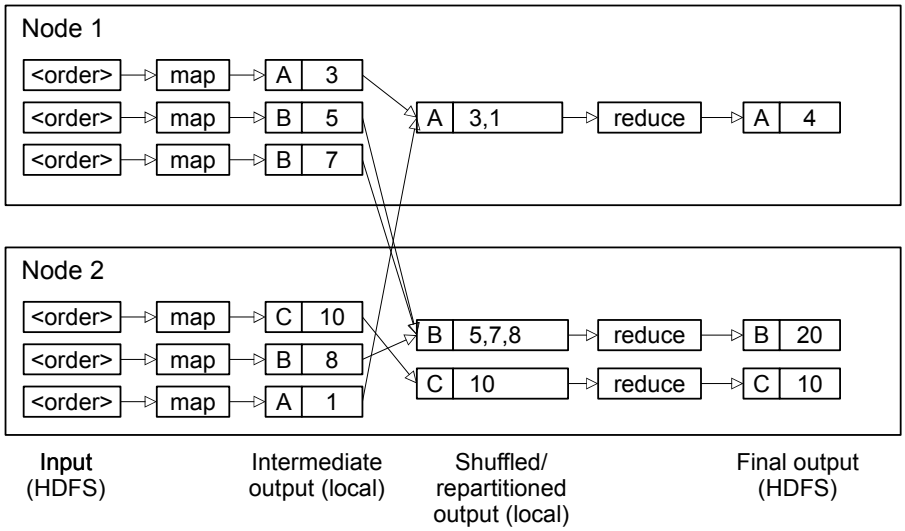

Figure 5: A simplified MapReduce example focusing on the semantics of Map and Reduce and on the data flow. The job computes the total revenue for each customer (A, B and C) based on a collection of orders stored in HDFS in any serializable format.

The programming model consists of two second-order functions Map and Reduce with

[9]See http://wiki.apache.org/hadoop/PoweredByYarn for a list of Yarn applications.

an optional Combine method in between. To write a program in MapReduce one has to write custom implementations (first-order functions) for Map and Reduce. These implementations are called user defined functions (UDF). We illustrate the semantics of Map and Reduce in Figure 5, showing the example of computing the total revenue for a group of customers. The Map UDF gets called for every input unit. It processes the input and emits any number of key-value pairs as the output. In our case the UDF receives a single order object, extracts the customer-id and revenue and emits the customer-id as key and revenue as value. The system then repartitions by the key: It usually applies a hash function on the key to determine where to send the output and transmits the output over network if necessary. After the last map finished, the system calls the reduce function for every key and passes an iterator over all values of the map phase output having this key. Hadoop MapReduce gives the additional guarantee that the Reduce calls are made in sort order of the key on every node, so that B will be called before C. This is because it uses sorting as an internal strategy to group by the key. The output of the Reduce function is the final output and stored on HDFS.

As we saw previously a central goal of shared nothing systems is to reduce the network traffic. To reduce the traffic in the shuffle phase between Map and Reduce, an optional Combiner UDF can be implemented that allows to pre-aggregate the results on every machine. In our case the Combiner is identical to the Reduce method and would emit a single record for B with value 12 on the first node.

The flow of a MapReduce program is static: Map, combine, reduce and materialize the output to HDFS. To map more realistic or iterative algorithms to MapReduce one has to chain multiple programs together or use additional constructs such as the Distributed Cache. The Distributed Cache is designed to broadcast small files to the local disks of all nodes to make them available in the UDF calls.

2.1.3 Stratosphere & PACT

„Everything flows - Panta rhei“, Heraklit

This old wisdom describes best the feeling when using Stratosphere for the first time and comparing it to Hadoop. From a user perspective, the biggest contrast to MapReduce is that operators such as Map, Reduce and others can be chained together in a directed acyclic graph (DAG) so that even more complex algorithms can be expressed in a single data flow.

Stratosphere is an open source software stack consisting of a parallel execution engine called Nephele and a programming model called PACT [4] [10]. We explain the

[10]The system additionally includes higher level interfaces that are not used in this theses and not

Stratosphere top-down.

PACT is a programming model based on so called Parallelization Contracts (PACTs)[11]. PACTs are second-order functions, and Stratosphere currently supports five of them: Map, Reduce, CoGroup, Match and Cross. Each PACT has a semantic of how many instances of the UDF will be started and what part of the input will be passed to those instances. Figure 6 summarizes the semantics for all PACTs and we shortly define them informally. We have to mention beforehand that PACT uses a record data model and the input and output of any PACT is always zero or more records. A record is comparable to a database row containing objects from any serializable type. PACT records, however, do not have a schema and fields can only be accessed by their index.

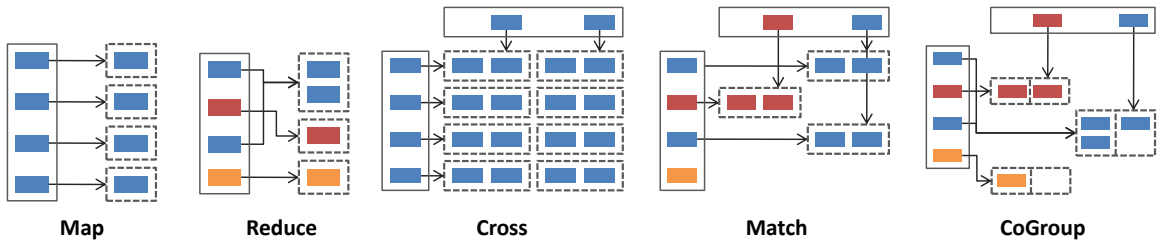

Figure 6: Semantics of the currently available PACTs. Each solid box denotes an input with multiple records. Each dashed box denotes an independent subset. The UDF gets called once for every independent subset. The color denotes the value of the key, except for Map and Cross which have no notion of a key. The graphic were extracted and adopted from slides created by Stephan Ewen from TU Berlin.

Map and *Reduce* have the same semantics as in Hadoop. For Reduce, the user has to specify the index of the record which holds the key to group by. Similar is true for other contracts that use a key.

CoGroup behaves like a Reduce for multiple inputs. For every group of records with the same key one UDF instances gets called. The UDF actually receives two iterators with the records from the first and the second input.

Cross realizes the cross product of two inputs: Every possible pair of records from different inputs forms the input for the UDF. In fact each side of the Cross can be a list of inputs.

Match realizes an Inner Join for two inputs, as known from SQL: From all record-tuples in the cross product of both inputs, those that share the same key form the input. For each of these inputs one UDF instance gets called.

explained for size constraints. We would have liked to develop SFS in the Scala interface but it was still under development.

[11]PACT can mean two things: The whole programming model and a concrete Parallelization Contract such as Map. It should be clear from the context what we refer to.

Up to now we looked at PACTs in terms of their input semantics, called *input contracts*. Additionally any PACT can be manually annotated to inform the system about special properties holding for the output. Without these so called *output contracts*, a UDF is just a blackbox and the system cannot make any assumptions about the partitioning of the output. Currently `ConstantFields(int[] indices)` is supported, stating that every output record has the same values as the input for the specified positions. In the case where a UDF does not change the key of a partitioned input, the partitioning will be retained for the output. As explained in the fundamentals, it is important for the system to know about this.

PACTs can be assembled to a DAG with the vertices being the PACTs and their related UDF and the edges being the connections defining the data flow. A plan has multiple input source vertices, e.g. a HDFS file, and one or more data sinks vertices, usually writing to HDFS. Let us look into how a PACT plan comes to execution.

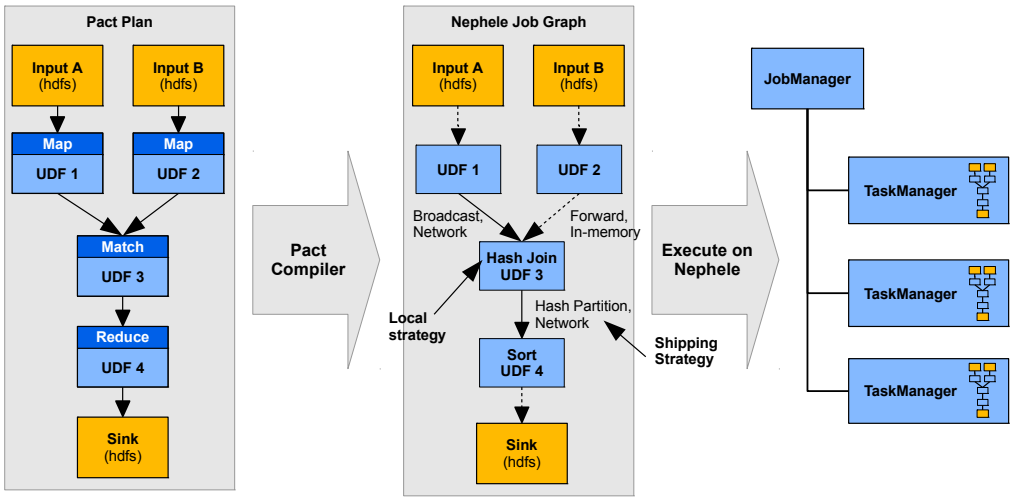

Figure 7: From PACT plan to execution, inspired by [4].

PACT Compiler and Nephele. The so called *PACT Compiler* transforms a PACT plan into a *Nephele Job Graph*, which will be executed on the parallel execution engine called *Nephele* [42]. A Nephele Job Graph is also a DAG and can be seen as an optimized version of the PACT plan which additionally specifies the strategies to fulfil the semantics of the PACT plan.

The central component of the PACT Compiler is the *optimizer*. The optimizer lives from the declarative nature of PACT: Each PACT declares the desired behaviour but not how it is achieved. Similar to a database optimizer it tries to find the execution plan that minimizes the costs, primarily for network communication and secondarily for disk I/O. The computation of the costs is non trivial since the UDFs can emit any number of arbitrary records. We will discuss this during the experimental evaluation

and see that Stratosphere often fails to compute the costs and thus makes unfortunate decisions for SFS.

A prominent example to illustrate the different strategies is a Join (Match contract) as known from SQL, where a big variety of strategies known from parallel database systems can be applied, including sort-merge join or hash-based join. In our example in Figure 7 the optimizer choose to broadcast input A to all nodes over the network, probably because it is assumed to be small, and executes a hash-based join locally. Another strategy is to repartition both inputs by the join key, resulting in far more network traffic if input B is big. For complex PACT plans there can be a big search space of possible execution plans and the optimizer has a high degree of freedom. We will see, however, that for other plans such as the SFS algorithm, the degree of freedom can be very small. The complete discussion of the different execution strategies would exceed the space and we refer to Battré et al. [4] for a detailed discussion.

As the final step of execution, Nephele *spans* the Job Graph over the nodes in the cluster and executes the job. Nephele has a single master process, called JobTracker, and one or many TaskTracker processes, where the actual execution of the job takes place. The JobTracker has responsibilities to manage resources, handle failures, schedule jobs but also to overlook the execution of each job.

The degree of freedom for Nephele is to choose the dop and to choose for each task a node to execute it. Commonly, however, the dop is manually specified by the developer in the PACT plan, leaving a very small or even zero degree of freedom for Nephele.

Our introduction highlights that the PACT programming model and the PACT Compiler can be seen as the heart of Stratosphere for two reasons: First, because PACT yields additional degrees of freedom and the PACT Compiler consumes them to apply automatic optimizations and to improve performance. Second, because the added expressiveness shall enable a more natural fit for many problems and thus better user experience. These two points, visualized in Figure 8, are main distinctions from Hadoop MapReduce. The upper part will be evaluated during our programming model evaluation, the lower part during our experiments.

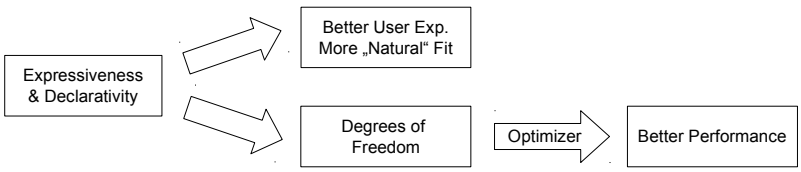

Figure 8: Desired implications of extended expressiveness and declarativity.

2.2 Machine Learning and Feature Selection

2.2.1 Machine Learning Basics

The fundamentals for Machine Learning are well established and the information in this chapter are taken out of standard machine learning text books from Hastie et al. [21] and Bishop [7].

We will use the example of text categorization to introduce Machine Learning. Let us assume we have a collection of text documents describing news articles and every article was manually annotated to belong to the category „Economics“ or not. In machine learning terminology this categorization is called the *class*, *label* or *target variable* and can be described mathematically as a vector $y_i \in \{0, 1\}$. As an example, $y_1 = 0$ denotes that the first document is not about economics. Let us now formalize the input, i.e. the documents: A single document can be described as a vector $x_i \in \mathbb{R}^D$ where D is the size of the dictionary containing all distinct words from all documents. As an example, the vector

$$x_1 = (0, \ldots, 0, \underbrace{1}_{\text{index 1000}}, 0, \ldots, 0, \underbrace{1}_{\text{index 2000}}, 0, \ldots, 0)^T$$

describes the first document that solely consists of the two words „I am“, assuming that „I“ has the index 2000 and „am“ the index 1000 in our dictionary. Each document is interpreted as a so called *bag of words*, where the ordering of the words is ignored and the representation as a vector is done according to the *vector space model*. More generally spoken, each dimension of the vector is called *feature* or *independent variable*, which means that features capture all we know about the input. Text is a typical example of a high dimensional feature space and if we look at the occurrence of combinations of words, so called *n-grams*, the feature space can grow into millions. For example when using 3-grams, all possible combinations of three words form a feature. Such high dimensional scenarios will be of particular interest in this thesis.

Let us generalize an input of size N to a Matrix X with the rows being the individual input vectors $x_i \in \mathcal{X}$. And let us formalize the output to a vector $y = (y_1, y_2, \ldots, y_N)^T$ with $y_i \in \mathcal{Y}$. x_i and y are commonly defined as a column vectors and we omit the vector arrow for simplicity. This input is referred to as the training dataset.

Classification. Now that we defined the input and the labels in a numeric way we can apply any machine learning algorithm to learn a model that allows us to predict the category of an unseen document. This discipline is called *supervised learning* due to the existence of labels. More specifically it is called *binary classification*: Classification because the target space $\mathcal{Y}$ is discrete, and binary because $\mathcal{Y}$ actually only consists of zero and one. If the target variable is real valued, e.g. when predicting the future income of a student based on his grades at university, then the problem

is called *regression*, which is not discussed here. The model we aim to learn is a function $h : \mathcal{X} \to \mathcal{Y}$, also called *hypothesis*, that maps a document to the category 0 or 1. We want the hypothesis to be as close as possible to the so called *target function* $t : \mathcal{X} \to \mathcal{Y}$ that always makes the correct prediction but at the same time is impossible to learn and chronically unknown for most real world problems. How close we are to the target function is measured by an error function or loss function which receives the prediction and the true labels to compute a score. A common and intuitive error function is residual sum of squares, shown next, or negative log likelihood, which is used for SFS and will be introduced later.

$$RSS = \sum_{i=1}^{N} (y_i - h(x_i))^2$$

Generalization. We are especially interested in a hypothesis that has a good *generalization* behaviour, i.e. that approximates the target function well for unseen documents. To train a model that always makes the correct prediction on the training data is easy: The model, assuming it has enough complexity, could simply memorize the complete dataset, which is always of finite size. Such a memorizing model would suffer the problem of *overfitting*, meaning that it fits the training data too much, including the irregularities or noise, but performs bad for unseen documents. Even if the model is not complex enough to memorize all data, it could take into account too many irrelevant features that might improve the accuracy for the training data but have only little statistical significance and therefore often describe the noise. The real generalization behaviour is unknown until the trained model is applied to unseen data. To get an earlier approximation, the input is usually split into two parts: The first being used for training and the second for testing, i.e. to evaluate the hypothesis using the error function. Next we will introduce feature selection which is a central approach to reduce overfitting and thus improve generalization.

2.2.2 Feature Selection

An extensive introduction to Feature selection is given by Guyon and Elisseeff [19]. We will summarize here what feature selection is, why it is applied and how SFS fits into this.

Feature selection is the process of selecting a subset of all available features in such a way that the removed features are either *irrelevant* or *redundant*: irrelevant features do not correlate with the target variable and redundant features bring little or no improvement because they are usually correlated with other features that were already selected to be in the model. The training algorithm has less chances to overfit if the data only has relevant features because we simply removed a lot of irrelevant features,

yielding improved generalization behaviour. A second benefit is that the data and the model become smaller and the time for training and prediction decreases. This is particular important for high dimensional data such as text. As a third benefit, we gain insights on the relevance of the features and increase their interpretability.

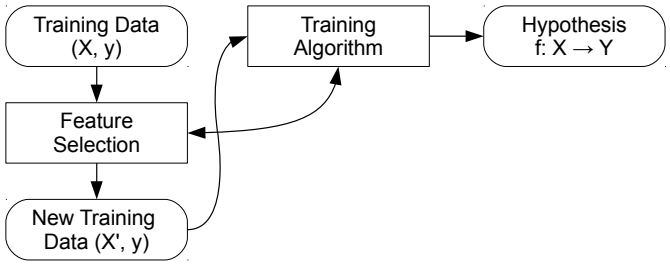

Figure 9: Feature selection as a wrapper method. The training algorithm is used to assess the features for usefulness.

Our feature selection algorithm is in the family of *wrapper methods*. As visualized in Figure 9 such a wrapper method uses the training algorithm, in our case logistic regression, as a black box to find the most useful features. There are other so called *embedded* methods where feature selection is an integral part of the training algorithm. To give an example, L1-Regularization is often proposed for logistic regression [23, 15] because it produces a sparse model containing only the relevant features, however the error function is no longer differentiable and so the optimization process becomes more complex and difficult to parallelize. A further and more generic approach to feature selection, better known as dimensionality reduction, is to project the input into a new and smaller feature space, containing features that can be used to approximately reconstruct the original input. Principle Component Analysis and Matrix Factorization such as Singular Value Decomposition are methods in this field. However, since these methods do not select a subset from existing feature it is harder to reason about the relevancy of the original features. We refer to [19] for an in-depth discussion of the various approaches to feature selection and their tradeoffs.

2.2.3 Logistic Regression

Before we continue with our specific feature selection algorithm we have to introduce logistic regression, the method for classification that is used in SFS.

Logistic regression has a long history as a statistical technique used in fields like economics, social science or medicine and is now also recognized as a workhorse of machine learning [27, 32]. We refer to logistic regression here as a method for binary classification.

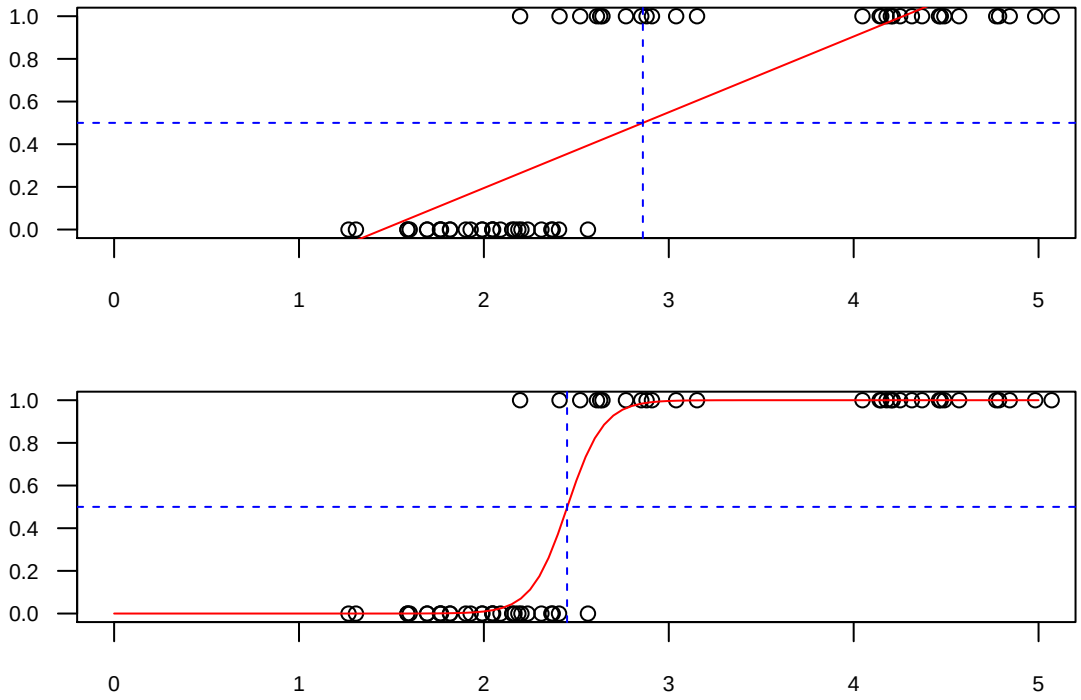

Figure 10: Linear regression and logistic regression applied to the same single-dimensional binary classification problem, where the two classes are encoded with 0 and 1. The red line is the function curve of the hypothesis and the vertical blue line visualizes the decision boundary. Linear regression was trained using least squares, which is similar to the previously introduced RSS and minimize the squared horizontal distance from the line to the points. For logistic regression we used maximum likelihood.

Despite the simplicity of the underlying linear model, logistic regression is a good fit for large scale high-dimensional data such as text or life science data for two reasons. First, its qualitative classification performance is competitive to more sophisticated methods like Support Vector Machines if the number of dimensions is very high [9]. Second, relatively simple and fast algorithms exist for the computation of the logistic regression coefficients [24, 38].

To understand logistic regression we need to shortly introduce the notion of a linear model. A linear model is a linear combination of x of the form

$$h_w(x) = w_0 + w_1 x_1 + \ldots + w_D x_D. \tag{1}$$

The output of this model can be used as a hypothesis and we call it linear regression then, a method for regression. For a one-dimensional problem it simply describes a line, as shown on top of Figure 10. Each hypothesis is completely defined by a vector w of weights, or coefficients, which is why we denote it explicitly as h_w. w_0 is called

bias or intercept term and defines a fixed offset in the prediction. For mathematical convenience we assume in the following that x is also extended by $x_0 = 1$ so that the dot product $w^T x$ can be used to express the linear model including the bias.

We can now introduce logistic regression as a generalized linear model where the output of the linear model $w^T x$ is fed into the logistic function $\sigma(x) = \frac{1}{1+e^{-x}}$ which is also called sigmoid function due to its s-shape.

$$h_w(x) = P(y = 1) = \sigma(w^T x) = \frac{1}{1 + e^{-w^T x}}. \tag{2}$$

The logistic regression hypothesis $h_w(x)$ can be interpreted as the probability $P(y = 1)$ that x belongs to the positive class 1. The coefficient w_i represents the impact of the i-th feature on the prediction. In a sparse model, after feature selection was applied, w_i should be zero for irrelevant features. We must be cautious, however, that a feature with weight 2 does not necessarily have to be more important than a feature with weight 1, since features might have different ranges of values. To fully understand the meaning of the coefficients of logistic regression we would need to introduce another interpretation of logistic regression as logit or log-odds model that is common in statistics. Since we do not use this interpretation we refer to Hastie et al. for a discussion [21].

We visualize logistic regression and the closely related linear regression in Figure 10. First, we see that logistic regression successfully handles outliers in the positive class and leads to a very good decision boundary. Second, we see that the prediction can be interpreted as a confidence, or probability. Third, we see that logistic regression can be interpreted as a generalized linear regression, where the logistic function (the S-curve) is applied to a linear regression curve.

Learning as an optimization problem. Given an input of labeled training data $\{(x_i, y_i)\}_{1 \le i \le N}$, the machine learning problem is to learn the model w that best fits these data, or equally, to minimize the error functions. There are many error functions and here we use maximum likelihood. Informally, a maximum likelihood solution is a model w that maximizes the likelihood $L(w)$ that, given this model, our training data were observed with their labels. It is equivalent to find the w that minimizes the negative log likelihood $-\ell(w) = -\ln(L(w))$, yielding the following mathematical optimization problem for the training:

$$\arg\max_w -\ell(w) = \arg\max_w -\sum_{i=1}^{N} y_i \ln(h_w(x_i)) + (1 - y_i) \ln(1 - h_w(x_i)) \tag{3}$$

Since there is no closed form solution for this problem, iterative methods such as Gradient Descent, Newton-Raphson or Quasi-Newton methods have to be applied.

We use Newton-Raphson, a method that converges faster than Gradient Descent but is computationally more expensive. Newton-Raphson starts with any initial model w and improves this model iteratively until convergence, using the first and second derivative (gradient and Hessian matrix) of the function to minimize. For negative log likelihood the update rule for the i-th iteration is the following, where $E(w)$ denotes the error function to minimize:

$$w_{i+1} = w_i - \frac{\nabla E(w)}{\nabla^2 E(w)} = w_i - \frac{\nabla - \ell(w)}{\nabla^2 - \ell(w)} \tag{4}$$

What makes logistic regression attractive for large scale problems is that both derivatives have a closed form and can be computed quickly. The derivatives will show up in our algorithm in the next section.

2.2.4 Scalable Feature Selection

This section summarizes the feature selection approach from Singh et al. [38] that serves as the use case for our system evaluation and that we already referred to as SFS.

The intuitive approach to a feature selection wrapper method would train a model for all possible 2^D subsets of D features, evaluate all those models on test data and choose the subset that is the best compromise of predictive performance and number of features. Given the high dimensionality of problems like text categorization and the iterative nature of most logistic regression algorithms, solving 2^D logistic regression problems is not feasible. Singh et al. use three techniques to reduce this to a linear runtime.

1) Forward Feature Selection. They adopted the well known *forward feature selection* technique where we start with an empty model and in every iteration add the single feature which brings the highest gain when being added to the current set of features (base model). In the i-th iteration our base model will have $i-1$ features and we have to train $D - i + 1$ models with dimensionality i. Repeating this nearly D times still yields a quadratic number of problems.

2) Single Feature Optimization. Singh et al. introduced a new approach to compute *approximate models* for the new features: Given a base model and a new feature to evaluate, the training algorithm keeps all parameters of the base model constant and only optimizes the single dimension for the new feature. They call this *Single Feature Optimization*. Assuming our base model has $50,000$ dimensions and we want to test another $100,000$ newly designed features, we now have to solve $100,000$ problems of dimensionality 1 instead of $100,000$ problems of dimensionality $50,000$. As a second step we have to compute the gain metric for $100,000$ approximate models

to get a *ranking* of the features. The last step is to add the best feature to the base model and retrain the complete base model to remove the error from approximation. The step of retraining is not discussed by Singh et al. and neither part of this thesis as it opens a completely new range of topics.

3) Single-Pass training and validation. We saw that for the feature evaluation of D new features we still have to run D single dimensional training algorithms and the same number of gain computations. This is where parallelism comes into play, and Singh et al. showed that the training and the evaluation can be done in a single pass over the training and the test data. We will come back to this when we discuss the parallel implementations for Stratosphere and Hadoop and now focus on the algorithm.

Up to now we described SFS as a forward feature selection method, where we build a complete model from scratch. According to Singh et al. and Konda et al. [25], however, feature selection is often an iterative process where features are being designed and evaluated and an existing model is improved by adding or removing features. Similarly, Anderson et al. describe feature engineering as an "interaction loop of Explore-Extract-Evaluate" [3]. SFS can be used in such a setting to quickly and efficiently evaluate and *rank* a large set of new features regarding their usefulness. Furthermore this ranking gives deep insights into the usefulness of individual features and thereby increases domain knowledge. In our introduction we used the example of an online learning system to motivate this. This ranking is what makes SFS special, compared to an embedded algorithm that only emits a single set of features that are all considered somehow useful.

Algorithm 1 defines a single SFS iteration. The first input is a training and test dataset with D dimensions, containing the features that are already in the base model as well as the new features to be evaluated. Additional input is the current base model w and a set of new features D_{new} to be evaluated, defined as a vector holding all indices of the new features. The base model has D dimensions, but will be zero for all dimensions defined in D_{new}. The first output is a weight vector w where the dimensions defined in D_{new} contain the trained coefficient. The second output is a vector of gains, where $gain_i$ is the approximate increase of log likelihood we obtain if we add the i-th feature to the base model.

The algorithm incorporates the previously introduced methods of logistic regression and Newton-Raphson. $h_w(x_i)$ is the logistic regression prediction for x_i using the base model, according to the definition in Equation 2. As a special notation, $h_d(x_i)$ denotes the prediction for x_i when extending the base model by the coefficient we trained for dimension d. w^d denotes the base model extended by the coefficient trained for d, and the trained coefficients are stored in w_{new}. $\nabla_d E(w)$ and $\nabla_d^2 E(w)$ are the first and second partial derivatives of the negative log likelihood function from

Equation 3 with respect to the d-th dimension. This is the actual implementation of Single Feature Optimization, and indeed the optimization of single dimensions is what makes this algorithm feasible. Both derivatives are scalar values as we see in Algorithm 1, whereas the partial derivative with respect to the complete base model w is a matrix of size $D \times D$ that is expensive to compute and invert.

The function Train, as defined below, requires $|D_{new}| \cdot I$ scans over all training data, where I is the average number of iterations for Newton-Raphson[12]. For now we stay with this non-scalable version of the algorithm and we will see in Chapter 3.1 how we can bring this down to a single scan over the data.

Algorithm 1 Sequential version of SFS

function ForwardFeatureEvaluation($\{(x_i, y_i)\}_{tra}, \{(x_i, y_i)\}_{tes}, w, D_{new}$)
  w_{new} = Train($\{(x_i, y_i)\}_{tra}$, w , D_{new})
  $gains$ = Evaluate($\{(x_i, y_i)\}_{tes}$, w, w_{new} , D_{new})
  return $(w_{new}, gains)$
end function

function Train($\{(x_i, y_i)\}, w, D_{new}$)
  for all $d \in D_{new}$ **do**
    # Train feature d using Newton-Raphson and Single Feature Optimization
    while w_d did not converge **do**
      $d_1 = \nabla_d E(w) = \nabla_d - \ell(w) = -\sum_{i=1}^{N} x_{id}(y_i - h_d(x_i))$
      $d_2 = \nabla_d^2 E(w) = \nabla_d^2 - \ell(w) = \sum_{i=1}^{N} x_{id}^2 h_d(x_i)(1 - h_d(x_i))$
      w_d -= $\frac{d_1}{d_2}$ (Newton-Raphson Update)
    end while
  end for
  return w
end function

function Evaluate($\{(x_i, y_i)\}, w, w_{new}, D_{new}$)
  $ll_{base} = \ell(w) = \ln(L(w)) = \sum_{i=1}^{N} y_i \ln(h_w(x_i)) + (1 - y_i) \ln(1 - h_w(x_i))$
  for all $d \in D_{new}$ **do**
    # Compute gain in log-likelihood for feature d
    $ll_d = \ell(w^d) = \ln(L(w^d)) = \sum_{i=1}^{N} y_i \ln(h_d(x_i)) + (1 - y_i) \ln(1 - h_d(x_i))$
    $gains_d = ll_d - ll_{base}$
  end for
  return $gains$
end function

[12]Newton-Raphson converged after a few iterations, usually less than five, in the examples we looked at.

2.3 Related Work

In this section we will give a short survey of related work. Thereby we focus on the intersection of Parallel Data Processing Systems, machine learning and feature selection. We add many references as a footnote to avoid blowing up the list of references. All mentioned websites were accessed in October 2013.

Related Systems. There are a couple of systems with objectives comparable to Stratosphere. We also refer to Sakr et al. [37] who recently published a comprehensive survey of "the family of MapReduce and Large Scale Data Processing Systems". Spark [45] is a cluster computing system centered around so called resilient distributed datasets (RDDs), which are distributed datasets held in memory to enable iterative algorithms such as machine learning and interactive applications. RDDs can be used as a data structure inside regular Scala programs (driver) offering operators like map, filter or reduce that receive a custom function and yield either a new RDD or send the results to the driver program. What makes Spark interesting in comparison to PACT is that RDDs are designed to be embedded in arbitrary programs that might include control structures or even interact with the user, whereas a PACT plan describes a single data flow graph that is executed as a whole on the cluster. The current research interests of Spark include stream processing, efficient fault tolerance and data analysis frameworks on top of Spark, which will be mentioned soon.

Further related systems include ASTERIX [5] and SCOPE [46], both trying to combine the strengths of parallel databases and Big Data technologies like Hadoop. Similar to Stratosphere, both are based on separate execution engines (Hyracks and Dryad) and both incorporate a compiler trying to find the most efficient execution plan for higher level queries. The authors of ASTERIX give a broad survey of research challenges [5], including "modern storage and indexing", "text data and queries" and "dynamic parallel query processing". The latter one will become interesting in our discussion of the PACT Compiler. While the first papers did not mention machine learning, a more recent paper from 2012 [8] introduces a machine learning framework on top of Hyracks based on an "Iterative Map-Reduce-Update" model and a Scala based high level machine learning language, which could be used to implement feature selection or logistic regression. SCOPE is a system used at Microsoft offering a SQL-like declarative scripting language with C# integration for custom operators. Similar to what the authors of ASTERIX propose, the SCOPE optimizer leaves some decisions to runtime to enable dynamic optimizations.

Apache Tez[13] and Apache Drill[14] are two systems under development for Hadoop

[13]See http://hortonworks.com/hadoop/tez. The website describes that Tez "generalizes the MapReduce paradigm to a more powerful framework for executing a complex DAG of tasks" which reads like Stratospheres initial design.

[14]See http://incubator.apache.org/drill.

Yarn with similarities in the architecture to Stratosphere. Both are targeted to be a low latency execution engine to execute higher level languages such as Apache Hive[15], Apache Pig [33] or Cascading[16], which are currently compiled to Hadoop MapReduce jobs. Apache Pig and Cascading could have been also used to implement SFS. Lin and Kolcz [30] used the UDF capabilities of Apache Pig to implement machine learning frameworks for Twitter on top of Hadoop. Cascading offers a workflow oriented API for arbitrary data processing with operators such as Merge, GroupyBy, CoGroup and HashJoin yielding a DAG comparable to PACT plans.

Scalable Machine Learning and Feature Selection. Let us now discuss some frameworks dedicated to scalable machine learning. MLBase [26] is a recently published machine learning framework on top of Spark aiming to hide the complexity of Machine Learning and its scaling from the user. It offers a high level machine learning language including functions like `doClassify(X,y)` and `findTopFeatures(data)`, the latter coming very close to the SFS algorithm. Apache Mahout[17] is a library for scalable machine learning running on top of Hadoop MapReduce. Besides Clustering, Recommender Systems and Classification it contains algorithms for dimensionality reduction, but not for feature selection. Mahout also contains a Java library for sequential matrix and vector arithmetic, which we used for the implementation of SFS. SystemML [17] is another machine learning framework from IBM that compiles jobs to MapReduce. It exposes the "Declarative Machine learning Language" with an R-like [35] syntax offering mathematical constructs e.g. for linear algebra, but we are not aware of any built-in logic for feature selection. Other related frameworks we only like to mention briefly include GraphLab[18], a C++ based framework, and MADlib[19], aiming to execute machine learning tasks on parallel databases.

Since machine learning is a lot about statistics, a recent approach is to make the established statistical R language scalable. Several commercial database systems including SAP HANA[20] and Oracle R Enterprise[21] support the execution of R on their database. We already mentioned that the syntax of SystemML is also based on R. Konda et al. [25] short ago prototyped an R-based feature selection framework running on Oracle R Enterprise. They understand feature selection as an "iterative, ad-hoc and a subjective process that is driven largely by a user's understanding of the entities under consideration". They provide the operations they identified to be important for this process, including forward and backward feature selection,

[15]See http://hive.apache.org.
[16]See http://www.cascading.org.
[17]See http://mahout.apache.org.
[18]See http://graphlab.org.
[19]See http://madlib.net.
[20]See http://help.sap.com/hana/SAP_HANA_R_Integration_Guide_en.pdf.
[21]See http://www.oracle.com/technetwork/database/options/advanced-analytics/r-enterprise/index.html.

manual addition or removal of features, and training and scoring models using logistic regression (see Table 1 in their paper). Although they do not cover unstructured data and high dimensional problems, their work is closely related to SFS and gives a good impression how feature selection can be applied in real world scenarios.

At last, we would like to mention a group of well established tools for machine learning and data mining that are limited to a scaleup[22]: Libsvm and liblinear[23] both support embedded feature selection via L1-regularization. Weka[24] additionally supports wrapper feature selection. The Python library scikit-learn[25] supports a wider range of feature selection approaches[26], but we did not find a wrapper or forward feature selection approach such as SFS.

[22]Most of the tools make use of in-memory techniques. This makes them a good fit for smaller and medium sized datasets. Some might even incorporate a multi-threading approach, but we did not analyse this.

[23]See http://www.csie.ntu.edu.tw/~cjlin/libsvm and http://www.csie.ntu.edu.tw/~cjlin/liblinear.

[24]See http://www.cs.waikato.ac.nz/ml/weka.

[25]See http://scikit-learn.org.

[26]See http://scikit-learn.org/stable/modules/feature_selection.html.

3 Comparing Parallel Programming Models

This chapter presents a qualitative evaluation of Hadoop and Stratosphere with regard to their programming models. We proceed as summarized in Figure 11: First, we transform the sequential version of SFS into a scalable version and afterwards we evaluate how well it could be mapped to the programming models and discuss both implementations.

Figure 11: Strategy for the programming model evaluation for SFS.

3.1 Making Feature Selection Scalable

When we started to design SFS using PACT we tried to free our mind of the solution using MapReduce given by Singh et al. and also tried to ignore the specific contracts that are offered by PACT. Instead we searched for an intuitive and abstract scalable solution for the SFS problem. Given such scalable algorithm we can evaluate how naturally the algorithm can be expressed in MapReduce and PACT. Often this step is skipped by going directly from the problem to the implementation in the specific programming model, as done by Singh et al. Adding this step explicitly has some advantages: It enables a fair comparison, results in a better understanding of SFS and gives us a notion of what an ideal programming model for SFS would look like.

Starting point is the sequential SFS algorithm as defined in Chapter 2.2.4, which seems far away from being a scalable algorithm. Algorithm 2 is a copy of the Train function in Algorithm 1 with an explicit loop for the computation of the sum and explicit formulas. For clarity we only analyse the Train function in detail in this chapter because the ideas for the Evaluate function are the same. This new version already incorporates a first optimization to exploit the sparseness of the data: For the training of feature d only the records that have a non-zero value in this dimension are relevant. This highly reduces the number of records that are considered for the computation of the derivatives, however, we still have to scan over all records in this version.

In the next transformation, described in Algorithm 3, we aim to meet the requirements of a scalable algorithm and limit ourselves to a single scan over the data. For this we assume infinite memory and use an associative array (Map) data structure

Algorithm 2 A more explicit sequential version of SFS Train, exploiting sparseness

function TRAIN($\{(x_i, y_i)\}, w, D_{new}$)
  for all $d \in D_{new}$ **do**
    while w_d did not converge **do**
      double $d_1 = d_2 = 0$
      for all $(x_i, y_i) \in \{(x_i, y_i)\}$ where $x_{id} \neq 0$ **do**
        $a_i = w \cdot x_i$ (constant in all iterations, only place where x_i is used)
        $p'_i = h_d(x_i) = \frac{1}{1 + e^{-(a_i + w_d \cdot x_{id})}}$
        $d_1 = d_1 + x_{id}(y_i - p'_i)$
        $d_2 = d_2 - x_{id}^2 \, p'_i(1 - p'_i)$
      end for
      $w_d = w_d - \frac{d_1}{d_2}$ (Newton-Raphson Update)
    end while
  end for
  return w
end function

to cache all the data we need for the subsequent training. The training itself is completely done in memory[27]. Looking at the variables used inside the inner for-loop of Algorithm 2 we can derive what we need to cache for the training: For dimension d we need the values $(x_{id},\ y_i\ , a_i)$ for every record having feature d. It is interesting to note that it is sufficient to store the precomputed values $w \cdot x_i$ instead of the whole vectors x_i, which is due to Single Feature Optimization where the optimization is only concerned about a single dimension and the rest is fixed. A last change is that D_{new} is no longer needed, which eases the use of SFS. It is derived whether a feature is new by testing if it is already in the base model. This was not possible before since the algorithms iterated over D_{new}. D_{new} could be easily included if one does not want to evaluate all features in the input.

Let us briefly look at the memory requirements and runtime for Train. The memory footprint for the Map data structure depends on the number of non-zero fields in the input $\{(x_i, y_i)\}$: Every non-zero field x_{id} is stored exactly once, together with the value of $w \cdot x_i$ and y_i. Also the runtime depends on the number of non-zero values in a linear way: For every non-zero value, one value is written to the cache, and the inner loop of `TrainSingleDimension` is executed I times for every cache entry, where I is the average number of Newton-Raphson iterations. Similar observations apply to the Evaluate method in Algorithm 4. This highlights that it is useful to characterize datasets by their sparseness, i.e. by the number of non-zero values.

[27]This is in fact a technique used by machine learning software such as Weka or Liblinear to efficiently implement iterative learning algorithms.

The same idea of exploiting sparseness and using an in-memory data structure can be applied to Evaluate. The reason why we can exploit sparseness here is that for the computation of the gain in likelihood for feature d, we only need to consider the records containing d, since for all other records the likelihood will be unchanged. The result is shown in Algorithm 4 for completeness.

Algorithm 3 A scalable version of SFS Train that can be parallelized easily.

function TRAIN($\{(x_i, y_i)\}, w$)
  # Single pass: Cache everything needed for training in memory
  Map cache = new Map($d,\ [(x_{id},\ y_i,\ w \cdot x_i)]$)
  for all $(x_i, y_i) \in \{(x_i, y_i)\}$ **do**
    for all $x_{id} \in x_i$ where $w_d = 0$ **do**
      cache.add($d,\ (x_{id},\ y_i,\ w \cdot x_i)$)
    end for
  end for
  # Train all dimensions in-memory
  Vector w_{new}
  for all $d \in$ cache.keys **do**
    $w_{new,d}$ = TRAINSINGLEDIMENSIONNEWTON(d, cache.get(d))
  end for
  return w_{new}
end function

function TRAINSINGLEDIMENSION(d, cached-records)
  int $w_d = 0$
  while w_d did not converge **do**
    double $d_1 = d_2 = 0$
    for all $(x_{id},\ y_i,\ w \cdot x_i)$ in cached-records **do**
      $$p'_i = \frac{1}{1 + e^{-(a_i + w_d \cdot x_{id})}}$$
      $$d_1 += \frac{\partial L_i}{\partial w_d} = x_{id}(y_i - p'_i)$$
      $$d_2 -= \frac{\partial^2 L_i}{\partial w_d} = x_{id}^2 p'_i(1 - p'_i)$$
    end for
    w_d -= $\frac{d_1}{d_2}$ (Newton-Raphson Update)
    return w_d
  end while
end function

Algorithm 4 A scalable version of SFS Evaluate.

```
function EVALUATE({(x_i, y_i)}, w, w_new)
    # Single pass: Cache computed log-likelihood gain in memory
    Map cache = new Map(d, gain)
    for all (x_i, y_i) ∈ {(x_i, y_i)} do
        ll_base = y_i ln(h_w(x_i)) + (1 − y_i) ln(1 − h_w(x_i))
        for all x_id ∈ x_i where w_d = 0 do
            ll_d = y_i ln(h_d(x_i)) + (1 − y_i) ln(1 − h_d(x_i))    (h_d accesses w_new)
            cache.add(d, ll_d − ll_base)
        end for
    end for
    # Sum up gains of likelihood
    Vector gains
    for all d ∈ cache.keys do
        for all gain ∈ cache.get(d) do
            gains_d = gains_d + gain
        end for
    end for
    return gains
end function
```

An ideal programming model. Algorithm 3 is very friendly to data parallelism because the for loops over the input and over the features have almost no side effects. An ideal[28] parallel programming model for our use case would not require to rewrite the algorithm at all and parallelize it automatically by executing the code inside the loops in parallel for different parts of the data[29]. We would need only a few assumptions: First, we need to operate on a specialized Map data structure that allows concurrent additions for the same key. In the same way we need to be able to write to a vector w_{new}, however, there we do not need support for conflicting write access since every coefficient is written to only once. Second, if we move the code inside the for-loops into separate functions, or UDFs, to be executed in parallel, every parallel instance needs a reference to the base model w. This is a free, or non-local variable for the anonymous function and we need to make it available, as it is supported by closures in functional programming languages. The same applies to the trained coefficients w_{new} which have to be made available in the first for-loop of Evaluate. When we write about closures in the following, we refer mainly to the

[28]The notion of what is ideal is to a certain extend subjective, but it is difficult to avoid subjectivity in a qualitative evaluation. Important to note is also that ideal only refers to our use case.

[29]The Matlab programming language has built in support for such automatic parallelization: A loop can be parallelized using data parallelism and multithreading simply by using `parfor` (parallel for) instead of `for`.

ability of a UDF to access variables that become non-local variables after moving the code inside the loops into UDFs. In general, closures also describe the function itself having such capabilities, but the main motivation of closures was to enable the access to variables of the environment [40][30]. Finally, we must be aware of the limited space for pipeline and task parallelism because the cache has to be populated completely before we can use the `cache.get(d)` in training and so we need a synchronization barrier between the for-loops.

Evaluation methodology. Now as we defined SFS as a scalable algorithm and we developed the notion of an ideal programming model for SFS we can evaluate how well, or naturally, it can be expressed by using MapReduce and PACT. For this we will look at multiple criteria, the most important being expressiveness. This comprises that the system should offer constructs allowing us to express our algorithm declaratively, i.e. in a way that is not over or underspecified. We saw in the last paragraph that this is possible when we imagined a parallel programming model that offers a data structure and operators on it with additional support for closures. This gives the system degrees of freedom to choose a good parallelization strategy such as distributing the data structure and running the operators in parallel, or even a shared memory multi-threading approach. The system should offer such constructs for all places where data, task, pipeline parallelism or any other optimization can be applied. A second criterion will be compactness and readability, describing how close the resulting code is to the original algorithm in visual terms. As a last criterion we shortly look at the overall user experience, including the actual process of development.

3.2 Scalable Feature Selection on MapReduce

The question for this section is how well the previously defined scalable algorithm can be expressed using Hadoop MapReduce. The implementation, which was described by Singh et al. and implemented before the thesis, is visualized in Figure 12. SFS can be accessed from a single Java class, the driver, which owns the base model and internally runs the Hadoop jobs to compute the gain for new features. The driver can be instantiated within any JVM-based application that has a network connection to the cluster. We begin our evaluation with the positive impressions and relativise these afterwards.

Ignoring Hadoops implementation of MapReduce for a moment, Map and Reduce turn out to be excellent constructs to express SFS in. With a slightly different vocabulary

[30]We found that the authors of Spark also refer to closures in a similar meaning when writing about their shared variables concept: "Programmers invoke operations like map, filter and reduce by passing closures (functions) to Spark. As is typical in functional programming, these closures can refer to variables in the scope where they are created." [45]

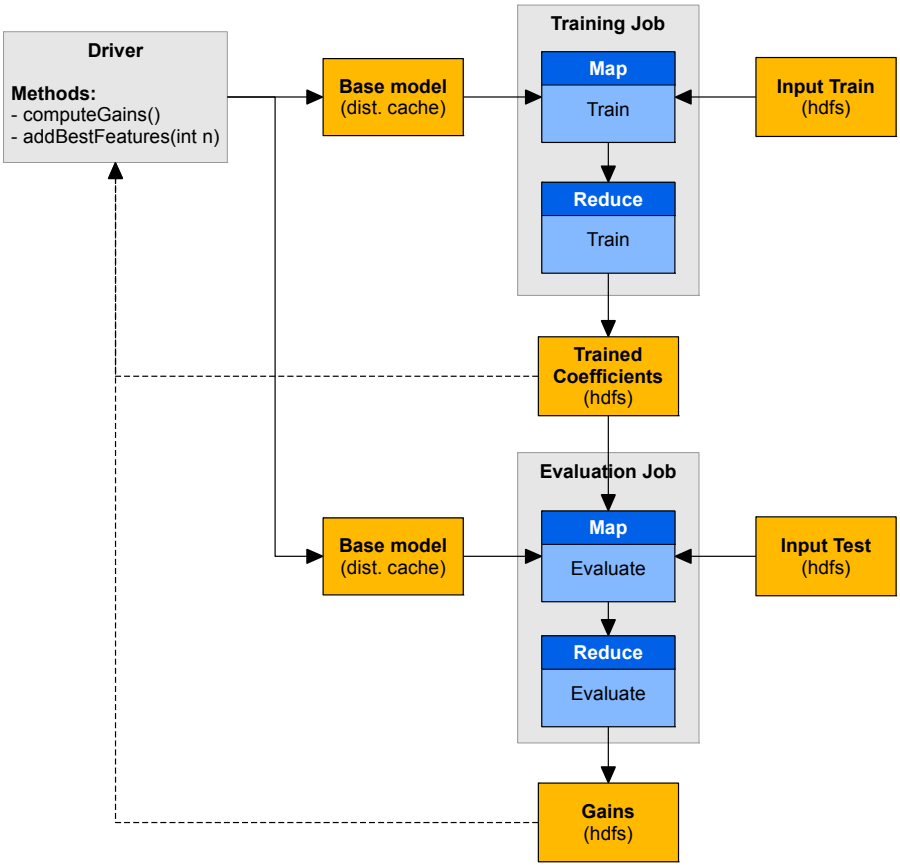

Figure 12: Job Graph for SFS using Hadoop MapReduce.

we can read the Train function as if it was already specified in MapReduce: Map is equivalent to the for-loop over the input dataset and applies the code inside the loop to all data records. The output of Map is equivalent to the associative array. Finally, Reduce runs an aggregation, the training, for every key in the cache and the output of Reduce is our vector with the trained coefficients. The same ideas apply to the Evaluate function. This looks very close to what we described as an ideal parallel programming model for our use case in the last section. Furthermore, the plan in Figure 12 is a great high level visualization of SFS, showing the dependencies and the output of the different phases. One might say that it is just coincidence that the static chain of Map and Reduce fits our needs, or that SFS was designed to fit MapReduce, but we think that both are very powerful primitives for parallelization, maybe the most important ones, because processing, grouping and aggregating is such a common task. This view is supported by the fact that Borkar et al. suggest a scalable machine learning framework [8] that is solely based on Map, Reduce and iterations. This positive impression has to be relativised for multiple reasons.

Verbosity, limited expressiveness and media disruptions. These are the

three main weaknesses we identified and in the following we will explain where we encountered them.

To express the algorithm in MapReduce one has to create multiple Java classes and split up the algorithm into many different parts. This makes it significantly harder to understand and maintain the program, a problem summarized by us as verbosity, the opposite of compactness. Since Hadoop jobs are hardcoded to consist of a Map followed by a Reduce and since there is no support for chained jobs we have to split up the algorithm into two Hadoop jobs and need to add a separate Java class (Driver) to execute them in a sequence. Both Hadoop jobs have to be further divided into three classes: One for the Map UDF, one for Reduce and a third for the job itself. Part of the problem is that Hadoop uses Java, which has very limited support for functional programming and is itself known for its verbosity. Verbosity can be resolved to a certain extent by using a higher level language such as Cascading or maybe Apache Pig. We find, however, that it is a bad user experience that Hadoop does not directly offer a concise programming model and instead puts the burden of finding a suitable and mature language on the user.

A second main problem arises since we need to make the base model object available in both Map UDFs via any kind of closure. In the same way, the output of the training phase needs to be available in the Evaluate Map UDF. As a limited expressiveness, Hadoop does not offer a direct construct for this. Instead one has to use the Distributed Cache that is designed to broadcast arbitrary files to all nodes before a job is executed. The Distributed Cache is one of the places where what we call media disruption occurs: Instead of working with the constructs of the programming model at a single place, the user has to write custom code in different classes to serialize the base model object to a file and to read it in the UDF. Additionally, due to its multi-purpose design the Distributed Cache removes the knowledge of what actually is transmitted and thereby removes any degrees of freedom for the system. In our case this means that the system does not detect that we transfer the same base model object to both jobs and therefore has to transmit it twice.

A further weakness related to limited expressiveness arises if we want to support multiple iterations to implement forward feature selection. Since there is no support for iterations the user is forced to implement a separate driver for this. We call this approach *driver iterations*. To run a single iteration of forward selection, one has to call `computeGains()`, which runs the training and evaluation job in sequence and read the results from hdfs, followed by `addBestFeatures(1)`, which adds the feature with the highest gain to the base model. This lack of native iterations adds complexity, but more importantly it removes the degree of freedom to cache the input, which is constant in all iterations, in memory.

User Experience. During the thesis we adapted the existing Hadoop SFS imple-

mentation to operate on Yarn and to be executed automatically for the experiments. Since the Java API of Hadoop is constantly evolving, it was sometimes difficult to find out how to do simple things such as running a job on a cluster or loading the data for the experiments into hdfs programmatically. For such tasks it is common to use shell scripts and the command line interface of Hadoop, however this would mean yet another media disruption. Another disruption in the development process is the need to build a jar file first in order to run a job on the cluster.

3.3 Scalable Feature Selection on PACT

One design goal of Stratosphere is to overcome the shortcomings of MapReduce. In a comparison paper of PACT and MapReduce the authors of Stratosphere claim that the improvements over Hadoop MapReduce such as the more expressive programming model yield a better performance and a better experience for the programmer [2]. In this section we re-evaluate the second part of this claim for SFS. We will begin to shortly explain relevant parts of our implementation and discuss the strengths and weaknesses we identified afterwards. The complete list of issues we identified, including several bugs, contains more than 30 issues and we can only present the most important here.

We implemented two versions of SFS for Stratosphere: Driver iterations, similar to Hadoop, and *native iterations*, using the built-in iteration support of Stratosphere. The driver iterations version was written to be comparable to Hadoop and to determine how big the performance gain of native iterations is. It can be accessed via a driver implementing the same methods as the Hadoop version. We list the constructor arguments of the driver in Table 9 in the Appendix. They include the specification of the input, the settings for Newton-Raphson optimization and the information about where to run the job. The native iteration version, shown in Figure 13, uses the Stratosphere bulk iteration feature [13] to implement forward feature selection. It can be accessed by the `forwardFeatureSelection` method. Every bulk iteration produces a complete new partial solution, which is in our case the new base model extended by the k features with the highest gain, where k is configurable. One main benefit from iterations, besides expressiveness, is that the system detects that the training and test input are constant in all iterations and thus it is free to keep these in main memory. Furthermore, the job has to be started once only for many iterations, which reduces the startup overhead.

Let us give a short overview of the PACT plan. If we ignore the non-bold nodes, we can recognize the similarity to the MapReduce solution, except Cross instead of Map is used and the contract "Apply Best" is added to extend the base model with the best features at the end of a single iteration. The Match UDF is used to join the trained coefficient and log-likelihood gain for each feature, so that "Apply Best" has

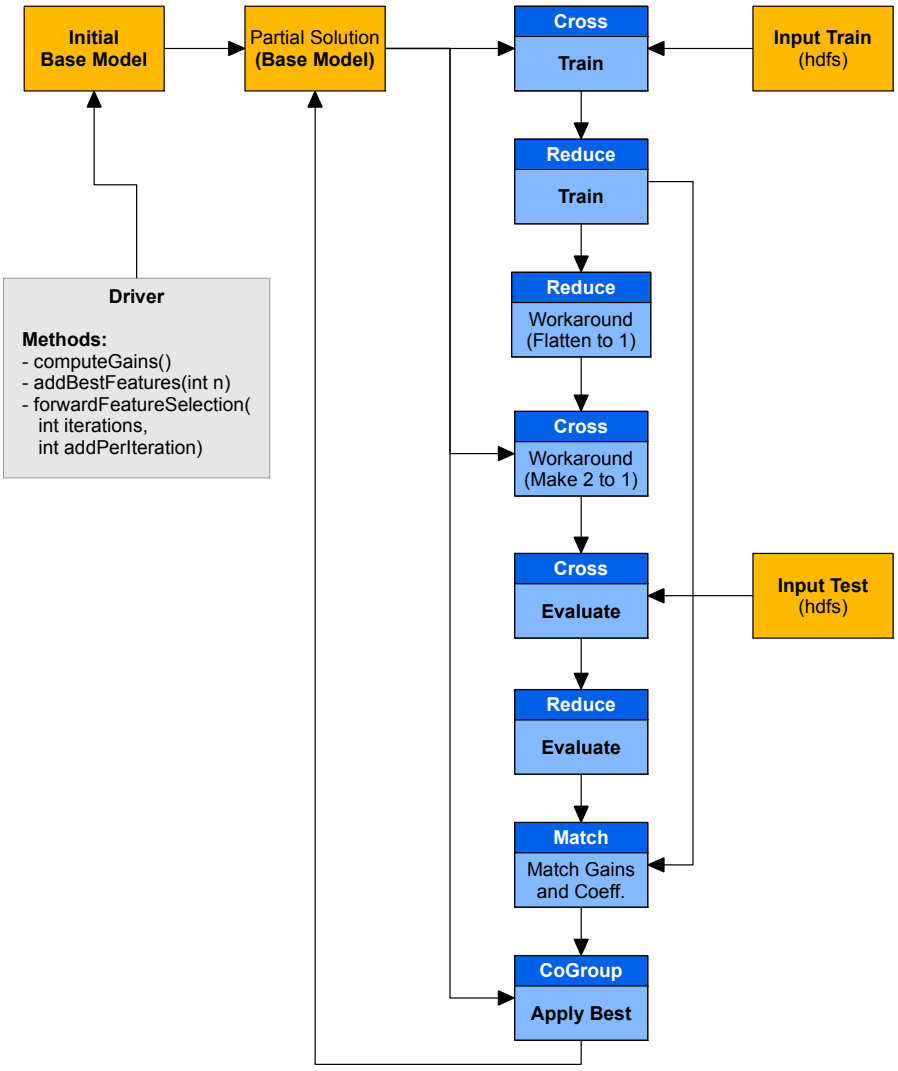

Figure 13: PACT plan and driver for SFS using the bulk iterations feature. The plan as shown will be executed by the driver method `forwardFeatureSelection`. The other methods implement driver iterations, similar to the Hadoop version. The plan for driver iterations is the same, except that it has no partial solution and ends after the Match node.

a convenient input. As in the MapReduce version, the code within the Train and Evaluate UDFs is analogous to the body of the for-loops of the scalable algorithm. The differences marked as "workaround" will be explained in a moment and we now begin with the evaluation.

"Everything flows". Being able to express our algorithm, including iterations, as a single job without leaving the constructs of PACT is the highest gain in expressiveness compared to MapReduce. We found that this greatly improves the user experience

since there is no more media disruption like manual chaining of jobs. Additionally the system has, at least theoretically, new degrees of freedom to choose from different execution strategies due to a more holistic view of the job. We also found that the addition of Cross, CoGroup and Match is a good trade-off between expressiveness and added complexity since the new constructs add a clear value and are relatively easy to understand, not least because they operate on the same level of abstraction as Map and Reduce. We found that the naming, however, could be more intuitive[31]. Similar to MapReduce, we found that PACT offers a good level of abstraction to write data processing tasks. To conclude, looking at what was added in PACT compared to MapReduce, we think that the benefits outweigh the added complexity. There are, however, some important things missing so that the overall picture is different.

Limited Expressiveness. The most significant shortage we identified is that PACT does not support closures and neither offers a construct like the Distributed Cache to be used as a workaround. The only way to solve this is to use Cross as a workaround in the following way: If we need to make a single record, such as the base model, available in a Map UDF, one can use Cross instead of Map with the second input being the base model. We consider Cross to be an unfortunate workaround for five reasons: First, it can be only applied to add closure support for Map. It was surprising to find that there is no way to make the context of a previous output available in a subsequent Reduce, Match or CoGroup, to the best of our knowledge. Second, it only works if the non-local variable is a single record and not a list of records. The first Evaluate UDF needs the output of the training phase, which is such a list of records, and therefore we had to write the Reduce workaround "Flatten to vector" to merge all trained coefficients to a single record. Third, it can be only applied if we have a single non-local variable because Cross is limited to two inputs. For this reason we had to introduce the workaround contract "Make 2 to 1" that makes a single record out of two. Fourth, it is computationally inefficient because in every UDF call we receive a tuple (x_i, non-local-variable), where the non-local variable has to be copied by the system for every call, despite being constant. Fifth, we consider closures to be a more intuitive description of what our original intention was than Cross. This opinion, however, depends on our view of an ideal programming model and on the way we specified our scalable algorithm.

Let us briefly discuss possible approaches for closures. More practical for us would have been a way to broadcast and attach the output of any contract as a preliminary input to a specific UDF. Each UDF instance could process the additional inputs first and store them in main memory. Figure 14 visualizes this approach. It is comparable to the Distributed Cache, with the difference that the user does not have to read from files. A problem arises if there is a large number of small records to be broadcasted,

[31]For users familiar with SQL, Join would more intuitive than Match, and CoGroup does not tell that it is a special case of Reduce for multiple inputs.

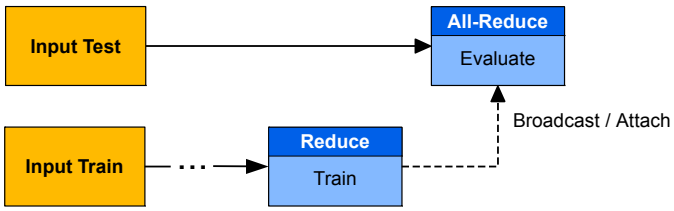

Figure 14: One approach to closures by broadcasting and attaching. This example shall be representative for the cases where the output of one UDF is required in a subsequent UDF other than Map. It has two phases: The first trains a model, e.g. a vector, and the second evaluates the accuracy of this model. All-Reduce is a Reduce where all items are sent to a single group.

like it is the case for the Reduce UDF in the training phase: It may emit millions of PACT records, each consisting solely of a single trained coefficient and the index. To avoid this overhead one could offer a machine learning specific data model so that the Reduce UDF can write in a distributed way to a vector, which would be much closer to our original intention. Such vector could be broadcasted and attached very efficiently as a single object[32]. In an ideal scenario, the system would be aware of the semantics and vector arithmetics within the UDF and decide whether it is feasible and fortunate to broadcast the whole object or whether it is better to use a join to realise the dot product[33]. We decided to broadcast the vector as a whole because even for very high dimensions the vector will fit into memory and there is currently no efficient support from Stratosphere for system-based vector arithmetics.

The full discussion of this and other approaches is out of scope for this thesis, since it raises a few more questions, for example how the system handles the additional dependencies arising from the attached outputs, as we see in Figure 14. We suggest that more use cases need to be analysed to find an appropriate solution. It is interesting to note, though, that the notion of closures only arises from the way we defined SFS as a scalable algorithm. This gives a different perspective on SFS that is more data and data structure oriented. In contrast, PACT appears more data flow and UDF oriented. As an example, the in-memory cache of Train was a central part of the scalable algorithm but becomes almost invisible in PACT: It is the equivalent to the intermediate output of the Train Reduce UDF, stored in schemaless records as we will discuss soon. The Hadoop plan appears to be more informative, since the output of the first job is written to a file, with a well defined schema. The introduction of higher level interfaces such as a Scala interface will change the perspective to be

[32]We simulated this behaviour with the "Flatten to 1" UDF.

[33]This thought is based on discussions with Sebastian Schelter and Alexander Alexandrov from the DIMA group, who both reminded of the fact that for a dot product of two sparse vectors, only the coefficients that are in both vectors need to be joined, multiplied and summed up afterwards. In this way, broadcasting the whole vector can be seen as a shortcut.

closer to our ideal programming model.

We shortly like to mention further shortcomings regarding expressiveness. First, the system is designed to read input from files, and it not well support to embed the job execution in an external driver application and use an object from runtime, such as our base model, as input[34]. This can be also interpreted as a limited support for closures, since objects or variables from the driver are non-local variables in the UDFs. Second, there is no keyless CoGroup where all records from both inputs are streamed into a single UDF instance, as we would have needed it for the "Apply Best" contract. Although the PACT Compiler has an extensible architecture, it is non trivial to add such new PACTs. A third limitation is the lack of global counters. Hadoop offers such counters that can be incremented in any UDF and read in the client after the job ended. Hadoop automatically collects many useful information via these counters such as number of records emitted in the different phases. Custom counters are very useful for debugging, performance analysis and to extract information such as number of non-zero values. Counters can also be interpreted as support of closures: A counter is a variable in the driver that is defined outside the UDF but accessed inside. We propose to additionally support global histograms, for example to quickly analyse the distribution of the number of Newton-Raphson iterations needed in SFS.

Verbosity. Stratosphere suffers from the same problems as Hadoop regarding verbosity. There is one class describing the plan, one for each UDF and in our case one for the driver, splitting the algorithm into many parts. The workarounds for closures make the plan overall even worse readable than the MapReduce plan. There are currently more concise high level languages under development, such as a Scala interface, which will probably improve the situation.

Schemaless records and UDFs. The sparse and schemaless record model of PACT has positive and negative aspects. It greatly simplifies the transfer of structured data such as two numeric values: In Hadoop we need to write a special serializable class containing nothing but the two values. However, it implies that UDFs are also schemaless, in the meaning that they do not specify what input they expect, what output they produce and which fields the input and output values reside in. The only way to derive the implicit schema of an UDF is to rely on code documentation or to read the UDF completely. As stated earlier this is in contrast to our ideal programming model where the cache data structure has an explicit type. This also contrasts to Hadoop where the signature of the UDF completely specifies the schema and where there is no risk to mix up the field indices. To highlight that all our UDFs have a static schema we introduced public fields such as `IDX_INPUT2_BASEMODEL=0`, stating that the basemodel is assumed to be stored at field 0 in the second input. We added the Evaluate Map UDF for Hadoop and Stratosphere in Listing 1 and 2 in the

[34]We wrote two extensions for this purpose, which can be found in the classes `RecordSequenceInputFormat` and `SingleValueDataSource`.

appendix to document this.

Immaturity, Traps and Bugs. We spent a significant amount of time with bugs or at least traps, since Stratosphere is still under development. To give an example, we tried to use the keyless Reducer, also known as AllReducer, where all records from the input are forwarded to a single UDF instance, but it yielded an unspecific error message. This bug was fixed, but after a while we found that the plan visualization does not work for keyless Reducers. This bug was also fixed, but during the implementation of native iterations we received an unspecific error message, that all contracts inside the iterations must have the same dop. After some debugging we realized that for the keyless Reducer the dop is automatically set to one and we finally had to use a workaround that does not use the keyless Reducer. Another main problem we faced was the complexity of the build system of Stratosphere that did not support different hadoop versions. A last problem is the lack of profiling and performance analysis tools, which is related to the lack of counters. Information such as in which UDF the most time was spent or whether the data fit into memory are important when writing a more complex program, since the user mostly also has some degrees of freedom to optimize the UDFs. Although thesis issues are less interesting from a scientific perspective, the system has to overcome them to be competitive regarding the user experience.

We summarize the results of our programming model evaluation in Table 7 in the conclusion, to present them together with the results from the experimental evaluation.

4 Experimental Evaluation

The goal of this section is to compare Hadoop and Stratosphere regarding their speedup and scaleout behaviour for the SFS algorithm. We start by explaining our hardware and software environment, continue by describing the two high dimensional datasets used and finally discuss the experiments and their results. In these experiments we will vary the cluster size, the size of the dataset and other interesting dimensions such as the number of iterations.

4.1 Experimental Setup

Hardware Environment. All experiments were executed on a cluster of 26 identical machines and one additional machine that served as the master. Each slave machine has two AMD Opteron 6128 2GHz CPU with 8 cores, 32 GB of RAM, 4 separate disks with 1 TB each, connected with gigabit ethernet. The master machine has two Intel Xeon E5620 2.4GHz CPU with 4 cores and hyperthreading enabled, 48 GB of RAM, and a RAID 5 array of 850 GB.

Software Versions. We used Hadoop version 2.1.0-beta, the latest version at the time of writing. We choose Hadoop YARN because it will be the successor of Hadoop 1.0 and any results for Hadoop 1.0 would soon become obsolete. As an example, the resource and memory management logic has been completely overhauled. Furthermore, YARN reached a certain stability as it is already in large-scale production use at companies such as Yahoo![35]. For Stratosphere, we used the version 0.2-ozone[36] which is currently under development. There is no support for previously published versions and some features, such as iterations, are only available in 0.2-ozone. Hadoop and Stratosphere were both executed on Java version 1.7.0_40 from Oracle. The slaves run Linux Ubuntu Server 12.04.1 and the master Ubuntu Server 10.04.4.

Configuration and Tuning. Both systems have a large number of configuration options which we do not show in detail for clarity. We generally configured the systems to use all available resources, especially memory and disk[37]. We found, however, that the optimal settings highly vary for each experiment and therefore most experiments were executed with individual settings.

[35] http://developer.yahoo.com/blogs/ydn/hadoop-yahoo-more-ever-54421.html, accessed Sept, 24th, 2013

[36] We built Stratosphere from the source code available at https://github.com/dimalabs/ozone.

[37] both systems allow specifying multiple directories on different disks for temporary data to spread disk I/O

4.2 Datasets

	# Records	# Features	File size	# Non-zero values
RCV1-v2	23,149 and 781,265	47,236	43 MB and 1.34 GB	59,157,312 for test (average 76 per record)
Webspam	350,000	16,609,143	23.3 GB	1,304,697,446 (average 3728 per record)

Table 1: Dataset properties. If two values are given they refer to the training and test part. We calculated the number of non-zero values using Hadoop counters.

We used two high dimensional real-world datasets for our evaluation. Both datasets are available in a preprocessed numerical form at the dataset library of the libsvm tool[38]. A dataset in the libsvm format is a simple text file with the label and a sparse representation of the vector x_i in each line. We now describe both datasets shortly.

Reuters RCV1-v2. This dataset consists of all 804,414 news articles in English that were published by Reuters between August 20, 1996 and August 19, 1997 [28]. All articles were manually categorized using a hierarchy of topic codes, where multiple topics can be selected at the same time. On the highest level there are four topic codes: CCAT (Corporate/Industrial), ECAT (Economics), GCAT (Government/Social), and MCAT (Markets). To translate this into a binary classification problem, we train a one-versus-all classifier that looks only at the presence or absence of a single topic code. If nothing else is stated we used the ECAT category, similar to Singh et al. Each of the 47,236 features describes the occurrence of a word according to the bags of word model. Additionally, high frequency words (stop words) like "the" were removed, all words where reduced to their stem (stemming) and tf-idf (term-frequency inverse document-frequency) encoding was applied to measure the importance of the occurrence. These are common methods from the fields of information retrieval and text mining. The dataset is split into a training dataset with 23,149 and a test dataset with 781,265 records and the uncompressed filesize is 43 MB and 1.34 GB. In most cases we are only interested in the runtime and therefore use the test file also in the training phase.

Webspam. This dataset consists of 350,000 websites that were categorized into spam websites, such as link farms and regular websites [43]. Each website is described in terms of 16,609,143 features, where each feature describes the occurrence of a tri-gram. Each continuous three bytes are treated as a tri-gram, so that the word "yes" would form a feature. The uncompressed filesize is 23.3 GB. We manually split the dataset in two equally sized files to obtain a training and test set.

Both datasets are not in the terabyte range, and thereby do not exploit the systems full potential. Nevertheless we think they are a good choice for multiple reasons.

[38]http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/, accessed Sept. 24th, 2013

First, as discussed in Chapter 2.1, we think that the speedup for real world and normal-sized datasets is equally important as the scalability to very large datasets. Second, both datasets are large in terms of dimensions, which is a challenge on its own for the systems because they have to shuffle and group the data into millions of groups and efficiently handle millions of UDF calls. To still be able to examine the systems scaleout behaviour we used a scaled version of the RCV1-v2 where we appended the whole test file multiple times to a larger single file.

4.3 Experiments

Let us start with a few words on the test methodology. Whenever the size of the dataset allowed it, we executed ten trials for every experiment to guarantee statistical significance. We discarded an additional first run after starting the system to consider only a "hot" state of the system.. The runtime was measured using wall-clock-time, i.e. total elapsed time for the job. The time for the system startup was not considered. Equally we ignored the time to load the dataset since it is equivalent for both systems. In most cases the plots should speak for themselves and only in a few cases we found it supportive to add some values as text.

Speedup for RCV1-v2 In our first experiment we used the RCV1-v2 dataset as a fixed size input and varied the number of nodes. The task was to run a single SFS iteration to compute a ranking of all features. Figure 15 shows the absolute runtime and the speedup, where the runtime for a single node serves as the reference value for the speedup. First we observe that Stratosphere has almost linear speedup until dop 10 whereas Hadoop drops of after dop 2 and has almost constant runtime above dop 5. A first reason is that Hadoop has to start up two jobs where Stratosphere uses a single. During our experiments we found that this can add several tens of seconds. Furthermore, the number of Map tasks is determined solely by the number of input splits and does not change with the number of nodes. Since there were only 44 map tasks for our input, the map phase did not become faster for bigger clusters and became a limiting factor. Consequently, changing the block size to 64 MB yielded even worse results for Hadoop since there was even less parallelism during the Map phase. This contrasts to Stratosphere where we did not see a big impact by the block size since the number of Cross instances is solely determined by the specified degree of parallelism.

We can further observe that due to the relatively small file size, the overhead for adding more nodes clearly becomes a dominating factor for both systems. Stratosphere keeps the job startup costs relatively low since all tasks for one node are executed as multiple threads in a single long-running Java Virtual Machine (JVM) that is started with the system startup on each node. Hadoop Yarn, which is designed for large clusters with multiple jobs running in parallel, isolates the long running services

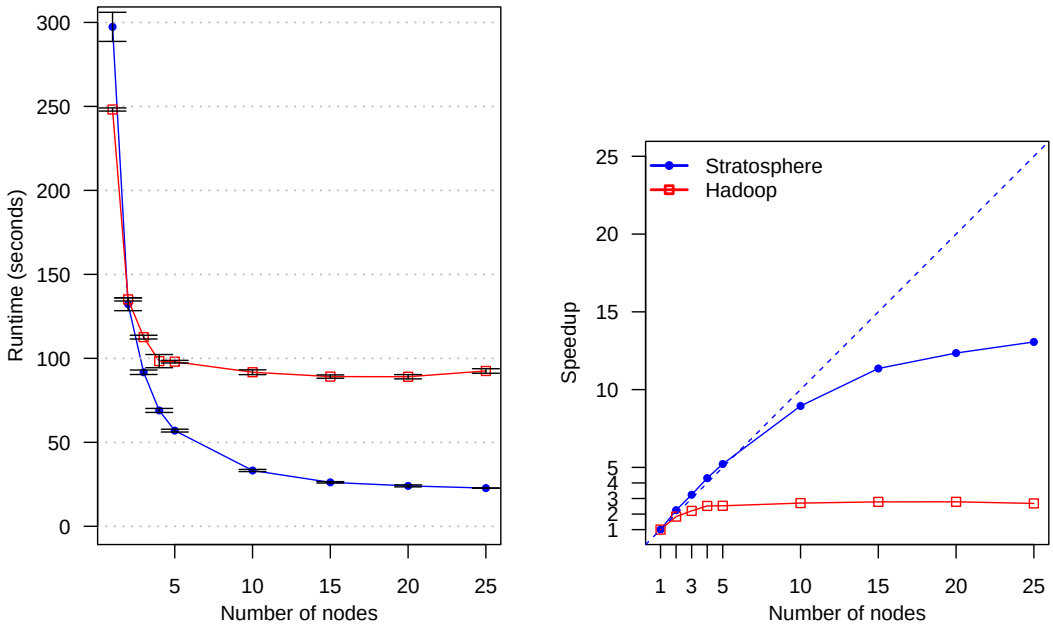

Figure 15: Absolute runtime and speedup for a single SFS iteration using the RCV1-v2 dataset for training and evaluation (two times 1.34 GB) and an empty base model. The blue dashed line denotes ideal speedup behaviour. Every point stands for the mean runtime of 10 repetitions and the error bars denote 95% confidence intervals. Both systems use intra-node dop 8, for Hadoop we set HDFS block size to 32 MB, for Stratosphere to 64 MB.

from the application specific tasks and creates a separate JVM, so called containers, for each task. In contrast to Hadoop v1.0 these containers cannot be reused for different tasks in Yarn. We visualized the different memory layouts in Figure 16 and explain them briefly in the caption of the Figure.

Let us shortly discuss some observations in favour of Hadoop. First we see that the runtime for dop 2 is almost similar for both systems, which can be explained by the smaller startup overhead which is no longer a dominating factor for Hadoop. Additionally we have to notice that Stratosphere has a relatively high reference runtime on a single node, making its speedup curve generally look better. Furthermore, it took some tuning and very specific settings to get the good results for Stratosphere. To give an example, the runtime for dop 15 halved when we changed the TaskManager memory manager size from 22GB to 21GB. To conclude this discussion we can say that Stratosphere demonstrated an overall better speedup behaviour than Hadoop for this smaller dataset, although being very sensitive to tuning, and we identified the smaller overhead as a main reasons.

Speedup for Webspam. Our second speedup experiment is similar to the first except we used the Webspam dataset. As we will see, the main new challenge is not

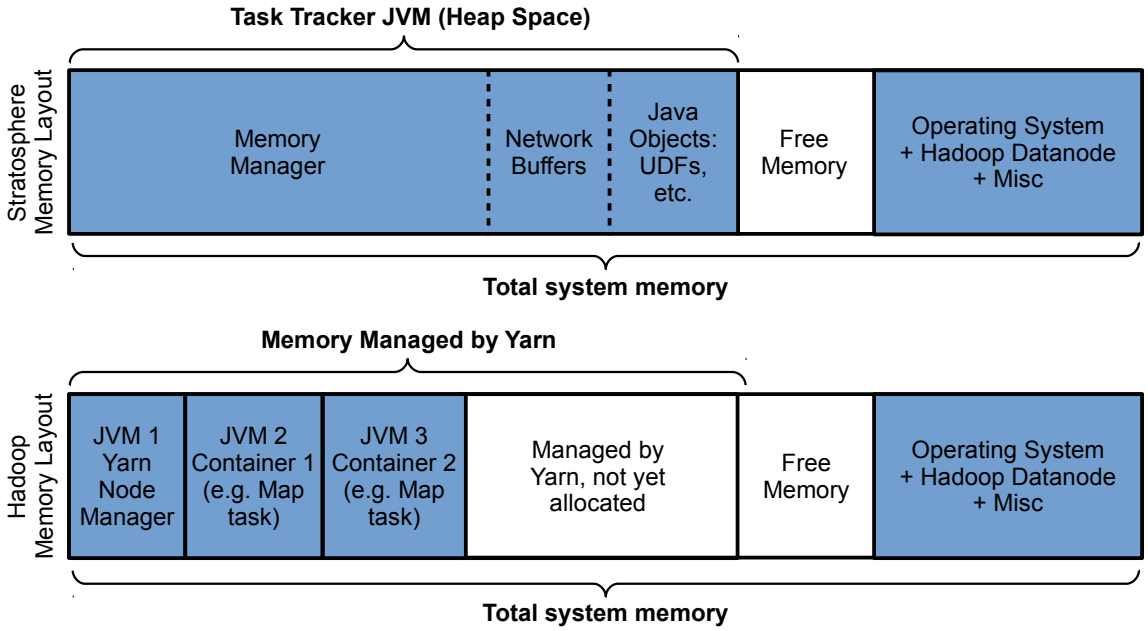

Figure 16: Comparing Memory Layouts of Hadoop and Stratosphere slaves (proportions are just an example). Stratosphere runs all tasks in a single JVM whereas Hadoop starts a separate JVM for every task. The memory size for the memory manager and the number and size of the network buffers are the main tuning options for Stratosphere. Our basic configuration was as follows: 29GB heap size for the TaskTracker, 21 GB for the memory manager, and 1GB for network buffers (32K buffers of 32KB size). Yarn was granted 29GB in total and each container had a size from 1GB to 8GB.

the increased file size but the number of features, which highly increased to 16 million. Initially we were not able to run the test for Stratosphere due to various errors where the system was running out of memory. Most errors also occurred for small samples of the dataset, indicating that the problem is the high number of features. Indeed, Stratosphere emits about 16 million records in the Reducer for the training phase, one for each trained coefficient. Additionally these records are merged to a single record holding a dense mahout vector of 16 million double-precision floating-point numbers, which should have a size of more than 127 MB (16,609,143 * 8 byte). Only after longer tuning we found that it works if we set the intra-node dop to one[39]. This partially explains the huge gap between Stratosphere and Hadoop because Hadoop could be executed with intra-node dop 8, yielding higher parallelism and a better resource utilization. But it is is probably not the only explanation since Stratosphere is 14 times slower than Hadoop for 10 nodes (63 minutes versus 4,5 minutes). We assume that another reason is that the large record with trained coefficients is crossed with the test data and the system has to copy the record for every UDF call.

[39]Additionally we had to set the number and size of network buffers to 32,000 and 128 KB and decrease the size for the memory manager to 15 GB.

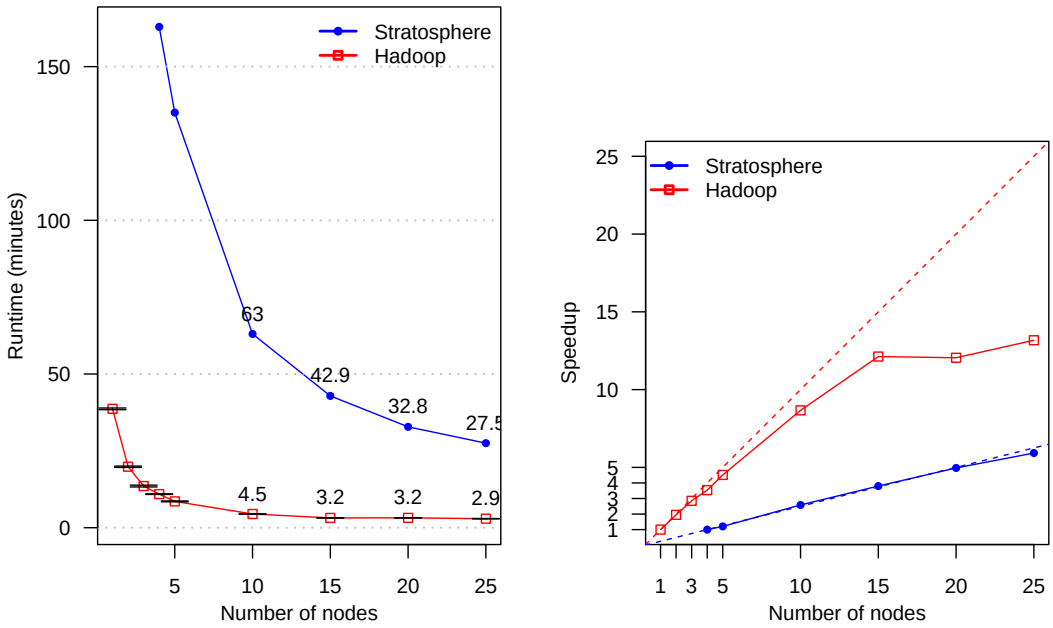

Figure 17: Absolute runtime and speedup for a single SFS iteration using a 50/50 training and test split of the Webspam dataset. For Hadoop we executed 10 repetitions, for Stratosphere we could only run a single repetition starting at dop 4 due to the long runtime (previous test runs yielded similar results). Therefore we had to plot a separate dashed line for the ideal speedup for Stratosphere, considering dop 4 as the reference runtime. Stratosphere required specific network buffer (32K buffers of size 128KB) and memory settings (15GB memory manager) and intra-node dop one, whereas Hadoop used intra-node dop 8.

Related to the Cross issue we have to mention that we introduced an optimization in the Stratosphere job: Instead of deserializing the base model and the trained coefficients for every UDF call we cached it to a local variable. This highly reduced the runtime, e.g. from 64 minutes to 28 minutes for dop 25. Please consult Listing 2 in the appendix to get a better understanding of this optimization. The job we used for the RCV1-v2 experiments only cached the base model, but we left these results because we think that the burden should not be put on the developer to remember this optimization[40]. We see that the different ways to implement closures have a big impact on the runtime if the vectors for the trained coefficients or the base model become very large. The Hadoop solution is not very elegant but performs very efficient: It write the vectors to a file, broadcasts these and reads from file directly into a vector, whereas Stratosphere transmits a single record for each number and requires additional workaround UDFs yielding a higher overhead.

[40]To introduce caching was also a pitfall: When used with iterations, one has to know that the same UDF instance is used in all iterations and to remember that the cache has to be repopulated after each iteration.

If we look at the speedup behaviour we see that Hadoop has a great speedup behaviour until dop 15 and the curve looks almost similar to the speedup curve of Stratosphere for RCV1-v2. Stratosphere, considering the values we have, has an almost perfect speedup behaviour. This makes sense because the overhead of adding a single node with intra-node dop one is very low compared to the overall runtime. To conclude this experiment, we see that both systems demonstrate a good speedup behaviour on this larger dataset, however, Stratosphere is much slower because its approach to closures does not scale to very high dimensional problems.

Scaleout Experiment. Since Stratosphere was too slow for the Webspam dataset we fall back to use a scaled version of RCV1-v2 for our scaleout experiments. As described in the background section, an ideal scaleout behaviour means that we can proportionally increase the number of nodes and the problem size and still solve the problem in the same time. It is not trivial to say what a scaled problem is in our case. For this experiment we only considered the file size, however, as a future work it would be beneficial to create a data generator that varies other equally important influencing variables: The number of dimensions, the sparseness of the records and the distribution of the number of records per feature.

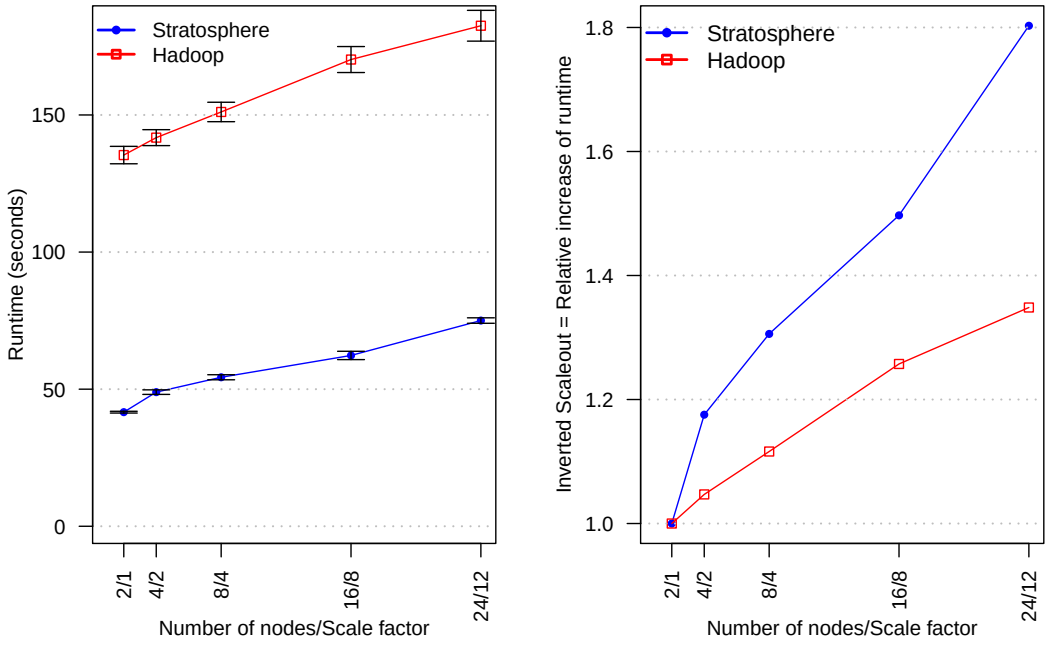

Figure 18: Scaleout experiment using the scaled version of the RCV1-v2 test dataset for training and evaluation. The number of nodes and the problem size are proportionally increased. The left side shows the runtime, the right side shows the inverted scaleout which is the runtime of the scaled problem divided by the runtime for two nodes. The runtime for Stratosphere and two nodes is better than in Figure 15 because the job used for RCV1-v2 did not have the caching optimization for the trained coefficients.

Let us analyse the results shown in Figure 18. The ideal scaleout behaviour would be a flat line in both diagrams, however this is impossible due to many factors such as network overhead [17]. Comparing the runtime of the smallest and the biggest problem, Stratosphere has a lower absolute increase in runtime (33 seconds versus 47 seconds) but a higher relative increase. More important to note is that the curves behave linear, with steady but small growth, which suggests that we can scale to very large problems while the runtime is increasing only slowly[41]. We can conclude that both system demonstrate a good scaleout behaviour, in the meaning that it is feasible to solve larger problems, in terms of file size, in almost the same amount of time by adding more nodes. There is no well defined threshold for a "good" scaleout behaviour, but the slope of our curves looks similar to an scaleout experiment for SystemML [17].

Dataset	File Size	Non-zeros	Features	Nodes	t Stratosphere	t Hadoop
RCV1-v2 scale 12	32,38 GB	1,419,775,488	47,236	24	75 sec.	183 sec.
Webspam	23.3 GB	1,304,697,446	16,609,143	25	1,650 sec.	176 sec.

Table 2: Comparison of the runtime (denoted by t) for Webspam and the scaled RCV1-v2 problem.

We can also use our previous results for Webspam to analyse the scaleout behaviour for higher dimensionality. As we see in Table 2, Webspam is comparable to the scaled RCV1-v2 regarding the size and number of non-zero values but highly differs in the number of features. If we compare the runtime for Stratosphere we see that it fails to handle the high dimensionality, for reasons we already discussed, and we can conclude that the good scaleout behaviour we identified before does no longer hold if we scale the number of dimensions. In contrast to this, the results for Hadoop indicate that the good scaleout behaviour also holds if we scale the number of dimensions.

Revisiting Amdahl and Gustafson. Let us shortly interpret the results from our experiments in terms of Amdahl's and Gustafson's law. Gustafson refers to our speedup experiments when he writes "One does not take a fixed-sized problem and run it on various numbers of processors, except when doing academic research." [18]. And he refers to our scaleout experiments when writing that "in practice, the problem size scales with the number of processors" and "it may be most realistic to assume run time, not problem size, is constant". Although we do not share the position that speedup scenarios are unrealistic[42] we could observe the limits in such scenarios: For example, the speedup curve for Stratosphere and RCV1-v2 reminds of the speedup

[41] We did not test how many additional nodes we need to keep the runtime constant, but we assume that the number is relatively small compared to the total number of nodes. Also we did not test the scaleout for clusters with thousands of nodes and terabytes of data and there is probably a limit.

[42] We already cited Yanpei Chen et al. [44] who showed the existence and relevancy of medium and small jobs within big clusters, i.e. jobs that did not grow with cluster size.

curve according to Amdahl for a problem with serial fraction of about $\frac{1}{15}$ that could never experience a speedup greater than 15 because the serial fraction becomes the dominating part. In our experiments, however, it was more the overhead than the serial fraction that became the dominating factor. Since Amdahl and Gustafson both ignore the overhead one should be slow to take Amdahl's law as an excuse for a limited speedup[43]. We could also observe that the scaleout behaviour of both systems is not as limited as the speedup behaviour. This is in line with Gustafson who wrote that it is "much easier to achieve efficient parallel performance than is implied by Amdahl's paradigm" if we scale out, i.e. we scale the problem and then speed it up.

Iterations for RCV1-v2. The last experiment aims to mimic a forward feature selection use case with multiple iterations, where in each iteration one or more features are being added. Thereby we evaluate the different approaches to iterations, namely driver and native iterations, which are described in Chapter 3.2 and 3.3. It will be particularly interesting to see how big the performance gain is when using the native iterations feature of Stratosphere. We can expect two improvements from native iterations: First, only a single job needs to be started and second, the system can cache the training and test files in memory. We choose a setup of 15 nodes with RCV1-v2 scale-factor 8 for training and evaluation, yielding an input size of 21.6 GB in total and $1,44$ GB to cache for every node.

Let us first analyse the total runtime in Figure 19 for 10 iterations, adding one feature per iteration. As expected, native iterations are fastest, followed by Stratosphere driver iterations and Hadoop. The gain in performance for the native variant is very small, which suggests that neither the disk/io nor the job startup times are dominating factors of SFS. This seems natural since big parts of the job are not affected by iterations, particularly the repartitioning of all data via Reduce.

Let us now look at the runtime when adding 10 features each iteration. The first surprise is that this has a high impact on the runtime. The second surprise is that Hadoop performs better in this situation and has a significantly smaller increase of runtime. A first explanation for the increasing runtime is that the computations we have to run on each non-zero value, almost 1 billion for scale-factor 8, become more complex if the base model is no longer an empty vector. For this reason, the experiment can be seen as a scaleout experiment where the size of the base model is varied. As an explanation for the high increase of Stratosphere, the previously discussed inefficiency of Cross becomes visible if the base model is no longer empty.

Let us look at the runtime for the individual iterations in Figure 20 to find out more. We see that in most cases the runtime increases linearly with the size of the

[43]The good speedup for Stratosphere and RCV1-v2 showed that there is room for improvement. Parallel databases are an example of what is possible: They are written from beginning on to minimize the overhead and they almost approach linear speedup and scaleout [12].

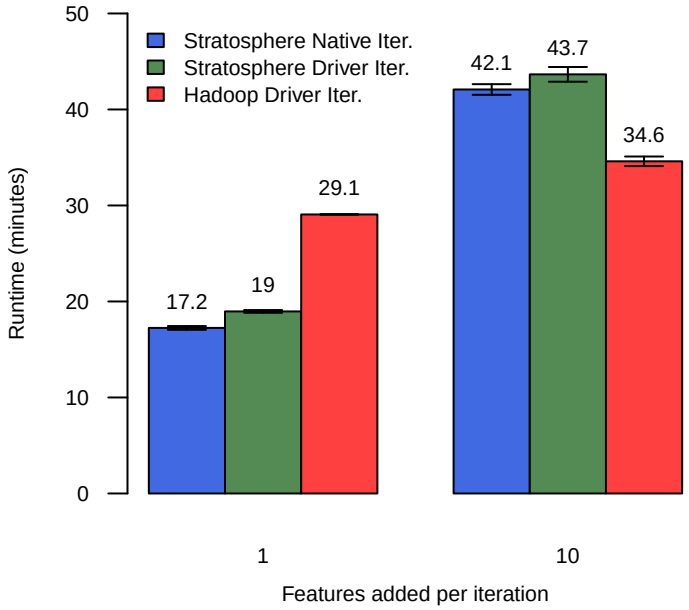

Figure 19: Absolute runtime for 10 iterations, using different approaches to iterations. We varied the number of features added per iteration. The dataset RCV1-v2 with scale-factor 8 is used for training and evaluation. We show the mean and the 95% confidence intervals (very small) of 3 executions.

base model, which is the only parameter changed. For Hadoop we can only see a small increase of the runtime after we added many features, which suggests that the increasing complexity of the computations, that holds for both systems, is just a small factor. This leaves the inefficiency of Cross or any other detail we missed as an explanation for the high increase of Stratosphere. We would like to further investigate the reasons since we could not do this due to a lack of time and simple profiling tools for Stratosphere. In the course of this we also like to analyse the irregularities in the curves at dop 6 and 7. As a last observation we see that the curve for Stratosphere driver iterations slightly decreases after adding a single feature, which can be seen as the savings of having the input stored in-memory.

To summarize, we saw that native iterations bring only little performance improvements for SFS. Additionally the runtime increases with the size of the base model, slightly for Hadoop and significantly for Stratosphere. This is disappointing because even with native iterations it would take long to build a complete model for high dimensional and large scale data, since the number of relevant features can be large. Singh et al. also did not include such a performance experiment in their paper, and instead focused on the use case to evaluate, or rank, a set of new features in a single iteration.

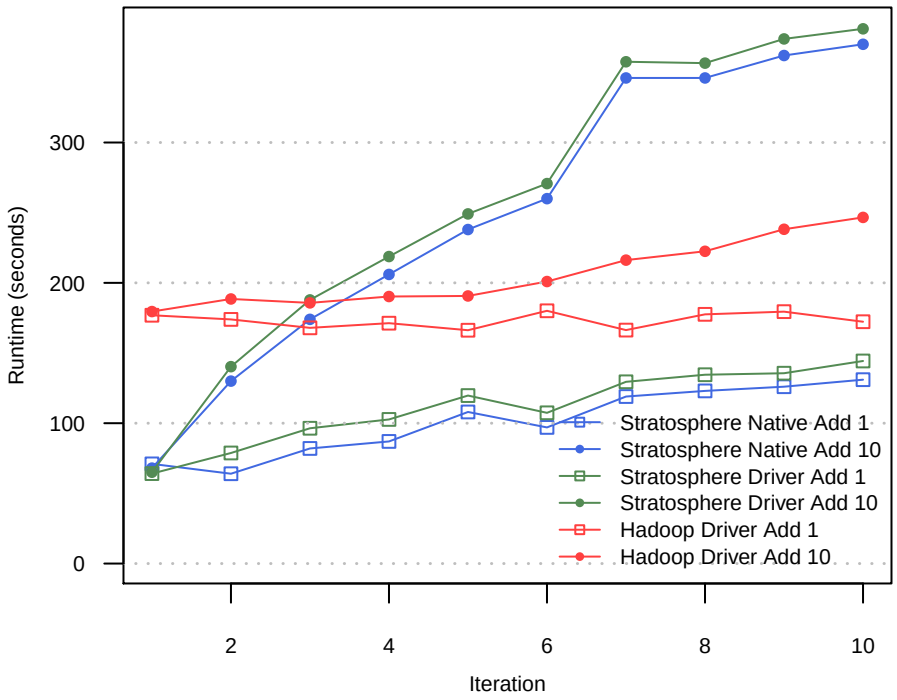

Figure 20: Runtime for the individual iterations using RCV1-v2 scale-factor 8 for training and evaluation. We show the mean of three runs. The runtime for a driver iteration is solely the time for the job, without the time of the driver to read the result, add the best feature and call the job. The runtime for native iterations were extracted from logfiles.

As the last part of our experimental evaluation, we will have a closer look at the job characteristics of SFS and the PACT Compiler, which both had an impact on all previous experiments.

Limited Degree of Freedom. To recap, we saw that Stratosphere shows a better runtime in all experiments, ignoring the high dimensional case, and we identified the reduced overhead as one main factor. However, we could not observe that the additional degrees of freedom and the ability of the PACT Compiler to choose from multiple execution strategies made an overall positive contribution. A first explanation for this is the characteristic of our algorithm. The main computational effort for SFS is to process and repartition the test and training data by dimension, which is done by Cross and Reduce. Stratosphere implements only a single strategy for Reduce relying on hash partitioning and sorting, comparable to Hadoop[44], so that it has no degrees of freedom at this place. Furthermore, the large number of records

[44]The strategies for Reduce differ in detail so that Stratosphere sorts on the reducer side (Reduce) whereas Hadoop sorts on the sender side (Map). The implementations also differ in the way they use memory.

between Cross and Reduce cannot be pipelined as we saw during the analysis of our scalable algorithm. This means that for big parts of the job there is limited space to choose from different strategies or to apply other optimizations that distinguish Stratosphere from Hadoop, such as pipelining parallelism or special join strategies. This highlights that the efficient implementation of Cross (or Map) and Reduce is crucial for SFS. The good speedup results indicate that the existing implementation is already very efficient.

We also want to mention a situation where Stratosphere correctly used its degree of freedom. Stratosphere avoids the repartitioning of the inputs to match the gains and trained coefficients at the end of our plan since it recognizes that both outputs are already partitioned by the same key, the dimension[45]. Although the data are relatively small here, this optimization has the potential to improve the performance significantly. The interested reader can also consult the whole execution plan in Figure 23 that visualizes some of the issues discussed here. For clarity we decided to put the execution plan in the appendix because it would require to explain too many details. We will now discuss situations that are less specific to SFS, where Stratosphere can choose from different strategies but makes an unfortunate decision.

PACT Compiler: Lost in Unknowns. A general problem we identified is that the cost-based PACT Compiler is often unable to choose a good strategy because it simply does not know all costs and has to apply a default strategy. To be more precise, when talking about the compiler we mostly refer to the optimizer component of the PACT Compiler. The compiler computes cost information for every node of the plan based on information such as the number of records emitted (output cardinality) or the size of the output in bytes. The only place where all these information are known is a file input, but after the first UDF the system does not know anything about the output since the user can emit any number of arbitrary records. The system therefore relies on manual compiler hints, given by the developer in the code, or it falls back to default estimations: Every UDF call is assumed to result in one output record of unknown size. This often leads to situations where the compiler knows the cardinality, but not the size.

Figure 21 shows such situation that we encountered twice in our plan, in the Cross UDF for training and evaluation. In both cases a large file from HDFS is crossed with a single record whose size in bytes is unknown at compile time[46] and all we could do is to give a hint that the cardinality is one. The compiler, however, applied

[45]This only works because we manually annotated both preceding Reduce UDFs with `@ConstantFields(0)`, stating that field 0 of any output record will be equal to field 0 of the input records. This implies that the partitioning that was created for Reduce is preserved.

[46]Since the base model is a non-file input and can be the output of the previous iteration we do not know its size in bytes at compile time. Similarly we do not know the size of record that contains the base model and the trained coefficients which is the input of the evaluation Cross.

the strategy "unknown costs are always higher than known costs" and assumed that the 24GB file is smaller and should be broadcasted to all nodes to realize the Cross. For this reason, Stratosphere was significantly slower than Hadoop in all our first experiments until we found the problem and hardcoded the strategy to be the other way.

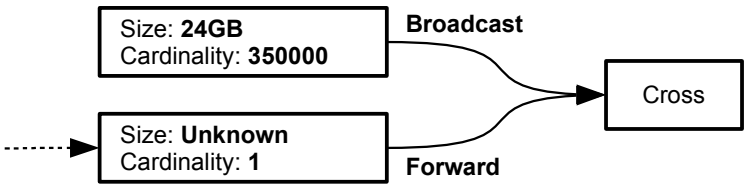

Figure 21: Illustration of a PACT Compiler decision for Cross. The values are chosen arbitrarily. The compiler follows a conservative strategy, avoids the risk that the first input is larger in terms of size in bytes and broadcasts the second input to all nodes.

Unknown costs also prevent an optimization for Match, where Stratosphere tries to broadcast one input if it is relatively small and thereby avoids the need to repartition the other input. If the size of the small input is unknown, the system will fall back to the conservative strategy to repartition both inputs, yielding a significant higher runtime if the other input is large. To conclude, we think that the current cost based compiler rules are too simple and do not reflect the reality where cost information are often partially or completely unknown.

There are multiple approaches to address this. First the compiler rules for partially or unknown costs could be revised, for example to handle the cases where only the cardinality is known. Second, the system could try to collect more cost information, either by offering more expressive and specialized contracts, such as a Map or Reduce where only a single record can be emitted, or by offering a more unified and user friendly compiler hint programming interface[47]. One could also apply static code analysis to automatically infer cost information. The feasibility of this approach was shown by Hueske et al. who created a prototype for Stratosphere [22] that is not yet part of the system. A last approach is to use dynamic execution plans that are adapted during execution based on collected statistics or new resource situations. Behm et al. describe this approach for Hyracks and propose plans that depend on variables to be bound later, i.e. at runtime [5]. SCOPE follows a similar approach, described in section 7.2 of [46]. But inspiration can also be gained from traditional database systems like Oracle[48], who only recently introduced "adaptive query optimization" in their

[47]There are multiple places in the code to give these hints, making them harder to maintain and understand.

[48]See Oracle Whitepaper "Optimizer with Oracle Database 12c" http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-optimizer-with-oracledb-12c-1963236.pdf, accessed 2013, Oct. 21th

optimizer. As long as non of these approaches is taken, it should be considered to delegate the decision to the user, e.g. by adding contracts like `CrossBroadcastFirst` or `MatchBroadcastFirst` that dictate which of the inputs will be broadcasted[49]. This reduces declarativity and degrees of freedom but also reduces the risk that the system makes a decision that is either unfortunate or too conservative.

A full discussion of this issue is out of scope for this thesis. To conclude, we summarize important aspects in Table 3: At compile time, the true output size and cardinality is always unknown, and only in a few cases cardinality estimations are available, which might be wrong. More cardinality information can be made available via specialized contracts. What the table does not show is that all information can be derived at runtime, which highlights that dynamic optimization is the most powerful approach. The aforementioned work from other systems suggests that this is an active field of research.

Operation	Knowledge of output cardinality and size at compile time	Estimation of output cardinality at compile time
Map	unknown	N
Reduce	unknown	unknown
Cross	unknown	$N_1 \cdot N_2$
Match	unknown	unknown
CoGroup	unknown	unknown
Map one-to-one	N (size unknown)	N
Map filter	$\leq N$ (size unknown)	$\leq N$

Table 3: Summary of the information about the output of an UDF that can be derived from the input by the PACT Compiler. N is the number of records of the input, with an index if there are multiple inputs. The bottom adds specialized versions of Map which are currently not available but considered to be added.

4.4 Discussion of Scalable Feature Selection

The qualitative analysis of the SFS algorithm was not a part of this thesis and is covered deeply by Singh et al. [38]. However, to evaluate the correctness of our algorithm we compared our results to the results of Singh et al. and to the features that are selected by a embedded feature selection method.

Comparison with Singh et al. Table 4 shows the first five SFS iterations adding the best feature in each iteration. Similar to Singh et al. we used the training set (RCV1-v2, ECAT) for training and testing. The results reported by Singh are not the same but there are some similarities. First, the word-stems “shar” and “clos” were

[49]Meanwhile, this approach was realised for Cross by Stephan Ewen after discussing the shortcomings described here with him.

	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
Rank 1	compan 5578	market	shar	day	lead 869
Rank 2	shar 5566	clos	day	lead	group 830
Rank 3	million 5302	day	year	year	year 779
Rank 4	year 5184	shar	million	peopl	ton 705
Rank 5	market 5146	trad	lead	wednesday	peopl 691

Table 4: Top five features for five SFS iterations using the RCV1-v2 training dataset for test and training (ECAT), adding one feature each iteration. For iteration 1 and 5 we added the approximated gain in log likelihood.

	Iteration 1	Iteration 2
Rank 1	compan 5578	econom 1367
Rank 2	shar 5566	deficit 1083
Rank 3	million 5302	inflat 1001
Rank 4	year 5184	growth 934
Rank 5	market 5146	gdp 8837

Table 5: Top five features for two SFS iterations using the RCV1-v2 training dataset for test and training (ECAT). Now we added the five best features after the first iteration.

commonly ranked high. Additionally the feature "econom", which was ranked top 1 with a gain of 283.7 for Singh et al. has an almost equal gain of 281.87 in our results. Furthermore, similar to Singh et al. the ranking changes after adding a feature, since the gain in log likelihood reduces for correlated features. One partial explanation for the difference is that we did not implement retraining and therefore the results after the first iteration must be different. A second explanation is that Singh et al. neither explain the actual implementation of their training and evaluation step, nor they explain which constant intercept term they used, which both makes a difference in the computed gain. To understand why the gains are higher in our results we analysed the features "compan" and "econom": "Compan" occurs in 6041 test records, of which 476 are positive and 5565 are negative. This makes it a good distinguishing feature and by adding it we can improve the log likelihood of most records by almost one, which explains the high gain. The feature "econom", however, occurs only 3149 times with 1331 positive and 1818 negative records, leading to an overall smaller gain in likelihood. This analysis explains our results, although it leaves the question open why features such as "compan" are ranked lower for Singh et al.

During our analysis we found that we receive almost identical results to Singh et al. after we added the best 5 features. The top 5 features of the second iteration in Table 5 are equal to the first ranking of Singh et al., except "growth" was added before "gdp". Due to a lack of time and information on the implementation of Singh we could not further investigate the differences and instead continue with a second approach to evaluate our results.

Comparison with Liblinear. As second evaluation we compare our ranking to

the results of an embedded feature selection method. We use the classification library Liblinear [14] to train a sparse model using L1-regularized Logistic Regression for the rcv1.binary dataset[50]. Liblinear selected 184 out of 47,236 features for a regularization parameter of 0.2[51]. Despite the low number of features the accuracy for the test dataset was still as good as 92.6, indicating that the few selected features are highly relevant. We now sorted the resulting features by the absolute value of their coefficient to obtain a ranking. In general such ranking can not be interpreted as a ranking of importance, however, since all feature values are encoded in a range from zero to one the ranking can be a good indicator for importance. We now applied SFS to the same dataset, with three iterations, adding one feature at a time. The results in Table 6 show that all features ranked top 10 from SFS were also selected by Liblinear. Furthermore, highly ranked features are mostly in the upper quarter of the Liblinear ranking.

To conclude, although we could not reproduce the exact results of Singh et al., our experiments suggest that the ranking produced by our SFS implementation gives valuable insights into the relevancy of individual features. As a future work we would like to incorporate retraining after each iteration to evaluate the predictive accuracy when training a complete model using SFS.

	Iteration 1	Iteration 2	Iteration 3
Rank 1	trad 4	peopl 8	compan 2
Rank 2	day 7	kill 17	net 40
Rank 3	week 13	polit 51	day 7
Rank 4	clos 16	party 12	shar 10
Rank 5	market 18	day 7	play 11
Rank 6	point 5	lead 122	digest 1
Rank 7	peopl 8	elect 38	polit 51
Rank 8	lead 122	compan 2	party 12
Rank 9	high 163	milit 21	saturday 31
Rank 10	futur 39	peac 9	lead 122

Table 6: Top 10 features for the first five SFS iterations using the rcv1.binary training and test dataset. The number denotes the ranking of the feature within the 184 features that were selected from Liblinear when using L1-Regularization with a regularization parameter 0.2. The dataset uses CCAT and ECAT as positive and GCAT and MCAT as negative classes.

[50]The dataset is a binary variant of the predecessor of RCV1-v2 and can be downloaded under http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html#rcv1.binary

[51]For Liblinear, a low regularization value actually means high regularization and vice versa.

5 Conclusion & Future Work

The goal of this thesis was to test the claim of Stratosphere that the improvements over Hadoop and MapReduce such as the more expressive programming model significantly ease the implementation of many complex data processing tasks and yield a better performance. We reached this goal for one use case with three main contributions.

First, we implemented SFS for Stratosphere using PACT, which can be seen as a starting point for more algorithms from the area of predictive analytics. The source code is available at https://github.com/andrehacker/logreg.

As a second contribution, we evaluated the programming models to test the first part of the claim. We used the method to define SFS as an abstract scalable algorithm first, to see how well it can be mapped to the programming models afterwards. This method gave us a good understanding of an ideal programming model for SFS and revealed a larger number of issues for both systems, the most important being summarized in Table 7. While PACT improves the expressiveness significantly, it lacks a good solution for closures, which turned out to be the first of two main weak spots we identified.

As a third contribution we conducted a series of experiments to compare the performance of both systems and to test the second part of the claim. The results are summarized in Table 8. While Hadoop demonstrated a solid performance in all experiments, Stratosphere showed impressive results in some experiments, but disappointing results in others. We attribute this to the inefficiency of Cross and the limitations of the PACT Compiler, which can be seen as a second main weak spot.

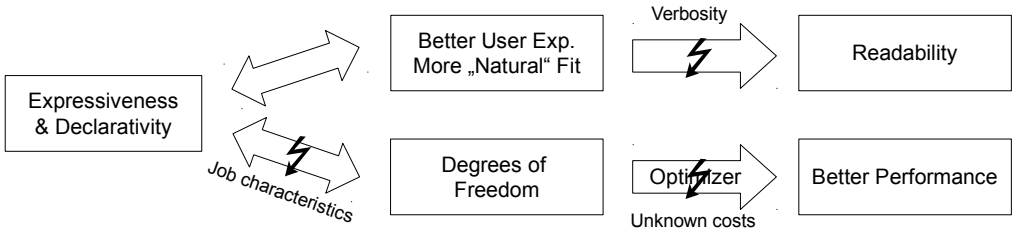

Figure 22: The implications of expressiveness and declarativity for Stratosphere and SFS. The flash indicates where we encountered exceptions to the rule.

Our mixed results suggest that the claim of Stratosphere can only be partially confirmed for SFS. At the same time the results emphasize the potential of Stratosphere. Figure 22 takes up our initial expectations from Figure 8 and adds where they could be not fulfilled. This leads us to two conclusions, that can be derived from the upper and lower part of Figure 22.

First, our experiences suggest that expressiveness highly correlates with the user experience: The benefits of well fitting constructs surpass the added complexity, whereas a lack of expressiveness highly dampens the user experience. We also observed that the readability suffers from verbosity since both implementations were far less concise than our scalable algorithm. We conclude that Stratosphere is on the right path to improve the overall user experience, assuming that good solutions are found for the missing constructs and more concise programming interfaces become available.

Second, the experiments revealed that it is a long and stony path from expressiveness to better performance: The algorithm may leave only few degrees of freedom or the system may not be able to correctly use its freedom, which might even result in worse performance. We observed instead that efficient implementations and reduced overhead are equally important for better performance. This highlights the need for a strategy to overcome the PACT Compiler limitations. It also highlights the potential to further increase the efficiency.

As a last conclusion related to SFS, we saw that it is a good method to quickly rank a large number of features, but our experiments raised doubts whether it is a well suited method to build a complete model for high dimensional data.

Programming Model Evaluation	
Hadoop	**Stratosphere**
+ Map and Reduce fit well	+ Added expressiveness: Single job, iterations
− Limited expressiveness: Two jobs needed, no iterations, distributed cache underspecified	− Limited expressiveness for closures required workarounds
− Verbosity	− Verbosity
	− Immaturity (Bugs, Traps)
	− No counters, few profiling tools
	± Schemaless record model has pros and cons

Table 7: Summary of the most important findings of the programming model evaluation.

Experimental Evaluation	
Hadoop	**Stratosphere**
+ Good scaleout	+ Mostly good scaleout (*)
± Good speedup, but only for larger input	+ Mostly good speedup (*)
+ Scaled to high dim. and larger base models	+ Mostly faster (*), efficient, low overhead
− Mostly slower (*), high overhead	− Problems for high dim. and larger base models
	− Compiler problems, required hints

Table 8: Important results of the experimental evaluation. (*) denotes that this does hold for the cases with high dimensionality and larger base models.

Future work. In the following we briefly discuss potential areas for future work. We begin with work related to evaluation, continue with work related to the system and end with the machine learning specific topics. This should be more understood as a summary since most issues were already discussed.

As a first future work we propose to extend the evaluation to more systems and high level languages. SFS could be implemented for example on Spark and in the Scala interface of Stratosphere. A second area is to extend the evaluation to more algorithms. An obvious example would be to implement an iterative training algorithm in Stratosphere, which would be also required to extend SFS by retraining. For each algorithm we propose to analyse several characteristics, in a way as done in this thesis: First, analyse which constructs are required and how well does the programming model support them. Second, analyse which degrees of freedom are theoretically[52] possible for the execution of the algorithm and how many does the system actually have after the algorithm was implemented. Third, analyse which optimizations were applied and how much do the additional degrees of freedom contribute to better performance. Since the methodology was developed during this thesis and we had limited time we could not analyse all mentioned aspects. This thesis is only a first step in this direction. Particularly we did not analyse the large space of theoretical degrees of freedom.

Our results show the importance of several issues for Stratosphere. The first is to investigate into strategies to overcome the limitations of the optimizer. Our discussion at the end of Chapter 4.3 suggest that the computation of costs in the presence of custom UDFs as well as dynamic optimization are still active areas of research. To lay the groundwork for the optimizer improvements, it might be benefitial to categorize or even formalize[53] all degrees of freedom as well as the information necessary to make sound optimization decisions. A second main question is how to implement support for closures. Our experiences suggest that it is best to look at this from the perspective of a more concise high level language, where the notion of closures arises more naturally, and to take into account a wider range of algorithms to gather the requirements.

There is a wide area of potential research at the intersection of Parallel Data Processing Systems and machine learning or predictive analytics. First, a parallel generator for supervised learning experiments is required, as we discussed briefly in Chapter 4.3[54]. A further question is how Stratosphere shall support machine learning tasks. In

[52]It is interesting to note that the efficiency of the system, such as sophisticated algorithms to implement joins and exploitation of new hardware architectures, can be seen as theoretical degrees of freedom and thus will be considered.

[53]Zhou et al. formalized the query optimizer of SCOPE in [46] which goes in a similar direction.

[54] The Myriad toolkit developed at the TU Berlin might be adapted for this purpose. See https://github.com/TU-Berlin-DIMA/myriad-toolkit/wiki.

chapter Related Work we discussed approaches ranging from low level interfaces for scalable matrix arithmetics to very high level frameworks such as MLBase. The complexity and the wild variety of algorithms suggest that a cooperation with a machine learning oriented research group should be intensified[55].

[55]The Data Analytics Laboratory already has this target. See http://www.analytics.tu-berlin.de/

A Appendix

Constructor Parameter	Description
String inputPathTrain	Path of training input file (libsvm format), typically on hdfs.
String inputPathTest	Path of evaluation input file (libsvm format).
boolean isMultilabelInput	True, if the input files are multi-class files (i.e. each record may have multiple labels), false otherwise.
int positiveClass	Id of the class that will be used as positive class in a one-versus-all classifier (only relevant for multi-class input).
String outputPath	Output path of the whole job, typically hdfs.
int numFeatures	Total number of features, or to be more concise, highest feature id. Required to construct sparse vectors.
double newtonTolerance	Tolerance for newton raphson convergence, e.g. 0.000001. If the change in trained coefficient is smaller, convergence is assumed.
int newtonMaxIterations	Maximum number of newton raphson iterations, e.g. 5.
double regularization	L2-regularization penalty term. Set to 0 for no regularization and increase for higher regularization. A high value keeps the coefficient smaller.
boolean runLocal	False, to execute the job regularily on a cluster. False, to execute the job in local mode (via LocalExecutor).
String confPath	Path to the config directory of Stratosphere. Required to start the job.
String jarPath	Local path of the jar file containing the job. Required to start the job.

Table 9: Constructor arguments of the SFS driver class for Stratosphere. The arguments for the PACT job are almost the same.

Listing 1: User defined function for Evaluation Map in MapReduce.

```
// The signature of the UDF defines the input and output schema
public class SFOEvalMapper extends Mapper<LongWritable, Text, IntWritable,
    DoubleWritable> {

  private static IntWritable outputKey = new IntWritable();
  private static DoubleWritable outputValue = new DoubleWritable();

  private boolean isMultilabelInput;
  private int positiveClass;
  private int numFeatures;
  private String trainOutputPath;
  private boolean collectDatasetStats;

  private IncrementalModel baseModel;
  List<Double> coefficients;
```

```
@Override
protected void setup(Context context) throws IOException, InterruptedException {
  super.setup(context);

  this.isMultilabelInput = Boolean.parseBoolean(
    context.getConfiguration().get(SFOEvalJob.CONF_KEY_IS_MULTILABEL_INPUT));
  this.positiveClass = Integer.parseInt(
    context.getConfiguration().get(SFOEvalJob.CONF_KEY_POSITIVE_CLASS));
  this.numFeatures = Integer.parseInt(
    context.getConfiguration().get(SFOEvalJob.CONF_KEY_NUM_FEATURES));
  this.trainOutputPath =
      context.getConfiguration().get(SFOEvalJob.CONF_KEY_TRAIN_OUTPUT);
  this.collectDatasetStats = Boolean.parseBoolean(
    context.getConfiguration().get(SFOEvalJob.CONF_KEY_COLLECT_DATASET_STATS));

  baseModel = SFOToolsHadoop.readBaseModel(context.getConfiguration());

  coefficients =
      SFOToolsHadoop.readTrainedCoefficients(context.getConfiguration(),
      numFeatures, trainOutputPath);
}

@Override
public void map(LongWritable ignore, Text line, Context context) throws
    IOException, InterruptedException {

  Vector xi = new RandomAccessSparseVector(numFeatures);
  int y;
  if (isMultilabelInput) {
    y = LibSvmVectorReader.readVectorMultiLabel(xi, line.toString(),
        positiveClass);
  } else {
    y = LibSvmVectorReader.readVectorSingleLabel(xi, line.toString());
  }

  // Compute the log-likelihood for the current record using the base model
  double piBase = LogRegMath.predict(xi, baseModel.getW(),
      SFOGlobalSettings.INTERCEPT);
  double llBase = LogRegMath.logLikelihood(y, piBase);

  // Compute the gain in log-likelihood for all non-zeros in this record
  for (Vector.Element feature : xi.nonZeroes()) {
    int dim = feature.index();
    if (! baseModel.isFeatureUsed(dim)) {
      Double coefficient = coefficients.get(dim);
      // Features we did not have in our training data won't have a coefficient
      if (coefficient != null) {
        // Extend the base model by the current coefficient, revert afterwards
        baseModel.getW().set(dim, coefficient);
```

```
          double piNew = LogRegMath.logisticFunction(xi.dot(baseModel.getW()) +
              SFOGlobalSettings.INTERCEPT);
          baseModel.getW().set(dim, 0d);

          double llNew = LogRegMath.logLikelihood(y, piNew);

          outputKey.set(feature.index());
          outputValue.set(llNew - llBase);
          context.write(outputKey, outputValue);
        }
      }
      if (collectDatasetStats) {
        context.getCounter(SFOEvalJob.SFO_EVAL_COUNTER.NUM_NON_ZEROS).increment(1);
      }
    }
  }
}
```

Listing 2: User defined function for Evaluation Map in PACT.

```
public class EvalComputeLikelihoods extends CrossStub {

  public static final int IDX_INPUT1_INPUT_RECORD = 0;
  public static final int IDX_INPUT1_LABEL = 1;

  public static final int IDX_INPUT2_BASEMODEL = 0;
  public static final int IDX_INPUT2_TRAINED_COEFFICIENTS = 1;

  public static final int IDX_OUT_DIMENSION = EvalSumLikelihoods.IDX_DIMENSION;
  public static final int IDX_OUT_LL_BASE = EvalSumLikelihoods.IDX_LL_BASE;
  public static final int IDX_OUT_LL_NEW = EvalSumLikelihoods.IDX_LL_NEW;

  private boolean baseModelAndCoefficientsCached = false;
  private IncrementalModel baseModel = null;
  Vector coefficients = null;

  // It is a common optimization to reuse PactRecord objects instead of creating
      them at the udf
  // This is, however, not intuitive and adds verbosity
  private final PactRecord recordOut = new PactRecord(3);

  @Override
  public void open(Configuration parameters) throws Exception {
        // Dangerous: When using iterations, the udf instance will be reused
        // and we have to make sure to deserialize again.
        baseModelAndCoefficientsCached = false;
  }

  // The system has to create a new copy of baseModelAndCoefficients for every
      call to guaranty that it is the same for every call
```

```
  // If the system would pass a reference the udf could modify it
  @Override
  public void cross(PactRecord testRecord, PactRecord baseModelAndCoefficients,
      Collector<PactRecord> out) throws Exception {

    int y = testRecord.getField(IDX_INPUT1_INPUT_RECORD,
        PactInteger.class).getValue();
    Vector xi = testRecord.getField(IDX_INPUT1_LABEL,
        PactVector.class).getValue();

    // Manual optimization: Cache base model and trained coefficients
    if (!baseModelAndCoefficientsCached) {
      baseModel = baseModelAndCoefficients.getField(IDX_INPUT2_BASEMODEL,
          PactIncrementalModel.class).getValue();
      coefficients =
          baseModelAndCoefficients.getField(IDX_INPUT2_TRAINED_COEFFICIENTS,
          PactVector.class).getValue();
      baseModelAndCoefficientsCached = true;
    }

    // Compute the log-likelihood for the current record using the base model
    double piBase = LogRegMath.predict(xi, baseModel.getW(),
        SFOGlobalSettings.INTERCEPT);
    double llBase = LogRegMath.logLikelihood(y, piBase);

    // Compute the gain in log-likelihood for all non-zeros in this record
    for (Vector.Element feature : xi.nonZeroes()) {
      int dim = feature.index();
      if (! baseModel.isFeatureUsed(dim)) {
        double coefficient = coefficients.get(dim);
        // Features with coefficient 0 were either not in our training data
        //  or were considered to be not important. We don't need to evaluate
            these features
        if (coefficient != 0) {
          // Extend the base model by the current coefficient, revert afterwards
          baseModel.getW().set(dim, coefficient);
          double piNew = LogRegMath.logisticFunction(xi.dot(baseModel.getW()) +
              SFOGlobalSettings.INTERCEPT);
          baseModel.getW().set(dim, 0d);

          double llNew = LogRegMath.logLikelihood(y, piNew);

          recordOut.setField(IDX_OUT_DIMENSION, new PactInteger(dim));
          recordOut.setField(IDX_OUT_LL_BASE, new PactDouble(llNew - llBase));
          out.collect(recordOut);
        }
      }
    }
  }
}
```

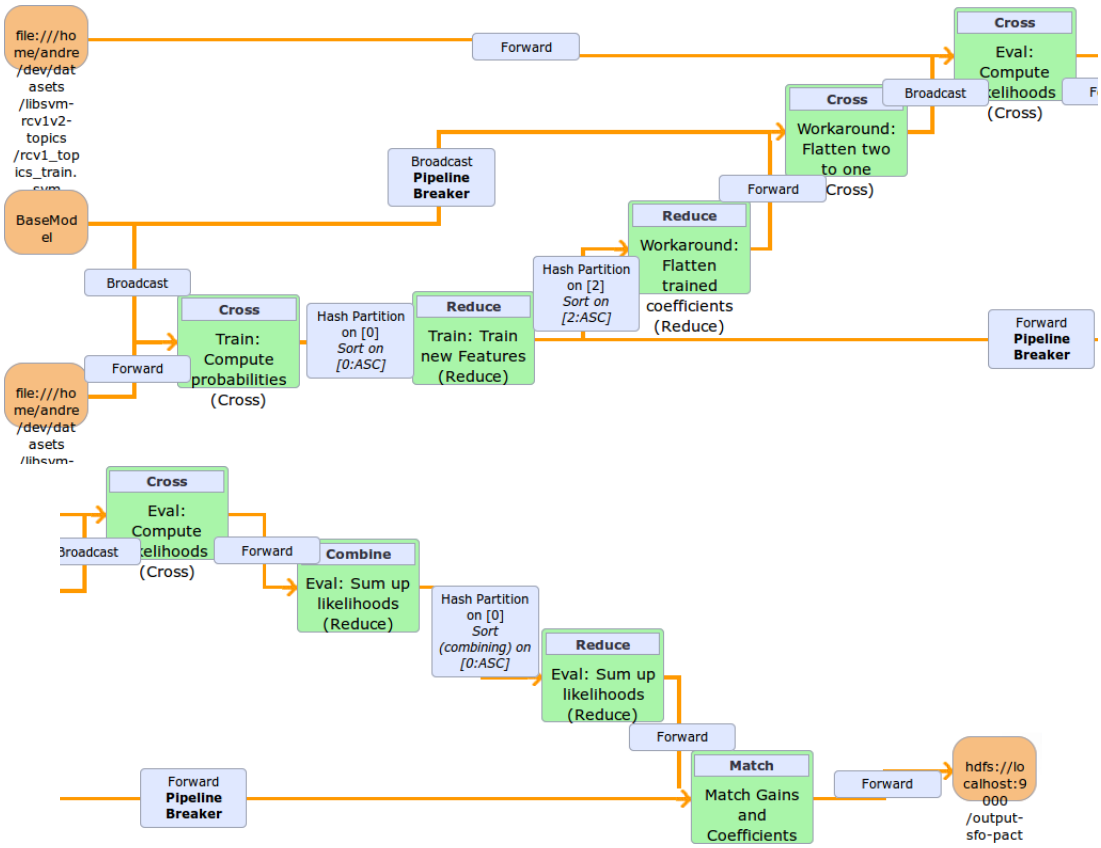

Figure 23: Stratosphere execution plan for SFS, without iterations, splitted into two parts to make it fit on the page.

References

[1] Apache Hadoop, http://hadoop.apache.org/, accessed 2013, Aug. 26.

[2] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke. MapReduce and PACT - Comparing Data Parallel Programming Models. *Proceedings of Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 25–44, 2011.

[3] M. Anderson, D. Antenucci, and V. Bittorf. Brainwash: A Data System for Feature Engineering. *CIDR*, 2013.

[4] D. Battré, S. Ewen, and F. Hueske. Nephele / PACTs : A Programming Model and Execution Framework for Web-Scale Analytical Processing. *Proceedings of the 1st ACM symposium on Cloud computing*, 3(1-2):119–130, 2010.

[5] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. ASTERIX: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, Mar. 2011.

[6] M. Beyer and D. Laney. The Importance of «Big Data»: A Definition. *Stamford, CT: Gartner*, 2012.

[7] C. M. Bishop. *Pattern Recognition and Machine Learning*, volume 4 of *Information science and statistics*. Springer, 2006.

[8] V. Borkar, Y. Bu, and M. Carey. Declarative Systems for Large-Scale Machine Learning. *IEEE Data Eng. ...*, 2012.

[9] R. Caruana, N. Karampatziakis, and A. Yessenalina. An empirical evaluation of supervised learning in high dimensions. *Proceedings of the 25th International Conference on Machine Learning (2008)*, pages 96–103, 2008.

[10] J. Cohen, B. Dolan, and M. Dunlap. MAD skills: new analysis practices for big data. *Proceedings of the ...*, 2009.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, page 10, Dec. 2004.

[12] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):1–26, 1992.

[13] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *Proceedings of the VLDB ...*, 2012.

[14] R. Fan, K. Chang, and C. Hsieh. LIBLINEAR: A library for large linear classification. *The Journal of Machine ...*, 2008.

[15] A. Genkin, D. D. Lewis, and D. Madigan. Large-Scale Bayesian Logistic Regression for Text Categorization. *Technometrics*, 49(3):291–304, 2007.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29, 2003.

[17] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce, 2011.

[18] J. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 1988.

[19] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 2003.

[20] A. Halevy, P. Norvig, and F. Pereira. The Unreasonable Effectiveness of Data, 2009.

[21] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, volume 27 of *Springer Series in Statistics*. Springer, 2009.

[22] F. Hueske, M. Peters, A. Krettek, and M. Ringwald. Peeking into the Optimization of Data Flow Programs with MapReduce-style UDFs. 2013.

[23] K. Koh, S.-J. Kim, and S. Boyd. An interior-point method for large-scale l1-regularized logistic regression. *Journal of Machine Learning Research*, 8(8):1519–1555, 2007.

[24] P. Komarek and A. W. Moore. Making logistic regression a core data mining tool with TR-IRLS, 2005.

[25] A. P. Konda. Feature Selection in Enterprise Analytics: A Demonstration using an R-based Data Analytics System. *Proceedings of the VLDB ...*, 2013.

[26] T. Kraska, A. Talwalkar, J. Duchi, and R. Griffith. MLbase: A Distributed Machine-learning System. *CIDR*, 2013.

[27] S. S.-i. Lee, H. Lee, P. Abbeel, and A. Y. A. Ng. Efficient L 1 Regularized Logistic Regression. *Compute*, 21(1):401, 2004.

[28] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li. RCV1: A New Benchmark Collection for Text Categorization Research. *Journal of Machine Learning Research*, 5:361–397, 2004.

[29] J. Lin and C. Dyer. Data-Intensive Text Processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.

[30] J. Lin and A. Kolcz. Large-scale machine learning at twitter. *Proceedings of the 2012 international conference on Management of Data SIGMOD 12*, page 793, 2012.

[31] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data : The next frontier for innovation , competition , and productivity. *McKinsey Global Institute*, 364(May):156, 2011.

[32] T. Minka. A comparison of numerical optimizers for logistic regression. *Unpublished draft*, 2003.

[33] C. Olston, B. Reed, and U. Srivastava. Pig latin: a not-so-foreign language for data processing. *... on Management of data*, 2008.

[34] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, page 165, New York, New York, USA, 2009. ACM Press.

[35] R. R Development Core Team. R: A Language and Environment for Statistical Computing, 2011.

[36] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using Hadoop on a cluster. *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing HotCDP 12*, pages 1–5, 2012.

[37] S. Sakr, A. Liu, and A. Fayoumi. The Family of MapReduce and Large Scale Data Processing Systems. *arXiv preprint arXiv:1302.2966*, 2013.

[38] S. Singh, J. Kubica, S. Larsen, and D. Sorokina. Parallel Large Scale Feature Selection for Logistic Regression. *SIAM International Conference on Data Mining (SDM)*, pages 1172–1183, 2009.

[39] M. Stonebraker. The Case for Shared Nothing. *Database Engineering Bulletin*, 9(1):4–9, 1986.

[40] G. Sussman and G. S. Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 1998.

[41] J. S. Ward and A. Barker. Undefined By Data: A Survey of Big Data Definitions. page 2, Sept. 2013.

[42] D. Warneke and O. Kao. Nephele: Efficient Parallel Data Processing in the Cloud. In *Proceedings of the 2nd Workshop on ManyTask Computing on Grids*

and Supercomputers MTAGS 09 November 2009 Portland OR USA, MTAGS '09, pages 1–10. ACM, 2009.

[43] S. Webb. Introducing the Webb spam corpus: Using email spam to identify Web spam automatically.

[44] S. A. Yanpei Chen, R. H. Katz, Y. Chen, S. Alspaugh, and R. Katz. Interactive Query Processing in Big Data Systems: A Cross Industry Study of MapReduce Workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[45] M. Zaharia and M. Chowdhury. Spark: cluster computing with working sets. *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[46] J. Zhou, N. Bruno, M. Wu, and P. Larson. SCOPE: parallel databases meet MapReduce. *The VLDB Journal*, 2012.

List of Abbreviations

DAG Direct acyclic graph

DOP Degree of parallelism

HDFS Hadoop Distributed File System

JVM Java Virtual Machine

PACT Parallelization Contract

SFS Scalable feature selection

UDF User defined function

List of Figures

List of Tables