Scaling Data Mining in Massively Parallel Dataflow Systems

vorgelegt von
Dipl.-Inf. Sebastian Schelter
aus Selb

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

genehmigte Dissertation

Promotionsausschuss:

Gutachter: Prof. Dr. Volker Markl
Database Systems and Information Management Group, TU Berlin
Prof. Dr. Klaus Robert Müller
Machine Learning and Intelligent Data Analysis Group, TU Berlin
Prof. Reza Zadeh, PhD
Institute for Computational & Mathematical Engineering, Stanford University

Berlin 2015
D 83

Acknowledgements

I would like to express my gratitude to the people who accompanied me in the course of my studies and helped me shape this thesis.

I'd like to first and foremost thank my advisor Volker Markl. He introduced me to the academic world, gave me freedom in choosing my research topics, and motivated me many times to leave my comfort zone and take on the challenges involved in conducting a PhD. Furthermore, I want to express my appreciation to Klaus-Robert Müller and Reza Zadeh for agreeing to review this thesis.

Very importantly, I'd like to thank my colleagues Christoph Boden, Max Heimel, Alan Akbik, Stephan Ewen, Fabian Hüske, Kostas Tzoumas, Asterios Katsifodimos and Isabel Drost at the Database and Information Systems Group of TU Berlin, who helped me advance my research through collaboration, advice and discussions. I also enjoyed working with the DIMA students Christoph Nagel, Janani Chakkaradhari, and Till Rohrmann who conducted excellent master thesis in the context of my line of research. Furthermore, I'd like to thank my colleagues Mikio Braun, Jérôme Kunegis, Jörg Fliege and Mathias Peters for ideas and discussions in the research projects in which we worked together. I also thank Zeno Gantner for reviewing my work related to recommender systems.

A special thanks goes out to Ted Dunning, who showed me the beauty of Linear Algebra and initially inspired me to work on data mining and contribute to Apache Mahout.

Finally, I'd like to thank my colleagues Douglas Burdick, Berthold Reinwald, Alexandre Evfimievski and Matthias Böhm at IBM Research Almaden, as well as my colleagues Venu Satuluri, Reza Zadeh, Ajeet Grewal, Siva Gurumurthy, Aneesh Sharma and Volodymyr Zhabiuk from Twitter. It was a pleasure to work at your side and learn from you.

Dedicated to Lea, who always had my back during the highs and lows of the PhD studies.

Abstract

This thesis lays the ground work for enabling scalable data mining in massively parallel dataflow systems, using large datasets. Such datasets have become ubiquitous. We illustrate common fallacies with respect to scalable data mining: It is in no way sufficient to naively implement textbook algorithms on parallel systems; bottlenecks on all layers of the stack prevent the scalability of such naive implementations. We argue that scalability in data mining is a multi-leveled problem and must therefore be approached on the interplay of algorithms, systems, and applications. We therefore discuss a selection of scalability problems on these different levels.

We investigate *algorithm-specific scalability aspects* of collaborative filtering algorithms for computing recommendations, a popular data mining use case with many industry deployments. We show how to efficiently execute the two most common approaches, namely neighborhood methods and latent factor models on MapReduce, and describe a specialized architecture for scaling collaborative filtering to extremely large datasets which we implemented at Twitter. We turn to *system-specific scalability aspects*, where we improve system performance during the distributed execution of a special class of iterative algorithms by drastically reducing the overhead required for guaranteeing fault tolerance. Therefore we propose a novel optimistic approach to fault-tolerance which exploits the robust convergence properties of a large class of fixpoint algorithms and does not incur measurable overhead in failure-free cases. Finally, we present work on an *application-specific scalability aspect* of scalable data mining. A common problem when deploying machine learning applications in real-world scenarios is that the prediction quality of ML models heavily depends on hyperparameters that have to be chosen in advance. We propose an algorithmic framework for an important subproblem occuring during hyperparameter search at scale: efficiently generating samples from block-partitioned matrices in a shared-nothing environment.

For every selected problem, we show how to execute the resulting computation automatically in a parallel and scalable manner, and evaluate our proposed solution on large datasets with billions of datapoints.

Zusammenfassung

Diese Doktorarbeit befasst sich mit den technischen Grundlagen, die notwendig sind, um skalierbares Data Mining auf heutigen, großen Datensätzen mithilfe massiv-paralleler Datenflusssysteme zu ermöglichen. Sie beschreibt gängige Fehlannahmen im Bezug auf skalierbares Data Mining. Es reicht in keinster Weise aus, Algorithmen in ihrer herkömmlichen Formulierung auf parallelen Systemen zu implementieren. Engpässe auf allen Schichten verhindern die Skalierbarkeit solch naiver Implementierungen. Die Arbeit legt dar, dass Skalierbarkeit im Data Mining ein mehrschichtiges Problem ist und daher im Zusammenspiel von Algorithmen, Systemen und Anwendungen angegangen werden muss. Deshalb befasst sich diese Arbeit mit einer Auswahl von Skalierbarkeitsproblemen in verschiedenen Schichten.

Die Arbeit untersucht *algorithmus-spezifische Skalierbarkeitsaspekte* von 'Collaborative Filtering'-Algorithmen zur Empfehlungsberechnung, eine beliebte Data Mining-Technik, die häufig in der Industrie Anwendung findet. Es wird dargelegt, wie die beiden vorherrschenden Ansätze, 'Neighborhood Methods' und 'Latent Factor Models' mithilfe des MapReduce Paradigmas skaliert werden können. Desweiteren beschreibt die Arbeit ein spezialisierte Architektur, die bei Twitter implementiert wurde, um Collaborative Filtering auf extrem große Datensätze anwenden zu können. Im Folgenden wird sich mit *system-spezischen Skalierbarkeitsaspekten* befasst: die Arbeit beschreibt, wie man die Systemleistung während der verteilten Ausführung einer speziellen Klasse iterativer Algorithmen verbessern kann, indem man den Mehraufwand drastisch reduziert, der für die Garantie von Fehlertoleranz notwendig ist. Die Arbeit führt einen neuartigen optimistischen Ansatz zur Fehlertoleranz ein, der die robusten Konvergenzeigenschaften einer großen Klasse von Fixpunktalgorithmen ausnutzt und während fehlerfreier Ausführung keinen messbaren Mehraufwand verursacht. Schlussendlich widmet sich die Arbeit einem *anwendungsspezifischen Skalierbarkeitsaspekt*. Ein gängiges Problem beim Einsatz von Anwendungen des Maschinellen Lernens ist, dass die Vorhersagequalität der Modelle häufig stark von Hyperparametern abhängt, die im Vorhinein gewählt werden müssen. Die Arbeit beschreibt ein algorithmisches Framework für ein wichtiges Unterproblem bei der Suche nach Hyperparameter auf großen Datensätzen: die effiziente Generierung von Stichproben aus block-partitionierten Matrizen in verteilten Systemen.

Für jedes ausgewählte Problem legt die Arbeit dar, wie die zugrundeliegenden Berechnungen automatisch auf eine parallele und skalierbare Weise ausgeführt werden können. Die präsentierten Lösungen werden experimentell auf großen Datensätzen bestehend aus Milliarden einzelner Datenpunkte evaluiert.

Contents

1 Introduction

> *"These inventions gave rise, as is well known, to an industrial revolution, a revolution which altered the whole civil society; one, the historical importance of which is only now beginning to be recognised."*
>
> *—Friedrich Engels, 1844*

In recent years, the cost of acquiring and storing data of large volume has dropped significantly. The growing ability to process and analyze these datasets presents a multitude of previously unimaginable possibilities. Scientists can test hypotheses on data several orders of magnitude larger than before. Companies apply advanced data analysis for creating disruptive applications and services, often based on machine learning (ML) and graph mining approaches. Examples include relevance-based ranking of trillions of web pages [49] to improve search engines, self-driving cars [122], statistical speech recognition and machine translation learned from web-derived corpora [83] as well as realtime article recommendation in news aggregation platforms [46].

Due to size and complexity of the data, scaling-out the analysis to parallel processing platforms running on large, shared-nothing commodity clusters is popular [49]. In the last decade, MapReduce has emerged as a de-facto standard for complex, distributed data processing applications in such a setting. Although parallel relational database systems for shared-nothing architectures exist since the mid of the 1980s, the conceivably simpler and in many cases less performant family of MapReduce-based systems have become the dominant choice for many large-scale data analysis cases. This popularity stems from several reasons. Analogous to parallel databases, MapReduce relieves the programmer from having to handle machine failures and concurrency, and automatically executes the user's program in parallel on a cluster. Yet, MapReduce-based systems typically have a lower cost of ownership compared to parallel databases, given that they aim at datacenters built from many low-end commodity servers instead of a few high-end servers. Moreover, in contrast to classical database systems, MapReduce-based systems are independent of the underlying storage system, which makes them a highly flexible tool and the prime choice for "extract-transform-load" and other read-once workloads [50, 165].

An aspect that makes MapReduce attractive for data mining applications is that MapReduce programs allow the programmer to express complex transformations much more

easily than SQL-based programming extensions such as PL/SQL. Aditionally, MapReduce-based systems have been designed for handling semi-structured data (typically in the form of key-value pairs) which occurs often in modern analysis use cases. Such data is often ingested from the web or social media. Such semi-structured data is traditionally very problematic for relational databases that rely on fixed, well-defined schemata. Yet, MapReduce alone does not pose a solution for scaling data mining applications to today's datasets. Scalable data mining is still a very hard problem in real-world applications and an active area of research [2, 76, 89, 108, 117, 120, 126, 163], for a variety of reasons. Most importantly, existing data mining algorithms have been designed to minimize CPU cycles on a classical von Neumann architecture, often with the assumption of quick random access to the input data. In the distributed shared-nothing setting however, the input data is always partitioned across several machines and performance is typically reached by minimizing inter-machine communication rather than CPU cycles, as network bandwidth is usually the most scarce resource in large clusters. Therefore, a large number of algorithms have to be reformulated and redesigned to scale in distributed architectures. Moreover, MapReduce has serious deficiencies, such as its lack of declarativity, inefficient performance when executing iterative computations, as well as missing operators for combining multiple datasets [18]. This situation currently leads to the substitution of MapReduce by a new generation of systems built on more advanced parallel processing paradigms [2, 7, 186]. Scalable data mining applications have to adapt to and embrace the possibilities offered by these new paradigms. A further challenge is that successfully putting scalable data mining applications into production today requires a strong background in data management and distributed systems, due to the current immaturity of parallel processing systems. This fact unnecessarily constrains the number of people that can conduct such tasks, as the majority of data analysts come from a background in statistics and machine learning [121]. Combining the aforementioned challenges in designing scalable data mining systems and applications leads to the core statement of the thesis:

Thesis Statement: *Scalability in data mining is a multi-leveled problem and must therefore be tackled on the interplay of algorithms, systems, and applications. None of these layers is sufficient to provide scalability on its own. Therefore, this thesis presents work on scaling a selection of problems in these layers. For every selected problem, we show how to execute the resulting computation automatically in a parallel and scalable manner. Subsequently, we isolate user-facing functionality into a simple user-defined function, to enable usage without a background in systems programming.*

1.1 Example: Scalability Bottlenecks in Principal Components Analysis

We use an example to illustrate the difficulty of scaling data mining in current MapReduce-based systems: *Principal Component Analysis* (PCA) gives insight to the internal structure of a dataset and constitutes a standard analysis technique in a wide range of scientific fields (e.g. signal processing, psychology, quantitative political science). PCA projects the data onto a lower dimensional space, such that the variance of the data is maximized [24]. Algorithm 1 shows the three steps that a textbook PCA implementation conducts to compute the first k principal components, the basis of this space. The dataset to analyze consists of N observations. A row x_i of X corresponds to the i-th observation. At first, the data has to be centered, i.e., we substract the vector $\frac{1}{N}\sum_i x_i$ of column means from every row of X (cf., line 1). Next, we compute the matrix multiplication $XX^\top$ and divide the result by the number of observations N to obtain the covariance matrix C (c.f., line 2). In line 3, we compute the first k eigenvectors of the covariance matrix C which correspond to the first k principal components of X.

Algorithm 1: Computing principal components of a matrix X

1	$X = X - \frac{1}{N}\sum_i x_i$	`# centering of the data`
2	$C = \frac{1}{N}XX^\top$	`# computation of the covariance matrix`
3	$P = \text{eigs}(C, k)$	`# computation of leading eigenvectors`

Imagine a real-world application, where we want to apply PCA to a large, sparse dataset such as a text corpus in 'bag-of-words' representation to gain insights about the relationships of term occurrences. It is not sufficient to implement the simple steps from Algorithm 1 on a MapReduce-based system such as Hadoop [11] to get a scalable and efficient PCA implementation. The reasons why it is hard to scale these three simple lines of PCA very well illustrate some major pain points about scaling-out data mining algorithms.

Algorithm Level: The first and most fundamental aspect is the scalability of the mathematical operations of the algorithm itself. Careful investigation of the operations applied reveals two scalability bottlenecks in the algorithm. The first bottleneck is caused by the centering of the data. A lot of modern data analysis tasks deal with very large and at the same time very sparse matrices (e.g., adjacency matrices in social network analysis [39] or matrices representing interactions between users and items in recommendation mining [182]). Often, the ratio of non-zero entries to the overall number of cells in the matrix is one-thousandth or less [39, 182]. These large matrices are efficiently stored by only materializing the non-zero entries. Applying the first step in PCA (line 1 in Algorithm 1), which centers the data, to such sparse matrices would have devastating consequences. The majority of zero entries become non-zero values and the materialized data size increases

by several orders of magnitude, thereby posing a serious scalability bottleneck. The second bottleneck is caused by the matrix multiplication $XX^\top$ conducted to compute the covariance matrix in line 2 of Algorithm 1. Its runtime is quadratic in the number of observations that our dataset contains (which is equal to the number of rows of the input matrix X). Computations with quadratic runtime quickly become intractable on large inputs. Therefore, a direct implementation of the textbook PCA algorithm will not scale[1]. In general, there are three dimensions of scaling algorithms. For the algorithm to scale to large amounts of input data ("data scaling"), its operations must preserve data sparsity, a data-parallel execution strategy on partitioned input must exist, and their runtime must be at most linear. If the size of the model of the algorithm grows proportionally to the cardinality of the input data, it is necessary to find a way to execute the algorithm efficiently with a partitioned model ("model scaling"). A third strategy is to train many small models on samples of the data and combine their predictions later.

System Level: Many data mining algorithms are iterative. Such algorithms start from an initial state and operate on the data to iteratively refine this state until a convergence criterion is met. In our example, we need a scalable way to compute the k first eigenvectors of the covariance matrix in the implementation of the *eigs* function in line 3 of Algorithm 1. The iterative Lanczos algorithm is the predominant scalable method to compute these eigenvectors [97]. A parallel processing system suitable for data mining must be able to efficiently run such iterative programs. During the execution of such computations in large clusters, guaranteeing fault tolerance is a mandatory task. Current systems for executing these iterative tasks typically achieve fault tolerance by checkpointing the result of an iteration as a whole [117, 120, 186]. In case of a failure, the system halts execution, restores a consistent state from a previously saved checkpoint and resumes execution from that point. This approach is problematic as it potentially incurs a high overhead. In order to tolerate machine failures, every machine has to replicate its checkpointed data to other machines in the cluster. Due to the pessimistic nature of the approach, this overhead is always incurred, even if no failures happen during the execution, as the checkpoints must be written in advance. It is highly desirable to find a mechanism that guarantees fault tolerance for iterative computations and at the same time minimizes checkpointing overhead in failure-free cases.

Application Level: Research on scalable data mining often exclusively focuses on finding efficient ways to train machine learning models. In order to apply these models in real applications in industry use cases however, additional work is necessary. Most models have hyperparameters that need to be carefully chosen as they heavily influence the quality of a model (a task known as parameter selection). Another important task is feature selection, where we aim to select a subset of the features that provides sufficient model quality. In PCA, we might for example want to eliminate columns from the data matrix

[1] randomized algorithms provide scalable PCA [84]

X that have negligible impact on the resulting principal components [109]. These tasks are solved via meta learning techniques such as cross-validation. These meta learning techniques typically first generate samples from the input data, and secondly train and evaluate models on these samples. Existing data parallel techniques for model training and evaluation have been proposed recently [26], but efficiently generating samples from large datasets is still problematic in many cases. This is especially true when the input dataset is physically represented as block-partitioned matrix [76, 89]. Automation of meta learning functionality in existing large scale data mining frameworks has therefore been recognized as a major challenge [108, 126], as data scientists usually spend a major amount of their time on selecting well-working features and hyperparameters.

The three pitfalls listed illustrate what we already mentioned in the thesis statement. Scalable data mining is such a hard problem because there is no 'silver bullet' solution to the problem. Scalability has to be achieved on many different levels at once, and all of these levels require methods from separate fields of computer science. For algorithm scalability, a careful choice of the mathematical operations involved is required. In order for systems to efficiently execute scalable algorithms, techniques from distributed systems, data management and system engineering have to be combined. On the application level, we must scale common use cases and the same time carefully design the implementations, such that they allow adaption without a background in systems programming.

1.2 Contributions and Greater Impact

In reference to the thesis statement, we present the following contributions in scaling a selection of problems on the level of algorithms, systems and applications:

We reformulate the two most prominent collaborative filtering algorithms for recommendation mining to scale on MapReduce. Our formulation includes an abstraction of different similarity measures and least squares solvers used in the algorithms, which allows for their adaption without a background in systems programming. We showcase linear scalability of the resulting algorithms with growing a user-base and perform an experimental evaluation on real-world and synthetic datasets with up to 5 billion data points.

On the systems side, we propose a novel optimistic recovery mechanism for distributed processing of a large class of iterative algorithms. This mechanism provides optimal failure-free performance with respect to the overhead required to guarantee fault tolerance. The main idea is to exploit the robustness of a large class of fixpoint algorithms, which converge from many different intermediate states. In case of a failure, a user-defined compensation function restores a valid intermediate state which allows the algorithm to converge to the correct solution afterwards. We evaluate our approach with simulated

failures during processing of datasets with billions of data points.
Further systems-based research resulted in the proposal and implementation of a distributed, asynchronous parameter server for matrix factorization. Here, we allow adaptation to different loss functions with a simple integrated user-defined function. The system supports scalable hyperparameter search using model multiplexing techniques which enable it to train and evaluate many models in a single run. We present an experimental evaluation on a matrix with more than 200 million rows and columns and 38.5 billion entries, which to the best of our knowledge is the largest dataset used so far in the academic literature on collaborative filtering.

Finally, we present work on scaling import application scenarios in data mining, such as cross-validation and ensemble learning. We focus on the subproblem of efficient sample generation from distributed, block-partitioned matrices. This functionality is an integral part of enabling meta learning in large-scale ML systems with block-aware operators. We propose an efficient, scalable single-pass algorithm for sample generation that allows for a high degree of parallelism. This algorithm supports distributed sampling with and without replacement as well as sample generation for common meta learning techniques. Furthermore, it scales linearly with the number of observations in the dataset and the number of sample matrices to generate. We evaluate our algorithm in three different systems, using matrices with up to 10 billion entries, and find that we outperform a linear algebra-based sample generation method by more than an order of magnitude.

Parts of this thesis have already been published as follows:

1. **Sebastian Schelter, Douglas Burdick, Berthold Reinwald, Alexandre Evfimievski, Juan Soto, Volker Markl:** *Efficient Sample Generation for Scalable Meta Learning*, 31st IEEE International Conference on Data Engineering (ICDE), 2015

2. **Sebastian Schelter, Venu Satuluri, Reza Zadeh:** *Factorbird – a Parameter Server Approach to Distributed Matrix Factorization*, Distributed Machine Learning and Matrix Computations workshop in conjunction with NIPS 2014

3. **Sebastian Schelter:** *Scaling Data Mining in Massively Parallel Dataflow Systems*, PhD Symposium at ACM SIGMOD, pp. 11-14, 2014

4. **Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, Daniel Warneke:** *The Stratosphere platform for big data analytics*, VLDB Journal, 23(6), pp. 939-964, 2014

5. **Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, Volker Markl:** *"All Roads Lead to Rome:" Optimistic Recovery for Distributed Iterative Data Processing*, ACM Conference on Information and Knowledge Management (CIKM), pp. 1919-1928, 2013

6. **Sebastian Schelter, Christoph Boden, Martin Schenck, Alexander Alexandrov, Volker Markl:** *Distributed Matrix Factorization with MapReduce using a series of Broadcast-Joins*, ACM Conference on Recommender Systems (RecSys), pp. 281-284, 2013

7. **Sebastian Schelter, Christoph Boden, Volker Markl:** *Scalable Similarity-Based Neighborhood Methods with MapReduce*, ACM Conference on Recommender Systems (RecSys), pp. 163-170, 2012

Early results of the thesis work have been contributed to open source software libraries and are already deployed in industry use cases. An implementation of our approach to scale out neighborhood-based recommendation mining on MapReduce has been contributed to the popular open source machine learning library *Apache Mahout* [13]. This implementation serves as the foundation for many large-scale, real-world recommender systems, e.g. place recommendation at *Foursquare* [67], scientific article recommendation in the academic network *Mendeley* [125] and diverse recommendations in *ResearchGate* [148], the world's largest social network for researchers. Our scalable matrix factorization-based recommender is part of the open source machine learning server *PredictionIO* [144] and forms the basis for the recommender platform *Kornakapi* [107] maintained by the digital advertising company *Plista* [143]. During our work on optimistic recovery, we implemented and contributed large parts of the iteration runtime of the distributed data processing engine *Apache Flink* [65]. Furthermore, *Twitter* is considering to conduct user-facing experiments with *Factorbird* [157], our proposed asynchronous matrix factorization system. Additionally, Twitter is currently issuing a patent on the resulting system.

We observe several adoptions of the thesis work in the research world. Our work on scaling similarity-based recommendation mining has been cited many times and is referenced in a survey about parallel and distributed collaborative filtering [100]. Our MapReduce-based matrix factorization serves as a baseline for the experimental evaluation of modern parallel processing platforms such as *Graphlab* [118] and *Spark* [186] that have been designed for high performance in distributed machine learning tasks. It is again cited in the survey [100]. An implementation of our distributed sample generation machinery for scalable meta learning has been contributed to the proprietary *SystemML* [76] platform for large-scale machine learning developed by IBM Research. Lastly, our work on optimistic recovery for iterative computations forms the basis for further research in this direction at TU Berlin [52].

For the future we expect that large scale machine learning systems will be enhanced with a comprehensive and scalable meta learning layer as envisioned by the *MLBase* project [108, 126]. We are confident that our sample generation machinery for blocked matrices can be an important part of such a layer. At the same time, we see that further projects are exploiting the robust, 'self-healing' nature of a lot of fixpoint algorithms on which we built our optimistic recovery mechanism. A prime example is Google's coming *ASyMP* system for distributed graph processing [113].

1.3 Outline and Summary of Key Results

The remainder of the thesis is structured as follows:

Chapter 2 introduces the background required for the thesis. We describe technological developments that lead to the emergence of massively parallel dataflow systems and briefly discuss why relational databases have been insufficient to meet these requirements. Next, we introduce the most prominent abstractions for massively parallel dataflow processing, namely MapReduce, Resilient Distributed Datasets (RDDs) and Parallelization Contracts (PACTs). On the algorithmic side, we present a selection of approaches from the fields of recommendation mining and graph mining. Major parts of this thesis focus on scaling out exactly these families of algorithms. Finally, we introduce basic mathematical methods that are used by the selected algorithms.

The following chapters 3 to 5 present the details of our contributions on scaling a selection of problems in the algorithm, system, and application layer as proposed in the thesis statement.

In Chapter 3, we present our work on *algorithm scalability*. We investigate the most prominent approaches to recommendation mining, similarity-based neighborhood methods and latent factor models, and describe how to reformulate and execute the underlying mathematical operations efficiently in a MapReduce-based system. We provide *a UDF-centric abstraction for a wide range of similiarity measures and least squares solvers* that allows the system to automatically parallelize the resulting computations. We experimentally evaluate our implementations on several datasets, consisting of up to 25 million users and 5 billion interactions, to showcase scalability to industry use cases. Additionally, we give an outlook on non-dataflow architectures for asynchronous distributed machine learning. We revisit the problem of training latent factor models for recommendation mining and present 'Factorbird', a prototypical system for factorizing large matrices with Stochastic Gradient Descent-based algorithms. We design this system to meet the following desiderata: scalability to tall and wide matrices with dozens of billions of non-zeros, extensibility to different kinds of models and loss functions, and adaptability to both batch and streaming scenarios. Factorbird uses a parameter server in order to scale to models that exceed the memory of an individual machine, and employs lock-free Hogwild!-style learning with a special partitioning scheme to drastically reduce conflicting updates. We *abstract the update step of Stochastic Gradient Descent into a simple UDF*, which allows for using Factorbird without a systems programming background. Furthermore, we discuss additional aspects of the design of our system such as how to efficiently grid search for hyperparameters at scale using so-called model multiplexing. We present experiments of Factorbird on a matrix built from a subset of Twitter's interaction graph, consisting of more than 38 billion non-zeros and about 200 million rows and columns, which is to the best of our knowledge the largest matrix on which factorization results

have been reported in the literature. Furthermore, we investigate the scalability of two network analysis algorithms for ranking vertices in large graphs, using different centrality measures. We compare scalability and expressitivity of the programming model of the two families of systems for these tasks. We find that dataflow systems subsume the vertex-centric model of the graph processing systems and at the same time offer a more flexible programming model that fits a wider range of algorithms.

Chapter 4 gives details on our systems-based work on the *efficient execution of iterative computations*. We focus on drastically reducing the overhead for guaranteeing fault tolerance during the distributed execution of fixpoint algorithms. Current systems typically achieve fault tolerance through rollback recovery. The principle behind this pessimistic approach is to periodically checkpoint the algorithm state. Upon failure, the system restores a consistent state from a previously written checkpoint and resumes execution from that point. We propose an optimistic recovery mechanism using so-called algorithmic compensations. Our method leverages the robust, self-correcting nature of a large class of fixpoint algorithms, which converge to the correct solution from various intermediate consistent states. In the case of a failure, we apply a *user-defined compensate function* that algorithmically creates such a consistent state, instead of rolling back to a previous checkpointed state. Our optimistic recovery does not checkpoint any state and hence achieves optimal failure-free performance with respect to the overhead necessary for guaranteeing fault tolerance. We illustrate the applicability of this approach for three wide classes of problems and provide an experimental evaluation using simulated failures during processing large datasets. In the absence of failures our optimistic scheme is able to outperform a pessimistic approach by a factor of two to five. In presence of failures, our approach provides fast recovery and outperforms pessimistic approaches in the majority of cases.

Our application-layer work on *scalable meta learning* is given in Chapter 5. We investigate how to scale out meta learning techniques such as cross-validation and ensemble learning, which are crucial for applying machine learning to real-world use cases. These techniques first generate samples from input data, and then train and evaluate machine learning models on these samples. For meta learning on large datasets, we illustrate why the efficient generation of samples becomes problematic, especially when the data is stored and processed in a block-partitioned representation. In order to solve this challenge, we present a novel, parallel algorithm for efficient sample generation from large, block-partitioned datasets in a shared-nothing architecture. This algorithm executes in a single pass over the data, and minimizes inter-machine communication through clever use of so-called 'skip-ahead' random number generators. The algorithm supports a wide variety of sample generation techniques through an *embedded user-defined sampling function*. We illustrate how to implement distributed sample generation for popular meta learning techniques such as hold-out tests, k-fold cross-validation, and bagging, using our algorithm. Furthermore, we detail how our approach enables distributed sampling with

replacement. Finally, we present an extensive experimental evaluation on datasets with billions of datapoints, using three different distributed data processing systems. Our results show that the runtime of our algorithm increases only linearly with the number of samples to generate and that we are able to generate a large number of samples efficiently with a single pass over the data. Furthermore, our algorithm beats a linear algebra-based sample generation approach by an order of magnitude, which indicates that distributed matrix operations do not fit the sample generation problem well and large-scale ML systems benefit from a dedicated sample generation machinery.

Chapter 6 summarizes and concludes the findings of the thesis, discusses gaps in the corresponding research landscape and proposes directions for future investigations.

2 Background

This chapter introduces the necessary preliminaries for the remainder of the thesis. We start with a brief historical overview on why a new generation of dataflow systems emerged (c.f., Section 2.1). Afterwards, we turn our focus to actual systems and paradigms for performing large scale data analysis. We first introduce distributed filesystems (c.f., Section 2.2), the dominant data store nowadays. Next, we describe the most prominent abstractions for distributed data processing (c.f., Section 2.3), which we use for scaling out data mining algorithms later. In the second part of this chapter, we introduce the two most important data mining use cases for this thesis: collaborative filtering for recommendation mining and the computation of centralities on large networks (c.f., Section 2.4). We introduce the models behind these methods as well as algorithms to learn them. The remaining parts of the thesis will investigate several aspects of scaling these algorithms in massively parallel dataflow systems.

2.1 From Relational Databases to Massively Parallel Dataflow Systems

We give a brief historic overview [33] of the developments in the database and distributed systems communities that led to today's massively parallel dataflow systems. With the growing need for enterprise data management, data in filesystems was more and more moved to specialized database systems. In 1970, Codd proposed the relational model [44], which gave rise to relational database management systems that quickly saw commercial adoption. The database community encountered large data processing challenges in enterprise data management: Enterprises established the collection of their historical business data in data warehouses in order to conduct reporting and business analytics. The encountered scalability issues during these tasks led to the development of software-based parallel database systems in the mid 1980s. These systems leveraged a so-called shared-nothing architecture, which means that they run on a cluster of autonomous machines, each with their own hardware, disk and operating system that only communicate over the network via message passing. They introduced divide-and-conquer parallelism based on hash-partitioning the data for storage and relational query processing tasks. Examples are Gamma [51], GRACE [69], Teradata [159] and DB2 Parallel Edition [17]. After the commercial adoption of parallel database systems in the mid 1990's, the database community considered parallel query processing solved and moved on to work on other problems.

The rise of the World Wide Web in the late 1990's produced a growing need to index and query the data available online. The companies working in this field considered database technology for this usecase, but found it to be neither well-suited nor cost-effective [35]. Web-scale search engines belong to the largest data management systems in the world, especially in terms of query volume. Yet, the ACID[1] paradigm which forms the basis for relational databases is a mismatch for the web search case which features only read-only queries and considers high availability to be much more important than consistency. Furthermore, the web search scenario led to new types of data and use cases that were very different from traditional enterprise data management. A factor for Google's success for example was their novel way of ranking of search results based on the link structure of the web [138], a large-scale graph processing task. Another example use case is personalized advertising based on users' search histories, a machine learning problem, which is also very different from traditional relational query processing. During work on such problems, Google developed a breed of novel storage and processing systems, aimed at cost-effective shared-nothing clusters built from commodity hardware, which, in hindsight, started the modern "Big Data" revolution. They created a distributed, web-scale storage system, called the Google File System (GFS) [75], together with MapReduce [49], a simple paradigm for distributed data processing. MapReduce is often considered to be "parallel programming for dummies", as the users only have to implement two simple functions to write their programs. Analogously to parallel database systems in the eighties, MapReduce leverages partitioned parallelism for distributed query processing on shared-nothing clusters. Google's publications about GFS and MapReduce [49, 75] gave birth to a free and open source variant of Google's technology stack in the form of Apache Hadoop MapReduce and the Hadoop Distributed File System (HDFS) [11], which was quickly adopted by industry. This was followed by the development of higher level languages, that are compiled to MapReduce programs, such as Pig [137] from Yahoo!, Jaql [23] from IBM and Hive [170] from Facebook. The open source Apache stack produced a striving ecosystem around Hadoop with projects such as Mahout, a library for scalable machine learning on MapReduce [13] and further equivalents of Google technologies such as ZooKeeper [14] (a clone of Chubby [38]) and HBase [12] (a clone of BigTable [42]).

An intense debate about the advantages and disadvantages of MapReduce with respect to parallel databases was led [50, 165]. The discussion concluded that the success of MapReduce was due to four factors. First, MapReduce is cost effective, as it allows for building datacenters from many low-end servers rather than a few high-end servers, and it runs robustly in such clusters due to its conservative pull-model for data exchange. A second success factor is the storage and schema independence of MapReduce, it can read from the huge variety of data sources that make up today's production systems. Third, it is very effective for "read-once" tasks such as web indexing, as it does not require

[1] transactions that provide atomicity, consistency, isolation and durability

importing the data before processing it (in contrast to relational databases). Finally, MapReduce allows to express complex transformations (e.g. information extraction tasks) much more easily than SQL. Unfortunately, MapReduce still has serious deficiencies, e.g. inefficient performance when executing iterative computations and no support for asynchronicity, which make it suboptimal for certain complex applications such as machine learning and graph processing. This has led to a variety of specialized systems for these tasks, for example Pregel [120], Giraph [10] and Pegasus [99] for graph processing and GraphLab [117], Parameter Server [114], Terascale Learner [1], h20 [82], SystemML [76] and Cumulon [89] for machine learning. While these systems perform extremely well in their specialized area, their specialization comes with a price: pre- and postprocessing of the data, which often requires relational operations, has to be done outside of the specialized system, which leads to processing pipelines that involve many different systems. These pipelines are difficult to operate as they require a scattered codebase and data movement between the individual systems.

In the last years, a new breed of massively parallel dataflow systems emerged, which finally brings together the world of parallel databases with MapReduce-like systems. These systems provide declarative query languages [4], operators for combining multiple datasets [18] and automatic query optimization [91] from parallel databases together with the storage-independence, scalability and programming flexibility of MapReduce systems. They are able to effectively execute relational operations at scale [15], and due to dedicated support for iterative computations, can emulate many of the specialized systems [62, 177, 180]. Microsoft runs large parts of its infrastructure on a stack of massively parallel dataflow systems consisting of Dryad [92], DryadLINQ [183] and SCOPE [40]. The most prominent examples for massively parallel dataflow systems in the research world are Apache Flink (formerly Stratosphere [2]), Apache Spark [186] and Hyracks/AsterixDB [7, 32]. The majority of the work in this thesis focuses on scaling out data mining algorithms in massively parallel, MapReduce-like dataflow systems. We present research on scaling out data mining in classical systems such as MapReduce and parallel databases, as well as in novel systems like Apache Flink and Apache Spark.

2.2 Distributed Shared-Nothing File Systems

The data analysis systems discussed in this thesis all process large datasets stored in distributed file systems modeled after the Google File System (GFS) [75]. Google developed GFS as storage platform for their web search engine, where it handled hundreds of terabytes of data in 2003 already. GFS is a scalable, shared-nothing file system for large distributed data-intensive applications built on the following design decisions. The main goal is to achieve high fault tolerance, as GFS is aimed at clusters consisting of hundreds or thousands of commodity machines, where high failure rates are common.

Furthermore, the file system is supposed to store large, multi-gigabyte files. The write workload consists of large sequential writes that append only to existing data. Random updates of existing files are not possible. This pattern matches web crawling and indexing scenarios very well. The read workload is expected to be mostly large streaming reads with a few small random reads. The design clearly favors high bandwidth for bulk reads over low latency access to individual files.

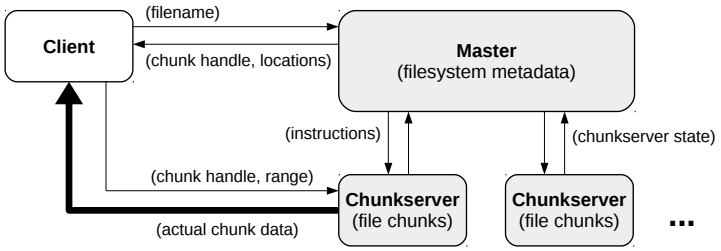


Figure 2.1: Communication patterns in GFS as depicted in [75].

GFS is implemented with a master-slave architecture. The master server orchestrates the file operations and stores the metadata of the file system. It maintains a replicated write-ahead log of critical metadata changes in order to guarantee fault tolerance. The files are to store are divided into so-called chunks with a typical size of several dozen megabytes. The slaves, called chunk servers, store replicas of the chunks (the actual file data) on their local disks. The master ensures that there is always a given number of replicas of each chunk stored in the file system. Clients of the file system communicate with the master for metadata only, e.g. for querying the location of the chunks belonging to a certain file (c.f. Figure 2.1). The master redirects them to the actual chunk servers, which contain the replica and the client directly conducts reads and writes of the chunks on the corresponding chunk server. The master regularly sends out so-called heartbeat messages to the chunk servers, to which these answer with their current state. The master piggybacks control messages on the heartbeat, e.g. to re-replicate chunks in case of machine failures or data corruption. A popular open source implementation of GFS is the Hadoop Distributed File System (HDFS) [11], which we use for our experiments throughout this thesis.

2.3 Abstractions for Massively Parallel Dataflow Processing

Next, we introduce the dominant paradigms for conducting parallel computations on data stored in distributed file systems. The majority of algorithm implementations in this thesis will use one of these abstractions.

2.3.1 MapReduce

MapReduce is a programming model as well as a paradigm for distributed data processing, based on two second-order functions called `map` and `reduce` [49]. The data model of MapReduce consists of simple key-value tuples. A MapReduce program is expressed in the form of two user-defind first-order functions f_m and f_r. The function f_m takes a tuple with key of type k_1 and value of type v_1 and transforms this tuple to zero, one or more tuples of type k_2 for the key and v_2 for the value:

$$f_m : (k_1, v_1) \rightarrow \text{list}(k_2, v_2)$$

The function f_r takes a key of type k_2 and a list of values of type v_2 as input and computes another list of values of type v_2 as output:

$$f_r : (k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)$$

A MapReduce system applies the user-defined first-order functions f_m and f_r in parallel to the input using the system-defined second-order functions `map` and `reduce` as illustrated in Figure 2.2. In the so-called map-phase, the system applies the function f_m individually to all key-value tuples from the input data. This phase is followed by the shuffle-phase, where the system groups all tuples produced by the map-phase by the key k_2. In the final reduce-phase, the system applies the function f_r to all the groups produced in the shuffle-phase. This execution style amends itself to massive parallelism. The maximum degree of parallelism in the map-phase is equal to the number of input tuples, as f_m can be applied to every input tuple in isolation. The maximum degree of parallelism in the reduce-phase is equal to the number of distinct keys of the tuples produced in the map-phase.

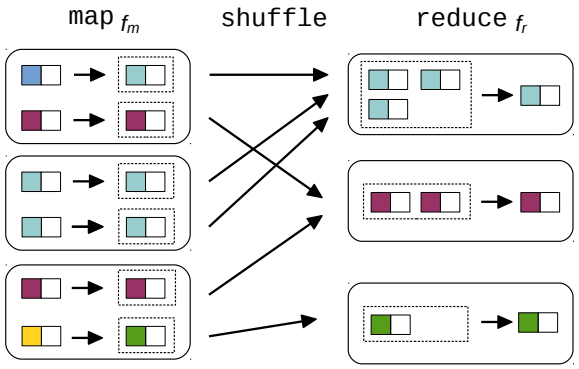


Figure 2.2: Dataflow of key-value tuples in MapReduce.

The architecture of a distributed MapReduce system is complementary to the architecture of the underlying distributed file system (c.f., Section 2.2). Again, a master-slave

architecture is used, where the master orchestrates the system by scheduling the individual map and reduce tasks on the slave machines, by monitoring their progress and reacting to failures [49,179]. The input to a MapReduce program typically consists of key-value tuples (typically serialized to a binary representation) that are physically stored in many chunks on the distributed file system. For the execution of a particular job, the master first decides which slave machines should participate and transfers the compiled MapReduce program to these machines. Next, the input data is logically divided into fixed-size input splits. In order to minimize data movement over the network, the master tries to exploit locality by scheduling map processing tasks for input splits to the machines that already store a local copy of the chunk containing the split.

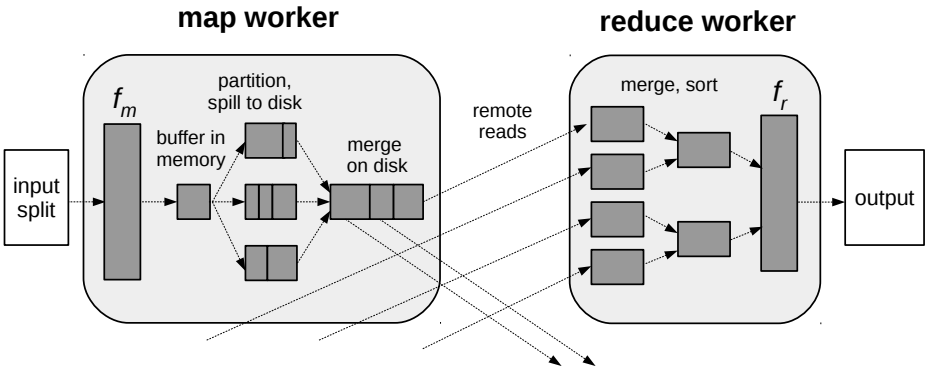
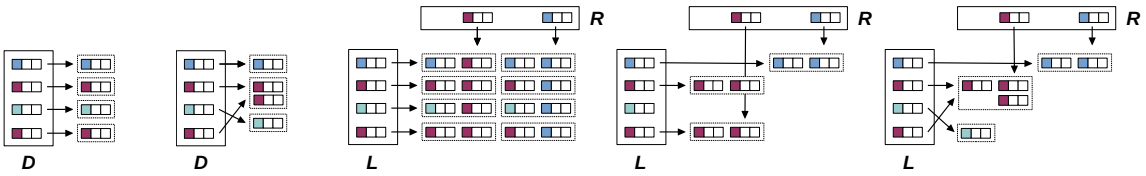


Figure 2.3: Distributed execution of a MapReduce program.

The slaves execute their assigned map and reduce tasks as follows (c.f. Figure 2.3). Every map worker reads its designated input splits and deserializes the contained key-value pairs of type (k_1, v_1). Next, the worker passes each pair to the user-defined function f_m and stores the key-value pairs of type (k_2, v_2) emitted by f_m in memory. These buffered output pairs are partitioned on k_2 according to the number of reduce workers and periodically written to the map worker's local disk. Once a map worker finishes processing an input split, it reports this progress to the master, which notifies the reduce workers. Subsequently, the reduce workers fetch the partitions of the map output intended for them via remote reads from the map workers. Finally, each reduce worker sorts all fetched partitions to group them by the key k_2 as required for the user-defined function f_r. Next, each reduce worker iterates over the grouped key-value pairs, invokes f_r for every group $(k_2, \text{list}(v_2))$ and appends output of f_r to its output file in the distributed file system. The execution of the MapReduce job finishes once all reduce workers have finished. Similar to GFS, the master uses a heartbeat to check the workers presence. In case of failures, MapReduce allows for a remarkibly simple way of guaranteeing fault tolerance. The master only needs to reschedule the tasks assigned to failed map or reduce workers to other machines in the cluster. The correctness of the output of a deterministic



(a) Map (b) Reduce (c) Cross (d) Join (e) CoGroup

Figure 2.4: Second-order functions forming the Pact operators.

MapReduce program in presence of failures is guaranteed by using atomic commits for the intermediate map and reduce outputs to avoid data corruption.

2.3.2 Parallelization Contracts & Iterative Dataflows

Parallelization Contracts (Pacts) [18] are the core abstraction of the parallel data processing system *Apache Flink*, formerly known as *Stratosphere*. Flink offers in-situ data processing, a wide variety of operators, treats user defined functions as first-class citizens, automatically optimizes programs and has dedicated support for iterative computations. In this section, we focus on a discussion of its core concepts. We redirect the interested reader to [2] for an exhaustive description of Flink's complete software stack. Pact extends the MapReduce paradigm (c.f. Section 2.3.1) and allows for the specification of parallelizable transformations on sets using the following data and programming model. A dataset D is an unordered collection $D = \{r_1, \ldots, r_n\}$ of n records. Every record r_i comprises an ordered tuple of m values $r_i = \{v_1, \ldots, v_m\}$. The system is agnostic of both, the semantics and type of the individual values v_j. Their interpretation is left to the user-defined functions supplied inside a Pact-program. The Pact programming model generalizes the MapReduce model: a Pact operator combines a second-order function with a user-defined first-order function (UDF) [3]. Furthermore, the second-order function might require the user to specify key fields of the records to process. A UDF contains arbitrary code and can output an arbitrary number of records. The UDF can modify, remove and add values in the records it processes. The system-provided second-order function dictates the partitioning of the dataset to produce, and applies its associated UDF to the groups resulting from the partitioning. These groups can than be processed in a data-parallel fashion by many independent tasks. The executing system derives the parallelization strategies from the partitioning semantics of its provided second-order functions, the Pact operators. Pact inherits the two unary operators map and reduce from the MapReduce paradigm (c.f., Section 2.3.1). Both operate on a single input dataset D as follows. The map operator (c.f. Figure 2.4a) applies the user-defined first order function f to all individual records $r_1, \ldots, r_n$ from D:

$$\texttt{map} : D \times f \rightarrow \left\{ f(r_1), \ldots, f(r_n) \right\}$$

The reduce operator (c.f. Figure 2.4b) works on a user-defined keyspace K over the attributes of D with t distinct keys. The active domain of K is $\{k_1, \ldots, k_t\}$, and $r_{(k)}$ denotes that $r.K = k$. The user defined function f takes a group of records with the same key as input here:

$$\texttt{reduce} : D \times f \times K \rightarrow \left\{ f(r_{1_1}^{(k_1)}, \ldots, r_{n_1}^{(k_1)}), \ldots, f(r_{1_t}^{(k_t)}, \ldots, r_{n_t}^{(k_t)}) \right\}$$

Additionally, Pact defines three binary operators that work on two input datasets, to which we refer to as $L = \{l_1, \ldots, l_p\}$ and $R = \{r_1, \ldots, r_q\}$. The dataset L contains p records, and the dataset R consists of q records. Note that the UDF f expects two inputs for these operators. The cross operator (c.f. Figure 2.4b) builds the cartesian product between all records of L and R and invokes the user-defined first order function f on every single resulting pair of records:

$$\texttt{cross} : L \times R \times f \rightarrow \left\{ f(l_1, r_1), f(l_1, r_2), \ldots, f(l_p, r_q) \right\}$$

The join operator (c.f. Figure 2.4d) performs an equi-join between L and R, using the key spaces K and Q. Next, it applies the user-defined first-order function f to all pairs of records in the join result:

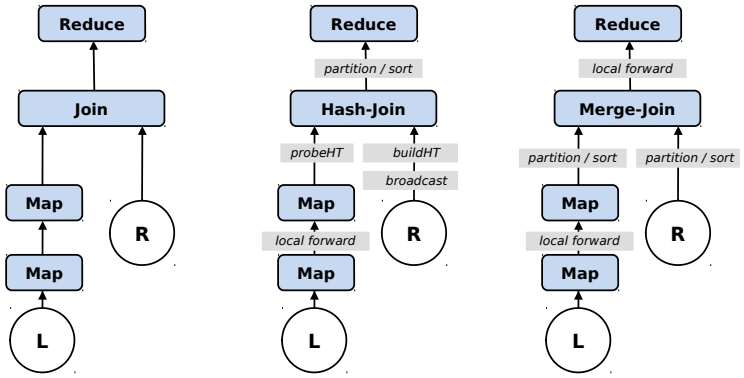
$$\texttt{join} : L \times R \times K \times Q \times f \rightarrow \left\{ \{ f(l, r) \mid l.K = r.Q \} \right\}$$

The cogroup operator (c.f. Figure 2.4e) is a binary extension of the *reduce* operator. Records from the datasets L and R are grouped using the key spaces K and Q. Afterwards, every pair of groups from L and R with the same grouping key w from the combined active domain of K and Q is fed to the user-defined first order function f:

$$\texttt{cogroup} : L \times R \times K \times Q \times f \rightarrow \left\{ f(l_{1_1}^{(w_1)}, \ldots, l_{n_1}^{(w_1)}, r_{1_1}^{(w_1)}, \ldots, r_{n_1}^{(w_1)}), \ldots, l_{1_t}^{(w_t)}, \ldots, l_{n_t}^{(w_t)}, r_{1_t}^{(w_t)}, \ldots, r_{n_t}^{(w_t)}) \right\}$$

A Pact program consists of a directed aclycic graph (DAG), where Pact operators and their corresponding user-defined first-order functions form the vertices and edges represent exchange of data between operators. The Flink optimizer transforms such a Pact program into a low-level job graph of the distributed processing engine *Nephele* [175] which executes in parallel on a cluster. Flink's optimizer is inspired by optimizers proposed for parallel database systems, and analogously bases its optimizations on logical plan equivalences, cost models and reasoning about interesting properties [80, 158]. The optimization of Pact programs is more difficult than in the relational case for several reasons. Most importantly, the operators do not have fully specified semantics due to the execution of user-defined first-order functions. The unknown UDFs also complicate deriving estimates for intermediate result sizes. A further challenge is that there is no pre-defined schema present in the Pact programming model. The optimization is

conducted in two phases: a logical plan rewriting phase is followed by the selection of a physical execution plan. During the logical rewriting, the optimizer generates equivalent plans by reordering operators. The reordering is conditioned on conflicting value accesses and preservation of group cardinalities [90, 91]. The subsequent physical optimization follows a cost-based approach by picking strategies for data shipping and local operator execution using estimated execution costs [18]. Examples for such strategies are broadcast- or re-partition-based data shipping and sort- or hash-based join execution. During the selection process, the optimizer keeps track of interesting properties such as sorting, grouping and partitioning.



(a) Logical plan (b) Physical plan A (c) Physical plan B

Figure 2.5: Optimization example in Pact.

Figure 2.5a shows an example of a logical plan for which a physical plan must be chosen. The example program applies two subsequent map operations to an input dataset L, joins the outcome with another input R and finally aggregates the join result with a reduce operation. Figures 2.5b and 2.5c show two possible physical plans for executing the program. Both plans connect the map operators with a local-forward, to run them subsequently on the same machine, which is the most efficient way to chain tuple-at-a-time operations, such as maps. The main difference between the two physical plans is the execution strategy of the join. Plan A broadcasts R and uses it as the build side of a hash-join. Therefore, no re-partitioning of L is necessary in order to execute the join. The result of the join must be re-partitioned and transferred over the network in order to execute the reduce operator[2]. This plan makes sense when both R and the join result are very small. Plan B is only valid if the reduce operation groups on the join key. It re-partitions both R and the output of the map operations applied to L, sorts them and applies a merge-join afterwards. This choice enables the local execution of the final reduce operator, which re-uses the partitioning and sorting of the join result. Plan B

[2]assuming that the join result is not incidentally partitioned on the grouping key already

Algorithm 2: Bulk-iterative computation.

1 **while** $\neg c(\, S^{(t)}, f(S^{(t)})\,)$
2 $\quad S^{(t+1)} \leftarrow f(S^{(t)})$

makes sense when the join result is very huge. The optimal plan choice depends on the size and cardinality of the inputs, as well as on the intermediate results sizes produced by the UDFs.

A distinctive feature of Flink is its special support for embedding iterative computations in a dataflow. Efficiently executing such computations is crucial for use cases such as machine learning and graph processing. In general, an iterative computation starts with an initial state $s^{(0)}$ and aims to find the fixpoint s^* of the iterated step function f, such that $s^* = f(s^*)$. Flink integrates iterative computations by allowing the user to mark a part of the DAG as iterative [61, 62]. The system then repeatedly executes this part of the DAG by forwarding the output of its last operator to its first operator. Flink provides two ways to execute these iterative parts of a Pact program. *Bulk iterations* apply the step function f (which can be an arbitrary Pact program) of the iterative computation to re-compute the intermediate result, represented by a dataset $S^{(t+1)}$ in iteration t as a whole (c.f. Algorithm 2). Bulk iterations stop execution once a termination criterion c is satisfied. In many cases however, parts of the intermediate state converge at different speeds. An example are single-source shortest path computations in large graphs, where the shortest paths to vertices close to the source vertex are typically found much earlier than those of vertices far away from the source. In such cases, the system would waste resources by always re-computing the whole intermediate state $S^{(t)}$, which includes the parts that do not change anymore. To alleviate this issue, Flink offers *delta iterations*. Delta iterations model an iterative computation with two datasets (c.f. Algorithm 3) and a special step function. The solution set $S^{(t)}$ holds the current intermediate result, while the working set $W^{(t)}$ holds algorithm-specific data to compute updates to the solution. During a delta iteration the system uses the specialized step function called Δ which consumes the current solution $S^{(t)}$ as well as the current working set $W^{(t)}$. This Δ function then computes a set of updates $D^{(t+1)}$ to the solution as well as the next working set $W^{(t+1)}$ (c.f. line 2). The system uses $D^{(t+1)}$ to selectively update the solution set $S^{(t)}$ and thereby forms the next intermediate solution $S^{(t+1)}$ (c.f., line 3). A delta iteration automatically terminates once the working set $W^{(t)}$ becomes empty (c.f. line 1). It is important to note that Flink's optimizer is aware of the iterative parts of Pact programs and can apply specialized optimizations, e.g. caching of the static parts of iterative Pact programs.

A wide variety of applications have been built on top of the Pact model. Examples include Meteor, an operator model with application-specific functions as first-class operators that

Algorithm 3: Delta-iterative computation.

1 **while** $W^{(t)} \neq \emptyset$
2 $\quad (D^{(t+1)}, W^{(t+1)}) \leftarrow \Delta(S^{(t)}, W^{(t)})$
3 $\quad S^{(t+1)} \leftarrow S^{(t)} \cup D$

allows for operator semantics to be evaluated and exploited for optimization at compile time [86]. The high level language Pig as well as the deeply embedded data processing language Emma have been compiled to Pacts [4, 95]. Furthermore, Pact's iteration model has been shown to optimize for the asymmetry in the convergence of graph processing algorithms [94].

2.3.3 Resilient Distributed Datasets

The third abstraction for parallel processing we introduce are Resilient Distributed Datasets (RDDs), the core abstraction behind the distributed data processing system Apache Spark [186, 187]. RDDs have been motivated by the growing need to efficiently execute applications that re-use intermediate results across multiple operations. Prime examples for such applications are interactive ad-hoc data analysis and iterative algorithms that occur in domains such as machine learning and graph processing. MapReduce-based systems typically show bad performance in these kinds of applications, as they materialize the intermediate data between subsqeuent MapReduce jobs in the distributed filesystem, which leads to large I/O overheads that dominate the execution time.

RDDs are a distributed shared memory abstraction[3] for MapReduce-like computations on large clusters in a fault tolerant way. Despite of their shared memory character, RDDs are based on coarse-grained transformations rather than fine-grained updates, in order to allow for simple and effective fault tolerance. Analogously to Pacts (c.f. Section 2.3.2) and MapReduce (c.f. Section 2.3.2), users conduct parallel computations using a set of high-level operators and user-defined functions. The system automatically handles parallelization, work distribution and fault tolerance. RDDs are fault tolerant, parallel data structures which allow users to explicity persist intermediate results in memory and control the partitioning of their data for placement optimization. An RDD is an immutable, read-only, partitioned collection of records, created through deterministic operations, either from data on stable storage or by transforming other RDDs. RDDs allow the users to manipulate the data through a rich set of transformations that are mostly analogous to the Pact operators 2.3.2 (e.g., map, reduce, cogroup). The programming

[3] We characterize RDDs as a shared memory abstraction because they provide bulk transformations over datasets partitioned over the memory of a cluster.

interface is inspired by the functional API of DryadLINQ [183]. These transformations again apply the same operation to all elements (or groups of elements) in parallel. The main purpose of RDDs is to enable an order of magnitude faster execution times (compared to disk based MapReduce systems like Hadoop [11]) by re-using intermediate results across multiple operations. Analogous to Pacts, RDDs are an abstraction for general distributed processing and subsume specialized systems such as the Pregel graph processing system [120]. The immutability of RDDs imposes a rigorous restriction in contrast to traditional distributed shared memory abstractions: RDDs only make sense for applications doing bulk writes, as efficient random writes are not possible by design. This choice offers lots of advantages however: it allows for straggler mitigation through speculative execution of transformations on partitions of RDDs, it enables bulk-reads to gracefully go out-of-core in case of memory pressure and allows scheduling based on data locality.

The main advantage of the immutability in the design of RDDs is that it enables a novel, efficient fault tolerance scheme called lineage-based recovery. As RDDs are based on coarse-grained bulk transformations and not on fine-grained element-wise updates, they allow the system to log the transformations (the lineage of the data), thereby freeing it from having to replicate the data itself. In the system, the lineage is represented as a graph, where the partitions of the individual RDDs are vertices, and edges represent the data dependencies of the transformations between RDDs. Spark distinguishes between two different kinds of dependencies between RDDs: *Narrow dependencies* have a one-to-one relation between the parent and the child RDD (the result of a transformation), narrow dependencies occur through map transformations or joins on co-partitioned data. In case of the loss a partition of the child RDD, only a single partition of the parent RDD must be re-computed. This allows for very inexpensive recovery in case of failures. *Wide dependencies*, on the contrary, occur when there is an all-to-all relation between the partitions of the parent and child RDDs of a transformation (e.g. in transformations that require the re-partitioning of an RDD). Recovery in that case is expensive, as the recovery of a single lost partition of the child RDD requires a full re-computation of all partitions of the parent RDD. Spark usually materializes intermediate data for operations with wide dependencies to simplify fault tolerance. Additionally, Spark offers support for checkpointing an RDD's data which is more efficient for special applications with very long lineage chains such as iterative graph computations.

Spark uses a master/slave architecture for execution. A driver program connects to the master and to a cluster of workers. Next, the driver program issues instructions to create and transform RDDs. The execution of the issued transformations is deferred until the program requests a so-called action, an operation that either sends data to the driver program (e.g. a count) or materializes an RDD (e.g. to write out results). The driver tracks the lineage, while the workers store and process partitioned RDDs. Spark's scheduler is aware of the locality of RDD partitions and its scheduling decisions

take into consideration whether partitions are pinned in memory or not. Spark divides the execution of a program into so-called stages. Every stage contains all pipelined transformations with narrow dependencies until a transformation with wide dependencies is encountered. The system will then execute these transformations and materialize the result of the wide dependency operation as last part of the stage.

The simplicity of the RDD abstraction allowed the Spark community to build lightweight libraries on top of their system, such as Spark Streaming [188], a mini-batch oriented stream processing engine, GraphX [180] for distributed graph processing, a SQL engine called SparkSQL [15] and MLlib [126], a library for distributed machine learning. RDDs and Pacts are very related in the abstraction which they provide. The main difference is that Pacts take an operator centric view and focus on the execution of query-like DAGs (which provides more optimization opportunities), while RDDs provide a data centric view and focus on program rather than query execution (e.g. including control flow).

2.3.4 Vertex-Centric Graph Processing

The last approach to massively parallel data processing which we introduce is *vertex-centric graph processing*, as proposed in the Google Pregel system [120]. In contrast to general data flow system, Pregel is a system specialized for a single use case, namely graph processing. We will however see that the computational model proposed in Pregel can easily be emulated by general data flow systems, which make Pregel-like systems a valid target of research in this thesis.

Pregel defines a computational model for large-scale graph processing, where a program consists of a sequence of iterations, in which vertices get messages from the previous iteration, modify their own state and send messages to other vertices. Pregel enables efficient, scalable and fault-tolerant graph processing on clusters of commodity computers. A main motivation for the development of Pregel was the high overhead incurred for iterations by MapReduce-like systems. Its computational model is heavily inspired by 'Bulk Synchronous Processing' (BSP), a general model for parallel computations proposed by Leslie Valiant [174]. Analogous to BSP, iterations are called supersteps in vertex centric graph processing. During a superstep, the system invokes a UDF for every vertex (conceptually in parallel). The UDF defines the behavior of a vertex v in superstep s; it receives messages sent to v in superstep $s-1$, can modify state of v, as well as the graph topology, and typically sends messages to other vertices, which will be (logically) delivered in the subsequent superstep $s+1$. The execution is synchronized via a global superstep barrier. Pregel again uses a master-slave architecture. The execution of a vertex-centric program proceeds as follows. The graph is by default hash-partitioned on the vertex id, such that the vertex and its whole adjacency list live in the same partition. The master assigns these partitions to the workers which load the portions of the graph

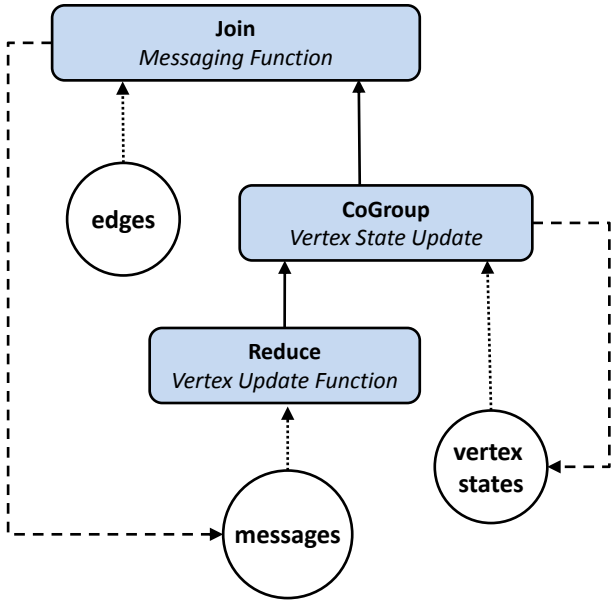


Figure 2.6: Vertex centric graph processing as iterative dataflow plan in Pact.

in parallel into their local memory. Next, the master coordinates the supersteps. During a superstep, the workers invoke the vertex update UDF on their local partitions and asynchronously deliver messages between themselves. A superstep is synchronized via a global barrier (typically implemented using a distributed locking service [38]). The execution continues as long as there are active vertices or messages to be delivered. Finally, the master instructs the workers to save their vertex states to the DFS and the program terminates. Pregel achieves fault tolerance via checkpointing of vertex values, edge values and incoming messages. Failures are detected by checking timeouts while pinging the participating machines. In case of failure, the computation is restarted from the last checkpoint.

The popularity and ease of the computational model in Pregel has lead to many implementations of the system [10, 152, 153]. Furthermore, researchers have shown that Pregel-like programs easily execute in general dataflow systems that offer an abstraction for the efficient execution of iterative programs [36, 62, 177, 180]. Figure 2.6 exemplarily illustrates how Flink models Pregel-like programs using its delta iteration abstraction (c.f., Section 2.3.2). The input to the Pact-program is threefold: the initial states of the vertices which form the solution set of the iteration, the initial messages (the workset of the delta iteration) as well as the edges of the graph that we process. In this plan, we divide the vertex-centric UDF into two Pact-UDFs. First, we compute a pontential new state for every vertex by aggregating the messages with a reduce-operator (the ‘vertex update

function'). Next, we join the candidate update with the current vertex state using a cogroup-operator and update it if has changed. Finally, we join the updated vertex states with their corresponding adjacent edges using a join-operator and run the user-specified 'messaging function' (the second part of the Pregel-UDF). This will produce a new set of messages (the workset of the delta iteration) and the system runs the next iteration. The computation terminates once no more new messages are produced (meaning the workset becomes empty).

2.4 Exemplary Data Mining Use Cases And Their Mathematical Foundations

We move from the system-specific background of the thesis to the algorithmic background for the data mining tasks, which we scale to large datasets in the remainder. We introduce the mathematical background for the two main use cases discussed in the thesis: collaborative filtering and network centrality.

2.4.1 Collaborative Filtering

Recommender systems are ubiquitous in modern internet platforms, e.g. in recommending products in online shops, suggesting friends in social networks or proposing similar videos on video sharing websites. There are two different general approaches for generating recommendations for users that interact with some kind of items. In *content-based filtering*, we build content-related profiles for users and items. These profiles might for example be based on product attributes or user demographics. The next step is to associate users with matching products, based on the profiles. The content-based approach has the huge disadvantage that it requires a deep understanding of the domain, and potentially a lof of manual effort to build the profiles. The alternative approach (which will be one of the main applications of this thesis) is so-called *collaborative filtering* [77], which bases its recommendations solely on observed historical interactions between users and items. Thereby, collaborative filtering is independent of the domain for which it is applied.

Collaborative filtering is a form of dyadic learning: we are given a set of interactions M between a set of users C and a set of items P. M is typcially represented as a matrix from users (rows) to items (columns). An entry m_{ij} denotes that user i interacted with item j in the past. There exist two different kinds of interaction data in collaborative filtering scenarios [106]. In the case of *explicit feedback*, the users explicitly assign numerical scores to items which they prefer (e.g. by rating movies with 1 to 5 stars). In this case, the matrix representing M is only partially defined, as only the cells that hold observed ratings are known. In the case of *implicit feedback*, the users' behavior is passively observed and the entries in M typically holds counts of observed behavior (e.g. user i

visited website of item j three times). Implicit feedback produces a completely defined matrix, yet the confidence in non-observed interactions (the zero entries) is very low, due to lack of negative feedback. It is non-obvious whether users did not interact with an item because they did not like that item or whether they did simply not have the chance to interact with it yet. Furthermore, it is important to note that real world interaction datasets are typically extremely sparse, as most users only interact with a tiny fraction of the available items (e.g., less than 0.1% of entries are known in [182]). The goal in collaborative filtering is to suggest new items for users which they might enjoy. Formally, collaborative filtering is typically cast as a supervised learning problem, where the task is to accurately predict future interactions. There are many ways to evaluate the quality of a recommender, e.g. by the root mean squared error (RMSE) to held-out ratings [21], the mean average precision in predicting the top k items of users, or the users personalized ranking of the items [147]. There are two main algorithmic families in collaborative filtering, whose scale-out will be in the focus of this thesis. The *similarity-based neighborhood methods* [54, 77, 116, 149, 154] compute relationships (in terms of interaction behavior) between either users or items, and subsequently derive their predictions from these relationships. The most prominent approach from this family is *item-based collaborative filtering* [116, 154]. The idea is to first compute a matrix S which holds pairwise similarities among item interaction vectors (the columns of M). Typical choices for measuring the similarity among item vectors are variations of cosine, pearson correlation or pointwise mutual information [164]. A simple way to estimate an unobserved rating $\hat{m}_{ij}$ of user i towards item j is by using a weighted sum over similarities and past ratings (c.f., Equation 2.1). Let h_i denote the set of all items in the interaction history of user i. Then we estimate the rating of user i towards item j as follows. We multiply the similarity s_{jk} of j towards each known item k with the rating m_{ik} that user i gave for item k. We finally normalize their sum by the sum of the absolute values of the similarities:

$$\hat{m}_{ij} = \frac{\sum_{k \in h_i} s_{jk}\, m_{ik}}{\sum_{k \in h_i} |s_{jk}|} \tag{2.1}$$

We find the top k items to recommend for a particular user by predicting the ratings for all her unknown items and retaining the k items with the highest estimated rating. Item-based collaborative filtering is very popular because it is a simple approach and empirically works well with a slightly outdated version of the similarity matrix S, due to the static nature of the relationship between items. This allows to leverage a pre-computed version of S for fast prediction computations.

$$\underset{U,V}{\operatorname{argmin}} \sum_{(i,j) \in M} (m_{ij} - u_i^T v_j)^2 + \lambda(\|u_i\|^2 + \|v_j\|^2) \tag{2.2}$$

The second prominent approach to collaborative filtering are *latent factor models* [88, 106, 189]. The idea is to characterize both users and items by a set of automatically derived

latent variables. These variables can correspond to distinguishing features (e.g. the genre of movies), but will not necessarily be interpretable by humans. Latent factor models project users and items onto a joint latent factor space of low rank f, such that closeness in that space denotes similarity and high preference. We aim to learn a vector $u_i \in \mathbb{R}^f$ for every user i, as well as a vector $v_j \in \mathbb{R}^f$ for every item j, such that their inner product models the interaction between the user and the item. This inner product allows us to estimate the strength of unobserved interactions $\hat{m}_{ij} = u_i^T v_j$. Mathematically, this leads to a matrix factorization problem: Find a matrix of user vectors U and a matrix of item vectors V, such that their product approximates observed parts of M and generalizes well to unseen parts. This problem is closely related to singular value decomposition, with the difference that we operate on a partially defined matrix M in explicit feedback cases. A standard technique is to choose U and V such that they minimize the regularized empirical squared error of the predicted interaction values to the observed values in M. A common way to formulate this as an optimization problem is shown in Equation 2.2, where λ denotes a hyperparameter for weighting the regularization term. This model is easily extendable to incorporate biases for users and items, temporal dynamics [106] or implicit feedback data with varying confidence levels [88].

2.4.2 First-Order Optimization Methods for Latent Factor Models

In this section, we introduce mathematical techniques for learning latent factor models. A popular way to learn model parameters p that minimize a differentiable error function $E(p)$ are *gradient descent* methods [24]. The idea is to iteratively take steps into the negative direction of E's gradient: $p^{(t+1)} = p^{(t)} - \eta \nabla E(p^{(t)})$. This approach has the drawback that a single parameter update requires a full pass through the whole dataset. Online versions of gradient descent have been proposed to mitigate this problem when the error function is comprised of a sum of error terms for independent observations in the dataset: $E(p) = \sum_i E_i(p)$. The resulting online version, called *stochastic gradient descent* (SGD), updates the model parameters based on a single observation at a time: $p^{(t+1)} = p^{(t)} - \eta \nabla E_i(p^{(t)})$, and converges much faster in many cases. SGD is the most popular method for latent factor models [68, 106]. Its application to Equation 2.2 leads to the simple learning algorithm depicted in Algorithm 4. We randomly initialize the model parameters U and V (c.f. line 1) and loop through the dataset in random order until convergence (c.f. lines 2 & 3). For every interaction m_{ij} between a user i and an item j, we compute the error e_{ij} of the prediction $u_i^T v_j$ using the factor vectors corresponding to user i and item j. Next, we update the factor vectors in the negative direction of the gradient with respect to the learning rate η and the regularization term λ. The disadvantage of SGD in its classical form is that it is an inherently sequential algorithm, as the model parameters change after each data point processed. Therefore, another approach to learning latent factor models called *alternating least squares* (ALS) [88, 178, 189] became

Algorithm 4: Learning latent factor models with SGD.

1 randomly initialize U and V
2 **while** not converged
3 **foreach** $m_{ij} \in M$ in random order
4 $e_{ij} \leftarrow m_{ij} - u_i^T v_j$
5 $u_i \leftarrow u_i - \eta \left(e_{ij} v_j - \lambda u_i\right)$
6 $v_j \leftarrow v_j - \eta \left(e_{ij} u_i - \lambda v_j\right)$

popular. ALS is computationally more expensive than SGD but amends itself to an easy parallelization scheme. ALS repeatedly keeps one of the unknown matrices (either U or V) fixed, so that the other one can be optimally re-computed by solving a least squares problem. ALS then rotates between re-computing the rows of U in one step and the columns of V in the subsequent step, as shown in Algorithm 5. We update every user vector u_i in parallel by solving a least squares problem (c.f. lines 3 & 4). This least squares problem involves V_{h_i} which denotes a submatrix of V built from the factor vectors of all items in the interaction history h_i of user i, E which is the $k \times k$ identity matrix where k is the dimensionality of the factor space, and $m_{h_i}^T$ which the i-th row of M with only the entries retained for items in the interaction history h_i of user i. In the subsequent step in lines 5 & 6, we analogously update all factor vectors for the items in parallel. Here U_{h_j} denotes a submatrix of U with all factor vectors retained for users in the interaction history h_j of item j, and m_{h_j} stands for the j column of M with all entries kept for users in h_j.

Algorithm 5: Learning latent factor models with ALS.

1 randomly initialize V
2 **while** not converged
3 **forall** users i **do in parallel**
4 $u_i \leftarrow (V_{h_i} V_{h_i}^T - \lambda E)^{-1} (V_{h_i} m_{h_i}^T)$
5 **forall** items j **do in parallel**
6 $v_j \leftarrow (U_{h_j} U_{h_j}^T - \lambda E)^{-1} (U_{h_j} m_{h_j})$

It is important to note that these approaches assume a low rank structure in the data (e.g., interaction vectors of items and users that we wish to be close in latent factor space must roughly be linear combinations of each other). Furthermore, these methods expose a large number of hyperparameters (e.g., the number of factors f, the regularization constant λ, the learning rate η) which have to be determined in order to achieve high quality results.

2.4.3 Centralities in Large Networks

A further use case that we study in this thesis concerns finding important objects in large networks with billions of edges (e.g. social networks or the web graph). This problem is crucial for ranking purposes, e.g. to rank the results of a web search or to rank publications in a citation network. In general, a network describes pairwise relationships among n objects [132], which are represented by a set V of vertices of the network. The set of edges E represents the pairwise relationships between the vertices. In the following, we will focus on directed, unweighted networks and represent them mathematically by their $n \times n$ adjacency matrix A, defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if the edge from vertex } i \text{ to vertex } j \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

We tackle the centrality problem [30, 132], where the goal is to measure the importance of the individual vertices of a network. The task is to compute a centrality vector $x \in \mathbb{R}^n$ for all vertices, where x_i denotes a score of importance assigned to vertex i, and higher scores mean higher importance. We introduce the basic centrality measures from the literature; later parts of the thesis will describe how to compute these in a scalable manner on large networks. The simplest form of centrality is *degree centrality*. The degree k_i of a vertex i is the number edges which are connected to i. In directed networks we differentiate between the in-degree k_i^{in} (which equals the number of incoming edges in vertex i) and the outdegree k_i^{out} (which equals the number of outgoing edges from vertex i). A simple centrality is to use the out-degree: $x_i = \sum_j a_{ij}$. In this case, we assume that vertices with more connections are more important. Unfortunately, degree centrality ignores the global structure of the network and is easy to trick (e.g. by spammers forming many random connections in a social network). *Spectral centrality measures* are more elaborate by taking the structure of the network into account. Spectral measures define a recursive notion of centrality: the centrality of a vertex is the sum of the centrality of the vertices of its neighbors, with a higher contribution for more important vertices. This was first formulated as *eigenvector centrality* [31], in which x is equal to the leading eigenvector of A (c.f., Equation 2.3):

$$x_i = \sum_j a_{ij} x_j \quad (2.3) \qquad x_i = \alpha \sum_j a_{ij} x_j + \beta \quad (2.4) \qquad x_i = \alpha \sum_j \frac{1}{k_j^{\text{out}}} a_{ij} x_j + \beta \quad (2.5)$$

The problem of this simple formulation is that vertices with no incoming edges get assigned zero centrality, as well as vertices that are only pointed to by such vertices (e.g., all vertices that are not in a strongly connected component of the network). This is especially problematic for acyclic networks such as citation networks which have no strongly connected component. *Katz centrality* [101] alleviates this by assigning a

constant amount β of centrality to every vertex (c.f., Equation 2.4). Katz centrality has the problem that a vertex gives the same contribution to all its neighbors, regardless of its out-degree, which is counter-intuitive. *PageRank* [138] corrects for this by weighing the contribution to neighbors of a vertex by its out-degree (c.f. Equation 2.5). PageRank has been instrumental as a ranking measure for Google's web search results. Note that special care is necessary for handling vertices with out-degree zero.

An alternative family of centrality measures are *path-based measures*. *Closeness centrality*, for example, measures the importance of a vertex by its geodesic distances to all other vertices in the network. The geodesic distance d_{ij} from vertex i to vertex j is the number of edges contained in a shortest path from i to j. The closeness centrality of a vertex is defined as the inverse of the sum of its geodesic distances to all other vertices (c.f., Equation 2.6). As the classical closeness centrality assigns zero to vertices that cannot reach all other vertices (through an ∞ term in the denominator), *harmonic centrality* [27, 29, 30] was proposed, which only considers reachable vertices (c.f., Equation 2.7). Both of these centralities measures are hard to scale as they require to solve an all-pairs distance problem on the vertex set, which is quadratic in the number of vertices. Scalable substitutes use probabilistic data structures to approximate the neighborhood function of the graph, which gives the number of reachable vertices for a particular vertex in a given distance [29, 98, 139].

$$x_i = \frac{1}{\sum_j d_{ij}} \quad (2.6) \qquad x_i = \frac{1}{\sum_{j;d_{ij}<\infty} d_{ij}} \quad (2.7) \qquad x_i = \sum_{s,t \in V; s \neq t} \frac{n_{st}^i}{g_{st}} \quad (2.8)$$

A second prominent path-based measure is *betweeness centrality*, which calculates the importance of a vertex i by comparing the number of geodesic paths n_{st}^i between any pair of vertices s and t that go through vertex i with the overall number of geodesic paths g_{st} between s and t (c.f., Equation 2.8). The intuition here is that we consider a vertex to be important for the flow of information in a network if it lies on many shortest paths. Betweeness centrality is again hard to scale to large networks, as it contains an all-pairs shortest paths problem, which is quadratic in the number of vertices. A scalable substitue called LineRank [97] has been proposed. The idea behind LineRank is to compute the steady state probabilities of a random walk on the line graph of the network and to aggregate the resulting probabilities assigned to edges as a centrality score for the vertices.

2.4.4 Computation via Generalized Iterative Matrix Vector Multiplication

In this section, we show that a single computational abstraction, namely *generalized iterative matrix vector multiplication*, suffices to compute all the centralities introduced in Section 2.4.3. We describe how spectral measures map to an eigenvector problem,

which we solve with iterated matrix vector multiplications. Similarly, we detail how path-based measures are connected to the powers of a matrix defined over a semiring. We exemplarily detail how a simple algorithm for computing closeness centrality maps to iterative matrix vector multiplications defined over a semiring.

The computation of spectral centralities reduces to finding the leading eigenvector of a certain matrix representing the network. In eigenvector centrality, for example, we compute the leading eigenvector of the adjacency matrix A. For Katz centrality, the matrix to consider is $\alpha A + \beta\mathbb{1}$, where *alpha* and *beta* control the constant amount of centrality which we assign to every vertex. The PageRank vector equals to the normalized leading eigenvector of the matrix $\alpha AD^{-1} + \beta\mathbb{1}$, where D is a diagonal matrix holding the vertices' out-degrees ($d_{ii} = k_i^{out}$). Analogously, LineRank requires us to compute the leading eigenvector of the matrix given by $\alpha\, TS^T D^{-1} + \beta\mathbb{1}$, where TS^T is a decomposition of the adjacency matrix of the network's line graph. A simple way to obtain the leading eigenvector v_1 of a matrix M is the Power Method [78], which computes v_1 as the fixpoint of the following sequence.

$$x^{(t+1)} = \frac{Mx^{(t)}}{\|Mx^{(t)}\|_2}$$

This method easily applies to large datasets, as every iteration just conducts a matrix vector multiplication and scales the resulting vector to unit length. The power method makes $x^{(t)}$ converge to the leading eigenvector v_1 if its corresponding eigenvalue λ_1 is strictly greater than the remaining eigenvalues. As a further condition, we must choose the initial vector $x^{(0)}$ such that it does not contain a zero component in the direction of v_1. Let $c_1\lambda_1 v_1 + c_2\lambda_2 v_2 + \cdots + c_n\lambda_n v_n$ be the representation of $x^{(0)}$ in the n-dimensional eigenbasis of A. As the power method multiplies $x^{(0)}$ by powers of A, it converges to a multiple of v_1 as follows[4]:

$$\begin{aligned} A^t x^{(0)} &= c_1\lambda_1^t v_1 + c_2\lambda_2 v_2 + \cdots + c_n\lambda_n^t v_n \\ &= \lambda_1^t v_1 \left(c_1 + \tfrac{c_2}{c_1}(\tfrac{\lambda_2}{\lambda_1})^t v_2 + \cdots + \tfrac{c_n}{c_1}(\tfrac{\lambda_n}{\lambda_1})^t v_n \right) \end{aligned}$$

Path-based measures such as closeness centrality and harmonic centrality require us to compute the neighborhood function $N(t, i)$ of a network, which gives the number of vertices that are reachable from a vertex i within t edge hops [98]. The neighborhood function immediately allows us to compute the number of vertices $N_i(t)$ whose geodesic distance to i is t as $N(t, i) - N(t-1, i)$. The sum of geodesic distances $\sum_j d_{ij}$ of all vertices to vertex i from Equation 2.6 equals the fixed point of the series $\sum_{r=0}^{\infty} N_i(t)$ which is reached once t becomes equal to the diameter (the longest shortest path) of the network. Figure 2.7 illustrates a simple algorithm to compute the neighborhood function. We assign a bitstring of length n to every vertex, where n is the number of vertices in the

[4] note that all terms $(\frac{\lambda_i}{\lambda_1})^t$ approach zero, as $\lambda_n > \lambda_i$ for all eigenvalues λ_i with $i \geq 2$

network. If bit j is set in the bitstring of a vertex i, it means that vertex i has reached vertex j via edge traversals. In the first iteration $t = 0$, every vertex only reached itself. In every iteration t, each vertex mimicks edge traversals by collecting the bitstrings of its adjacent neighbors and merging them with its own bitstring via logical *OR*. Then, the resulting bitstring of a particular vertex has all the bits sets for vertices that it reaches with at most t edge traversals. We compute the neighborhood function $N(t, i)$ for a vertex i as the number of bits set in its bitstring at iteration t. The algorithm converges once the bitstrings stop changing, which happens when t becomes equal to the diameter of the network. Interestingly, paths in a network are also connected to powers of a matrix,

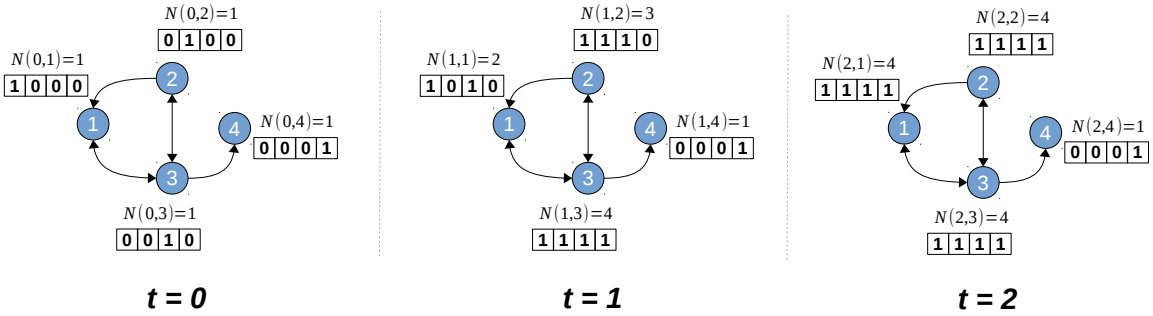


Figure 2.7: Computing the neighborhood function of a network through a series of bitstring exchanges between its vertices.

similar to eigenvectors. For example, an entry a_{ij}^t in the t-th power of the adjacency matrix A denotes the number of paths of length t from vertex i to vertex j [110]. It has been shown that many popular graph algorithms such as shortest paths, connected components, reachability or transitive closure can be computed by iterative matrix vector multiplications on matrices defined over certain semirings $(E, \oplus, \otimes)$. Here E denotes the set of elements on which we operate and $\oplus$ and $\otimes$ are binary operations that fulfill the algebraic properties of a semiring [79, 99].

We exemplarily discuss how to arrive at the algorithm for computing the neighborhood function depicted in Figure 2.7 by conducting generalized matrix vector multiplications on matrix defined over a semiring. We choose the semiring $(\{0,1\}^n, \vee, \wedge)$ which operates on the set of bitstrings of length n and substitutes addition with logical *OR* and multiplication with logical *AND*. We initialize our vector $x^{(0)} \in \{0,1\}^{(n \times n)}$ such that its i-th component $x_i^{(0)}$ contains the bitstring of length n with the i-th bit set. We replace the adjacency matrix with a matrix $A \in \{0,1\}^{(n \times n \times n)}$ such that an entry a_{ij} consists of the bitstring $\mathbb{1}$ of length n with all bits set if an edge from vertex i to vertex j exists or if $i = j$. The remaining entries comprise of the bitstring $\mathbb{0}$ of length n with no bit set. Note that $\mathbb{0}$ and $\mathbb{1}$ correspond to the identity elements of our substituted addition and multiplication operations. Computing the neighborhood function then reduces to computing the fixed

point of the series $x^{(t+1)} = Ax^{(t)}$ in our chosen semiring. Equation 2.9 shows the update for a single component $x_i^{(t+1)}$ of $x^{(t+1)}$ in iteration $t+1$.

$$\begin{aligned} x_i^{(t+1)} &= A_{i\bullet}\, x^{(t)} &&= \bigvee_{j=1}^{n} a_{ij} \wedge x_j^{(t)} \\ &= \bigvee_{j \in adj(i)} \mathbb{1} \wedge x_j^{(t)} &&= \bigvee_{j \in adj(i)} x_j^{(t)} \end{aligned} \tag{2.9}$$

We obtain x_i as the dot product between the i-th row $A_{i\bullet}$ of A and the vector $x^{(t)}$ holding the bitstrings of the vertices at iteration t. In our chosen semiring, we compute this dot product by first applying a logical *AND* between the bitstrings contained in the entries a_{ij} and their corresponding components $x_j^{(t)}$ of $x^{(t)}$. Second, we merge the resulting bitstrings with a logical *OR*. For non-adjacent neighbors of vertex i, we apply an *AND* with $\mathbb{0}$, which annihilates E as the result is always $\mathbb{0}$, the identity element of our addition substitute $\vee$. Therefore, these terms have no influence on the computation and we safely ignore them. That leaves only terms that perform a conjunction with $\mathbb{1}$. We ignore this conjunction too, as $\mathbb{1}$ is the identity element of our multiplication substitute $\wedge$. The remaining operation to obtain $x_i^{(t)}$ is $\bigvee_{j \in adj(i)} x_j^{(t)}$: merging the bitstrings of its adjacent neighbors with a logical *OR*. We have now arrived at exactly the operation depicted in our algorithm from Figure 2.7, which is what we wanted to show. In this formulation, the algorithm does not scale because it requires memory quadratic in the number of the vertices of the network for the bitstrings. Therefore, usually an approximation is computed where the bitstrings are substitued with a probabilistic datastructure for estimating set cardinality, e.g., a hyperloglog sketch [27, 98, 99].

We have seen that generalized iterative matrix vector multiplication is a powerful abstraction for computing centralities in networks. Scaling this computation will allow us to easily execute a wide variety of centrality algorithms on large networks, by simply substituting the contained matrices and binary operations.

3 Scalable Collaborative Filtering & Graph Mining

3.1 Problem Statement

In this chapter, we focus on algorithm-level scalability (c.f., Section 1.1) of our examplary data mining use cases, collaborative filtering and centralities in large networks (c.f., Section 2.4). We discuss the mathematical operations of the algorithms involved in these cases (c.f., Sections 2.4.2 and 2.4.4), and map them to the parallel programming paradigm of distributed data processing systems. We start with collaborative filtering on MapReduce-based systems, investigate a prototype of a specialized system for collaborative filtering tasks, and in the end compare graph mining algorithms on Pregel-like systems and dataflow systems like Apache Flink.

Recommender systems are a prime example for data mining applications that need scalable technology, as the size of the models produced is typically proportional to the input cardinality (e.g., the number of users in a social network or the number of items in a market place). With rapidly growing data sizes, the processing efficiency and scalability of recommender systems and their underlying computations becomes a major concern [8]. In a production environment, the offline computations necessary for running a recommender system must be periodically executed as part of larger analytical workflows and thereby underly strict time and resource constraints. For economic and operational reasons it is often undesirable to execute these offline computations on a single machine: this machine might fail and with growing data sizes constant hardware upgrades might be necessary to improve the machine's performance to meet the time constraints. Due to these disadvantages, a single machine solution can quickly become expensive and hard to operate. Therefore, it is desirable to run data-intensive, analytical computations in a parallel and fault-tolerant manner on a large number of commodity machines [49, 115]. This makes the execution independent of single machine failures and furthermore allows for the increase of computational performance by simply adding more machines to the cluster; thereby obviating the need for constant hardware upgrades to a single machine. As discussed in Section 1.1, this technical approach requires the re-phrasing of existing algorithms to enable them to utilize a parallel processing platform.

We re-phrase and scale-out the similarity-based neighborhood methods (c.f., Section 2.4.1). They have the advantage of being simple and intuitive to understand, as they are directly inspired by recommendation in everyday life, where we tend to check out things we heard about from like-minded friends or things that seem similar to what we already like. They

capture local associations in the data which increases serendipity [151] and they are necessary as part of ensembles to reach optimal prediction quality [20]. The item-based variants [154] of the neighborhood methods are highly stable and allow us to compute recommendations for new users without the need to re-build the model. Due to these properties, neighborhood methods are popular in industrial use cases [6, 47, 116, 164]. We improve the scalability of the similarity-based neighborhood methods by re-phrasing the underlying algorithm for similarity computations to MapReduce. In the next piece of work, we focus on the second-most popular family of collaborative filtering algorithms: latent factor models (c.f., Section 2.4.1). We discuss means to efficiently conduct the computations required on a MapReduce-based system, despite of the limitations of the MapReduce execution model. We propose low-rank matrix factorization with Alternating Least Squares, which uses a series of broadcast-joins that can be efficiently executed with MapReduce. Additionally, we propose a set of changes to the execution model to mitigate short-comings in MapReduce-based systems.

In cases with extreme model sizes (e.g., low-rank factorizations of datasets containing hundreds of millions of users), it is beneficiary to investigate alternative system architectures. In such cases, the model size exceeds the memory of an individual machine. This leads to a set of challenges, e.g. how to partition the model among the participating machines and how to correctly execute learning algorithms in such a setting. Furthermore asynchronous learning approaches are desirable as they typically converge faster [22]. To meet these challenges, we describe "Factorbird", a prototypical system that leverages a parameter server architecture [114] for learning large matrix factorization models for recommendation mining. We furthermore discuss how to build such a system on top of the existing technology stack of Twitter, the largest micro-blogging service in the world and present experiments on datasets with dozens of billions of interactions.

In the last part of this chapter, we discuss two algorithms for computing centralities in large networks (c.f., Section 2.4.3). We map these algorithms to the vertex-centric programming model of a Pregel-like graph processing system (c.f., Section 2.3.4) as well as to a general dataflow system. We investigate how well the operations of the algorithms fit into the vertex-centric paradigm and discuss scalability issues specific to graph datasets.

3.2 Contributions

In our work on scaling collaborative filtering on MapReduce in Section 3.3, we first introduce an algorithmic framework that enables scalable neighborhood-based recommendation. Next, we introduce a UDF which allows to implement a variety of similarity measures in a highly efficient manner in our framework. Furthermore, we describe a

selective down-sampling technique to handle scaling issues introduced by the heavy tailed distribution of user interactions commonly encountered in recommendation mining scenarios. Subsequently, we show how to execute an iterative low-rank matrix factorization algorithm with MapReduce in a scalable manner and propose a set of changes to the MapReduce execution model to increase performance. For the approaches proposed, we conduct an extensive set of experiments on several publicly available datasets and on a synthetic dataset generated from the Netflix dataset. We show that our approaches scale to these datasets containing hundreds of millions to billions of interactions, and therefore efficiently handle real-world workloads.

We revisit the matrix factorization problem for collaborative filtering in Section 3.4, where we present 'Factorbird', a prototype of a non-dataflow architecture approach for factorizing large matrices. We design Factorbird to meet the following desiderata: (a) scalability to tall and wide matrices with dozens of billions of non-zeros, (b) extensibility to different kinds of models and loss functions as long as they can be optimized using Stochastic Gradient Descent, and (c) adaptability to both batch and streaming scenarios. We discuss how to combine lock-free Hogwild!-style learning with a special partitioning scheme to drastically reduce network traffic and conflicting updates. We furthermore discuss how to efficiently grid search for hyperparameters at scale. In the end, we present experiments on a matrix built from a subset of Twitter's interaction graph, consisting of more than 38 billion non-zeros and about 200 million rows and columns, which is to the best of our knowledge the largest matrix on which factorization results have been reported in the literature.

Lastly, we investigate two network centrality use cases in Section 3.5. During our investigation, we highlight a scalability bottleneck connected to the memory requirements in distributed graph processing systems and point out short-comings in the generality of the vertex-centric abstraction using a spectral centrality measure as example.

3.3 Collaborative Filtering with MapReduce

In the following, we present our work on scaling out the two most popular families of collaborative filtering algorithms (c.f., Section 2.4.1), namely similarity-based neighborhood methods and latent factor models, on MapReduce-based systems.

3.3.1 Distributed Similarity-Based Neighborhood Methods

Recall that collaborative filtering approaches operate on a $|C| \times |P|$ matrix M holding all known interactions between a set of users C and a set of items P (c.f., Section 2.4.1). A user i is represented by her item interaction history $m_{i\bullet}$, the i-th row of M. The top

recommendations for this user correspond to top items selected from a ranking r_i of all items according to how strongly they would be preferred by the user. This ranking is inferred from patterns found in M. We express the similarity-based neighborhood approach in terms of linear algebraic operations. Neighborhood-based methods find and rank items that have been preferred by other users who share parts of the interaction history $m_{i\bullet}$. For simplicity, assume that M is a binary matrix with $M_{ij} = 1$ if a user i has interacted with an item j and $M_{ij} = 0$ otherwise. For pairwise comparison between users, a dot product of rows of M gives the number of items that the corresponding users have in common. Similarly, a dot product of columns of M gives the number of users who have interacted with both items corresponding to the columns. When computing recommendations for a particular user with *User-Based Collaborative Filtering* [149], first a search for other users with similar taste is conducted. This translates to multiplying the matrix M by the user's interaction history $m_{i\bullet}$, which results in a ranking of all users. Secondly, the active user's preference for an item is estimated by computing the weighted sum of all other users' preferences for this item and the corresponding ranking. In our simple model this translates to multiplying the ranking of all users with M^{T}. This means the whole approach can be summarized by the following two multiplications:

$$r_i = M^{\mathrm{T}}(M\, m_{i\bullet})$$

To exploit the higher stability of relations among items, a variant of the neighborhood methods called *Item-Based Collaborative Filtering* [154] has been proposed, which looks at items first and weighs their co-occurrences. This approach computes a matrix of item-to-item similarities and allows for fast recommendations as this matrix does not have to be re-computed for new users. Modeling the item-based approach with linear algebra simply results in moving the parentheses in our formula, as $M^{\mathrm{T}}M$ gives exactly the matrix of the item co-occurrences. This shows that both user- and item-based collaborative filtering share the same fundamental computational model. In the rest of this section, we focus on the more popular item-based variant.

$$r_i = (M^{\mathrm{T}}M)\, m_{i\bullet}$$

Algorithm 6: Sequential approach for computing item co-occurrences.

```
1 foreach item j1:
2   foreach user i who interacted with j1:
3     foreach item j2 that i also interacted with:
4       S_{j1 j2} = S_{j1 j2} + 1
```

The standard sequential approach [116] for computing the item similarity matrix $S = M^{\mathrm{T}}M$ is shown in Algorithm 6. For each item j_1, we need to look up each user i who

interacted with j_1. Then we iterate over each other item j_2 from i's interaction history and record the co-occurrence of j_1 and j_2. We mathematically express the approach using three nested summations:

$$S = M^{\mathrm{T}}M = \sum_{j_1=1}^{|P|} \sum_{i=1}^{|C|} \sum_{j_2=1}^{|P|} M_{j_1 i} \cdot M_{i j_2}$$

If we were to distribute the computation across several machines on a shared-nothing cluster, this approach becomes infeasible as it requires random access to both users and items in its inner loops. Its random access pattern cannot be realized efficiently when we work on partitioned data. Furthermore, the complexity of the item-based approach is quadratic in the number of items, as each item has to be compared with every other item. However, the interaction matrix M is usually very sparse. It is common that only a small fraction of all cells are known, (e.g., in the datasets we use for our experiments, this ratio varies from 0.1% to 4.5%), and that the number of non-zero elements in M is linear in the number of rows of M. This severely limits the number of item pairs to be compared, as only pairs that share at least one interacting user have to be taken into consideration. It decreases the complexity of the algorithm to quadratic in the number of non-zeros in the densest row rather than quadratic in the number of columns. The cost of the algorithm is expressed as the sum of processing the square of the number of interactions of each single user. Unfortunately, collaborative filtering datasets share a property that is common among datasets generated by human interactions: the number of interactions per user follows a heavy tailed distribution which means that processing a small number of 'power users' dominates the cost of the algorithm. In the following, we develop a parallelizable formulation of the computation to scale out this approach on a parallel processing platform. Our algorithm must be able to work with partitioned input data. Furthermore, it must scale linearly with respect to a growing number of users. We additionally enable the usage of a wide range of similarity measures and add means to handle the computational overhead introduced by 'power users'.

We discuss the step-by-step development of our algorithmic framework. We start with showing how to conduct distributed item co-occurrence counting for our simple model that uses binary data. After that, we generalize the approach to non-binary data and enable the usage of a wide variety of similarity measures through a similarity UDF. Finally, we discuss means to sparsify the similarity matrix and apply selective down sampling to achieve linear scalability with a growing number of users. In order to scale out the similarity computation from Algorithm 6, it needs to be phrased as a parallel algorithm, to make its runtime speedup proportional to the number of machines in the cluster. This is not possible with the standard sequential approach, which requires random access to both the rows and columns of M in the inner loops of Algorithm 6. A way of executing this multiplication that is better suited to the MapReduce paradigm and has an access

pattern that is compatible to partitioned data is to re-arrange the loops of Algorithm 6 to the row outer product formulation of matrix multiplication. Because the i-th column of M^{T} is identical to the i-th row of M, we can compute S with access to the rows of M only:

$$S = M^{\mathrm{T}}M = \sum_{i=1}^{|C|} \sum_{j_1=1}^{|P|} \sum_{j_2=1}^{|P|} M_{j_1,i}^{\mathrm{T}} \cdot M_{i,j_2} = \sum_{i=1}^{|C|} m_{i\bullet}(m_{i\bullet})^{\mathrm{T}}$$

Following this finding, we partition M by its rows (the users) and store it in the distributed file system. Each map function reads a single row of M, computes the row's outer product with itself and sends the resulting intermediary matrix row-wise over the network. The reduce function sums up all partial results, thereby computing a row of S per invocation (c.f., Algorithm 7). This approach exploits the sparsity of the intermediary outer product matrices by making the map function only return non-zero entries. At the same time we apply a combiner (which is identical to the reducer) on the vectors emitted by the mappers, which reduces the amount of data sent over the network. Additionally, we only compute the upper triangular half of S, as the resulting similarity matrix is symmetric.

Algorithm 7: Counting item coocurrences with MapReduce.

```
1  function map(m_i•):
2    foreach j_1 ∈ m_i•
3      c ← sparse_vector()
4      foreach j_2 ∈ m_i• with j_2 > j_1
5        c[j_2] ← 1
6      emit(j_1, c)
7  function combine(j, c_1, ..., c_n):
8    c ← Σ_{k=1}^{n} c_k
9    emit(j, c)
10 function reduce(j, c_1, ..., c_n):
11   c ← Σ_{k=1}^{n} c_k
12   emit(j, c)
```

Real world datasets contain richer representations of the user interactions than a simple binary encoding. They either consist of *explicit feedback* like numerical ratings that the users chose from a predefined scale or of *implicit feedback* where we count how often a particular behavior such as a click or a page view was observed (c.f., Section 2.4.1). We want to choose from a variety of similarity measures for comparing these item interactions, in order to be able to best capture the relationships inherent in the data. We drop the assumption that M contains only binary entries and assume that it holds such explicit or implicit feedback data. We incorporate a wide range of measures for comparing the

interactions of two items j_1 and j_2 by integrating a **similarity UDF** consisting of three canonical functions into our algorithm. We first adjust each item rating vector via a function *preprocess()*:

$$\widehat{m_{\bullet j_1}} = \text{udf.preprocess}(\, m_{\bullet j_1}\,) \quad \widehat{m_{\bullet j_2}} = \text{udf.preprocess}(\, m_{\bullet j_2}\,)$$

Next, the second function *norm()* computes a single number from the preprocessed vector of an item:

$$n_{j_1} = \text{udf.norm}(\, \widehat{m_{\bullet j_1}}\,) \quad n_{j_2} = \text{udf.norm}(\, \widehat{m_{\bullet j_2}}\,)$$

These preprocessing and norm computations are conducted in an additional single pass over the data, which starts with M^{T}, applies the two functions and transposes M^{T} to form M. The next pass over the data is a modification of the previously described co-occurence counting approach. Instead of summing up co-occurrence counts, we now compute the dot products of the preprocessed vectors.

$$dot_{j_1 j_2} = \widehat{m_{\bullet j_1}} \cdot \widehat{m_{\bullet j_2}}$$

We provide those together with the numbers we computed via the *norm* function to a third function called *similarity()* which computes a measure-specific similarity value (c.f., Algorithm 8).

$$S_{j_1 j_2} = \text{udf.similarity}(\, dot_{j_1 j_2}, n_{j_1}, n_{j_2}\,)$$

With this approach we are able to incorporate a wide variety of different similarity measures which can be re-phrased as a variant of computing a dot product. Note that this technique preserves the ability to apply a combiner in each pass over the data. Table 3.1 describes how to express several common similarity measures through our similarity UDF, including cosine, Pearson correlation and a couple of others evaluated by Google for recommending communities in its social network Orkut [164].

Example: The Jaccard coefficient between items j_1 and j_2 (two columns from the interaction matrix M) is defined as the ratio of the number of users interacting with both items to the number of users interacting with at least one of those items. We can easily express it with our similarity UDF, as this example shows:

$$m_{\bullet j_1} = \begin{bmatrix} 1 \\ - \\ 3 \end{bmatrix} \quad m_{\bullet j_2} = \begin{bmatrix} 2 \\ 1 \\ 5 \end{bmatrix}$$

Pass 1: We start by having *udf.preprocess* binarize the vectors:

$$\widehat{m_{\bullet j_1}} = bin(m_{\bullet j_1}) = \begin{bmatrix}1\\0\\1\end{bmatrix} \quad \widehat{m_{\bullet j_2}} = bin(m_{\bullet j_2}) = \begin{bmatrix}1\\1\\1\end{bmatrix}$$

The second function that is invoked for each of the vectors is *udf.norm*. We let it return the L_1 norm, which gives us the number of non-zero components of each of the binary vectors:

$$n_{j_1} = \| \widehat{m_{\bullet j_1}} \|_1 = 2 \quad n_{j_2} = \| \widehat{m_{\bullet j_2}} \|_1 = 3$$

Pass 2: Finally, the function *udf.similarity* will be given the dot product between the preprocessed vectors and their precomputed norms. We have to rearrange the formula of the Jaccard coefficient so that it computes from the numbers we have at hand:

$$jaccard(m_{\bullet j_1}, m_{\bullet j_2}) = \frac{|m_{\bullet j_1} \cap m_{\bullet j_2}|}{|m_{\bullet j_1} \cup m_{\bullet j_2}|} = \frac{dot_{j_1 j_2}}{n_{j_1} + n_{j_2} - dot_{j_1 j_2}} = \frac{2}{2+3-2} = \frac{2}{3}$$

Algorithm 8: Computing item similarities with MapReduce.

```
1  function map(m_{i•}^):
2    foreach (j_1, k_1) ∈ m_{i•}^
3      d ← sparse_vector()
4      foreach (j_2, k_2) ∈ m_{i•}^ with j_2 > j_1
5        d[j_2] ← k_1 k_2
6      emit(j_1, d)
7  function combine(j, d_1, ..., d_n):
8    d ← Σ_{k=1}^{n} d_k
9    emit(j, d)
10 function initialize_reducer():
11   n ← load_norms()
12 function reduce(j_1, d_1, ..., d_n):
13   dots ← Σ_{k=1}^{n} d_k
14   foreach (j_2, d) ∈ dots
15     s[j_2] = udf.similarity(d, n[j_1], n[j_2])
16   emit(j_1, s)
```

In order to be able to handle cases with an enormous number of items, we add means to decrease the density of the similarity matrix S to our final implementation. To get rid of pairs with near-zero similarity, we allow specifying a similarity threshold, for which we

evaluate a size constraint to prune lower scoring item pairs early in the process [19] and eventually remove all entries from S that are smaller than the threshold. This threshold is data dependent and must be determined experimentally to avoid negative effects on prediction quality. Furthermore, the prediction quality of the item-based approach is sufficient if only the top fraction of the similar items is used [154], therefore we add another MapReduce step that only retains these top similar items per item in a single pass over the data.

Measure	*preprocess*	*norm*	*similarity*
Cosine	$\frac{v}{\|v\|_2}$	-	$dot_{j_1 j_2}$
Pearson correlation	$\frac{(v-\bar{v})}{\|v-\bar{v}\|_2}$	-	$dot_{j_1 j_2}$
Euclidean distance	-	$\hat{v}^2$	$\sqrt{n_{j_1} - 2 \cdot dot_{j_1 j_2} + n_{j_2}}$
Common neighbors	$bin(v)$	-	$dot_{j_1 j_2}$
Jaccard coefficient	$bin(v)$	$\|\hat{v}\|_1$	$\frac{dot_{j_1 j_2}}{n_{j_1} + n_{j_2} - dot_{j_1 j_2}}$
Manhattan distance	$bin(v)$	$\|\hat{v}\|_1$	$n_{j_1} + n_{j_2} - 2 \cdot dot_{j_1 j_2}$
Pointwise Mutual Information	$bin(v)$	$\|\hat{v}\|_1$	$\frac{dot_{j_1 j_2}}{\|C\|} \log \frac{dot_{j_1 j_2}}{n_{j_1} n_{j_2}}$
Salton IDF	$bin(v)$	$\|\hat{v}\|_1$	$\frac{\|C\| \cdot dot_{j_1 j_2}}{n_{j_1} n_{j_2}^2} (-\log \frac{n_{j_1}}{\|C\|})$
Log Odds	$bin(v)$	$\|\hat{v}\|_1$	$\log \frac{\frac{\|C\| \cdot dot_{j_1 j_2}}{n_{j_1} n_{j_2}^2}}{1 - \frac{\|C\| \cdot dot_{j_1 j_2}}{n_{j_1} n_j^2}}$
Log-likelihood ratio [53]	$bin(v)$	$\|\hat{v}\|_1$	$2 \cdot (H(dot_{j_1 j_2}, n_{j_2} - dot_{j_1 j_2}, n_{j_1} - dot_{j_1 j_2}, \|C\| - n_{j_1} - n_{j_2} + dot_{j_1 j_2}) - H(n_{j_2}, \|C\| - n_{j_2}) - H(n_{j_1}, \|C\| - n_{j_1}))$

Table 3.1: Expressing similarity measures with the proposed similarity UDF.

Recall that our goal is to develop an algorithmic framework that scales linearly with respect to a growing user base. The cost of the item-based approach is dominated by the densest rows of M, which correspond to the users with the most interactions. This cost, which we express as the number of item co-occurrences to consider, is the sum of the squares of the number of interactions of each user. The number of interactions per user usually follows a heavy tailed distribution as illustrated in Figure 3.2a which plots the ratio of users with more than n interactions to the number of interactions n on a logarithmic scale. Therefore, there exists a small number of 'power users' with an unproportionally high amount of interactions. These drastically increase the runtime, as the cost produced by them is quadratic with the number of their interactions. If we only look at the fact whether a user interacted with an item or not, then we would intuitively not learn very much from a 'power user': each additional item she interacts

with will co-occur with the vast amount of items she already preferred. We would expect to gain more information from users with less interactions but a highly differentiated taste. Furthermore, as the relations between items tend to stabilize quickly [151], we presume that a moderately sized number of observations per item is sufficient to find its most similar items. Following this rationale, we decided to apply what we call an *interaction-cut*: we only partially take the interaction histories of the 'power users' into account. We apply the interaction cut by randomly sampling p interactions from each power user's history, thereby limiting the maximum number of interactions per user in the dataset to p. Note that this sampling is only applied to the small group of 'power users', it does not affect the data contributed by the vast majority of non-'power users' in the long tail. Capping the effort per user in this way limits the overall cost of our approach to $|C|\ p^2$. We will experimentally verify that a moderately sized p is sufficient to achieve prediction quality close to that of unsampled data. An optimal value for p is data dependent and must be determined by hold-out tests.

3.3.2 Experiments

In this section we present the results of a sensitivity analysis for the interaction-cut and conduct an experimental evaluation of our parallel algorithm on a large dataset. We empirically verify that the prediction quality achieved by using an interaction cut quickly reaches the prediction quality achieved with unsampled data. Subsequently, we analyze the relationship between the size of the interaction-cut, the quality achieved and the runtime for the similarity computation for our large dataset. After that we study the effects on the runtime speedup if we add more machines to the cluster as well as the scaling behavior with a growing user base. Prediction is conducted with weighted sum estimation enhanced by baseline estimates [105]. In a preprocessing step that has negligible computation cost, we estimate global user and item biases b_i and b_j that describe the tendency to deviate from the average rating μ. This gives us the simple baseline prediction $b_{ij} = \mu + b_i + b_j$ for the rating of a user i to an item j. To finally predict the rating r_{ij}, we use the normalized weighted sum over the user's ratings to the k most similar items of j, incorporating the baseline predictions:

$$r_{ij} = b_{ij} + \frac{\sum_{j_2 \in S^k(j_1,i)} S_{j_1 j_2}(M_{ij_2} - b_{ij})}{\sum_{j_2 \in S^k(j_1,j_2)} S_{j_1 j_2}}$$

Effects of the interaction-cut: We conduct a sensitivity analysis of the effects of the interaction-cut. We measure the effect on the probability of interaction with an item and on prediction quality for varying p on the Movielens dataset [129], which consists of 1,000,209 ratings that 6,040 users gave to 3,706 movies. For our first experiment, we rank the items by their probability of interaction as shown in the plot in the top left corner in

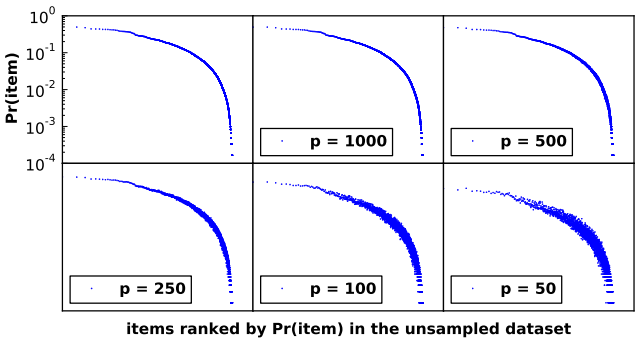
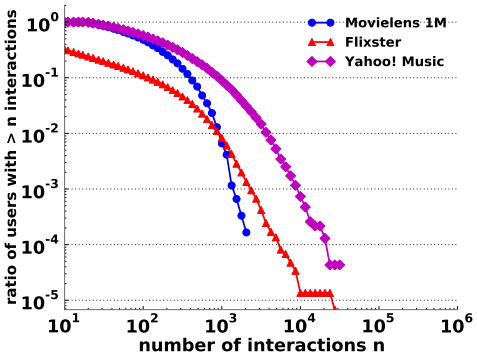
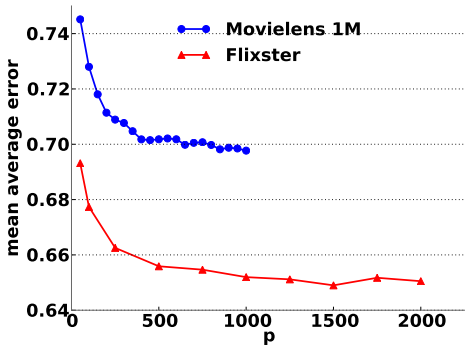


Figure 3.1: Sensitivity of the probability of interaction with an item to an interaction-cut of size p in the Movielens dataset.



(a) Long tailed distribution of the number of interactions per user in various datasets.



(b) Effects of an interaction-cut of size p on prediction quality in various small datasets.

Figure 3.2: Skew in interaction datasets and its effect on prediction quality.

Figure 3.1. Next, we apply the interaction-cut by sampling down the ratings of users whose number of interactions in the training set exceeds p, and repeat this for several values of p. In the remaining plots of Figure 3.1, we retain the order of the items found in the unsampled data and plot their probabilities after applying the interaction-cut for a particular p. We see that for $p \geq 500$ there is no observable distortion in the ranking of the items, which is a hint that this distribution is independent of the data omitted by sampling down the interactions of the 'power users'.

In our second experiment, we compute the prediction quality (by mean average error) achieved by a particular p by randomly splitting the rating data into 80% training and 20% test set based on the number of users. For this experiment, we additionaly use the Flixster dataset [66], consisting of 8,196,077 ratings from 147,612 users to 48,794 movies.

Again, we apply the interaction-cut by sampling down the ratings of users whose number of interactions in the training set exceeds p and use the 80 most similar items per item for rating prediction. For these small datasets, we conduct the tests on a single machine using a modified version of Apache Mahout's [13] *GenericItemBasedRecommender*. Figure 3.2b shows the results of our experiments. Note that the right-most data points are equivalent to the prediction quality of the unsampled dataset. In the Movielens dataset we see that for $p > 400$ the prediction quality converges to the prediction quality of the unsampled data, in the Flixster dataset this happens at $p > 750$. There is no significant decrease in the error for incorporating more interactions from the 'power users' after that. This confirms our expectation that we can compute recommendations based on the user data from the long tail and samples from the 'power users' without sacrificing prediction quality.

Parallel computation with MapReduce: We conduct the following experiments on a Hadoop cluster with a MapReduce implementation of our approach. The cluster consists of six machines running Apache Hadoop 0.20.203 [11] with each machine having two 8-core Opteron CPUs, 32 GB memory and four 1 TB disk drives. The experiments for showing the linear speedup with the number of machines run on Amazon's computing infrastructure, where we rent *m1.xlarge* instances, 64-bit machines with 15 GB memory and eight virtual cores each. In order to test our approach in a demanding setting, we use a large ratings dataset [182] which represents a snapshot of the Yahoo! Music community's preferences for various songs that were collected between 2002 and 2006. The data consists of 717,872,016 ratings that 1,823,179 users gave to 136,736 songs. We use the 699 million training ratings provided in the dataset to compute item similarities and measure the prediction quality for the remaining 18 million held out ratings. We compute the 50 most similar items per item with Pearson correlation as similarity measure, applying a threshold of 0.01. Figure 3.3 shows the results we get for differently sized interaction cuts. We see that the prediction quality converges for $p > 600$, similar to what we previously observed for the smaller datasets. We additionally measure the root mean squared error and observe the same behavior, the prediction quality converged to an error of 1.16 here. We see the expected quadratic increase in the runtime for a growing p, which is weakened by the fact there is a quickly shrinking number of users with more than p interactions (from Figure 3.2a we know for example that only approximately 9% of the users have more than 1000 interactions).

The computational overhead introduced by the 'power users' is best illustrated when we compare the numbers for $p = 750$ and $p = 1000$: we see a decrease of only 0.0007 in the mean average error, yet the higher value of p accounts for a nearly doubled runtime. We conclude that we are able to achieve satisfying prediction quality on this large dataset with a p that is extremely low compared to the overall number of items and thereby results in a computation cost that is easily manageable even by our small Hadoop cluster. In order to conduct a comparison to a single machine

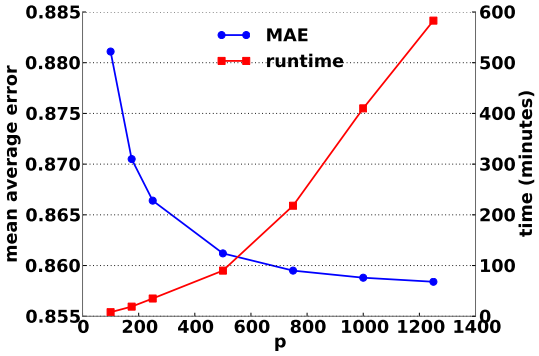
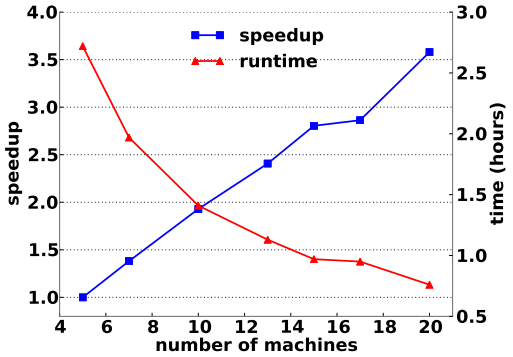
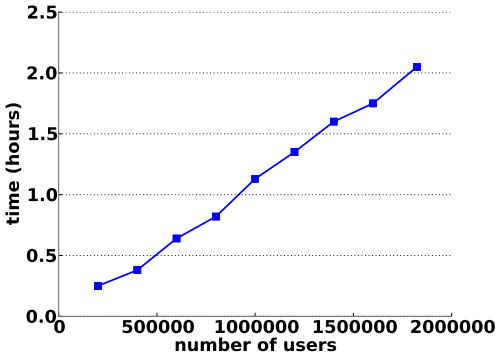


Figure 3.3: Prediction quality and runtime for an interaction-cut of size p in the Yahoo! Music dataset.

implementation, we experiment with the *item-based kNN* recommender of MyMediaLite [70], the *GenericItemBasedRecommender* provided by Apache Mahout [13] and the *ItemRecommender* from LensKit [58]. Unfortunately, not one of these is able to complete the computation due to problems with memory, although we run them on a machine which had 48 GB of RAM available. Based on our findings, we choose to set p to 600 for the following scalability experiments.



(a) Speedup for a growing number of machines in Amazon EC2.



(b) Linear runtime increase for a growing user base.

Figure 3.4: Scale-out experiments with MapReduce on the Yahoo! Music dataset.

Linear speedup with the number of machines: The major promise of parallel processing platforms is seamless horizontal scale-out by increasing the number of machines in the cluster. This requires the computation speedup to be proportional to the number of machines. To experimentally evaluate this property of our algorithm, we use the *ElasticMapReduce* computing infrastructure provided by Amazon, which allows us to

run our algorithm on a customly sized Hadoop cluster. We repeatedly run the similarity computation with an increasing number of cluster machines. Figure 3.4a shows the linear speedup as expected by us. With 15 machines, we are able to reduce the runtime of the similarity computation to less than one hour.

Linear scale with a growing number of users: Finally, we evaluate the scaling behaviour of our approach in the case of a rapid growth of the number of users. The Yahoo! Music dataset is already partitioned into several files, with each file containing approximately 77 million ratings given by 200,000 unique users. In order to simulate the growth of the user base, we use an increasing number of these files as input to our parallel algorithm and measure the duration of the similarity computation. Figure 3.4b shows the algorithm's runtime when scaling from 200,000 to 1,8 million users. We see a linear increase in the runtime which confirms the applicability of our approach in scenarios with enormously growing user bases. As both, the speedup with the number of machines, as well as the runtime for a growing number of users scale linearly, we can counter such growth by simply adding more machines to the cluster to keep the computation time constant.

3.3.3 Distributed Matrix Factorization with Alternating Least Squares

In the following, we move to the second most prominent approach to collaborative filtering: latent factor models. We discuss how to implement a scalable matrix factorization approach on MapReduce. We describe how we choose the algorithm to implement and what techniques we employ to push the performance boundaries of MapReduce-based systems. Recall that latent factor models compute a low-rank factorization of rank k of a $|C| \times |P|$ matrix M comprised of interactions between users and items (c.f., Section 2.4.1). The product UV^T of the resulting feature matrices U and V approximates M. For single machine implementations, SGD is the preferred technique to compute a low-rank matrix factorization. SGD is easy to implement and computationally less expensive than ALS, where every iteration runs in $O((|C| + |P|) * k^3)$, as $|C| + |P|$ linear systems have to be solved (c.f., Section 2.4.2). Unfortunately, SGD is inherently sequential, because it updates the model parameters after each processed interaction. Following this rationale, we chose ALS for our parallel implementation. It is computationally more expensive than SGD, yet provides higher quality updates and naturally amends itself to parallelization. When we re-compute the user feature matrix U for example, u_i, the i-th row of U, can be re-computed by solving a least squares problem only including m_i, the i-th row of M, which holds user i's interactions, and all the columns v_j of V that correspond to non-zero entries in m_i, (c.f., Figure 3.5). This re-computation of u_i is independent from the re-computation of all other rows of U and therefore, the re-computation of U is easy to parallelize if we manage to guarantee efficient data access to the rows of M and the

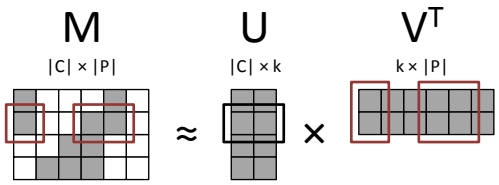


Figure 3.5: Dependencies for re-computing a row of U with Alternating Least Squares.

corresponding columns from V. In the following we refer to the sequence of re-computing of U followed by re-computing V as a single iteration in ALS.

From a data processing perspective, we conduct a parallel join between the interaction data M and V (the item features) in order to re-compute the rows of U. Analogously, we conduct a parallel join between M and U (the user features) to re-compute V. Finding an efficient execution strategy for these joins is crucial for the performance of our proposed parallel solution. When executing joins in a shared-nothing environment, minimizing the required amount of inter-machine communication is decisive for the performance of the execution, as network bandwidth is the most scarce resource in a cluster. For matrix factorization with ALS, we have to alternatingly join M with V and U. In both cases, the interaction matrix M is usually much larger than any of the feature matrices. For the datasets used in our experiments and a factorization of rank 20, the interaction matrix M is 10 to 20 times larger than U and 250 to 2500 times larger than V. We limit our approach to use-cases where U nor V individually fit into the memory of a single machine of the cluster (Section 3.4 will show how to efficiently handle use cases where this assumption is violated). A rough estimate of the required memory for the re-computation steps in ALS is $\max(|C|, |P|) * r * 8$ byte, as alternatingly, a single dense double-precision representation of the matrices U or V has to be stored in memory on each machine. Even for 10 million users or items and a rank of 100, the estimated required memory would be less than 8 gigabyte, which today's commodity hardware can easily handle. Our experiments in Section 3.3.4 show that, despite this limitation, we handle datasets with billions of data points. In such a setting, an efficient way to implement the necessary joins for ALS in MapReduce is to use a parallel broadcast-join [25]. The smaller dataset (U or V) is replicated to every machine of the cluster. Each machine already holds a local partition of M which is stored in the DFS. Then the join between the local partition of M and the replicated copy of V (and analogously between the local partition of M and U) executes with a map operator. This operator additionally implements the logic to re-compute the feature vectors from the join result, which means that we execute a whole re-computation of U or V using a single map operator. We broadcast M to all participating machines, which create a hashtable for its contents, the item feature vectors. M is stored in the DFS partitioned by its rows and forms the input for the map operator. The map operator reads a row m_i of M (the interaction history

of user i) and selects all the item feature vectors v_j from the hashtable holding V that correspond to non-zero entries j in m_i. Next, the map operator solves a linear system created from the interactions and item feature vectors and writes back its result, the re-computed feature vector u_i for user i. The re-computation of V works analogously, with the only difference that we need to broadcast U and store M partitioned by its columns (the interactions per item) in the DFS.

We integrate a **UDF for solving the least-squares problems** into our algorithmic framework. This UDF is invoked within the map-operator and generalizes our implementation such that different flavors of ALS are supported without having to change the distributed algorithm. Examples for such flavors are ALS formulations optimized for explicit and implicit feedback data [88, 189]. The UDF takes four input arguments: the features and interaction history which the map-operator uses, as well as a regularization constant λ and the rank of the factorization k:

$$\texttt{solve} : (features, interactions, lambda, rank) \rightarrow features$$

We replace lines 4 & 6 in Algorithm 5 responsible for the parallel feature vector re-computation with the corresponding UDF calls $u_i \leftarrow \texttt{solve}(V_{h_i}, m_{h_i}^T, \lambda, k)$ and $v_j \leftarrow \texttt{solve}(U_{h_j}, m_{h_j}, \lambda, k)$.

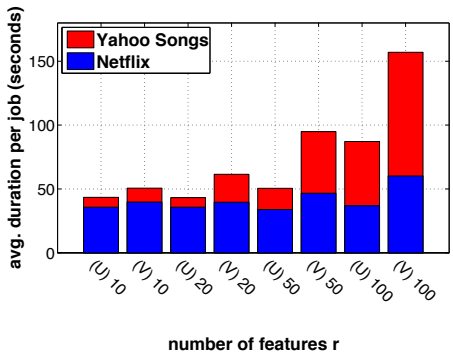
The proposed approach is able to avoid some of the drawbacks of MapReduce and common implementations (c.f., Section 2.3.3). It uses only map jobs that are easier to schedule than jobs containing map and reduce operators. Additionally, the costly shuffle-phase is avoided, in which all data would be sorted and sent over the network. As we execute the join and the re-computation using a single job, we also spare to materialize the join result in the DFS. Furthermore, we change two characteristics of the typical MapReduce execution model to further improve the performance: (1) Our implementation applies multithreaded mappers that leverage all cores of the worker machines for the re-computation of the feature matrices. We run a single map-operator per machine and all cores share the same read-only copy of the broadcasted feature matrix. Thereby we avoid having to store multiple copies of that matrix in memory for several map tasks that run on the same machine; (2) We configure Hadoop to re-use the VMs on the worker machines and cache the feature matrices in memory to avoid that later scheduled mappers have to re-read the data from disk each time upon a task invocation.

The main drawback of a broadcast approach is that every additional machine in the cluster requires another copy of the feature matrix to be sent over the network.An alternative would be to use a repartition join [25] with constant network traffic and linear scale-out. However, this technique has an enormous constant overhead. Let $|M|$ denote the number of interactions, m the number of machines in the cluster, e the replication factor of the DFS and $|T|$ the number of users or items. We assume that rating values are

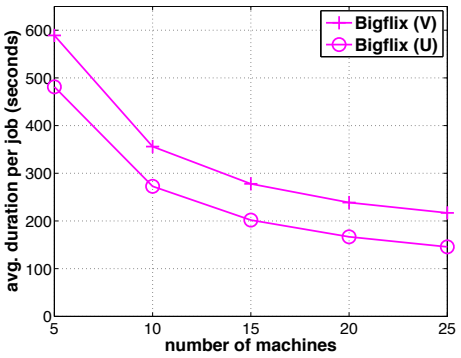
stored with single precision. If we employ a repartition-join we need two map-reduce jobs per re-computation then. In the first job, we conduct the join which sends the interaction matrix M and either U or V over the network, accounting to $|M| * 4 + |T| * e * 8$ bytes of network traffic. The result must be materialized in the DFS, which requires another $(|M| * 4 + |T| * e * 8) * n_r$ bytes of traffic. The re-computation of the feature matrix must be conducted via a second map-reduce job that sends all the ratings and corresponding feature vectors per user or item over the network, accounting for an additional $|M| * e * 8$ bytes. On the contrary, the traffic for our proposed broadcast-join technique depends on the number of machines and accounts to $|T| * e * m * 8$ bytes. Applying these estimations to the datasets used for our experiments, the cluster size would have to exceed several hundred machines to have our broadcast-join technique cause more network traffic than a computation via repartition-join. Further, this argumentation only looks at the required network traffic and does not account for the fact that the computation via repartition-join would also need to sort and materialize the intermediate data in each step. We conclude from these cost estimations that our approach with a series of broadcast-joins is to be preferred for common production scenarios.

3.3.4 Experiments

In this section, we experimentally illustrate that our approach is suitable for real-world production settings where an approach has to run in a few hours to be integratable into analytical workflows. We solely focus on measuring the runtime of computing a factorization, as the prediction quality of the ALS approach is well studied [73, 88, 189]. We use a formulation of ALS that is aimed at rating prediction [189]. The cluster for our experiments consists of 26 machines running Java 7 and Hadoop 1.0.4 [11]. Each machine has two 8-core Opteron CPUs, 32 GB memory and four 1 TB disk drives. We use two publicly available datasets for our experiments: A set of 100,480,507 ratings given to 17,770 movies by 480,189 users from the Netflix prize [21] and another set comprised of 717,872,016 ratings given to 136,736 songs by 1,823,179 users of the Yahoo! Music community [181]. For tests at industrial-scale, we generate a synthetic dataset termed Bigflix using the Myriad data generation toolkit [5]. From the training set of the Netflix prize, we extract the probability of rating each item and increase the item space by a factor of six to gain more than 100,000 movies. Next, we extract the distribution of ratings per user. We configure Myriad to create data for 25 million users by the following process: For each user, Myriad samples a corresponding number of ratings from the extracted distribution of ratings per user. Then, Myriad samples an item from the item probability distribution for each rating. The corresponding rating value is simply chosen from a uniform distribution, as we do not interpret the result anyways, but only want to look at the performance of computing the factorization. Bigflix contains 5,231,536,647



(a) Runtimes on Netflix and Yahoo Songs with different feature space sizes.



(b) Runtimes on Bigflix with different cluster sizes.

Figure 3.6: Scale-out experiments with MapReduce.

interactions and mimicks the 25 million subscribers and 5 billion ratings, which Netflix recently reported as its current production workload [8].

We start by measuring the average runtime per individual re-computation of U and V for different feature space sizes on the Netflix dataset and on the Yahoo Songs dataset (c.f., Figure 3.6a) using 26 machines. For Netflix, the average runtime per re-computation always lies between 35 and 60 seconds, regardless of the feature space size. This is a clear indication that for this small dataset the runtime is dominated by the scheduling overhead of Hadoop. For the larger Yahoo Songs dataset we see the same behavior for small feature space sizes and observe that the computation becomes dominated by the time to broadcast one feature matrix and re-compute the other one for larger feature space sizes of 50 and 100. We notice that re-computing V is much more costly then re-computing U. This happens because in this dataset, the number of users is nearly twenty times as large as the number of items, which means that it takes much longer to broadcast U. The experiments show that we can run 37 to 47 iterations per hour on Netflix and 15 to 38 iterations per hour on the Yahoo Songs dataset (e.g., ALS typically needs about 15 iterations to converge on Netflix [189]). This shows that our approach easily computes factorizations of datasets with hundreds of millions of interactions repeatedly per day.

Finally we run scale-out tests on Bigflix: we measure the average runtime per re-computation of U and V during five iterations of ALS on this dataset on clusters of 5, 10, 15, 20 and 25 machines (c.f., Figure 3.6b) conducting a factorization of rank ten. With 5 machines, an iteration takes about 19 minutes and with 25 machines, we are able to bring this down to 6 minutes. We observe that the computation speedup does not linearly scale with the number of machines. This behavior is expected because of the broadcast-join we employ, where every additional machine causes another copy of the feature matrix to be sent over the network. On clusters with less than several hundred machines, our

chosen approach is much more performant than a repartition-join, although the latter is linearly scaling. With 25 machines, we can run about 9 to 10 iterations per hour, which allows us to obtain a factorization in a few hours, a timeframe completely suitable for a production setting.

3.4 A Parameter Server Approach to Distributed Matrix Factorization

In this section, we revisit the matrix factorization problem, but drop the assumption that the model fits into the memory of a single machine, as we look at datasets with hundreds of millions of users. Furthermore, we exploratorily investigate the advantages and disadvantages of a non-dataflow architecture, which allows us to leverage asynchronicity for the factorization. Recommender systems are a prime example for ML problems where the size of the model is usually proportional to the input (e.g., the number of users in the dataset). In cases with hundreds of millions of users this model exceeds the memory of an individual machine and distributed approaches to model training that leverage multiple machines become necessary. This leads to a set of challenges, e.g. how to partition the model among the participating machines and how to correctly execute learning algorithms in such a setting. In the following, we describe 'Factorbird', a prototypical system that leverages a parameter server architecture [114] for learning large matrix factorization models for recommendation mining. Again, we compute a factorization of low-rank k of a $|C| \times |P|$ matrix M comprised of interaction data between users and items (c.f., Section 2.4.1). The factorization consists of two factor matrices U and V, such that their product UV approximates the observed parts of M and generalizes well to unobserved parts of M (c.f. Figure 3.7).

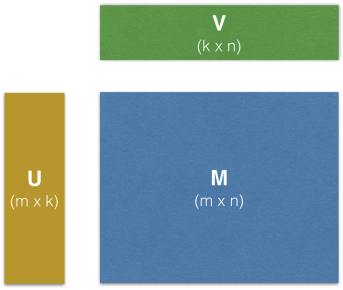


Figure 3.7: Latent factor models as matrix factorization problem.

Compared to the previous model in Section 3.3.3, we now adopt a more sophisticated approach outlined in [106]. First, we introduce a global bias term g, which captures the average interaction strength between a user and an item in the dataset. Next, we

introduce a user-specific bias term b_i^U for user i and an item-specific bias term bias b_j^V for item j. These biases model how strongly the interactions of certain users and items tend to deviate from the global average. We substitute the dot product $u_i^T v_j$ for predicting the strength of the interaction between a user i and an item j with a prediction function $p(i,j) = g + b_i^U + b_j^V + u_i^T v_j$ that takes the bias terms into account. Furthermore, we introduce a function $a(i,j)$ for determining the strength of the interaction between user i and item j. This allows us to transform the observed entries m_{ij} of M. Finally, we add a function $w(i,j)$ for weighting the prediction error for interaction between user i and item j. This function becomes useful when M consists of data from various sources with differing confidence in the observations. The loss function that we minimize for our latent factor model in Factorbird is the following:

$$\min_{g,b^U,b^V,U,V} \frac{1}{2}\left(\sum_{i,j\in M} w(i,j)(p(i,j)-a(i,j))^2\right) + \frac{\lambda}{2}\left(g^2 + \|b^U\|^2 + \|b^V\|^2 + \|U\|_F^2 + \|V\|_F^2\right)$$

We adopt a graph-specific terminology for the rest of this section, as this work originates from research on large network datasets from Twitter. Therefore, we assume that M represents a network (e.g. the network of user followings in Twitter), i and j reference vertices in this network (e.g. two users in this network) and the edges of the network correspond to observed entries of M, meaning that $a(i,j)$ depicts the weight of a directed edge between a vertex i and a vertex j in this network. Furthermore, we assume that the bias vectors b^U and b^V are stored in U and V (e.g., as first column and first row), to simplify notation.

3.4.1 Challenges in Distributed Asynchronous Matrix Factorization

First, Factorbird must handle factorizations of twitter-scale graphs with hundreds of millions of vertices and dozens of billions of edges. Scalability to datasets of this size is more important than high performance on small datasets commonly used in research (such as the Netflix [21] dataset). The matrices representing these graphs are either square (e.g. user to user followings) or 'tall-and-wide' for bipartite graphs (e.g. user to tweets). Second, the system has to be highly usable and adaptable for data scientists without making a systems background a necessity. No systems programming should be required to try out a variation of our model or a different loss function. Instead, the system should offer interfaces that abstract from the distributed execution model and are intuitive to implement given an ML background. Third, the system design shall be simple to keep the maintenance and debugging effort low. Additionally, the system design should be easily extendable into a streaming system in the future, where a previously learned matrix factorization is updated online given a stream of new observations. This

is the main reason why we decide against using ALS, which is much harder to adapt to a streaming scenario than SGD.

A delicate choice is which optimization technique to build the system on. We decided for SGD, as it is simple, provides fast convergence and is easy to adapt to different models and loss functions. Furthermore, SGD is much easier to adapt to a streaming scenario due its online nature, especially in comparison to ALS which relies on large batched updates (c.f. Section 2.4.2). Algorithm 9 shows the individual steps to conduct when learning the factorization with SGD for our model introduced in Section 3.4. First, we randomly initialize the factor matrices U and V (c.f. line 1). Next, we randomly pick an edge (i, j) from the graph and compute the weighted error e_{ij} of the prediction $p(i, j)$ against the actual edge strength $a(i, j)$ (c.f. lines 3 & 4). Next, we update the global bias term as well as the bias terms for i and j proportional to the prediction error e_{ij}, the learning rate η and the regularization constant λ (c.f. lines 5 to 7). We weight the regularization updates according the out-degree n_i of vertex i (the number of observed entries in the i-th row of M) and the in-degree n_j of j (the number of observed entries in the j-th column of M). We update the factor vectors u_i and v_j analogously (c.f. lines 8 & 9). The whole process is repeated until convergence.

Algorithm 9: Bias-aware variant of the SGD algorithm for matrix factorization.

1 randomly initialize U and V
2 **while** not converged
3 randomly pick edge (i, j)
4 $e_{ij} \leftarrow w(i, j)(a(i, j) - p(i, j))$
5 $g \leftarrow g - \eta\,(e_{ij} + \lambda g)$
6 $b_i^U \leftarrow b_i^U - \eta\left(e_{ij} + \frac{\lambda}{n_i} b_i^U\right)$
7 $b_j^V \leftarrow b_j^V - \eta\left(e_{ij} + \frac{\lambda}{n_j} b_j^V\right)$
8 $u_i \leftarrow u_i - \eta\left(e_{ij}\, v_j + \frac{\lambda}{n_i} u_i\right)$
9 $v_j \leftarrow v_j - \eta\left(e_{ij}\, u_i + \frac{\lambda}{n_j} v_j\right)$

Running SGD on datasets of Twitter scale inevitably leads to a set of challenges that have to be overcome when designing a scalable system. Our main focus is to present a prototype of a system that elegantly solves these challenges.

(1) The resulting factor matrices for a huge network quickly become larger than the memory available on an individual commodity machine [85, 173]. For example, U and V with $k = 100$ and a single precision factor representation for a graph with 250 million vertices already have a combined size of about 200 GB. This estimation does not even take operating system buffers and caches, object references and required statistics like degree distribution of the graph into account, which also compete for memory. Furthermore,

there might be cases with billions of vertices in the network (e.g., the users-to-tweets graph).

(2) Due to the sheer number of observations in large datasets, we aim to leverage multiple cores and machines to learn our desired matrix factorization. Unfortunately, SGD is an inherently sequential algorithm. It randomly picks an observation and updates the model parameters before proceeding to the next observation (c.f., Algorithm 9). When we run SGD in parallel on multiple cores, there is a chance that we concurrently try to update the same u_i or v_j, which results in conflicting writes and lost updates.

3.4.2 Parameter Server Architecture & Hogwild!-based Learning

In order to overcome the first challenge, we leverage a distributed architecture that allows us to partition the large model (the factor matrices) over several machines. We adapt a 'Parameter Server' architecture [114]. As illustrated in Figure 3.8, we partition the factor matrices over a set of machines, to which we refer to as *parameter machines*. At the same time, we partition the graph (our input data) over a set of so-called *learner machines*. Each learner machine runs multi-threaded SGD on its portions of the input data. For every observation to process, the learner machine has to fetch the corresponding factor vectors from the parameter machines, update them and write them back over the network.

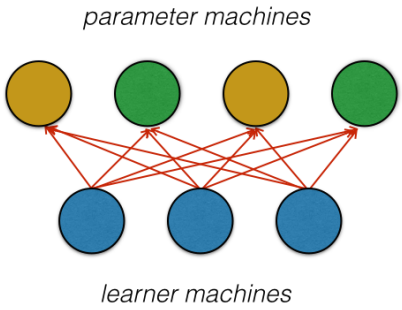


Figure 3.8: Parameter Server architecture.

This architecture inevitably leads to the second challege, the question of how to handle concurrent, possibly conflicting updates to the factor vectors. When two learner machines fetch, update and write back the same factor vector concurrently, one such update will be overridden. In the special case of matrix factorization, approaches to parallelizing SGD have been proposed that leverage a carefully chosen partitioning of the input data to avoid conflicting updates [73, 145]. As these approaches require complex data movement and

synchronization patterns and are at the same time hard to adapt to a streaming scenario, we decided for an alternative approach that is simpler to implement in a distributed setting. Instead of taking complex actions to prevent conflicting updates, Factorbird builds upon a recently proposed parallelization scheme for SGD-based learning called Hogwild! [146]. This work states that parallel SGD can be implemented *without any locking* if most updates only modify small parts of the model. The authors explicitly name latent factor models as one such case.

The special nature of the matrix factorization problem allows us for a further optimization in our system design, which reduces the required network traffic and at the same time greatly lowers the probability for conflicting overwrites. We reduce the communication cost by 50% through intelligent partitioning, as follows. If we partition M by either rows or columns over the learning machines, than the updates to either U or V become local, when we co-partition one of the factor matrices (U in the case of partitioning by rows, V in the case of partitioning by columns) with M on the learner machines. In the light of minimizing update conflict potential, we decide to co-locate V on the learner machines and keep U in the parameter machines (c.f., Figure 3.9). We choose this scheme for the following reasons: In case of the follower graph, the number of updates to a factor vector u_i in U is equal to the out-degree of the corresponding vertex i, while the number of updates to a factor vector v_j in V is equal to the in-degree of the corresponding vertex j. As the in-degree distribution of the follower graph has much higher skew than the out-degree distribution [130], we choose to localize the updates to V, which gives us a higher reduction in conflict potential than localizing U. Other graphs in twitter have similar skew in the degree distribution.

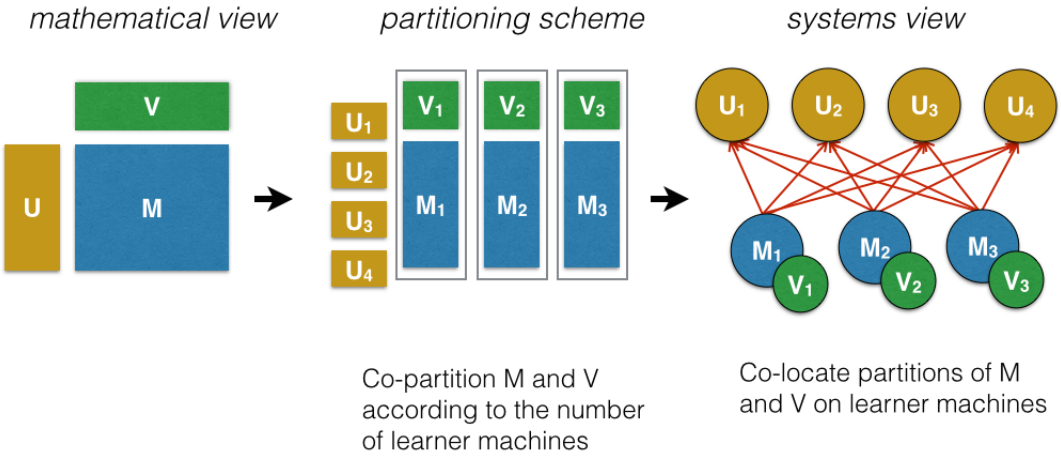


Figure 3.9: Partitioning scheme and system architecture.

Implementation: We implement Factorbird using Twitter's existing technology stack and infrastructure. For the parameter machines, we leverage an existing cluster of memcached machines, a popular distributed shared memory abstraction [63]. We implement the learner machines using Twitter's Finagle abstraction [60] for concurrent remote procedure calls. Hadoop's distributed filesystem serves as persistent datastore for reading inputs and storing the final results. We conduct prepartitioning and statistics computation of the input via MapReduce. The learner machines are assigned resources via Apache Mesos, a resource managing platform that allows different cluster computing frameworks to run on shared commodity clusters [87]. The typical execution steps on a learner machine are as follows. First, the machine loads statistics about the input graph and its assigned partition from HDFS, e.g. the number of edges and vertices, the average edge weight and the degree distribution of the graph. We use these statistics for learning as well as efficiently allocating memory for the required datastructures. Next, the learner machine instantiates its local partition of V. Subsequently, the model learning phase starts: the learner machine reads the edges of its assigned partition of the input graph in a streaming fashion. For each edge (i, j), it reads the corresponding factor vector u_i from the parameter machines and the factor vector v_j from its local partition of V. The reads from memcached are conducted in batches to increase throughput. Next, the learner machine updates the factor vectors using SGD and writes them back (c.f., Algorithm 9). Note that strictly speaking, we do not run vanilla Hogwild! on U, as memcached atomically updates the whole factor vector instead of individually updating the factors. Furthermore, we have every learner machine learn the global bias g for itself, as we would have an update conflict for every interaction otherwise.

The learner machines repeat these steps until a user-specified number of passes over the input graph has been conducted. Finally, the learner machines persist the learned matrices U and V in a partitioned manner in the distributed filesystem. Factorbird makes extensive use of memory-efficient data structures for managing factor vectors, the local partition of V and graph statistics such as the degree per vertex. The main objective of these data structures is to use as little memory as possible and to avoid object allocation and full garbage collection invocations in the JVM. A factor vector as well as a partition of a factor matrix are therefore internally represented by byte buffers and large primitive arrays, which are efficiently pre-allocated. For example, a learning machine determines the size of its local partition of V at startup time by reading the number of vertices assigned to its partition from the graph statistics. The partition of V is internally represented by a huge float array, into which the individual factor vectors (the columns of V) are packed. A mapping of vertex ids to offsets in this array is stored and update operations directly write to the underlying array. During the training phase, a learner machine directly reads the training edges from a compressed file in the distributed filesystem in a streaming fashion. If more than one pass through the training edges is necessary, the learner machine will on-the-fly create a copy of the

input file on local disk and switch to streaming edges from this local file for subsequent passes. Furthermore, some learning approaches require synthetically generated negative examples [140] (possibly taken into account with a lower confidence than observed positive examples). We therefore implement on-the-fly generation of such negative examples (with a configurable probability) and mix them into the original positive examples supplied on the learning machines.

3.4.3 Abstraction of Stochastic Gradient Descent Updates as UDF

The central abstraction for implementing the SGD updates of the learning algorithm within Factorbird is the `update` UDF. It is the main interface for data scientists wanting to try new models and shields the programmer from the complexity of the distributed nature of the underlying learning process. In this UDF, we implement the SGD-based update of two factor vectors u_i and v_j. The system provides the strength $a(i,j)$ of the edge between vertex i and j, as well the vertex degrees n_i and n_j and the error weight $w(i,j)$ as additional arguments. A typical implementation of this method conducts the steps from line 4 to 9 in Algorithm 9. This abstraction makes it very easy to implement learning algorithms for different models, e.g. it only requires a few lines of code to implement weighted matrix factorization for implicit feedback [88] in the UDF.

$$\texttt{update} : \left(g^{(t)}, b_i^{U(t)}, b_j^{V(t)}, u_i^{(t)}, v_j^{(t)}, a(i,j), n_i, n_j, w(i,j)\right) \rightarrow \left(g^{(t+1)}, b_i^{U(t+1)}, b_j^{V(t+1)}, u_i^{(t+1)}, v_j^{(t+1)}\right)$$

3.4.4 Distributed Cross-Validation using Model Multiplexing

The next aspect we focus on in Factorbird is the quality of the models we learn. Ultimately, these models have to be evaluated using online experiments with real users, but during the development and batch training phase, we concentrate on a simple offline metric: the prediction quality on held-out data, measured by the root mean squared error (RMSE). This metric has been the dominating evaluation criterion for collaborative filtering algorithms in academia in the last years [21]. Unfortunately, this prediction quality is heavily influenced by the choice of hyperparameters for our model, such as η (which controls rate of learning), λ (which controls regularization), or the number of factors k. In order to find a well-working hyperparameter combination, we conduct a grid search in the hyperparameter space. We extend Factorbird to enable hold-out tests at scale. During preprocessing, we randomly split the edges of the input graph into training set, validation set, and test set. Factorbird then learns a model on the training set, chooses the hyperparameter combination using the prediction quality on the validation set, and finally computes the RMSE on the test set.

However, conducting a single training run with Factorbird for each hyperparameter combination to inspect is tedious and takes a long time due to scheduling, resource negotation, and I/O overheads. We therefore describe a multiplexing technique to learn many models with different hyperparameters at once, in order to speed up the hyperparameter search. Given that we aim to inspect c hyperparameter combinations for a factorization of rank k, we pack c factor vectors into a large U of dimensionality $m \times c * k$ and a large V of dimensionality $c * k \times n$. We use a specialized implementation of the `update` UDF that is aware of the multiplexing (e.g. , it knows that the factors for the p-th model are contained in the $p * k$-th to $p * (k + 1)$-th entries of a factor vector) and learns c models at once. Figure 3.10 illustrates how the factor matrices for all possible combinations of two different learning rates η_1, η_2 and two different regularization constants λ_1, λ_2 would be packed into large matrices U and V.

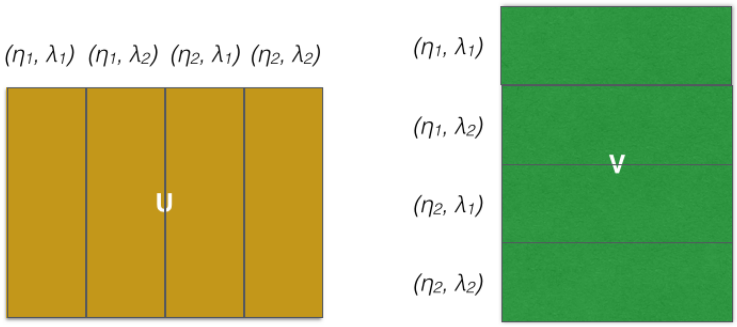


Figure 3.10: Packing many models into one for hyperparameter search.

In order to compute the prediction quality on held-out data, Factorbird requires another UDF called `predict`, which predicts the strength $\widehat{m_{ij}}$ of an unobserved edge (i, j) from the corresponding factor vectors u_i and v_j, as well as the biases g, b_i^U and b_j^V:

$$\texttt{predict} : \left(g, b_i^U, b_j^V, u_i, v_j\right) \rightarrow \widehat{m_{ij}}$$

Optionally, we provide functionality to estimate the current value of the loss function using samples of edges and factors, and make the resulting estimates inspectable via an external dashboard.

3.4.5 Experiments

We run experiments on various subsets of 'RealGraph', a graph that models various interactions between Twitter users [96]. The learner machines for our experiments are

provisioned by Apache Mesos. In all our experiments, we factorize the binarized adjacency matrix of the graph subset. That means, the transformation function $a(i, j)$ returns 1 if user i interacted with user j and 0 otherwise (in the case of a synthetic negative example). We equally weight all prediction errors ($w(i, j) = 1$). Due to the limited time in which we had access to Twitter's computing infrastructure, we only present preliminary experiments, aimed at validating the correctness of our system and showing its capacity to handle twitter-scale graphs. There is still a large potential for improvements in accuracy and performance that need to be tackled in future work by someone with access to a similar infrastrucuture. We run a first set of experiments on a small sample of the RealGraph, consisting of 100 million interactions between 440 thousand popular users. Additionally, we make Factorbird generate 500 million synthetic negative edges.

Benefits of increasing model complexity. In the first experiment, we show the positive effects on prediction quality of the individual parts of our chosen model. We randomly split the dataset into 80% training set, 10% validation set and 10% test set. We train models with increasing complexity and measure their prediction quality in terms of RMSE on the 10% held-out data in the test set (c.f., Figure 3.11). We start with a baseline that only uses the global bias (the average edge strength), followed by a more complex model that uses the global bias as well as vertex-specific bias terms. Finally, we train biased factorization models with $k \in \{2, 5, 10, 20\}$. We choose the hyperparameters using the validation set. The outcome confirms that an increase in the complexity of our model results in an increase in prediction quality. The global bias baseline provides an RMSE of 0.3727, adding the vertex-specific bias terms reduces the error to 0.3121 and incorporating the factors gives additional improvements, reducing the error to 0.2477 for $k = 20$.

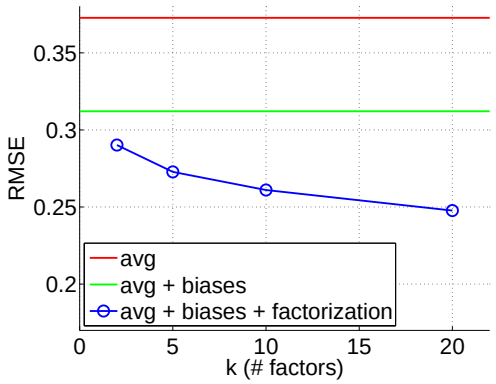


Figure 3.11: Prediction quality on held-out data with increasing model complexity.

Visual inspection. Next, we plot a selection of factor vectors from the V matrix of a biased factorization with $k = 2$. Two users i and j will be close in the resulting

low-dimensional space if their follower vectors are roughly linear combinations of each other. Due to homophily, we expect Twitter users sharing a common interest or characteristic to have a large overlap in followers and therefore to be much closer in the resulting space. Figure 3.12 indicates that factorizations produced by Factorbird have this property. We see several clusters of Twitter accounts of similar type, e.g. a cluster of european politicians, containing Sigmar Gabriel (`@sigmargabriel`) and Reinhard Buetikofer (`@bueti`) from Germany as well as Arseniy Yatsenyuk (`@Yatsenuk_AP`) from Ukraine. Another cluster consists of popstars such as Kanye West (`@kanyewest`), Justin Bieber (`@justinbieber`) and the boy-band *One Direction* (`@onedirection`). A third one related to U.S. sports contains Emma Span (`@emmaspan`), an editor for baseball at Sports Illustrated, Detroit's football team (`@lions`) and an account of the Daytona International Speedway (`@disupdates`).

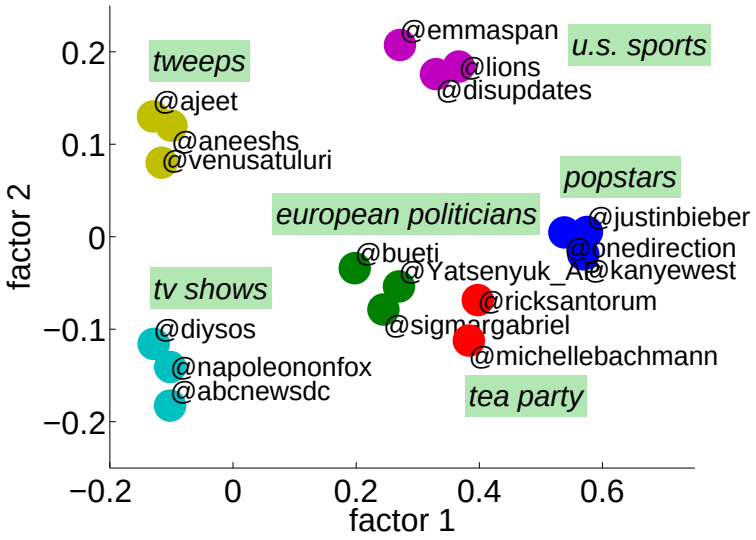


Figure 3.12: Plot of a selection of twitter users as positioned by a factorization with $k = 2$ of a sample of RealGraph.

Scale-out. For the scale-out experiments we build a matrix based on a large subset of Twitter's RealGraph [96] consisting of more than 38.5 billion non-zeros (half of which are synthetic negative examples). The dimensions of the matrix are 229 million $\times$ 195 million. To the best of our knowledge, this is the largest dataset on which collaborative filtering experiments have been published so far. In comparison to the Netflix prize dataset [21], which is often used for scale-out tests [117], our dataset is more than two orders of magnitude larger, has approximately 470 times more rows, 11,000 times more columns and 385 times more datapoints. We run Factorbird using 50 instances for the learner machines, provisioned in a Mesos cluster, with 16 cores and 16 GB of memory each. We leverage a shared memcached cluster for our parameter machines,

with a guaranteed quota of 5 million commands per second. We run two passes of hyperparameter search on 80% of the data for 16 different learners with $k = 5$. The rank of the factor matrices U and V is $(5 + 1) * 16 = 96$, which means we train a model with approximately $(229M + 195M) * 96 \approx 40B$ parameters. In this experiment, Factorbird issues more than 4.2 million commands per second to memcached on average and updates around 400M parameters per second. A single SGD pass through the training set finishes in about 2.5 hours on the majority of learner machines. Unfortunately, we experience that a few straggler machines take much longer to finish the computation. We find that these stragglers suffer from a heavily reduced network bandwidth to the parameter machines (up to only a third of the bandwidth of the non-straggling machines). This issue is caused by multi-tenancy in the cluster: we share our machines with other applications that in some cases cause extreme amounts of network traffic. Unfortunately, the resource manager Mesos cannot guarantee a defined network bandwidth for applications at the moment. For high performance, Factorbird needs to either be run in a different cluster or must implement work-stealing algorithms to mitigate the straggler effects. Nevertheless, this experiment clearly shows the potential of our prototype to handle Twitter-scale datasets.

3.5 Centrality Computation in Dataflow and Vertex-Centric Systems

In the following sections, we briefly discuss two negative results from our research on distributed graph processing. We first highlight a scalability bottleneck in path-based centrality computations, introduced by choosing a distributed architecture over a single machine setup. Secondly, we discuss a simple algorithm which only slightly varies from common examples like PageRank, but for which the vertex-centric programming model fails as easy-to-use abstraction.

3.5.1 Geometric Centrality with Hyperball

We put our focus on the distributed computation of geometric centralities such as closeness centrality and harmonic centrality in large networks (c.f., Section 2.4.3). A distributed algorithm to estimate the neighborhood function of the graph (from which these centralities are computed) nicely fits into vertex-centric graph processing (c.f., Section 2.3.4): We conduct a series of bitstring exchanges between the vertices of the network (c.f., Section 2.4.4).

An exact computation of these centrality measure would require memory quadratic in the number of vertices of the graph and would therefore not scale. We adopt a common approximation technique to overcome this bottleneck: we approximate the required

cardinality sets with a sketch, in our case with a hyperloglog counter [27, 64, 98, 139]. A vertex-centric variant of the algorithm is shown by the following pseudocode:

```
class HyperBall extends Vertex {

  compute(neighborCounters) {
    HyperLogLog counter = getValue()
    if (inFirstSuperstep()) {
      counter.observe(vertexId());
      sendMessageToAllEdges(vertex, counter);
    } else {
      numSeenBefore = counter.count();
      for (HyperLogLog neighborCounter : neighborCounters) {
        counter.merge(neighborCounter);
      }
      numSeenNow = counter.count();
      if (numSeenNow > numSeenBefore) {
        storeEstimate(vertexId(), getSuperstep(), numSeenNow);
        sendMessageToAllEdges(vertex, counter);
      }
    }
    vertex.voteToHalt();
  }
}
```

In the first iteration, every vertex adds its identifier to the hyperloglog counter and sends this counter to all incident neighbors. In a subsequent iteration t, each vertex i merges the counters received from its neighbors to re-compute its stored counter. If the value of the counter has changed, the vertex sends the updated counter to all its neighbors for receival in the next iteration. The updated value of the counter (represented by the variable `numSeenNow` in the pseudocode) gives the value of the neighborhood function $N(i, t)$ for this vertex i in iteration t. The algorithm convergences once no counter changes its value anymore. Therefore, the number of iterations required equals the diameter of the graph. Additionally, we use a combiner to pre-aggregate the counters destined for a particular vertex.

We aim to repeat the experiment presented in [29]: Computing closeness centrality on the large hyperlink graph of the Clueweb corpus [43], which consists of 4,780,950,903 vertices connected by 7,944,351,835 links. We therefore implement the vertex-centric variant in Apache Giraph 1.0 and use our cluster of 25 machines for the experiment. Each machine has two 8-core Opteron CPUs, 32 GB memory and four 1 TB disk drives. Unfortunately, the computation always crashes during the first iteration due to insufficient memory in the cluster. On the contrary, a single machine implementation on a machine with 40 cores and 1 TB of RAM computes these centralities in 96 and 422 minutes, dependening on

the desired accuracy of the computation [29]. Although our cluster has only slightly less memory in aggregate than the single machine (800 GB vs 1 TB), we find that the memory requirements imposed by the distributed architecture pose a scalability bottleneck for this particular dataset. The main problem is that Giraph requires two copies of the counter for every vertex to be in memory. The first copy holds the actual vertex value, while the second copy lives in a hash-table which is used for hash-aggregating the received messages. These memory requirements for the two copies (plus the memory required for auxiliary datastructures such as network buffers) combined with the extreme amount of vertices in the dataset cause the crashes due to insufficient memory. We conclude that there are graph datasets and use cases where the economics of distribution clearly speak against a vertex-centric system on a commodity cluster and favor a non-distributed solution on a single machine with large memory.

3.5.2 Flow-based Centrality with LineRank

In this section, we compare the vertex-centric programming model of Pregel-like systems to the general dataflow abstraction in Apache Flink for implementing the *LineRank* algorithm [97]. LineRank is a spectral centrality measure and conceptually very similar to PageRank which is often used as prime example for vertex-centric programs (c.f., Section 2.4.3). Analogous to PageRank, LineRank models centrality by the dominant eigenvector of a large matrix representing the network. While PageRank uses a markov chain directly derived from the links in the network, LineRank first weighs edges by computing the dominant eigenvector of a markov chain derived from the line graph of the network. The centrality of a vertex is then computed as the sum of the centralities of its adjacent edges. As the line graph L_G of a graph G quickly becomes intractably large, LineRank operates on a decomposition of L_G into the product of the source incidence matrix S_G with the target incidence matrix T_G of the network. Algorithm 10 shows the steps required for calculating LineRank. We first compute a normalization vector d (c.f., line 1). Next, we use the repeated matrix vector multiplications of the Power Method (c.f., Section 2.4.4) to compute the dominant eigenvector of the transition matrix of the line graph (c.f., lines 4 & 5). Note that we conduct two matrix vector multiplications as we leverage the decomposition $T_G S_G^T$ of the line graph to not have to materialize L_G. Finally, we multiply the sum of the incidence matrices S_G and T_G by the current eigenvector estimate to retrieve the LineRank (c.f., line 6).

In a series of experiments with a Flink and a Giraph implementation of the algorithm, we find that both systems provide comparable performance that only differs in a constant factor, which we attribute to implementation details [41]. However, we encountere huge difficulties in implementing LineRank using the vertex-centric abstraction, although it only slightly differs from PageRank, which fits very nicely into the vertex-centric

Algorithm 10: LineRank algorithm as proposed in [97].

1 $d \leftarrow 1 \,/\, (T_G S_G^T \mathbb{1})$
2 $v \leftarrow$ random vector of size m
3 $r \leftarrow \frac{1}{m}\mathbb{1}$
4 **while** v not converged:
5 $\quad v \leftarrow c\left(T_G S_G^T (d \otimes v)\right) + (1-c)\, r$
6 **return** $(S_G + T_G)^T v$

paradigm [120]. The first problem we ecounter is the creation of the incidence matrices T_G and S_G which represent the line graph of the network. The incidence matrices require a global sorting of the edges of the graph, as the row index in an incidence matrix refers to the sorting index of the corresponding edge. Dataflow systems like Flink offer operators for sorting data, but the vertex-centric paradigm lacks an abstraction for such simple tasks. We therefore manually pre-process the datasets before handing them to Giraph. The next difficulty lies in the two matrix-vector multiplications conducted in line 5 of Algorithm 10. While Flink allows us to simply model a matrix-vector multiplication using a join followed by a reduce operator, it is extremely tedious to implement these steps in Giraph. We have to represent both incident and adjacent edges of every vertex of the network and implement complex messaging logic in the vertex update function [41]. Finally, the addition of the incidence matrices in line 6 requires a join abstraction, which is completely lacking in Pregel-like systems; we therefore even refrained from implementing the last step in Giraph. We conclude that while the vertex-centric abstraction is popular and very naturally fits a set of example algorithms like PageRank or shortest paths, it fails to generalize to slightly more complex algorithms from the same field. These more complex algorithms are easily expressable in general dataflow systems such as Apache Flink, Hyracks or Apache Spark, which can also emulate vertex-centric programs [36, 62, 177, 180].

3.6 Related Work

Most closely related to our work on scaling item-based collaborative filtering is a MapReduce formulation presented by Jiang et al. [93]. However they do not show how to use a wide variety of similarity measures, they do not achieve linear scalability as they undertake no means to handle the quadratic complexity introduced by 'power users' and only present experiments on a small dataset. Another distributed implementation of an item-based approach is used by Youtube's recommender system [47], which applies a domain specific way of diversifying the recommendations by interpreting the pairwise

item similarities as a graph. Unfortunately this work does not include details that describe how the similarity computation is actually executed other than stating it uses a series of MapReduce computations walking through the user/video graph. Furthermore, a very early implementation of a distributed item-based approach was applied in the recommendation system of the TiVo set-top boxes [6], which suggests upcoming TV shows to its users. In a proprietary architecture, show correlations are computed on the server side and preference estimation is afterwards conducted on the client boxes using the precomputed correlations. While we only empirically motivate our sampling scheme, Zadeh et al. present a non-adaptive sampling-scheme for similarity matrix computations on MapReduce with proven statistical guarantees [185].

SGD- and ALS-based matrix factorization techniques for recommender systems have been extensively studied in the context of the Netflix Prize [106, 189] and beyond [85, 173]. These techniques have been extended to work on implicit feedback data [88] and to optimize metrics different from RMSE [147, 178]. This resulted in a huge body of work on parallelizing the computation of latent factor models: The recommender system of Google News [46] uses a MapReduce based implementation combining Probabilistic Latent Semantic Indexing and a neighborhood approach with a distributed hashtable that tracks item co-occurrences in realtime. As Google uses a proprietary implementation of MapReduce, it is hard to assess the technical details of this approach. Zhou et. al. describe a distributed implementation using Matlab [189]. GraphLab [117] is a specialized system for parallel machine learning, where programs operate on a graph expressing the computational dependencies of the data. GraphLab provides an asynchronous implementation of ALS that models the interaction matrix as a bipartite graph where users and items are represented by vertices and interactions by edges among them. A similar synchronous implementation has been proposed for the graph abstraction of Apache Spark [180]. Gemulla et. al presented a parallel matrix factorization using Stochastic Gradient Descent [73], which leverages an intelligent partitioning to avoid conflicting updates, where each iteration only works on a subset of the data and leverages an intelligent partitioning that avoids conflicting updates. While their approach converges faster than ALS, they switched their implementation to MPI [167] due to the overhead incurred by Hadoop. Recht et al. proposed a biased sampling approach to avoid conflicting updates during parallel training [145] and even proofed convergence under a minor amount of update conflicts [146]. Recently, Yun et. al. presented a decentralized algorithm with non-blocking communication for fast matrix factorization called NOMAD [184], where processors asynchronously exchange ownership of model partitions. Our work in matrix factorization on MapReduce builds on popular ALS formulations and has been used as a baseline for systems like GraphLab and Spark. Factorbird combines Hogwild!-style learning with a parameter server architecture [114] and presents a real-world system design based on current research.

Computing centrality measures at scale has been a much researched topic, both in MapReduce-based systems [49, 97–99], general dataflow systems [41, 180] as well as graph-based abstractions [117]. Recently, it has been shown that the large networks commonly used in academia can be efficiently processed on single machines [29, 112, 119, 123].

3.7 Conclusion

We showed how to build a scalable, neighboorhood-based recommender system based on the MapReduce paradigm. We re-phrased the underlying similiarity computation to run on a parallel processing platform with partitioned data and described how a wide variety of measures for comparing item interactions easily integrate into our method. We introduced a down sampling technique called interaction-cut to handle the computational overhead introduced by 'power users'. For a variety of datasets, we experimentally showed that the prediction quality quickly converges to that achieved with unsampled data for moderately sized interaction-cuts. We demonstrated a computation speedup that is linear in the number of machines on a huge dataset of 700 million interactions and showed the linear scale of the runtime with a growing number of users on that data. Continuing the ML on MapReduce work, we presented an efficient scalable approach for a data-parallel low-rank matrix factorization using Alternating Least Squares on a cluster of commodity machines. We explained our choice of ALS as algorithm and described how to implement it on MapReduce with a series of broadcast-joins, which can be efficiently executed using only map jobs. We experimentally validated our approach on two large datasets commonly used in research and on a synthetic dataset with more than 5 billion datapoints, which mimicks an industry usecase. There are two important limitations in the approaches presented. The similarity-based methods are highly dependent on the number of items in the dataset. While our selective-down sampling technique helps mitigate the quadratic character of the computation, the number of comparisons as well as the result size still heavily depend on the number of items in the dataset. Therefore this approach is best used in scenarios where the number of users largely exceeds the number of items (e.g., in most online shopping scenarios). Our presented ALS approach is limited by the assumption that the low-rank factor matrices fit into the main memory of the individual machines of the cluster. In cases where this assumption is violated, we recommend to switch to a parameter server-based approach.

We revisited the matrix factorization problem and combined a parameter server architecture with asynchronous, lock-free Hogwild!-style learning to a prototype named Factorbird. We proposed a special partitioning scheme to drastically reduce network traffic and conflicting updates. We furthermore showed how to efficiently grid search for hyperparameters at scale. In the end, we present experiments with our system on a matrix consisting of more than 38 billion non-zeros and about 200 million rows and

columns. Again, to the best of our knowledge, this is the largest matrix on which factorization results have been reported in the literature. In future work, Factorbird should be extended to a streaming scenario by bootstrapping it with a factorization that was trained offline and then having it update this factorization online from a stream of incoming real-time interactions. Furthermore, factorizing multiple matrices at once in order to incorporate different types of interactions could improve model quality. Biased sampling of the edges would allow us to use retrieve factor vectors for more than a single update and thereby reduce network traffic. There are many technical improvements to be made: e.g., a possible approach to recovery in case of failures in Factorbird could be to restart learner and parameter machines from asynchronously written checkpoints of the partitions of U and V which they hold [156]. It will be beneficiary to replace memcached with a custom application to be able to achieve higher throughput and conduct true Hogwild!-style updates on the parameter machines. Moreover, this would allow to run aggregations on the parameter machines. Additionally, dynamic load adaption in Factorbird might help to mitigate the negative effects of stragglers on the overall runtime. Another interesting research direction for future work would be to conduct an in-depth experimental comparison of synchronous ALS-based matrix factorization systems and asynchronous parameter-server based systems like Factorbird. We assume that parameter servers only make sense for extremely large datasets and models. In distributed systems, performance is heavily influenced by the amount of network traffic produced by a system. A rough comparison of the traffic caused by Factorbird versus the traffic caused by broadcast-based ALS shows different scaling behaviors. Let w be the number of workers in the cluster, $|M|$ the number of interactions (the number of non-zero entries in M), $|C|$ the number of users, $|P|$ the number of items and k the rank of the factorization. In every iteration, ALS needs to broadcast the two feature matrices to all participating machines, causing an estimated network traffic of $w * ((|C| + |P|) * k)$, which is proportional to the number of workers. Factorbird's produced network traffic on the other hand consists of the communication with the parameter machines and amounts to $2 * |M| * k$. We see that this traffic is independent of the number of workers and solely depends on the input data. For smaller datasets, ALS should be more efficient, but there should be a tipping point where parameter-server like architecture are to be favored. Note that this comparison does not take into account that asynchronous systems typically learn faster and therefore need fewer passes over the data.

With respect to distributed graph processing, we highlighted scalability bottlenecks in the systems themselves, as well as short-comings of the vertex-centric programming model.

4 Optimistic Recovery for Distributed Iterative Data Processing

4.1 Problem Statement

In the previous chapter, we focused on fitting existing algorithms into the programming models of parallel dataflow systems, thereby tackling algorithm-level bottlenecks in scaling data mining algorithms. Yet, as discussed in Section 1.1, further problems need to be addressed to achieve scalability in data mining. In this chapter, we concentrate on systems aspects arising during the distributed execution of iterative algorithms. Such algorithms repeat a step function until a termination condition is met; many important algorithms from the fields of machine learning and graph processing exhibit this nature. Examples include ALS and SGD for computing latent factor models as discussed in Section 2.4.2, as well as the large number of graph algorithms solvable by generalized iterative matrix vector multiplication as described in Section 2.4.4. The unique properties of iterative tasks open up a set of research questions related to building distributed data processing systems. We focus on improving the handling of machine and network failures during the distributed execution of iterative algorithms. We concentrate on problems where the size of the evolved solution is proportional to the input size and amends itself to be partitioned among the participating machines in order to scale to large datasets.

The traditional approach to fault tolerance under such circumstances is to periodically persist the application state as checkpoints and, upon failure, restore the state from previously written checkpoints and restart the execution. This pessimistic method is commonly referred to as *rollback recovery* [59]. Realizing fault tolerance in distributed data analysis systems is a complex task. The optimal approach to fault tolerance depends, among other parameters, on the size of the cluster, the characteristics of the hardware, the duration of the data analysis program, the duration of individual stages of the program (i.e., the duration of an iteration in our context), and the progress already made by the program at the time of the failure. Most fault tolerance mechanisms introduce overhead during normal (failure-free) operation, and recovery overhead in the case of failures [59]. An alternative approach to fault tolerance that does not rely on checkpointing is lineage-based recovery (c.f., Section 2.3.3). Lineage-based recovery is not applicable to many iterative algorithms however for the following reason. If the step function of the iteration contains an operation with dependencies between all parallel partitions (e.g., via a reduce operator), then lineage-based recovery results in a complete restart of the iterative computation: The only way to re-compute the transitive dependencies of lost partitions

in iteration i is to re-compute iteration $i-1$, which relies on the result of iteration $i-2$, etc. These dependencies cascade back to the initial iteration state. We classify recovery mechanisms for large-scale iterative computations in three broad categories, ranging from the most pessimistic to the most optimistic: operator-level pessimistic recovery, iteration-level pessimistic recovery, and the optimistic recovery mechanism proposed in this paper. Pessimistic approaches assume a high probability of failure, whereas optimistic approaches assume low failure probability.

Operator-level recovery, implemented in MapReduce [49], checkpoints the result of every individual program stage (the result of the Map stage in MapReduce). Its sweet spot is very large clusters with high failure rates. This recovery mechanism trades very high failure-free overhead for rapid recovery. In the iterative algorithms setting, such an approach would be desirable if failures occur at every iteration, where such an approach would be the only viable way to allow the computation to finish. For iterative algorithms, the amount of work per iteration is often much lower than that of a typical MapReduce job, rendering operator-level recovery an overkill. However, the total execution time across all iterations may still be significant. Hence, specialized systems for iterative algorithms typically follow a more optimistic approach. In *iteration-level recovery*, as implemented for example in graph processing systems [117, 120], the result of an iteration as a whole is checkpointed. In the case of failure, all participating machines need to revert to the last checkpointed state. Iteration-level recovery may skip checkpointing some iterations, trading better failure-free performance for higher recovery overhead in case of a failure. For pessimistic iteration-level recovery to tolerate machine failures, it must replicate the data to checkpoint to several machines, which requires extra network bandwidth and disk space during execution. The overhead incurred to the execution time is immense, in our experiments, we encounter that iterations where a checkpoint is taken take up to 5 times longer than iterations without a checkpoint. Even worse, a pessimistic approach always incurs this overhead, regardless whether a failure happens or not. An extreme point of iteration-level recovery is "no fault tolerance", where checkpoints are never taken, and the whole program is re-executed from scratch in the case of a failure.

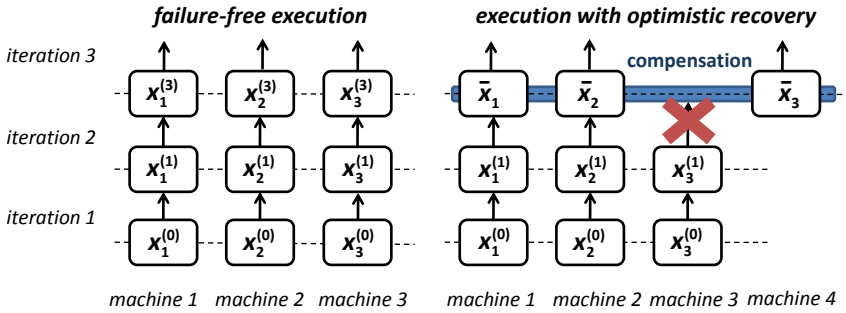


Figure 4.1: Optimistic recovery with compensations illustrated.

As alternative, we propose an optimistic recovery approach which never checkpoints any state. Thus, it provides optimal failure-free performance, as its failure-free overhead is virtually zero (Section 4.6 experimentally validates this) and at the same time, it requires much less resources in the cluster compared to a pessimistic approach. Furthermore, this optimistic approach only incurs overhead in case of failures, in the form of additional iterations required to compute the solution. Figure 4.1 illustrates the optimistic recovery scheme for iterative algorithms: Failure-free execution proceeds as if no fault tolerance is desired. In case of a failure, we propose to finish the current iteration ignoring the failed machines and simultaneously acquire new machines or re-distribute the work on existing machines. These initialize the relevant iteration-invariant data partitions. Then, the system applies a user-supplied piece of code that implements a *compensate* function, on every machine. This compensation function sets the algorithm state to a consistent state from which the algorithm will converge (e.g., if the algorithm computes a probability distribution, the compensation function could have to make sure that all partitions sum to unity). After that, the system proceeds with the execution. The compensation function can be thought of as bringing the computation "back on track", where the errors introduced by the data loss are corrected by the algorithm itself in the subsequent iterations. Figure 4.2 illustrates the different approaches. We see an exemplary iterative algorithm that converges to a fixpoint after four steps without failures. In the case of pessimistic failure recovery, it checkpoints the algorithm state after the first step and fails in the second step. The algorithm simply restarts from the first iteration and repeats exactly the same steps that it would have conducted in the failure-free case. With our proposed optimistic recovery, the algorithm never checkpoints any state. Instead, the compensation function restores a valid intermediate state after the failure in iteration two. This results in a jump to another point in the space of intermediate solutions from which the algorithm will still converge to the fixpoint. Note that this recovery might introduce additional steps after a failure.

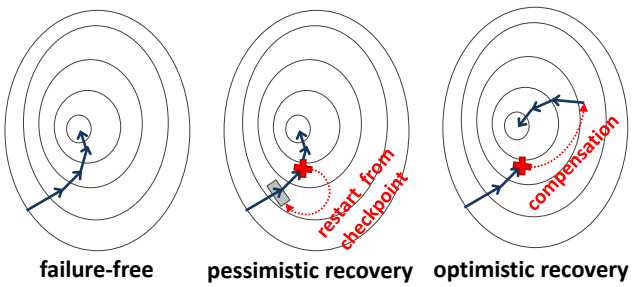


Figure 4.2: Different strategies to reach a fixpoint under potential failures in iterative data processing.

In the majority of our simulations, the optimistic approach provided a superior runtime, as additional iterations with a compensation function result in a shorter overall execution

time than writing checkpoints and repeating iterations with a pessimistic approach. Therefore, we conjecture that our proposed approach is a good fit for running iterative algorithms on clusters with moderate failure rates, given that a compensation for the algorithm to execute is known. Note that for many algorithms, it is easy to write a compensation function whose application will provably lead the algorithm to convergence (cf., Section 4.5, where we provide templates for compensating a variety of algorithms). Our experiments in Section 4.6 show that our proposed optimistic recovery combines optimal failure-free performance with fast recovery and at the same time outperforms a pessimistic approach in the vast majority of cases.

4.2 Contributions

We propose a novel optimistic recovery mechanism that does not checkpoint any state (c.f., Section 4.1). It provides optimal failure-free performance with respect to the overhead required for guaranteeing fault tolerance, and simultaneously uses less resources in the cluster than traditional approaches. Next, we discuss how to extend the programming model of a massively parallel data flow system to allow users to specify compensation functions. The compensation function becomes part of the execution plan, and is only executed in case of failures (c.f., Section 4.4). Additionally, we describe how our proposed recovery scheme fits into vertex-centric graph processing systems (c.f., Section 4.3.3). In order to showcase the applicability of our approach to a wide variety of problems, we explore three classes of problems from our exemplary data mining use cases (c.f., Section 2.4), which are commonly solved with distributed, iterative data processing. We start by looking at approaches to carry out link analysis and to compute centralities in large networks. Techniques from this field are extensively used in web mining for search engines and social network analysis. Next, we describe path problems in graphs which include standard problems such as reachability and shortest paths. Lastly, we look at distributed methods for factorizing large, sparse matrices, a field of high importance for personalization and recommendation mining. For each class, we discuss solving algorithms and provide blueprints for compensation functions (c.f., Section 4.5). A programmer or library implementor can directly use these functions for algorithms that fall into these classes. Finally, we evaluate our proposed recovery mechanism by applying several of the discussed algorithms to large datasets (c.f., Section 4.6). We compare the effort necessary to reach the solution after simulated failures with traditional pessimistic approaches and our proposed optimistic approach. Our results show that our proposed recovery mechanism provides optimal failure-free performance. In many failure cases, our recovery mechanism is superior to pessimistic approaches for recovering algorithms that incrementally compute their solution. For non-incremental algorithms that recompute the whole solution in each iteration, we find our optimistic scheme to be superior to pessimistic approaches for recovering from failures in early iterations.

4.3 Distributed Execution of Fixpoint Algorithms

In the following, we introduce a general model for expressing fixpoint algorithms. This model will form the basis for the distributed execution of fixpoint algorithms in different systems. We will furthermore describe the compensation function in terms of this model.

4.3.1 Fixpoint Algorithms as Dataflows

We restrict our discussion to algorithms that can be expressed by a general fixpoint paradigm taken from Bertsekas and Tsitsiklis [22]. Given an n-dimensional state vector $x \in \mathbb{R}^n$, and an update function $f(x) : \mathbb{R}^n \to \mathbb{R}^n$, each iteration consists of an application of f to x: $x^{(t+1)} = f(x^{(t)})$. The algorithm terminates when we find the fixpoint x^* of the series $x^{(0)}$, $x^{(1)}$, $x^{(2)}$, ..., such that $x^* = f(x^*)$. The update function f is decomposed into component-wise update functions f_i. The function $f_i(x)$ updates component $x_i^{(t)}$ such that $x_i^{(t+1)} = f_i(x_1^{(t)}, x_2^{(t)}, \ldots, x_n^{(t)})$ for $i = 1, \ldots, n$. An update function f_i might only depend on a few components of the state $x^{(t)}$. The structure of these *computational dependencies of the data* is described by the *dependency graph* $G_{dep} = (N, E)$, where the vertices $N = \{x_1, \ldots, x_n\}$ represent the components of x, and the edges E represent the dependencies among the components: $(i, j) \in E \Leftrightarrow f_i$ depends on x_j. The dependencies between the components might be subject to additional parameters, e.g., distance, conditional probability, transition probability, etc, depending on the semantics of the application. We denote the parameter of the dependency between components x_i and x_j with a_{ij}. This dependency parameter can be thought of as the weight of the edge e_{ij} in G_{dep} . We denote the adjacent neighbors of x_i in G_{dep}, i.e., the computational dependencies of x_i, as Γ_i (cf. Figure 4.3).

We define a simple programming model for implementing iterative algorithms based on the introduced notion of fixpoints. Each algorithm is expressed by two functions. The first function is called *initialize* and creates the components of the initial state $x^{(0)}$:

$$\texttt{initialize} : i \to x_i^{(0)}$$

The second function, termed *update*, implements the component update function f_i. This function needs as input the states of the components which x_i depends on, and possibly parameters for these dependencies. The states and dependency parameters necessary for recomputing component x_i at iteration t are captured in the *dependency set* $D_i^{(t)} = \{(x_j^{(t)}, a_{ij}) \,|\, x_j^{(t)} \in \Gamma_i\}$. The function computes $x_i^{(t+1)}$ from the dependency set $D_i^{(t)}$:

$$\texttt{update} : D_i^{(t)} \to x_i^{(t+1)}$$

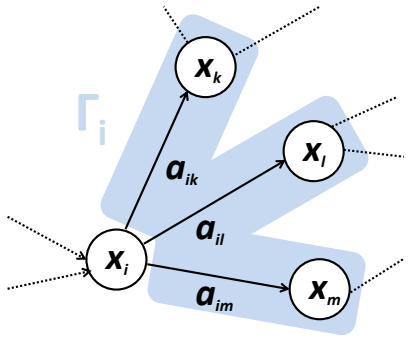


Figure 4.3: Adjacent neighbors Γ_i of x_i in the dependency graph, a representation of x_i's computational dependencies.

We refer to the union $D^{(t)}$ of the dependency sets $D_i^{(t)}$ for all components i as the *workset*. In order to detect the convergence of the fixpoint algorithm in distributed settings, we need two more functions. The function `aggregate` : $(x_i^{(t+1)}, x_i^{(t)}, agg) \rightarrow agg$ incrementally computes a global aggregate agg from the current value $x_i^{(t+1)}$ and the previous value $x_i^{(t)}$ of each component x_i. It is a commutative and associative function. The function `converged` : $agg \rightarrow \{true, false\}$ decides whether the algorithm has converged by inspecting the current global aggregate.

Example: PageRank (c.f., Section 2.4.3) is an iterative method for ranking web pages based on the underlying link structure. Initially, every page has the same rank. At each iteration, every page uniformly distributes its rank to all pages that it links to, and recomputes its rank by adding up the partial ranks it receives. The algorithm converges when the ranks of the individual pages do not change anymore. For PageRank, we start with a uniform rank distribution by initializing each x_i to $\frac{1}{n}$, where n is the total number of vertices in the graph. The components of x are the vertices in the graph, each vertex depends on its incident neighbors, and the dependency parameters correspond to the transition probabilities between vertices. At each iteration, every vertex recomputes its rank from its incident neighbors proportionally to the transition probabilities:

$$\texttt{update} : D_i^{(t)} \rightarrow 0.85 \sum_{D_i^{(t)}} a_{ij} x_j^{(t)} + 0.15 \frac{1}{n}.$$

The aggregation function computes the L1-norm of the difference between the previous and the current PageRank solution, by summing up the differences between the previous and current ranks, and the algorithm converges when this difference becomes less than a given threshold.

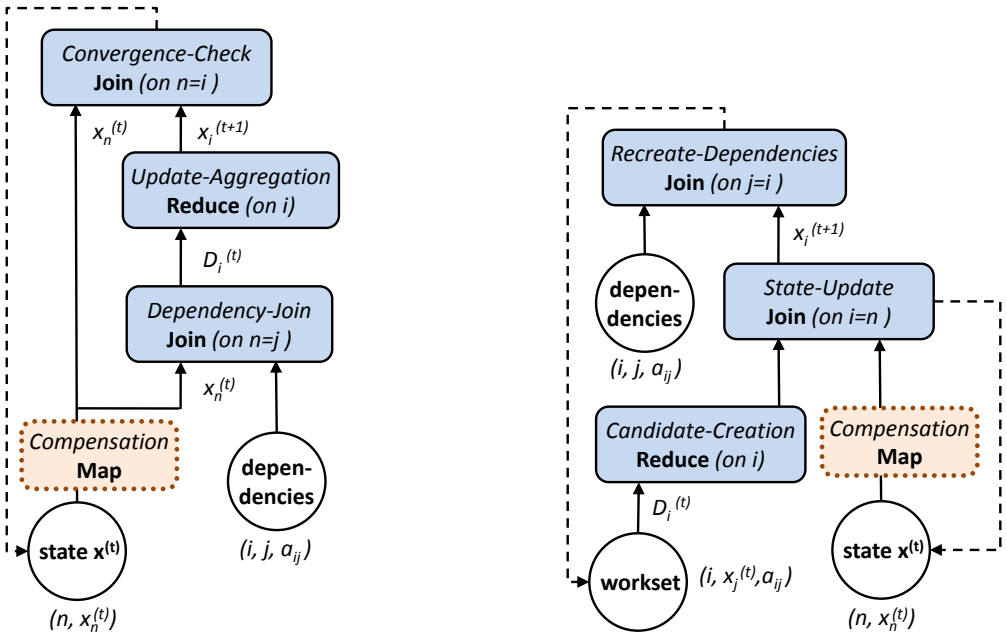
4.3.2 Bulk-Synchronous Parallel Execution of Fixpoint Algorithms

This fixpoint mathematical model is amenable to a simple synchronous parallelization scheme. Computation of $x_i^{(t+1)}$ involves two steps:

1. Collect the states $x_j^{(t)}$ and parameters a_{ij} for all dependencies $j \in \Gamma_i$.
2. Form the dependency set $D_i^{(t)}$, and invoke `update` to obtain $x_i^{(t+1)}$.

Assume that the vertices of the dependency graph are represented as tuples $(n, x_n^{(t)})$ of component index n and state $x_n^{(t)}$. The edges of the dependency graph are represented as tuples (i, j, a_{ij}), indicating that component x_i depends on component x_j with parameter a_{ij}. If the datasets containing the states and dependency graph are co-partitioned, then the first step can be executed by a local join between vertices and their corresponding edges on the component index $n = j$. For executing step 2, the result of the join is projected to the tuple $(i, x_j^{(t)}, a_{ij})$ and grouped on the component index i to form the dependency set $D_i^{(t)}$. The `update` function then aggregates $D_i^{(t)}$ to compute the new state $x_i^{(t+1)}$. This parallelization scheme, which can be summarized by treating a single iteration as *a join followed by an aggregation*, is a special case of the *Bulk Synchronous Parallel* (BSP) [174] paradigm. BSP models parallel computation as local computation (the *join* part) followed by message passing between independent processors (the *aggregation* part). Analogously to the execution of a *superstep* in BSP, we assume that the execution of a single iteration is synchronized among all participating computational units. We illustrate and prototype our proposed recovery scheme using a general data flow system with a programming model that extends MapReduce. For implementation, we use Apache Flink (c.f., Section 2.3.2) and will use its operator notation in the following. However, the ideas presented are applicable to other data flow systems with support for iterative or recursive queries.

To ensure efficient execution of fixpoint algorithms, Apache Flink offers two distinct programming abstractions for iterations, as introduced in Section 2.3.2. With *bulk iterations*, each iteration completely recomputes the state vector $x^{(t+1)}$ from the previous iteration's result $x^{(t)}$. Figure 4.4a shows a generic logical plan for modeling fixpoint algorithms as presented in Section 4.3.1 using the bulk iteration abstraction (ignore the dotted box for now). The input consists of records $(n, x_n^{(t)})$ representing the components of the state $x^{(t)}$ on the one hand, and of records (i, j, a_{ij}) representing the dependencies with parameters on the other hand (cf. Section 4.3.1). The data flow program starts with a "dependency join", performed by a *Join* operator, which joins the components $x_i^{(t)}$ with their corresponding dependencies and parameters to form the elements $(i, x_j^{(t)}, a_{ij})$ of the dependency set $D_i^{(t)}$. The "update aggregation" operator groups the result of the



(a) *Bulk* fixpoint algorithm in Apache Flink.

(b) *Incremental* fixpoint algorithm in Apache Flink.

Figure 4.4: Templates for modeling fixpoint algorithms as Flink dataflows.

join on the component index i to form the dependency set $D_i^{(t)}$, and applies the `update` function to compute $x_i^{(t+1)}$ from the dependency set. The "convergence check" operator joins and compares the newly computed state $x_i^{(t+1)}$ with the previous state $x_n^{(t)}$ on $n = i$. From the difference between the components, the operator computes a global aggregate using the `aggregate` function. A mechanism for efficient computation of distributive aggregates, similar to the aggregators in Pregel [120], invokes the `converged` function to decide whether to trigger a successive iteration (for simplicity reasons we omit this from the figure). If the algorithm has not converged, the "convergence check" operator feeds the records $(i, x_i^{(t+1)})$ that form $x^{(t+1)}$ into the next iteration.

The second form of iterations, called *delta iterations* are optimized for algorithms that only partially recompute the state $x^{(t)}$ in each iteration. A generic logical plan for fixpoint algorithms using this strategy is shown in Figure 4.4b. Like the bulk iteration variant, the plan models the two steps from the fixpoint model described in Section 4.3.1. It differs from the bulk iteration plan in that it does not feed back the entire state at the end of an iteration, but only the dependency sets $D_i^{(t)}$ for the fraction of components that will be updated in the next iteration (c.f., the feedback edge from the "recreate dependencies" operator to the workset in Figure 4.4b). The system updates the state of the algorithm

using the changed components rather than fully recomputing it. Hence, this plan exploits the fact that certain components converge earlier than others, as dependency sets are fed back selectively only for those components whose state needs to be updated. In addition to the two inputs from the bulk iteration variant, this plan has a third input with the initial version of the workset $D^{(0)}$. The creation of $D^{(0)}$ depends on the semantics of the application, but in most cases, $D^{(0)}$ is simply the union of all initial dependency sets.

In the plan shown in Figure 4.4b, the "candidate creation" operator groups the elements of the workset on the component index i to form the dependency sets $D_i^{(t)}$ and applies the `update` function to compute a candidate update for each $x_i^{(t+1)}$ from the corresponding dependency set. The "state update" operator joins the candidate with the corresponding component from $x^{(t)}$ on the component index i and decides whether to set $x_i^{(t+1)}$ to the candidate value. If an update occurs, the system emits a record $(i, x_i^{(t+1)})$ containing the updated component to the "recreate dependencies" operator, which joins the updated components with the dependencies and parameters. In addition, the records are efficiently merged with the current state (e.g., via index merging - see rightmost feedback edge in the figure). As in the bulk iteration variant, we represent the dependencies as (i, j, a_{ij}), and join them on $j = i$. The operator emits elements of the dependency sets to form the workset $D^{(t+1)}$ for the next iteration. The algorithm converges when an iteration creates no new dependency sets, i.e. when the workset $D^{(t+1)}$ is empty. We restrict ourselves to algorithms that follow the plans of Figures 4.4a and 4.4b, where the dataset holding the dependencies can be either a materialized or a non-materialized view, i.e., it may be the result of an arbitrary plan.

4.3.3 Fixpoint Algorithms as Vertex-Centric Programs

We furthermore discuss the execution of fixpoint algorithms in vertex-centric graph processing systems (c.f., Section 2.3.4). In these systems, programs directly operate on a graph in which vertices hold state and send messages to other vertices along the edges. By receiving messages, vertices update their state. Vertex-centric BSP systems are able to execute incremental iterations by maintaining mutable state and offering the choice whether to update a vertex's state and propagate the changes in the course of an iteration. Furthermore, a vertex can be de-activated, and will only be re-activated if it receives a message from another vertex. Our programming model defined in Section 4.3.1 executes in Pregel-like systems as follows. We use the component values as the vertex values of the graph that we operates on: the state of vertex i in iteration t is $x_i^{(t)}$. The edges of the dependency graph D correspond to the inverse of the edges of the graph modeled in the Pregel-like system. The collection of the dependency set $D_i^{(t)}$ for a vertex i happens by having all its incident neighbors j send their vertex value $x_j^{(t)}$ together with

the appropriate edge value a_{ij} (the dependency parameter) to i. We implement `update` function to compute $x_i^{(t)}$ in the vertex-update function of the Pregel-like system; the arguments to `update` are given by the messages received from neighbors. Convergence detection via the `agg` function is implemented using Pregel's distributed aggregators.

4.4 Forward Recovery via Compensation Functions

Next, we describe how to express compensations using a simple user-defined function and discuss how to integrate this UDF into large-scale data processing systems to enable optimistic recovery.

4.4.1 Abstraction of Compensation Functions as UDFs

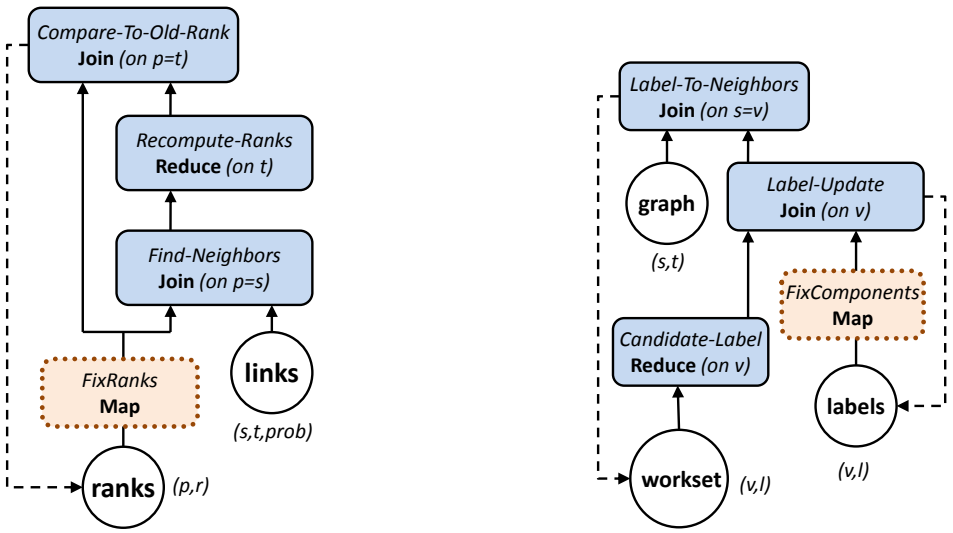
Compensation functions could be arbitrary data flow plans themselves. However, we find that it is sufficient to simply randomly re-initialize the lost parts of the state in many cases. The algorithms will then take care of restoring the individual components of the solution. Such compensation functions have the advantage that they are easy to write for library implementors and easy to integrate into existing systems. We define a simple signature for compensation functions, which takes the component index i, the current component state $x_i^{(t)}$ (which is null for lost components) and some algorithm specific metadata (e.g., the number of vertices in a graph) as input and produces a valid component state $x_i^{(t+1)}$ for the following iteration:

$$\texttt{compensate} : (i, x_i^{(t)}, metadata) \rightarrow x_i^{(t+1)}$$

In the following sections, we discuss how to efficiently integrate compensation functions of this shape in Apache Flink and Apache Giraph.

4.4.2 Recovery of Bulk Iterative Dataflows

To integrate optimistic recovery to the bulk iterations model, we introduce a "compensate" operator that takes as input the current state of components $x^{(t)}$ (dotted box in Figure 4.4a). The output of the compensation operator forms the input to the dependency join operator. The system only executes the compensate operator after a failure: e.g., in case of a failure at iteration t, the system finishes the current iteration and activates the additional operator in the plan for iteration $t+1$. We note that the compensate operator can be an arbitrary plan in itself. However, we found that compensation functions embedded in a simple Map operator are adequate for a wide class of algorithms (see



(a) PageRank as iterative data flow.

(b) Connected Components as iterative data flow.

Figure 4.5: Examples of fixpoint algorithms modeled as Flink dataflows.

Section 4.5). In addition, using a compensation function embedded in a Map operator ensures fast recovery times without the need for data shuffling. The compensation function often leverages lightweight meta-data from previous iterations that are stored in a distributed fault-tolerant manner using a distributed locking service [38].

PageRank: Figure 4.5a shows the data flow plan for the PageRank algorithm, as derived from the general plan in Section 4.3.2. The initial state $x^{(0)}$ consists of all pages p and initial ranks r. The dependencies $(s, t, prob)$ consist of the edges $s \rightarrow t$ of the graph, weighted by the transition probability $prob$ from page s to page t. The "find neighbors" operator joins the pages with their outgoing links on $p = s$ and creates tuples (t, c) holding the partial rank $c = r * prob$ for the neighbors. The "recompute ranks" operator groups the result of the "find neighbors" join on the target page t to recompute its rank using PageRank's update formula: $0.85 * \sum c + 0.15/n$. It emits a tuple (t, r_{new}) that contains the new rank to the "compare to old rank" operator, which joins these tuples with the previous ranks on $p = t$ and initiates the distributed aggregation necessary for the convergence check. Finally, the "compare to old rank" operator emits tuples (p, r) containing the pages and recomputed ranks. If PageRank has not converged, these tuples form the input to the next iteration $t + 1$.

In case of a failure, the system activates the additional Map operator called "fix ranks" in the plan, as shown in Figure 4.5a. This operator executes the compensation function. A simple compensation approach is to re-initialize the ranks of vertices in failed partitions

uniformly and re-scale the ranks of the non-failed vertices, so that all ranks still sum to unity (c.f., Algorithm 11). A requirement for this mechanism is that the system knows the total number of vertices n, and keeps an aggregate statistic about of the current number of vertices $n_{\text{nonfailed}}$ and the current total rank $r_{\text{nonfailed}}$. Note that these statistics can be maintained at virtually zero cost when computed together with the convergence check by the distributed aggregation mechanism. Furthermore, these statistics should be maintained anyways for ensuring and validating the numerical stability of the computation, regardless of the approach to fault tolerance.

Algorithm 11: Compensation function for PageRank.

```
1 function fix-ranks-uniform(pid, r, n, n_nonfailed, r_nonfailed)
2   if pid is in failed partition:
3     return 1/n
4   else:
5     return (n_nonfailed · r)/(n · r_nonfailed)
```

4.4.3 Recovery of Delta Iterative Dataflows

Analogously to bulk iterations, we compensate a large class of delta iterations by a simple Map operation, applied to the state at the beginning of the iteration subsequent to a failure. Additionally, for delta iterations, the system needs to recreate all dependency sets required to recompute the lost components. This is necessary because the recomputation of a failed component might depend on another already converged component whose state is not part of the workset anymore. In the plan from Figure 4.4b, the "recreate dependencies" operator produces dependency sets from all components of the state that were updated in the failing iteration, including components that were recreated or adjusted by the compensation function. The system hence only needs to recreate the necessary dependency sets originating from components that were not updated in the failing iteration. To identify those non-updated components, the "state-update" operator internally maintains a timestamp (e.g., the iteration number) for each component, indicating its last update. In the iteration subsequent to a failure, a record for each such component is emitted by the "state update" operator to the "recreate dependencies" operator. The "recreate dependencies" operator joins these record with the dependencies, creating the dependency sets $D^{(t+1)}$ for all components depending on it. From the output of the "recreate dependencies" operator, we prune all elements that do not belong to a lost component from these extra dependency sets. This is easily achieved by evaluating the partitioning function on the element's join key and checking whether it was assigned to a failed machine. By means of this optimization, the system does not unnecessarily recompute components that did not change in their dependent components and are not required for recomputing a failed component.

Connected Components [99]: This algorithm identifies the connected components of an undirected graph, the maximum cardinality sets of vertices that can reach each other. We initially assign to each vertex v a unique numeric label which serves as the vertex's state x_i. At every iteration of the algorithm, each vertex replaces its label with the minimum label of its neighbors. In our fixpoint programming model, we express it by the function `update` : $D_i^{(t)} \rightarrow \min_{D_i^{(t)}}(x_j^{(t)})$. At convergence, the states of all vertices in a connected component hold the same label, the minimum of the labels initially assigned to the vertices of this component. Figure 4.5b shows a data flow plan for Connected Components, derived from the general plan for delta iterations discussed in Section 4.3.2. There are three inputs: The initial state $x^{(0)}$ consists of the initial labels, a set of tuples (v, l) where v is a vertex of the graph and l is its label. Initially, each vertex has a unique label. The dependencies and parameters for this problem map directly to the graph structure, as each vertex depends on all its neighbors. We represent each edge of the graph by a tuple (s, t), referring to the source vertex s and target vertex t of the edge. The initial workset $D^{(0)}$ consists of candidate labels for all vertices, represented as tuples (v, l), where l is a label of v's neighbor (there is one tuple per neighbor of v).

Algorithm 12: Compensation function for Connected Components.

```
1 function fix-components(v, c)
2   if v is in failed partition:
3     return v
4   else:
5     return c
```

First, the "candidate label" operator groups the labels from the workset on the vertex v and computes the minimum label l_{new} for each vertex v. The "label update" operator joins the record containing the vertex and its candidate label l_{new} with the existing entry and its label on v. If the candidate label l_{new} is smaller than the current label l, the system updates the state $x_v^{(t)}$ and the "label update" operator emits a tuple (v, l_{new}) representing the vertex v with its new label l_{new}. The emitted tuples are joined with the graph structure on $s = v$ by the "label to neighbors" operator. This operator emits tuples (t, l), where l is the new label and t is a neighbor vertex of the updated vertex v. These tuples form the dependency sets for the next iteration. As $D^{(t+1)}$ only contains vertices for which a neighbor updated its label, we do not unnecessarily recompute components of vertices without a change in their neighborhood in the next iteration. The algorithm converges once the system observes no more label changes during an iteration by observing the number of records emitted from the "label update" operator. As discussed, the system automatically recomputes the necessary dependency sets in case of a failure in iteration t. Therefore, analogously to bulk iterations, the programmer's only task is to provide a record-at-a-time operation which applies the compensation

function to the state $x^{(t+1)}$. Here, it is sufficient to set the label of vertices in failed partitions back to its initial value (c.f., Algorithm 12).

4.4.4 Recovery in Vertex-Centric Programming

Analogously to Section 4.4.2, we propose to integrate the compensation function as a vertex-centric operator into the programming model of Pregel-like vertex-centric graph processing systems. Note that a compensation could be a full Pregel iteration. We will again make use of our observation that simple compensations provide fast recovery and are adequate for a wide variety of algorithms. We add the compensation function to the Pregel vertex class as a new method, in which the user-supplied compensation code can read aggregated values and modify the vertex state to bring it back to a condition from which the executed algorithm will still converge. The general process of optimistically recovering from a failure in Pregel works as described in Section 4.1: the system finishes the iteration where the failure occurred, acquires new machines to replace the failed ones (or re-distributes work to existing ones) and makes these replacement machines reload the inputs assigned to the failed ones. At the beginning of the iteration subsequent to the failure, the system invokes the compensate function on all vertices. Pseudocode for the simple compensation function for PageRank from Algorithm 11 is shown below:

```
class CompensablePageRankVertex extends PageRankVertex {
  ...
  void compensate(agg) {
    if (vertexInFailedPartition()) {
      setValue (1.0 / numVertices());
    } else {
      rescaledRank =
        (agg.nonFailedVertices * getValue()) /
        (agg.nonFailedRank * numVertices());
      setValue(rescaledRank);
} } }
```

The vertex checks whether it belongs to the failed partition and either re-initializes its value uniformly or re-scales it appropriately using aggregated values. Algorithms where early de-activation of some vertices lead to parts of the graph converging early again require special treatment: Vertices restored from failed partitions rely on having all their dependent values sent to them by their neighbors after the compensation function was applied. If a failed vertex had a neighbor that was inactive in the iteration subsequent to the failure, it would not receive all necessary messages and might not be able to correctly update its value. The system automatically solves this problem as follows: In the iteration subsequent to a failure, the system wakes up all inactive vertices and asks them to send their state to all adjacent neighbors that belong to a failed partition. Afterwards, it deactivates them again. We already stated that vertex-centric graph processing is simply

a special case of general distributed dataflow processing (c.f., Section 4.3.3). Hence, the integration is also similar to the already discussed data flow integration.

4.5 Templates for Compensation UDFs of Compensable Algorithms

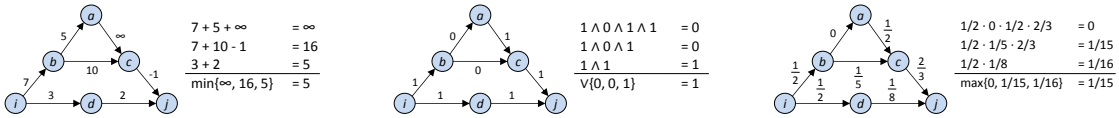
In order to demonstrate the applicability of our proposed approach to a wide range of problems, we discuss three classes of large-scale data mining problems, taken from our exemplary data mining problems in Section 2.4, which are solved by fixpoint algorithms. For each class, we list several problem instances together with a brief mathematical description and provide a generic compensation function as blueprint.

4.5.1 Link Analysis and Centrality in Networks

We first discuss centrality algorithms that compute and interpret the *dominant eigenvector of a large, sparse matrix* representing a network, as introduced in Section 2.4.3. Prominent examples are *PageRank*, *Eigenvector centrality*, *Katz centrality* and *LineRank*. Another example of an eigenvector problem is *Random Walk With Restart* [99], which uses a random walk biased towards a source vertex to compute the proximity of the remaining vertices of the network to this source vertex. Also, the dominant *eigenvector of the modularity matrix* [131] can be used to split the network into two communities of vertices with a higher than expected number of edges between them.

Recall that solutions for the problems of this class are computed by some variant of the *Power Method* [78], an iterative algorithm for computing the dominant eigenvector of a matrix M. An iteration of the algorithm consists of a matrix-vector multiplication followed by normalization to unit length. To model the Power Method as a fixpoint algorithm, we operate on a sparse $n \times n$ matrix, whose entries correspond to the dependency parameters. We uniformly initialize each component of the estimate of the dominant eigenvector to $\frac{1}{\sqrt{n}}$. The update function computes the dot product of the i-th row of the matrix M and the previous state $x^{(t)}$. We express it by the function `update` : $D_i^{(t)} \rightarrow \frac{1}{\|x^{(t)}\|_2} \sum_{D_i^{(t)}} a_{ij} x_j^{(t)}$ in our fixpoint programming model. The result is normalized by the L2-norm of the previous state, which can be efficiently computed using a distributed aggregation. The algorithm converges when the L2-norm of the difference between the previous and the current state becomes less than a given threshold. For failure compensation, it is enough to uniformly re-initialize the lost components to $\frac{1}{\sqrt{n}}$, as the power method will still provably converge to the dominant eigenvector afterwards [78]:

$$\texttt{compensate} : (i, x_i^{(t)}) \rightarrow \begin{cases} \frac{1}{\sqrt{n}} & \text{if } i \text{ belongs to a failed partition} \\ x_i^{(t)} & \text{otherwise} \end{cases}$$



(a) Shortest distance. (b) Reachability. (c) Maximum reliable path.

Figure 4.6: Exemplary path enumeration problems.

This function suffices as base for compensating all of the above problems when they are solved by the power method.

4.5.2 Path Enumeration Problems in Graphs

The second class of algorithms we discuss enumerate paths in large graphs and aggregate their weights. Instances of this problem class include *single-source reachability*, *single-source shortest paths*, *single-source maximum reliability paths*, *minimum spanning tree* [79], as well as finding the *connected components* [99] of an undirected graph. Based on the discussion in Section 2.4.4, instances of this problem class are solved by the *Generalized Jacobi algorithm* [79], which is defined on an idempotent semiring $(S, \oplus, \otimes)$. The state vector $x^{(0)}$ is initialized using the identity element e of $\otimes$ for the source vertex identity element ϵ of $\oplus$ for all other vertices. The algorithm operates on a given, application-specific graph. The dependency parameters correspond to the edge weights of this graph, taken from the set S, on which the semiring is defined. At iteration t, the algorithm enumerates paths of length t with relaxation operations, computes the weight of the paths with the $\otimes$ operation and aggregates these weights with the second, idempotent operation $\oplus$. In our fixpoint programming model, we express it by the function `update` : $D_i^{(t)} \rightarrow \bigoplus_{D_i^{(t)}} (x_j^{(t)} \otimes a_{ij})$.

The algorithm converges to the optimal solution when no component of $x^{(t)}$ changes. For single-source shortest distances, the algorithm works on the semiring $(\mathbb{R} \cup \infty, \min, +)$. For all vertices but the source, the initial distance is set to ∞. At each iteration, the algorithm tries to find a shorter distance by extending the current paths by one edge, using relaxation operations. The obtained algorithm is the well known Bellman-Ford algorithm (c.f. Figure 4.6a). For single-source reachability, the semiring used is $(\{0, 1\}, \wedge, \vee)$ (c.f. Figure 4.6b). A dependency parameter a_{ij} is 1, if i is reachable from j and 0 otherwise. The set of reachable vertices can then be found using boolean relaxation operations. Finally, in single-source maximum reliability paths, the goal is to find the most reliable paths from a given source vertex to all other vertices in the graph (c.f. Figure 4.6c). A dependency parameter a_{ij} represents the probability of reaching vertex j from vertex i. The semiring in use is $([0, 1], \max, \cdot)$. To compensate failures in the distributed execution

of the Generalized Jacobi algorithm, we can simply re-initialize the components lost due to failure to ϵ, the identity element of $\oplus$. Bellman-Ford, for example, converges to the optimal solution from any start vector $x^{(0)}$ which is element-wise greater than x^* [22]:

$$\texttt{compensate} : (i, x_i^{(t)}) \rightarrow \begin{cases} e & \text{if } i \text{ is the source vertex} \\ \epsilon & \text{if } i \text{ belongs to a failed partition} \\ x_i^{(t)} & \text{otherwise} \end{cases}$$

4.5.3 Low-Rank Matrix Factorization

The last class of algorithms we focus on are latent factor models for recommendation mining, as introduced in Section 2.4.1. The idea is to approximately factor a sparse $m \times n$ matrix M into the product of two matrices U and V, such that $M \approx UV^T$. The $m \times k$ matrix U models the latent features of the entities represented by the rows of M (e.g., users), while the $n \times k$ matrix V models the latent features of the entities represented by the columns of M (e.g., news stories or products). Due to its simplicity and popularity, we focus our efforts on the ALS technique for computing these models (c.f., Section 2.4.2).

Recall that, in order to find a factorization, ALS repeatedly keeps one of the unknown matrices fixed, so that the other one can be optimally recomputed. That means for example, that u_i, the i-th row of U, can be recomputed by solving a least squares problem including the i-th row of M and all the columns v_j of V that correspond to non-zero entries in the i-th row of M (c.f., Figure 4.7). ALS then rotates between recomputing the rows of U in one step and the columns of V in the subsequent step until the training error converges.

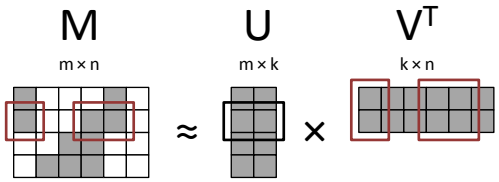


Figure 4.7: Dependencies for recomputing a row of R.

In the case of a failure, we lose rows of U, rows of V and parts of M. We compensate as follows: as we can always re-read the lost parts of M from stable storage, we approximately re-compute lost rows of U and V by solving the least squares problems using the remaining feature vectors (ignoring the lost ones). If all necessary feature vectors for a row of U or

V are lost, we randomly re-initialize it:

$$\texttt{compensate} : (i, x_i^{(t)}) \rightarrow \begin{cases} \text{random vector} \in [0,1]^k & \text{if } i \text{ is lost} \\ x_i^{(t)} & \text{otherwise} \end{cases}$$

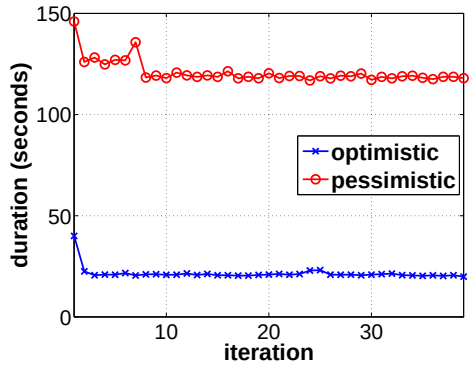
Matrix factorization with ALS is a non-convex problem [73], and our compensation function represents a jump in the parameter space that might lead to a different minimum. We therefore empirically validate our approach in Section 4.6.4 to show its applicability to real-world data.

4.6 Experiments

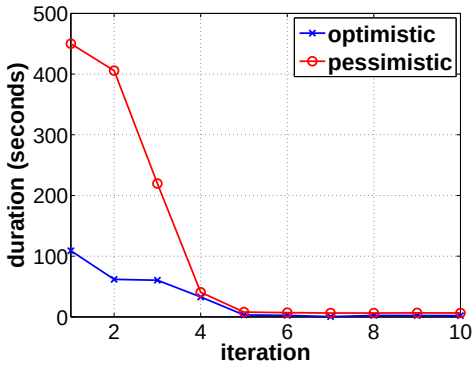
To evaluate the benefit of our proposed recovery mechanism, we run experiments on large datasets and simulate failures. Our simulation omits the time required for failure detection and provisioning of new machines or re-distributed work, as this has to be done for optimistic as well as pessimistic recovery in the same manner and therefore incurs the same overhead. The experimental setup for our evaluation is the following: the cluster consists of 26 machines running Java 7, Hadoop's distributed filesystem (HDFS) 1.0.4 and a customized version of Apache Flink. Each machine has two 4-core Opteron CPUs, 32 GB memory and four 1 TB disk drives. We also run experiments with Apache Giraph [10], an open-source implementation of Pregel [120], to validate our integration into vertex-centric systems. We use three publicy available datasets for our experiments: a webgraph called Webbase [28, 176], which consists of 1,019,903,190 links between 115,657,290 webpages, a snapshot of Twitter's social network [39, 172], which contains 1,963,263,821 follower links between 51,217,936 users and a dataset of 717,872,016 ratings that 1,823,179 users gave to 136,736 songs in the Yahoo! Music community [181].

4.6.1 Failure-free Performance

We compare our proposed optimistic approach to a pessimistic strategy that writes distributed checkpoints. When checkpointing is enabled, the checkpoints are written to HDFS in a binary format with the default replication factor of three. Analogously to the approaches taken in Pregel and GraphLab, we checkpoint state as well as communicated dependencies [117, 120]. We run each algorithm with a degree of parallelism of 208 (one worker per core). We look at the failure-free performance of optimistic and pessimistic recovery, in order to measure the overhead introduced by distributed checkpointing. Figure 4.8a shows the application of PageRank on the Webbase dataset. The x axis shows the iteration number and the y axis shows the time (in seconds) it took the system



(a) Failure-free performance of PageRank on the Webbase dataset.

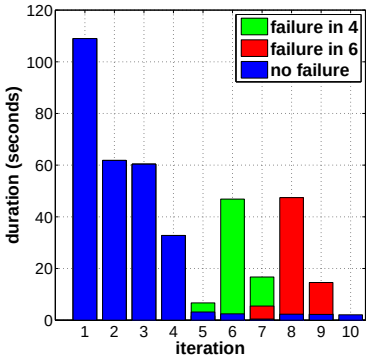


(b) Failure-free performance of Connected Components on the Twitter dataset.

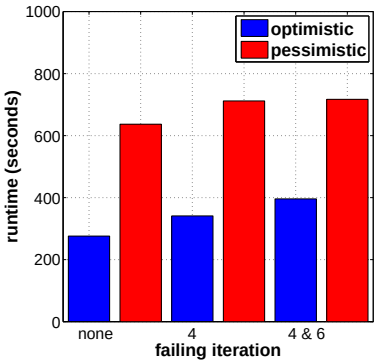
Figure 4.8: Failure-free performance.

to complete the iteration. The algorithm converges after 39 iterations. The runtime with checkpointing every iteration is 79 minutes, while optimistic recovery reduces the runtime to 14 minutes. The performance of the optimistic approach is equal to the performance of running without a fault tolerance mechanism, with an iteration lasting roughly 20 seconds. We see that the pessimistic approach introduces an overhead of more than factor 5 regarding the execution time of every iteration that includes a checkpoint.

Writing checkpoints only at a certain interval improves this situation, but trades off the time it takes to write a checkpoint with the time required to repeat the iterations conducted since the last checkpoint in case of a failure. If we for example checkpoint every fifth iteration, the incurred overhead to the runtime compared to the optimal failure-free performance is still 82%, approximately 11.5 minutes. We see that the optimistic approach is able to outperform a pessimistic one by a factor of two to five in this case. We note that in these experiments, this was the only job running in the cluster. In busy clusters, where a lot of concurrent jobs compete for I/O and network bandwidth, the negative effect of the overhead of checkpointing might be even more dramatic. Figure 4.8b shows the failure-free performance of Connected Components applied to the Twitter dataset. The checkpointing overhead varies between the iterations, due to the incremental character of the execution. With our optimistic approach, the execution takes less than 5 minutes. Activating checkpointing in every iteration increases the runtime to 19 minutes. We see that checkpointing results in a 3× to 4× increased runtime during the first iterations until the majority of vertices converge. Taking a checkpoint in the first iteration takes even longer than running the whole job with our optimistic scheme. After iteration 4, the majority of the state has converged, which results in a smaller workset and substantially reduces the checkpointing overhead. Again, the optimistic approach outperforms a pessimistic one that takes an early checkpoint by



(a) Additional work imposed by optimistic recovery of Connected Components on the Twitter dataset. A comparison to the overhead of pessimistic recovery is shown in Figure 4.9b.



(b) Runtime comparison of Connected Components on the Twitter dataset.

Figure 4.9: Recovery performance in Connected Components.

at least a factor of two. Our experiments suggest that the overhead of our optimistic approach in the failure-free case (collecting few global statistics using the distributed aggregation mechanism) is virtually zero. We observe that, in the absence of failures, our prototype has the same performance as Apache Flink with checkpointing turned off, therefore we conclude that it has optimal failure-free performance.

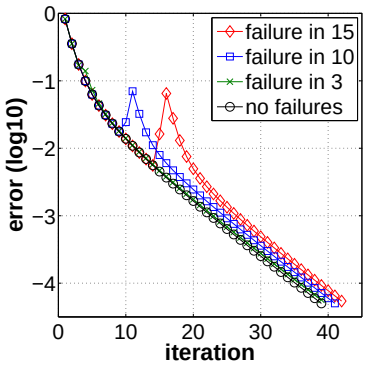
4.6.2 Simulated Recovery Performance

In the following, we simulate the failure of one machine, thereby losing 8 parallel partitions of the solution. We simulate failures by dropping the parts of the state x that were assigned to the failing partitions in the failing iteration. Additionally, we discard 50% of the messages sent from the failing partitions, simulating the fact that the machine failed during the iteration and did not complete all necessary inter-machine communication. After the failing iteration, our prototype applies the compensation function and finishes the execution.

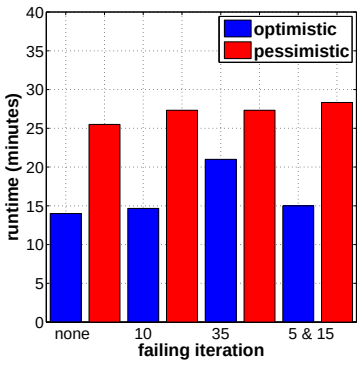
To evaluate the recovery of delta iterations, we run the Connected Components algorithm on the Twitter dataset. We simulate failures in different iterations and apply the compensation function. Figure 4.9a shows the overall number of iterations as well as their duration. The figure illustrates that Connected Components is very robust against failures when paired with our compensation function. Single failures in iterations 4 and 6 do not produce additional iterations, but only cause an increase in the runtime of the after next iteration by 45 seconds (which amounts to an increase of 15% in the

overall runtime). This happens because the system reactivates the neighbors of failed vertices in the iteration after the failure and by this, triggers the recomputation of the failed vertices in the next iteration. When we simulate multiple failures of different machines during one run, we observe that this process simply happens twice during the execution: The compensation of failures in iterations 4 and 6 or 5 and 8 during one run produces an overhead of approximately 35% compared to the failure-free execution. To investigate the effects of a drastic failure, we simulate a simultaneous failure of 5 machines in iteration 4 and repeat this for iteration 6. We again observe fast recovery: in both cases the overhead is less than 30% compared to the failure-free performance. In all experiments, the additional work caused by the optimistic recovery amounts to small a fraction of the cost of writing a single checkpoint in an early iteration. The execution with a compensated single machine failure takes at most 65 seconds longer than the failure-free run, while checkpointing a single early iteration alone induces an overhead of two to five minutes. Additional to writing the checkpoints, a pessimistic approach with a checkpointing interval would have to repeat all the iterations since the last checkpoint. Figure 4.9b illustrates the runtimes of a pessimistic approach (with a checkpoint interval of 2 iterations) to our optimistic approach. The times for the pessimistic approach are composed of the average execution time, the average time for writing checkpoints and the execution time for iterations that need to be repeated. The figure lists the runtime of both approaches for executions with no failures, a single failure in iteration 4 and multiple failures during one run in iterations 4 and 6. Figure 4.9b shows that our optimistic approach is more than twice as fast in the failure-free case and at the same time provides faster recovery than a pessimistic approach in all cases. The robustness in Connected Components is due to the sparse computational dependencies of the problem. Every minimum label propagates through its component of the graph. As social networks typically have a short average distance between vertices, the majority of the vertices find their minimum label relatively early and converge. After failure compensation in a later iteration, most vertices have a non-failed neighbor that has already found the minimum label, which they immediately receive and converge.

In order to evaluate the recovery of bulk iterations, we run PageRank on the Webbase dataset until convergence. We simulate failures in different iterations of PageRank and measure the number of iterations it takes the optimistic recovery mechanism to converge afterwards. Figure 4.10a shows the convergence behavior for the simulated failures in different early iterations. The x axis shows the iteration number. The y axis shows the L1-norm of the difference of the current estimate $x^{(t)}$ of the PageRank vector in iteration t to the PageRank vector $x^{(t-1)}$ in the previous iteration $t-1$ in logarithmic scale. We notice that the optimistic recovery is able to handle failures in early iterations such as the third, tenth, or fifteenth iteration extremely well with the compensation resulting only in maximum 3 additional iterations (as can be seen in the right bottom corner of Figure 4.10a). In additional experiments, we observe the same behavior for multiple



(a) Recovery convergence of PageRank on the Webbase dataset.



(b) Runtime comparison of PageRank on the Webbase dataset.

Figure 4.10: Recovery performance in PageRank.

failures during one run in early iterations: failures in iterations 2 and 10 or 5 and 15 also only result in at most 3 additional iterations. Next, we simulate a simultaneous failure of five machines to investigate the effect of drastic failures: such a failure costs no overhead when it happens in iteration 3 and results in only 6 more iterations when it happens in iteration 10.

When we simulate failures in later iterations, we note that they cause more overhead: a failure in iteration 25 needs 12 more iterations to reach convergence, while a failure in iteration 35 triggers 22 additional iterations compared to the failure-free case. This behavior can be explained as follows: a compensation conducts a random jump in the space of possible intermediate solutions. In later iterations, the algorithm is closer to the fixpoint, hence the random jump increases the distance to the fixpoint with a higher probability than in early iterations where the algorithm is far from the fixpoint. For a failure in iteration 35, executing these additional 22 iterations would incur an overhead of approximately eight minutes. Compared to a pessimistic approach with a checkpoint interval of five, the overhead is still less, as the pessimistic approach would have to restart from the checkpointed state of iteration 30 and again write a total of 7 checkpoints, amounting to a total overhead of more than 12 minutes. Figure 4.10b summarizes our findings about PageRank and compares the runtimes of our optimistic scheme to such a pessimistic approach with a checkpoint interval of five. The times for the pessimistic approach comprise the average execution time, the average time for writing checkpoints and the execution time for iterations that need to be repeated. We see that the optimistic approach outperforms the pessimistic one by nearly a factor of two in the failure-free case and for all cases, its overall runtime is shorther in light of failures.

For failures in later iterations, our findings suggest that hybrid approaches which use the optimistic approach for early iterations and switch to a pessimistic strategy later are needed. The decision when to switch could be made by observing the slope of the convergence rate. Once it starts flattening, the algorithm comes closer to the fixpoint and it would be beneficial to switch to a checkpoint-based recovery strategy. We plan to investigate this as part of our future work.

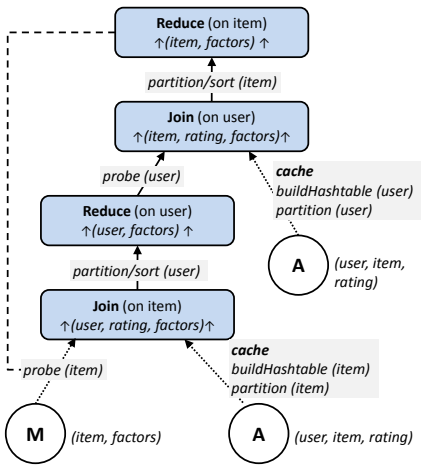
4.6.3 Recovery in Vertex-Centric BSP

We run an experiment with a Pregel-like system to showcase that optimistic recovery is applicable to vertex centric systems. We use a modified version of Apache Giraph 0.1 on a single machine with 16 workers and compute PageRank on a small social graph. We use the Slashdot-Zoo dataset [111, 161] which contains 79,120 vertices and 515,397 edges. Analogous to previous experiments, we simulate a failure of one worker in iteration 15 and compensate the intermediate. We observe an increase in the error directly after failure, yet the algorithm is able to quickly recover from the failure and steadily converges to the true PageRank, analogous to previous experiments.

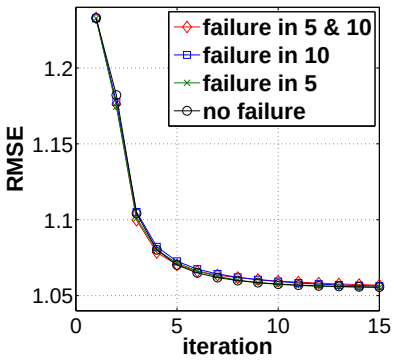
4.6.4 Empirical Validation of Compensability in ALS

We empirically validate the compensability of Alternating Least Squares for low-rank matrix factorization discussed in Section 4.5.3. We use the Yahoo Songs dataset for this. To show the compensability of Alternating Least Squares, we implement a popular variant of the algorithm aimed at handling ratings [189] as data flow program. Recall that ALS finds the factorization $M = UV^T$ by rotating between recomputing the rows of U in one step and the columns of V in the subsequent step until convergence. This is illustrated in the data flow plan in Figure Figure 4.11a: we re-compute U from an initial V using a join-operator followed by a reduce-operator and repeat these steps to subsequently re-compute V. Next, we feed back V and repeat this process until convergence (c.f., Algorithm 5 in Section 2.4.2).

Next, we simulate failures while computing a factorization of rank 10. We simulate a single failure in iteration 5, a single failure in iteration 10 and finally have a run with multiple failures where both failures happen after another. We measure the training error (by the root mean squared error, RMSE, shown in Figure 4.11b) of the factorization over 15 iterations. Our results show that failures result in a short increase of the error, yet until iteration 15, the error again equals that of the failure-free case. These results indicate that the ALS algorithm paired with the provided compensation strategy described in Section 4.5.3 is very robust to all failures, which only cause a short-lived distortion in the convergence behavior. We also repeat this experiment for the Movielens [129] dataset,



(a) ALS as fixpoint algorithm in Apache Flink.



(b) Optimistic recovery of failures in ALS on the Yahoo Songs dataset.

Figure 4.11: Experiments with Alternating Least Squares for matrix factorization.

which contains 1 million ratings. The results show in Figure 4.12 mirror our findings from the previous experiment. We first run ALS without any failures and convergence after 20 iterations, as illustrated in the top-left plot. Next, we simulate failures in the third, fifth and tenth iteration and again measure the convergence with respect to the training error. We run this experiment for a loss of 5%, 15% and 25% of the feature vectors in the factorization. We see that in all cases, our compensation for ALS quickly recovers from the failure and always manages to still convergence without requiring additional iterations.

4.7 Related Work

Executing iterative algorithms efficiently in parallel has received a lot of attention recently, resulting in graph-based systems [117, 120], and in the integration of iterations into data flow systems [32, 62, 124, 128, 186]. The techniques proposed in this chapter are applicable to all systems that follow a dataflow or a vertex-centric programming paradigm.

While the robust characteristics of fixpoint algorithms have been known for decades, we are not aware of an approach that leverages these characteristics for the recovery of distributed, data-parallel execution of such algorithms. Rather, most distributed data processing systems [49, 120], distributed storage systems [75], and recently real-time analytical systems [186] use pessimistic approaches based on periodic checkpointing and replay to recover lost state.

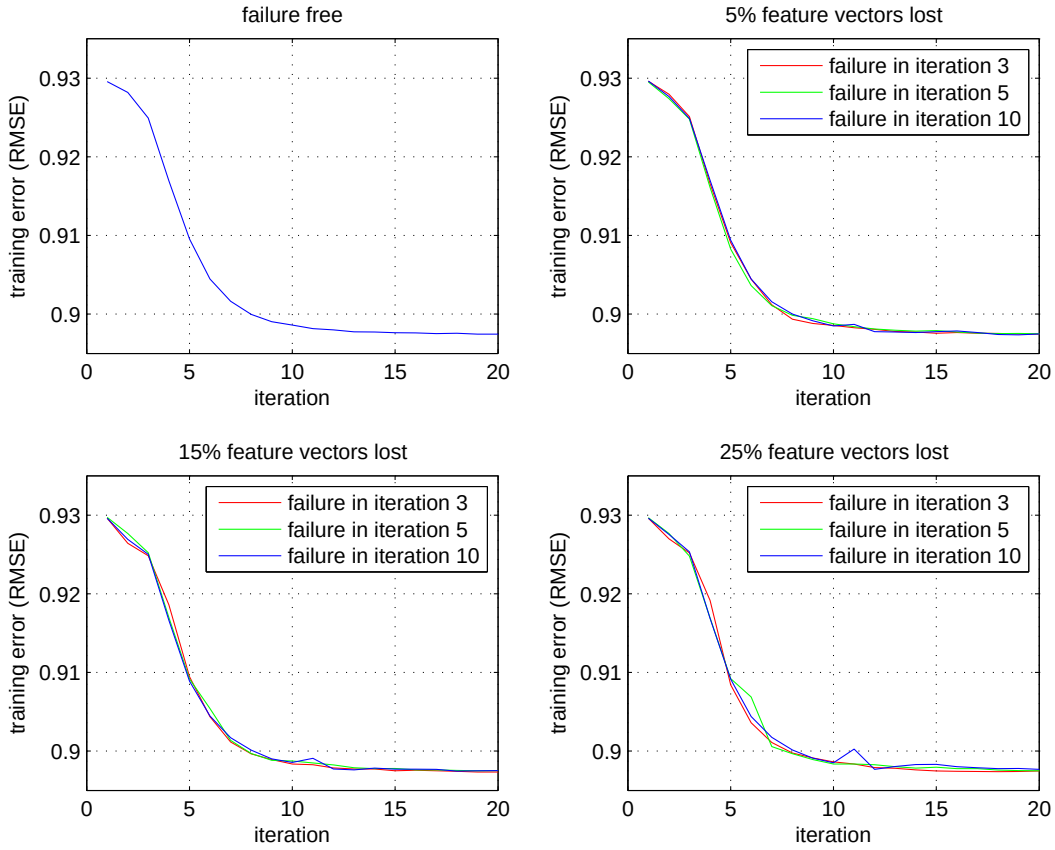


Figure 4.12: Optimistic recovery of failures in ALS on the Movielens dataset.

Systems such as Spark [186] offer recovery by recomputing lost partitions based on their lineage. The complex data dependencies of fixpoint algorithms however may require a full recomputation of the algorithm state (c.f., Section 4.1). While the authors propose to use efficient in-memory checkpoints in that case, such an approach still increases the resource usage of the cluster, as several copies of the data to checkpoint have to be held in memory. During execution such a recovery strategy competes with the actual algorithm in terms of memory and network bandwidth.

Confined Recovery [120] in Pregel limits recovery to the partitions lost in a failure. The state of lost partitions is recalculated from the logs of outgoing messages of all non-failed machines in case of a failure. Confined recovery is still a pessimistic approach, which requires an increase in the amount of checkpointed data, as all outgoing messages of the system have to be logged.

Similar to our compensation function, user-defined functions to enable optimistic recovery have been proposed for long-lived transactions. The *ConTract Model* is a mechanism for handling long-lived computations in a database context [150]. *Sagas* describe the concept of breaking long lived-transactions into a collection of subtransactions [71]. In such systems, user-defined compensation actions are triggered in response to violations of invariants or failures of nested sub-transations during execution. Due to the difference in context (transactional data processing versus analytical data processing) to our work, we could not compare against these approaches.

4.8 Conclusion

We present a novel optimistic recovery mechanism for distributed iterative data processing using a general fixpoint programming model. Our approach eliminates the need to checkpoint to stable storage. In case of a failure, we leverage a user-defined compensation function to algorithmically bring the intermediary state of the iterative algorithm back into a form from which the algorithm still converges to the correct solution. Furthermore, we discuss how to integrate the proposed recovery mechanism into a data flow system. We model three wide classes of problems (link analysis and centrality in networks, path enumeration in graphs, and low-rank matrix factorization) as fixpoint algorithms and describe generic compensation functions that in many cases provably converge. Finally, we present empirical evidence that shows that our proposed optimistic approach provides optimal failure-free performance (virtually zero overhead in absence of failures). At the same time, it provides faster recovery in the majority of cases. For incrementally iterative algorithms, the recovery overhead of our approach is less than a fifth of the time it takes a pessimistic approach to only checkpoint an early intermediate state of the computation. For recovery of early iterations of bulk iterative algorithms, the induced overhead to the runtime is less than 10% and again our approach outperforms a pessimistic one in all evaluated cases.

For convex problems with a single optimum, optimistic recovery from failures will provably retain the optimal solution, if the compensation is constructed accordingly. Examples of such cases are the compensation functions for spectral centrality measures in Section 4.5.1, as well as the compensation function for the path-enumeration-based graph algorithms in Section 4.5.2. The situation is different for non-convex problems with many local optima. Compensating a failure typically corresponds to a random jump in the parameter space of the algorithm and might lead to different local optima as a result in these cases. Often, this behavior might be acceptable. Yet it is important for users to be aware of the fact that optimistic recovery from a failure might result in an outcome different from the failure-free case in non-convex problems. Users can switch back to pessimistic recovery if they wish to prevent this behavior.

5 Efficient Sample Generation for Scalable Meta Learning

5.1 Problem Statement

The previous chapters 3 and 4 focused on efficient, scalable model training in massively parallel dataflow sytems and related systems issues. However, properly deploying ML in real-world settings requires more than efficient, scalable model training alone. Other 'meta' issues must be dealt with as well, such as model selection, parameter tuning and estimating model generalizability to new data. Another potential issue is that some complex ML applications may require more than one model to achieve satisfactory accuracy. Data scientists typically address the first set of issues using *cross-validation* for estimating prediction quality of a model for a given problem [37, 56, 72], while the latter is addressed using *ensemble learning* to create a stronger model by combining weaker models [34, 115]. The wide variety of meta learning techniques[1] have two common steps. First, samples are generated from the input data. Second, ML models are trained and built on these samples. Although efficiently conducting cross-validation and ensemble learning at scale has been recognized as an important challenge [108,126], existing systems for distributed, scalable ML [13, 76, 89, 163] lack a comprehensive meta learning layer supporting both steps of meta learning. These systems typically provide data-parallel techniques to efficiently perform model training and evaluation of the latter step, but do not readily support efficient, scalable sample generation required in the first step.

The focus of this chapter is in developing a scalable sample generation algorithm to support meta learning which meets three key desiderata. First, the common range of sampling methods used by different meta learning techniques must be supported. Adaption to a particular sample generation technique must only require minimal modification of the algorithm. Second, for reasons of efficiency, the algorithm must only conduct a single pass over the data, even for a large number of samples to generate. Third, the algorithm must be general enough to be used with a range of distributed data processing abstractions (c.f., Section 2.3). Obtaining these desiderata for sample generation from large datasets in a ML setting is challenging for three reasons. First, many scalable ML systems represent data as *block-partitioned matrices* [76, 89, 99], which leads to the rows and columns of the input matrix being spread across multiple blocks, each stored separately, potentially on different machines in a distributed file system. Thus, sample

[1] note that while we refer to cross-validation and ensemble learning as "meta learning", the term is often used in a different context [136]

generation must also consume distributed, blocked data as input and produce distributed, block-partitioned sample matrices as output. If we choose to sample a row or column from the given data, we must ensure that it is correctly restored in the newly formed blocks of the generated sample matrix. The second challenge is that meta learning techniques may require to read a composition of generated samples (e.g., during training on $k-1$ folds in a round of k-fold cross-validation, see Section 5.9.2 for details). This composition is not easily possible with a blocked representation. Naïve solutions for this challenge pose a scalability bottleneck as they generate many copies of the input data. The third challenge arises from the difficulty of mapping particular sampling techniques to a distributed, shared-nothing setting. For example, sampling with replacement is difficult to conduct in a distributed setting, as the number of occurrences of individual observations in a sample is correlated [48, 141]. Because of this correlation, we cannot easily generate the components of a sample with replacement in parallel, for reasons described in Section 5.8.

5.2 Contributions

We address the challenges introduced in Section 5.1 as follows. We present a general algorithm for distributed sample generation from block-partitioned matrices in a single pass over the data, which leverages *skip-ahead* random number generators to sample partitioned rows or columns in parallel (c.f., Section 5.7). We discuss how to efficiently generate samples for a variety of meta learning techniques via a user-defined sampling function embeddable in our algorithm (c.f., Section 5.9). This approach allows running different sampling techniques by modifying only a few lines of code while still guaranteeing efficient sample generation. We show how our method supports distributed sampling with replacement by leveraging an approach to compute a multinomial random variate in a distributed, parallel fashion (c.f. 5.8). Furthermore, we extend our algorithm to support generating blocked training sets for k-fold cross-validation, which requires aggregate size almost equal to the given data (c.f., Section 5.9.2). We close the chapter with a thorough experimental evaluation of our algorithm on datasets with billions of datapoints (c.f., Section 5.10).

5.3 The Role of Meta Learning in Deploying Machine Learning Models

This chapter's running example for the role of meta learning in ML involves the use case of automatic spam detection. Given a dataset of labeled emails, both spam and non-spam, obtain a predictive model which can accurately classify newly arriving emails as spam or non-spam. One example of a predictive model would be a k-nearest neighbor

(k-NN) classifier, which for a given unseen email determines its label using labels of the k most similar emails in the training data. The value of k serves as a *hyperparameter* for k-NN, and must be tuned to gain sufficient quality of the learned classifier. An example for automating the tuning of the hyperparameter in k-NN using *hold-out test* cross-validation would work as follows. First, we draw three disjoint samples from the mails. Next, we train nearest-neighbor classifiers with different values for k on the first sample (the training set). Then, we evaluate the accuracy of the resulting classifiers for predicting the labels of unseen mails in the second sample (the validation set), and select the value for k with the best performance. Finally, we evaluate the generalization error of the resulting classifier on the unseen mails in the third sample (the test set). Other ML classifiers have similar hyperparameters which can be tuned using cross-validation in a comparable way. For the sake of simplicity, we only talk about generating two different samples (training set and test set) in most examples of this chapter. Note that all techniques presented here apply for the general case with train set, validation set and test set.

A complementary meta learning technique is ensemble learning. Ensemble learning builds a stronger model by combining a set of weaker models [34, 115]. In our case, we draw many independent subsamples of the email corpus, train a classifier on each of them and combine the predictions of these classifiers afterwards. Datasets for such ML use cases have a matrix-representation, where rows correspond to training instances and columns to instance features. For the spam detection example, the matrix has a row for each email, and a column for each feature (e.g., occurrence count for a term 'mortgage' in email text, label as spam or non-spam). Thus, element (i, j) in the matrix would be the value of feature j for email i. Although our running example is small for explanatory purposes, we note a real-world dataset would have millions of rows (emails) and thousands of columns (features). Such a high number of features arises for example from a sparse representation of an email as vector in the space of all possible terms (hundreds of thousands for the english language alone) and their frequently occurring combinations (n-grams).

5.4 Physical Data Representation in Large-scale Machine Learning Systems

Although meta learning techniques are straightforward to implement on a single machine with random access to the data, efficient implementations are difficult when the input data is block-partitioned and stored on different machines in a distributed filesystem in a shared-nothing architecture. However, block-partitioning provides significant performance benefits, including a reduction in the number of tuples required to represent and process a matrix, block-level compression, and the optimization of operations like matrix multiplication on a block-level basis. Such benefits have led to the widespread adoption of block-partitioned matrices in several parallel data platforms [9, 76, 89, 99, 126].

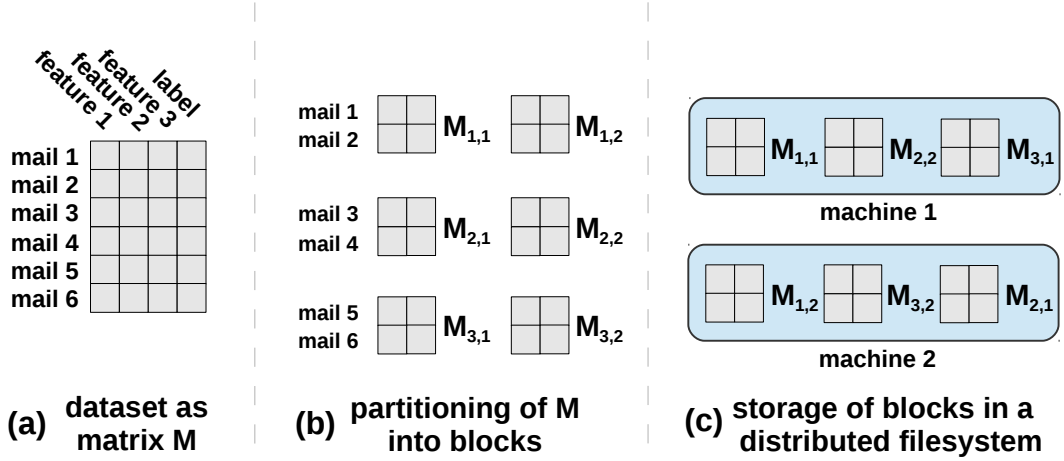


Figure 5.1: Sample dataset representation: matrix partitioned into blocks, stored on different machines in distributed filesystem.

Figure 5.1 illustrates our example dataset as a block-partitioned matrix in a distributed filesystem. As shown in Figure 5.1(a), the data is represented as a 6×4 matrix M where each row represents an email and each column a feature. The matrix is divided into square blocks according to a chosen block size. In our example, we choose a block size of 2×2, resulting in six different blocks, as shown in Figure 5.1(b). We refer to individual blocks using horizontal and vertical block indices, e.g., $M_{2,1}$ denotes the block in the second row and first column of blocks. Blocks are stored on different machines across the distributed file system, where the assignment of blocks to machines is usually random, as in Figure 5.1(c). As we discuss in subsequent sections, this distributed block-partitioned data representation creates multiple challenges for efficient sample generation for scalable meta learning.

5.5 Distributed Sampling with Skip-Ahead Pseudo Random Number Generators

In the following, we develop an algorithm for generating sample matrices from a large, block-partitioned matrix stored in a shared-nothing cluster. We discuss the case of generating sample matrices from rows of the input matrix. However, the same techniques apply to sampling columns of the input matrix as well. We define the sample generation task as follows. Given an $m \times n$ input matrix M, stored in blocks of size $d \times d$, we generate N sample matrices $S^{(1)}, \ldots, S^{(N)}$ such that the rows of a particular sample matrix $S^{(i)}$ are randomly drawn from M. We store the sample matrices in blocks of size $d \times d$. Let r be a row of M partitioned in blocks with the horizontal block index b. Let v be the number of vertical blocks of M. Then r is distributed over the v blocks, $M_{b,1}, \ldots, M_{b,v}$.

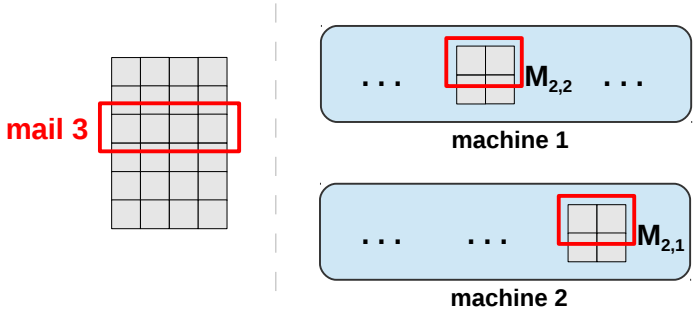


Figure 5.2: Parallel invocation of sampling on different machines that process partitions of the same row.

All machines processing one of these blocks must come to the same conclusion regarding the occurrences of r in a sample matrix $S^{(i)}$.

Figure 5.2 illustrates the scenario for our mail example. We must determine in which sample matrices mail 3 will occur. Due to the blocked representation, the row corresponding to mail 3 is partitioned and its partitions are contained in blocks $M_{2,1}$ and $M_{2,2}$ which are stored on different machines. The sample generation algorithm invokes sampling in parallel on both machines holding the blocks, and must give a deterministic and therefore reproducible result. For two invocations with the same row on different machines, it must return exactly the same destination in the sample matrices. Otherwise, we could not guarantee that the row is correctly restored in the sample matrix as row partitions might be missing or placed in the wrong order. Furthermore, this consistency should ideally be guaranteed without requiring inter-machine communication in order to achieve a high degree of parallelism. We solve this challenge by carefully choosing the pseudo random number generator (PRNG) which our algorithm deploys. A PRNG generates a sequence of values $p_0, p_1, p_2, \ldots$ that are approximately uniformly distributed. The generation is initialized with a seed s and typically uses a recursive transition function, φ, such that $p_i = \varphi(p_{i-1})$. A first step towards guaranteeing consistent results in our algorithm is to use the same seed s on all machines. This ensures all PRNGs produce the exact same random sequence. Second, we use the unique index of a row as the position to jump to in the random sequence. For a row with index j, sampling must leverage the j-th element, p_j, from the produced random sequence. This method will provide consistent results when partitions of row j are processed in parallel on different machines. However, this approach leaves open a performance question, as all random elements up to p_{j-1} would have to be generated in order to determine the value of p_j with the recursive function φ. To circumvent this performance issue, we use a special class of PRNGs, which enables us to directly skip to an arbitrary position in the random sequence (a so-called *skip-ahead*). Mathematically speaking, these generators allow us to compute the random element

p_j directly with a function $\hat{\varphi}(s, j)$ via the seed s and the position j. PRNGs with this property are also used in parallel data generation frameworks [5,74]. In general, any hash function that produces an approximately uniformly distributed output when supplied with the row indexes suffices in our case.

5.6 Abstraction of Sampling Techniques as UDF

We embed into the algorithm a user-defined sampling function, refered to as *sampling-UDF*, to specify the sampling technique to apply. This function allows adaption to a wide variety of sampling techniques with only a few lines of code. We assume during integration of our algorithm into a large-scale ML system that necessary sampling-UDFs are implemented by experts and available to end users. Our algorithm calls the sampling-UDF to determine occurrences of a row in the sample matrices. The sampling-UDF has the following signature:

$$row\_id, rng, params \rightarrow (samplematrix\_id, relative\_pos)^*$$

The first input parameter is the identifier of a particular row to process: row_id. The second input rng refers to a PRNG instantiated by the algorithm, which provides the 'skip-ahead' functionality discussed in Section 5.5. The third input $params$ denotes a set of user-defined parameters (e.g., the number of samples to draw). The implementation of the sampling-UDF must output a potentially empty set of $(samplematrix\_id, relative\_pos)$ tuples. Each tuple dictates an occurrence of the row in a sample matrix denoted by $samplematrix\_id$. This allows to generate occurrences in different sample matrices with a single invocation. The value $relative\_pos$ is used by the algorithm to determine the order of the rows in the sample matrices by sorting them according to $relative\_pos$. For instance, we can randomize a sample matrix by setting $relative\_pos$ to a random number drawn from rng. When operating with fixed precision numbers, it is possible that the same random number $relative\_pos$ is generated for two different rows. To alleviate that case, the algorithm also sorts by the unique row_id to guarantee a deterministic ordering in the sample matrix.

5.7 Distributed Sample Generation Algorithm

Our algorithm conducts the distributed sample generation in three phases: (i) the *'local-sampling'*-phase decides which rows of the input matrix M will occur in which sample matrices, (ii) the *'block-preparation'*-phase groups and sorts the selected rows, and (iii) the *'streaming-re-block'*-phase finally materializes the blocks of the sample matrices. Figure 5.3 shows an example of the distributed execution of the three phases. It details the

individual steps in the algorithm for generating two sample matrices, $S^{(1)}$ and $S^{(2)}$. Both sample matrices contain three mails out of our toy dataset of six emails M. We assume execution in a distributed data processing system that runs MapReduce-like programs, such as Hadoop [11], Flink [2], Spark [187], and Hyracks [32], or in a distributed database that supports UDTFs and UDAFs. Furthermore, we assume that the overall size of the dataset (i.e., the dimensions of the input matrix) is known. The three phases execute in a single pass over the data. A traditional MapReduce system runs the *'local-sampling'* within the map-stage, the *'block-preparation'* within the shuffle and the *'streaming-re-block'* during the reduce-stage. The 'local-sampling'-phase decides which rows of the input matrix occur in which sample matrices. Every machine in the cluster processes locally stored blocks of the input matrix in this phase and uses the sampling-UDF to sample partitions of rows from the blocks. Algorithm 13 illustrates the individual steps of this phase. Each machine processes a block of the input matrix at a time and iterates over the row partitions contained in the block (cf. line 4). For each such row partition, it invokes the sampling-UDF (cf. line 6) to determine the occurrences of this row in the sample matrices. For every (*samplematrix_id*, *relative_pos*) tuple returned by the sampling-UDF, the processing machine emits a record with the following structure: *samplematrix_id* | *vertical_block_index* | *relative_pos* | *row_id* | *row_partition* as shown in line 7 & 8. The field *samplematrix_id* denotes the sample matrix in which this row will occur, the field *vertical_block_index* is the vertical index of the block of the input matrix from which we sampled the row partition. The field *row_id* has the row index (e.g., the i-th row has index i) and the field *relative_pos* holds the *relative_pos* value from the tuple returned by the sampling-UDF. These two fields will later be used to determine the position of the row in the sample. The field *row_partition* holds the data from the row partition that we sampled. The 'local-sampling' phase is embarassingly parallel as every block of the input matrix is processed independently. Its maximum degree of parallelism is equal to the number of blocks of the input matrix. In Figure 5.3, we see that the 'local-sampling-phase' results in six records being emitted by the first machine that stores and processes the blocks, $M_{1,1}$, $M_{2,2}$, $M_{3,1}$, and another six records emitted by the second machine that processes the blocks, $M_{3,2}$, $M_{1,2}$, $M_{2,1}$.

Algorithm 13: 'Local-Sampling': Sampling from locally stored matrix blocks.

```
1 Input: matrix block blk, random number generator rng, sampling parameters params
2 Output: records containing the sampled rows
3 v ← read vertical index of blk
4 for row_partition in blk
5   j ← index of row
6   for occ ← sampling_UDF( j, rng, params )
7     emit_record( occ.samplematrix_id, v, occ.relative_pos, j, row_partition )
```

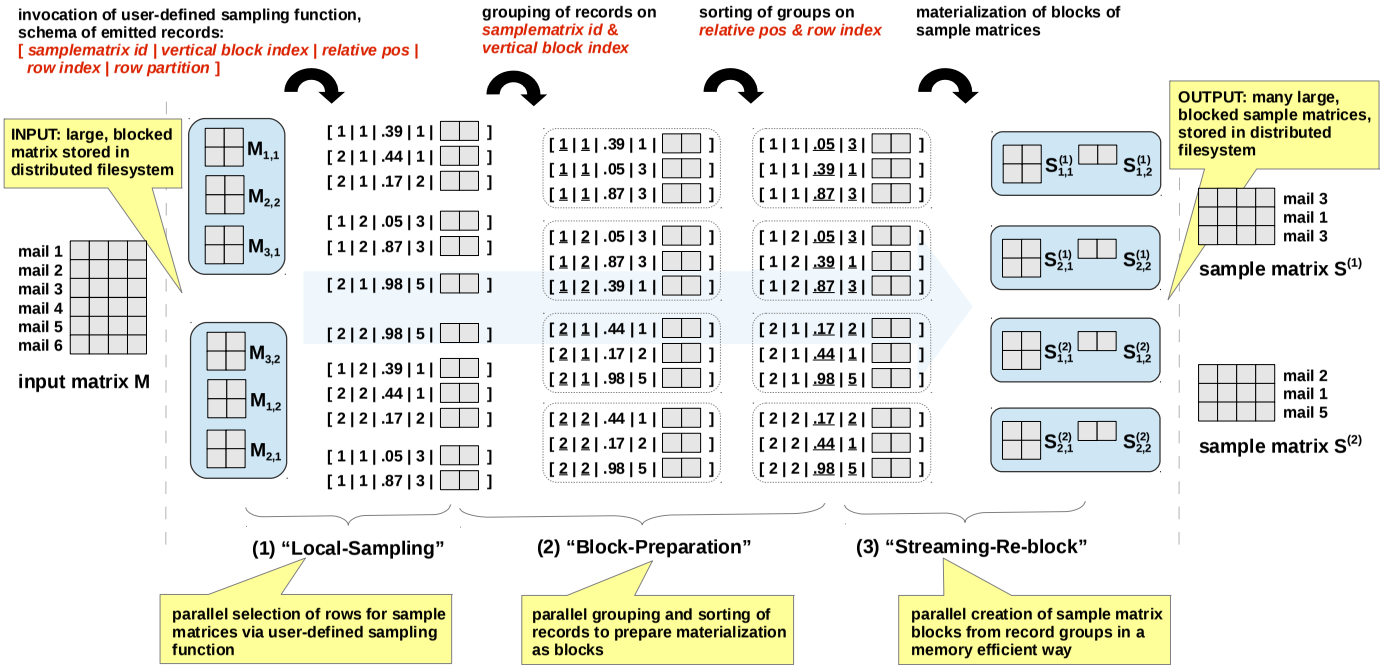


Figure 5.3: Example: Distributed generation of two blocked sample matrices, $S^{(1)}$ and $S^{(2)}$ from a blocked input matrix M.

The purpose of the next phase, called 'block-preparation'-phase, is to prepare the materialization of the records emitted from the 'local-sampling'-phase as blocked matrices. For that, we must group and sort the records. We use the data from the fields, *samplematrix_id* and *vertical_block_index* of the record as a composite key for grouping. After grouping, every resulting group contains the data for the column of blocks denoted by *vertical_block_index* of the sample matrix denoted by *samplematrix_id*. As different column blocks will be processed by different machines, we need to ensure that all of the groups contain their records in the same order. Failing to guarantee this would result in inconsistent sample matrices. Therefore, we sort the groups on the field *relative_pos* to ensure collection of partial rows in the same order on all machines. We additionally sort on the *row_id* to obtain a unique ordering even when *relative_pos* is the same random number for multiple rows. We omit pseudocode for this phase, since distributed data processing systems provide sorting and grouping functionality.

Figure 5.3 illustrates the two steps in this phase. First, we form four groups of three records from the twelve records emitted in the previous phase. Every group contains the row partitions for a column of blocks of a sample matrix. The topmost group, for example, contains the data for the first column of blocks of sample matrix, $S^{(1)}$, which consists of the blocks, $S^{(1)}_{1,1}$ and $S^{(1)}_{1,2}$. Analogously, the bottommost group contains the data for the second column of blocks of sample matrix $S^{(2)}$, namely, the blocks, $S^{(2)}_{2,1}$ and $S^{(2)}_{2,2}$. In a second step, we sort these groups on the *relative_pos* and *row_id* fields from the record, so that the records in each group have the same order in which the corresponding rows will later appear in their intended sample matrix.

The final 'streaming-re-block'-phase materializes the distributed blocked sample matrices. Every participating machine processes a sorted group from the 'block-preparation'-phase and forms the blocks for a column of blocks of the corresponding sample matrix. Algorithm 14 shows the required steps. We allocate a single block as buffer (c.f., line 3) which is filled with data from the field *row_partition* of incoming records in order (c.f., line 5). Whenever the buffer block is completely filled, we write it back to the DFS and clear it (cf. lines 7 & 8). We increment the horizontal index of the buffer block afterwards. Finally, we write back potentially remaining buffered data, to handle the case when the final buffer block is not filled (c.f., lines 10 & 11).

Re-blocking in this manner achieves a high degree of parallelism by allowing every column of blocks of every sample matrix to be processed independently. The maximum degree of parallelism of this phase is the number of sample matrices N to be generated multiplied by the number of column blocks (which equals the number of columns n of the input matrix divided by the block size d). We also achieve high memory efficiency, as only a single buffer block is necessary to materialize a column block of a sample matrix. A potential drawback of our approach is that the degree of parallelism in this last phase

Algorithm 14: 'Streaming-Re-block': Materialization of sample matrix blocks.

```
1  Input: sorted record group, sample matrix identifier s, vertical block index v
2  Output: sample matrix blocks written to DFS
3  allocate buffer block blk with indexes 1, v
4  for rec ← record group
5    add row partition from rec to blk
6    if blk is full then
7      write blk to DFS location for s
8      clear contents of blk
9      increment horizontal index of blk
10 if blk is not empty then
11   write blk to DFS location for s
```

depends on the shape of the input matrix, which determines the number of column blocks. It is therefore best applied in cases where at least one of the following conditions is true: (a) a large number of samples must be generated, (b) the input matrix has many columns, which is often the case when working with sparse datasets.

Figure 5.3 illustrates an example for this phase. We transform the four record groups from the previous phase into blocked matrices. Every group results in a column of blocks of a sample matrix. We transform the topmost group into the blocks, $S^{(1)}_{1,1}$ and $S^{(1)}_{1,2}$ of sample matrix $S^{(1)}$ and the next group into the blocks, $S^{(1)}_{2,1}$ and $S^{(1)}_{2,2}$. The remaining two groups analogously form the blocks, $S^{(2)}_{1,1}$, $S^{(2)}_{1,2}$, $S^{(2)}_{2,1}$, and $S^{(2)}_{2,2}$ of sample matrix $S^{(2)}$. We store the blocks in the distributed filesystem.

This section focuses on generating sample matrices by sampling rows of an input matrix, which is common for most supervised learning use cases where each row corresponds to a labeled training example. However, other ML use cases require sampling at matrix cell level. One example is recommendation mining, which typically analyzes a matrix of interactions between users and items to predict the strength of unseen interactions [106]. Here, cross-validation holds out interactions, which involves sampling matrix cells. Our algorithm can extend to such cases, by processing blocks of the input matrix cell-wise instead of row-wise, applying the sampling-UDF at the cell level, and lightly modifying the re-blocking to aggregate cells.

5.8 Distributed Sampling with Replacement

Sampling with replacement is required by multiple meta learning techniques, including approaches for both cross-validation and ensemble learning. Bootstrap is a powerful

statistical method for evaluating the quality of statistical estimators, and is applied to many cross-validation techniques [104].

Efficiently drawing bootstrap samples in a parallel, distributed way is a hard problem [141]. In the following, we will explain why this is the case and present our solution to the problem. So far, we viewed the sample generation techniques discussed as distributed generation of a categorial random variate. Take k-fold cross validation on an input matrix with m rows as an example: for each row we have to decide to which of the k folds it must be assigned. This is equivalent to generating a random variate $r \in \{1, \ldots, k\}^m$ from a categorial distribution of dimensionality m with possible outcomes $\{1, \ldots, k\}$, each having probability $\frac{1}{k}$. The j-th component r_j of r holds an outcome that describes to which of the k folds the row with index j is assigned. In our algorithm, every invocation of the sampling-UDF generates such a component. The generation of a categorial random variate is easy to conduct in parallel in a distributed system, as all components of such a random variate are independent of each other, allowing us to generate them in isolation.

In contrast to the sample generation that we have seen so far, bootstrap samples are drawn using *sampling with replacement*, which means every row occurs at each position of every sample matrix with equal chance. A single row can occur zero or many times in a generated sample matrix. Given an input matrix with m rows and a desired bootstrap sample size s, we have to generate a random variate from the multinomial distribution $M_m(s; \frac{1}{m}, \ldots, \frac{1}{m})$ for the creation of a sample matrix with bootstrap sampling. Here, a component r_j of r denotes the number of times that the row with index j occurs in the sample matrix. Unfortunately, the components of a multinomial random variate are not independent, but negatively correlated: Say $(r_1, \ldots, r_m) \sim M_m(s; \frac{1}{m}, \ldots, \frac{1}{m})$, then if we fix the first component r_1 to have value z, the variate composed of the remaining components of r follows a multinomial distribution with different probabilities [155]: $(r_2, \ldots, r_m) \sim M_{m-1}(s - z; \frac{1}{m-1}, \ldots, \frac{1}{m-1})$. Because of this correlation, we cannot easily generate the components of r in parallel in isolation (existing approaches usually generate multinomial random variates sequentially or with random access to the variate as a whole [48]). Another constraint in our case is that we do not know the exact number of rows that a particular machine of the cluster is going to process, as distributed data processing systems typically schedule processing tasks adaptively.

A simple solution to integrate sampling with replacement in our algorithm is to generate the whole random variate for every desired sample matrix once on every machine (our algorithm already exposes a random number generator with a fixed seed on every machine), and then to access the components corresponding to the rows to process. This approach however has the constraint that there must be enough memory available for all these variates. To circumvent this constraint, we present a method that enables us to compute a single compontent of a multinomial random variate in isolation with constant memory requirements. We use the property that a multinomial is partitionable

Algorithm 15: Recursive generation of a component of a multinomial random variate.

```
1  function sample_multinomial( j, start, end, t, rng )
2    if start = end or t = 0 then
3      return t
4    size ← end − start + 1
5    size_left ← ⌊size/2⌋
6    mid ← start + size_left − 1
7    skip ahead to position mid in rng
8    t_left ← binomial_rand( rng, t, size_left/size )
9    if j ≤ mid then
10     return sample_multinomial( j, start, mid, t_left )
11   else
12     return sample_multinomial( j, mid + 1, end, t − t_left )
```

by conditioning on the totals of subsets of its components. Let $r = (r_1, \ldots, r_m)$ be a random variate from the multinomial distribution $M_m(s; p_1, \ldots, p_n)$ and let $z = \sum_{j=1}^{i} r_j$ be the sum of the first i components of r. By fixing z we also fix the sum of the remaining components $\sum_{j=i+1}^{n} r_j$ as $s - z$. The left partition $(r_1, \ldots, r_i)$ then follows the multinomial distribution $M_i(z; \frac{p_1}{p_l}, \ldots, \frac{p_i}{p_l})$ with $p_l = \sum_{j=1}^{i} p_j$. Analogously, the right partition follows the multinomial distribution $M_{m-i}(s - z; \frac{p_{i+1}}{p_r}, \ldots, \frac{p_n}{p_r})$ with $p_r = \sum_{j=i+1}^{n} p_j$. The generation of the cumulative sum z necessary for the partitioning can be conducted by sampling a binomial distribution: $z = \sum_{j=1}^{i} r_j \sim B(s, \sum_{j=1}^{i} p_j)$.

We use this property to define a recursive algorithm that generates the components of a multinomial random variate in parallel in a distributed setting (c.f., Algorithm 15). In order to determine the component r_n of a multinomial random variate r of size t with equally probable outcomes $(p_1, \ldots, p_m)$, we proceed as follows. First, we determine the middle index of the current partition of the random variate that we look at (c.f., lines 4 to 6). Analogously to previous implementations, we skip the random number generator to this index to get repeatable results when processing this partition on different machines (c.f., line 7). Next, we sample from a binomial distribution to determine how many occurrences to assign to the left and right side of the current partition (c.f., line 8). Finally, we recursively invoke the function for the left or right side of the current partition, depending on the index j of the component we aim to compute. The algorithm terminates when we reach a single element partition (the component we look for) or when the current partition is assigned zero occurrences (c.f. lines 2 & 3). This approach enables us to compute a multinomial random variate in a distributed, parallel fashion with constant memory requirements and without any intermachine communication. While it fits our requirements from the perspective of minimal memory consumption, it has the drawback of requiring a logarithmic number of samples from binomials per component. Sequential

techniques, on the contrary, only require a single binomial sample per component [48]. We can improve the runtime however with additional memory by caching the sampling results from early invocations for large partitions that have to be computed for many components. Nevertheless, the presented algorithm is mostly academic in its nature.

However, the special case of block-partitioning of the input data gives rise to a much more practical implementation, which we will also use for our experiments. Every machine processes the data one block at a time, where a block contains a large number of rows (e.g., 1000 by default in SystemML [76]). We leverage this for an approach that uses a single partitioning step only: We first create a random variate $\hat{r}$ from the multinomial distribution $M_b(s; p_b)$, where b equals the number of blocks to consider (the number of rows divided by the blocksize) and the probability vector p_b assigns probabilities proportional to the number of rows per block[2]. A component $\hat{r}_h$ holds the totals of the components of r corresponding to the rows included in the block with horizontal index h. We use a fast sequential algorithm to generate $\hat{r}$ on every machine in the cluster [48]. When we encounter the first row of a block, we generate a temporary random variate of dimensionality l and cardinality $\hat{r}_h$, where l is the number of rows contained in the current block and h is the horizontal index of that block. This random variate contains the number of occurrences for every row of the current block and only is kept in memory until the current block is processed.

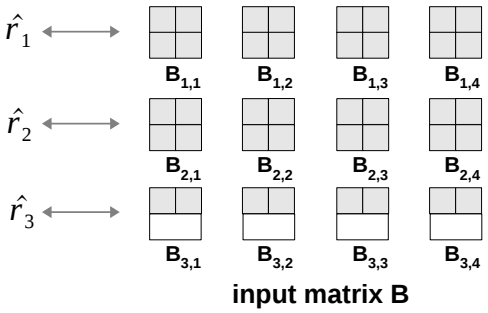


Figure 5.4: Correspondence of the components of a generated random variate to the horizontal columns of blocks in a blocked matrix.

Figure 5.4 illustrates this practical approach for a generating a sample of size s from a 5×8 example matrix B in a block size of 2×2. In the first step, a 3-dimensional random variate $\hat{r} \sim M_3(s; (\frac{2}{5}, \frac{2}{5}, \frac{1}{5}))$ is generated on every machine in the cluster. We use the same seed to guarantee that every machine generates the same random variate. We see that the probability for rows from the bottom blocks is lower ($\frac{1}{5}$) than in the remaining blocks ($\frac{2}{5}$), as the bottom block contains less rows than the remaining blocks. The components $\hat{r}_1$, $\hat{r}_2$ and $\hat{r}_3$ correspond to the number of occurrences of rows in the

[2]Non-fully filled bottom blocks have a lower probability of being selected.

first, second and third horizontal column of blocks in B (c.f., Figure 5.4). A machine that processes a block later in the 'local-sampling'-phase uses $\hat{r}$ to generate a second random variate which corresponds to the number of occurrences of the rows contained in that block. E.g., when a machine processes block $B_{2,3}$, which contains the rows 3 and 4, it uses $\hat{r}_2$ to generate a random variate which denotes the number of occurrences of these rows in the sample as follows: $(r_3, r_4) \sim M_2(\hat{r}_2\,;\,(\frac{1}{2},\frac{1}{2}))$. This approach is memory-efficient as only the small random variate $\hat{r}$ must be kept in memory, whose dimensionality is equal to the number of rows divided by the block size. At the same time, sampling from a block is very efficient, as we only need to generate an additional small random variate of dimensionality equal to the block size.

5.9 Sampling UDFs for a Variety of Meta Learning Techniques

In this section we show how to implement the sample generation for common meta learning techniques based on the previously described sample generation algorithm with the sampling-UDF. Users aiming to implement generation techniques beyond the ones described here can leverage the presented UDFs as a template.

5.9.1 Hold-Out Tests

Section 5.3 described a cross-validation technique known as the hold-out test. The idea behind hold-out tests is to randomly generate two sample matrices according to a user-defined train/test ratio using sampling without replacement. We use the data from the first sample matrix (the training set) to train a model and evaluate its prediction quality using the second sample matrix (the test set).

Algorithm 16: Sampling-UDF for a single hold-out test.

```
1 Input: row index j, random number generator rng, params: user-defined train/test ratio
2 Output: tuple denoting occurrence in training set or test set
3 skip ahead to position j in rng
4 r ← random number generated by rng
5 if r ≤ user-defined train/test ratio then
6   return occurrence in training set at position r
7 else
8   return occurrence in test set at position r
```

Figure 5.5 illustrates the sample generation for a hold-out test using our example dataset of six emails and a train/test ratio of two thirds. We randomly choose four emails to form the training set, and the remaining two comprise the test set. Algorithm 16 shows an

implementation of the sampling-UDF that generates the sample matrices for a hold-out test. As described in Section 5.7, the algorithm invokes the sampling-UDF for every row partition of a processed block, and the result describes in which sample matrices the row occurs. In the case of a hold-out test, every row either occurs in the training set or in the test set. In order to make this decision for a row with index j, we first skip the random number generator rng to the position j in its random sequence (c.f., line 3) to ensure reproducible results. Next, we generate a random number between zero and one from the rng and compare it to a user-defined train/test ratio to decide whether the row occurs in the training set or the test set (c.f., lines 4 to 8).

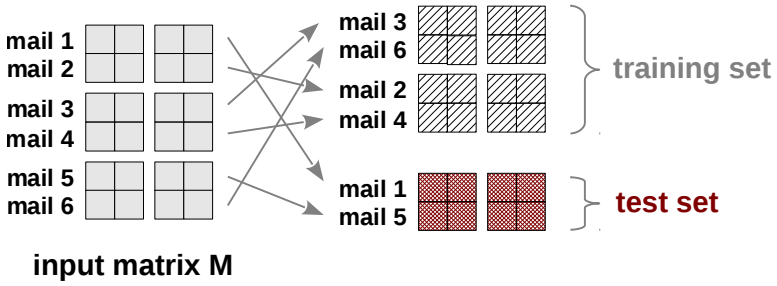


Figure 5.5: Sample generation for a hold-out test: Creating a training set from two thirds of the input rows and a test set from the remaining rows.

In order to improve the estimate of the prediction quality, we might want to conduct more than a single hold-out test. It is easy to adapt the implementation to generate several pairs (say w) of training sets and test sets for a series of hold-out tests. Algorithm 17 shows the necessary code. We simply repeat the approach of generating a random number and using this number to determine the occurrence of a row in the training set or test set w times (c.f., lines 5 to 10). There is one difficulty here: we need w random numbers for each invocation and we have to ensure that every position in the random sequence is only used for a single row to ensure that the results are uncorrelated. Therefore, for each row index $j \in \{1, \ldots, m\}$ (where m is the number of rows of the input matrix), we skip the rng to the position $(j-1) * w$ in its random sequence and use the subsequent w random numbers for the current invocation to determine the occurrences of row j (c.f., line 4). A further feature could be to only use a subset of the data for the training set and test set. This functionality is easily added by only executing lines 7-10 if the random number r is less than the ratio corresponding to the subset of data to be generated and ignoring non-matching rows.

Algorithm 17: Sampling-UDF for a series of hold-out tests.

```
1  Input: row index j, random number generator rng, params: number of hold-out tests w,
   user-defined train/test ratios
2  Output: tuples occs: occurrences in training set or test set
3  occs ← ∅
4  skip ahead to position (j − 1) ∗ w in rng
5  repeat w times
6    r ← random number generated by rng
7    if r ≤ user-defined w-th train/test ratio then
8      add occurrence in w-th training set at position r to occs
9    else
10     add occurrence in w-th test set at position r to occs
11 return occs
```

5.9.2 K-fold Cross-Validation

The most popular cross-validation approach is k-fold cross-validation [24, 72]. Here, we randomly divide the data into k disjoint logical subsets (called folds) using sampling without replacement, and execute k experiments afterwards. In every experiment, we train a parameterized model with all but the k-th fold as training data and test its prediction quality on the held-out data from the k-th fold. In the end, the overall estimate of the prediction quality of the model is computed from the outcomes of the individual experiments.

Figure 5.6 illustrates 3-fold cross-validation on our mail classification example. On a logical level, we assign the six emails randomly to the three folds: mail 2 and 4 form the first fold, mail 3 and 6 the second, and finally the third fold is comprised of mails 1 and 5. We have to create the training sets and tests from the logical folds for the three rounds of experiments. In the first round, the test set consists of the mails from the first fold and the training set is comprised of the rest of the data. In the second round, the second fold becomes the test set and the remaining data is used for training. In the third and final round, we train on folds one and two, and test on the third fold. Algorithm 18 shows a straightforward implementation of the sampling-UDF, which directly creates the k training sets and tests for the k rounds of k-fold cross-validation. Analogous to previous cases, we skip the random number generator rng to the position in the random sequence denoted by the row index j (c.f., line 3). Next, we generate a random number between zero and one and compute the target fold f for the row from that (c.f., lines 5 & 6). Then, we make the function emit k occurrences of the row: it occurs in the test set of the f-th round and in the $k - 1$ training sets of all other rounds (c.f., lines 7 to 11).

Algorithm 18: Naïve sampling-UDF for k-fold cross-validation.

```
1  Input: row index j, random number generator rng, params: number of folds k
2  Output: tuples occs: occurrences in training set or test set
3  occs ← ∅
4  skip ahead to position j in rng
5  r ← random number generated by rng
6  f ← compute fold to assign to as ⌈r ∗ k⌉
7  for w ← 1 . . . k
8    if w = f then
9      add occurrence in w-th test set at position r to occs
10   else
11     add occurrence in w-th training set at position r to occs
12 return occs
```

This naïve approach is highly problematic as it creates k copies of the input data, which poses a serious performance and scalability bottleneck. Ideally, we would only want to create the folds once (resulting in only a single copy of the input data) and read $k-1$ of them as training set in every round. Unfortunately, treating a set of generated sample matrices as if they were a single big matrix does not work out-of-the-box. The block-partitioned representation causes difficulties when the dimensions of the sample matrix for a fold are not an exact multiple of the chosen block size. In such cases, there will be partially filled blocks at the bottom of the sample matrices, which prevent us from treating a collection of sample matrices as a single big matrix. Figure 5.7 shows an example for such a case. We divide a matrix with ten rows into three folds, using a block size of 2×2. The resulting sample matrix for the first fold consists of four rows (with indices 1,2,3,4), while the sample matrices for the second and third fold only consist of three rows (with indices 5,6,7 and 8,9,10). This inevitably leads to non-fully filled blocks

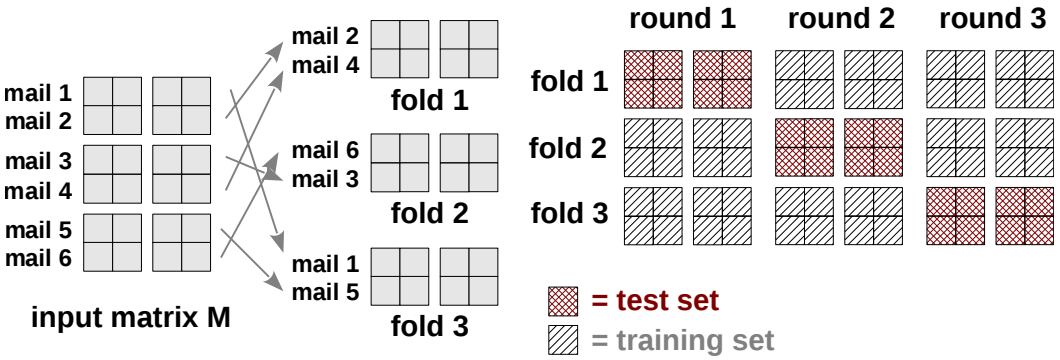


Figure 5.6: Sample generation for 3-fold cross-validation

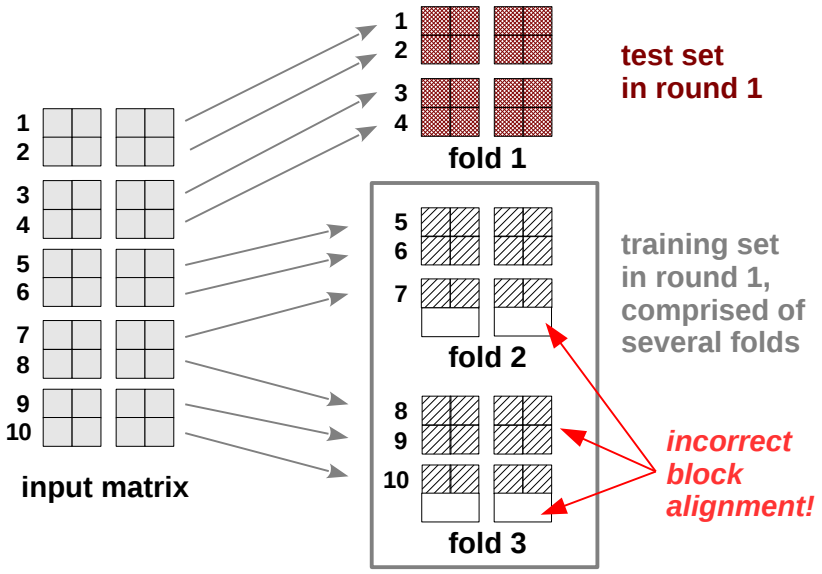


Figure 5.7: Difficulties of treating a set of sample matrices as a single large matrix during a round of k-fold cross validation when matrix dimensions and block size do not align.

at the bottom of the sample matrices for the second and third fold, as we are representing three rows with a block size of 2×2. In the first round of 3-fold cross-validation we need to treat the second and third fold as a single large matrix to be used as training set. Unfortunately, the rows of the matrix are not correctly aligned, as we represent six rows with four blocks of size 2×2. In a single big matrix, the rows with indices 7 and 8 should share a block, but are spread over two different blocks in our case, analogous to the rows with indices 9 and 10. Having operators of an ML system consume such an incorrectly blocked matrix would lead to inconsistent results. In order to circumvent the above issue in our algorithm, we introduce a special mode called 'dynamic sample matrix composition' for k-fold cross-validation. With this mode activated, the algorithm copies the non-fully filled bottom blocks of each generated sample matrix to a specific location in the distributed filesystem and runs an additional job that creates an additional matrix per training set from those blocks. This changes the order in the generated training set, but still provides a uniformly random order as result because we simply switch positions in an already randomized sample without looking at the content of the rows. Finally the algorithm only reads the full blocks of the sample matrices (neglecting the non-fully filled blocks) plus the additionally created matrix as one large composed matrix. 'Dynamic sample matrix composition' requires only a single copy of the input matrix plus k additional small matrices consisting of at most $k - 1$ blocks. With 'dynamic sample matrix composition' activated, we implement k-fold cross-validation with a very simple

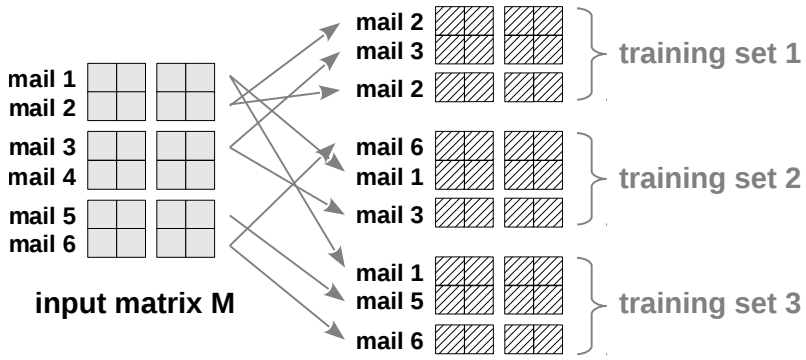


Figure 5.8: Generating bootstrap sample matrices from an input matrix. Note that individual rows can occur more than once in the sample matrices.

UDF that only triggers a single occurrence of each row, as shown in Algorithm 19. After computing the fold f for the row with index n analogous to Algorithm 18, we simply emit a single occurrence in the sample matrix of fold f (c.f., line 6). We refer to this as fold-based approach. We note that 'dynamic sample composition' violates our initial goal of restricting the algorithm to a single pass over the data. However, the second pass only reads a small fraction of the generated data (the non-fully filled blocks on the bottom of the sample matrices). Our experiments in Section 5.9.2 clearly indicate that this additional work pays off in terms of runtime and output size of the generated data.

Algorithm 19: Fold-based sampling-UDF for k-fold cross-validation.

```
1 Input: row index j, random number generator rng, params: number of folds k
2 Output: tuple denoting occurrence in fold
3 skip ahead to position j in rng
4 r ← random number generated by rng
5 f ← compute fold to assign to as ⌈r ∗ k⌉
6 return occurrence in f-th fold at position r
```

5.9.3 Bagging

A popular ensemble method leveraging sampling with replacement is bagging [34], which trains a set of models on bootstrap samples of the input data and combines their predictions afterwards. In our example, we would draw bootstrap samples of mails from our mail corpus (c.f., Figure 5.8), and train a classification model on each sample matrix in isolation. In order to decide whether to consider a newly arriving mail as spam, we would combine the predictions of the individual classifiers, i.e., by majority vote [24].

The sampling-UDF for generating bootstrap samples (c.f., Algorithm 20) directly invokes the recursive function *sample_multinomial* from Section 5.8 to determine the number of occurrences of a row with index j in the bootstrap sample (c.f., line 4). We skip the PRNG to the position $j * s$ (c.f., line 5), where $j \in \{1, \ldots, m\}$ is a row index and s is the desired sample size. This leaves the first s positions in the random sequence for the computation of the number of occurrences (via *sample_multinomial*) and another s positions for generating random numbers for the *relative_pos* values of the occurrences of row j.

Algorithm 20: Sampling-UDF for bootstrap sampling.

```
1 Input: row index j, random number generator rng, params: number of rows m, sample
  matrix size s
2 Output: tuples occs: occurrences in bootstrap sample
3 occs ← ∅
4 t ← sample_multinomial( j, 0, m, s, rng )
5 skip ahead to position j * s in rng
6 for 1 ... t
7   r ← random number generated by rng
8   add occurrence at position r to occs
9 return occs
```

5.10 Experiments

We conduct experiments with Apache Hadoop 1.0.4, Apache Spark 1.0 and Apache Hive 0.12 [169]. In every experiment, we spend effort to tune system parameters to improve performance. We run the evaluation on a 192-core cluster consisting of 24 machines. Each machine has two 4-core Opteron CPUs, 32 GB memory and four 1 TB disk drives. The datasets in use are synthetic. We generate random matrices with uniformly distributed non-zero cells for a specified dimensionality and sparsity. Analogous to [76], we employ a fixed block size of 1000×1000 and optimize the physical block representation for dense and sparse data. Dense matrices have smaller dimensions and a large data size, while sparse matrices have higher dimensionality and comparatively low data size. We run experiments with both to show that our algorithm efficiently handles each of these types of matrices.

5.10.1 Scalability

The goal of the first set of experiments is to evaluate the scalability of our algorithm with regard to the size of the input data, the number of machines in the cluster and the number of samples drawn.

Scalability with increasing data size: First, we look at the effects when generating samples from increasing input sizes on a fixed number of machines (the whole 192-core cluster). We start with a dense $100,000 \times 10,000$ matrix with 1 billion entries. We repeat this experiment on larger matrices by increasing the number of rows of the input, so that we obtain matrices with 2.5 billion, 5 billion, 7.5 billion and finally 10 billion entries. We test two sampling techniques on every input matrix. First, we generate 10 training set/test set pairs for hold-out-testing (c.f., Section 5.9.1). Second, we draw 20 bootstrap samples of the data (c.f., Section 5.9.3). In order to make the results comparable, we configure the sample sizes so that the aggregate size of the sample matrices is equal to the size of the input matrix. The disk-based Hadoop implementation reads the input matrix from the DFS and writes back the sample matrices to the DFS. For our memory-based Spark implementation, we test the sample generation from in-memory matrices: here the input matrices are already cached in-memory and we only materialize the resulting sample matrices in-memory. Figure 5.9(a) shows the results of this series of experiments. We observe a linear speedup for both implementations and sampling techniques. Bootstrap sampling incurs a higher overhead to the runtime, due to the higher complexity of sampling from a multinomial (c.f., Section 5.9.3). As expected, the Spark implementation is up to ten times faster than Hadoop, because it reads from and writes to memory and applies hash-based grouping. We repeat this set of experiments for matrices with 90% sparsity. We start with a $500,000 \times 20,000$ input matrix comprised of 10 billion entries (and 1 billion non-zeros). Again, we increase the number of rows to obtain matrices of size 25 billion, 50 billion, 75 billion and 100 billion, and execute both sampling techniques on those. Analogous to the previous experiment, the Hadoop implementation shows a linear speedup for both, bootstrap and hold-out sampling (c.f., Figure 5.9 (b)). Unfortunately, Spark was not able to efficiently handle the large matrices used in this set of experiments (we terminated the execution when Spark's shuffle did not complete after 15 minutes).

Scalability with increasing data and cluster size: Next, we investigate the effects of increasing the number of machines proportionally to a growing amount of input data. Again we generate 20 bootstrap and hold-out samples. We start with a dense $250,000 \times 10,000$ matrix (having 2.5 billion non-zeros) as input on a cluster of six machines. In the following experiments, we double the number of machines as well as the number of rows of the input matrix, up to 24 machines and 10 billion non-zeros. We measure the sample generation from memory-resident input with our Spark implementation. Figure 5.10 depicts the resulting execution times. Ideally, we would see a constant runtime. However, it is not possible to reach this ideal scale-out for many reasons (e.g., network overheads). The same effect has been observed in other large-scale ML systems [76]. Nonetheless, our algorithm achieves a steady increase in execution time with growing data and cluster size.

Scalability with increasing number of sample matrices: Finally, we focus on the effects of increasing the number of sample matrices to generate. We make our Hadoop

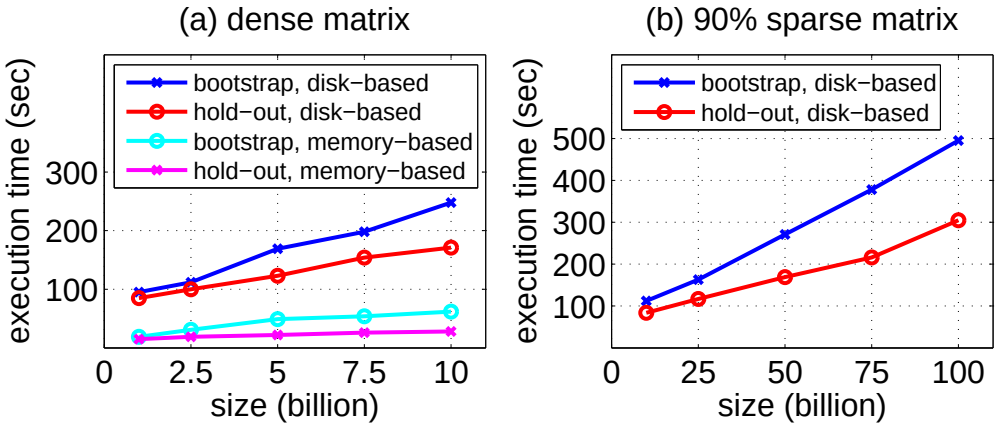


Figure 5.9: Linear speedup for the generation of 20 sample matrices when increasing the input data size on a fixed cluster size.

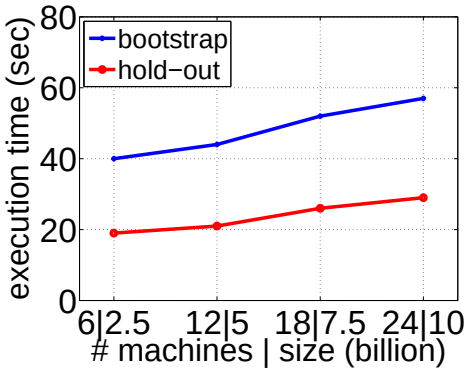


Figure 5.10: Effect of increasing the cluster and input data size.

implementation generate samples from a $500,000 \times 10,000$ matrix with 5 billion non-zeros. First, we generate up to 250 small sample matrices, where every sample matrix contains one percent of the rows of the input matrix (c.f., Figure5.11(a)). Next, we increase the size of the sample matrices to five percent of the input data and generate up to 50 sample matrices per run (c.f., Figure5.11(b)). We test both cases with bootstrap and hold-out sampling. Our results show that the runtime increases linearly with the number of samples to generate and that our algorithm is able to generate a large number of samples efficiently with a single pass over the data. Again, the runtime is higher for bootstrap sampling due to the higher complexity of the underlying sampling process.

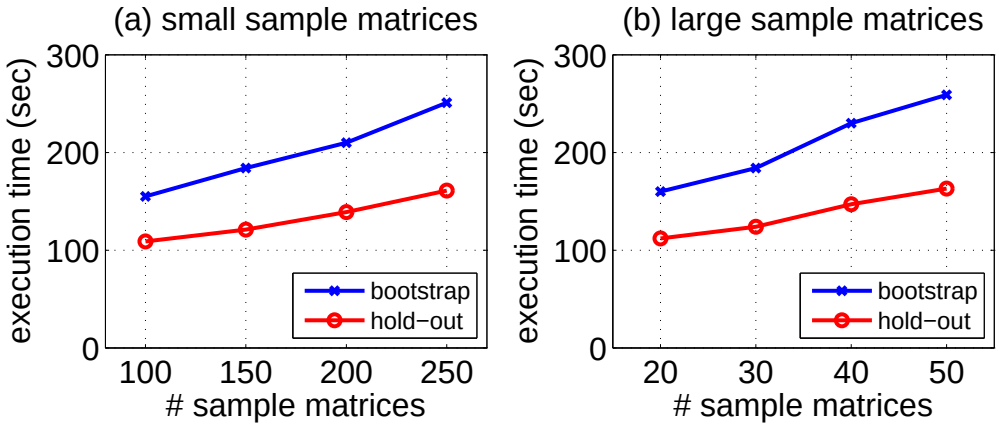


Figure 5.11: Linear speedup when increasing the number of sample matrices for different sampling techniques.

5.10.2 Dynamic Sample Matrix Composition

Next, we evaluate the benefits of our 'dynamic sample matrix composition' technique for k-fold cross-validation (c.f., Section 5.9.2). We make our Hadoop implementation generate the training sets and test sets for 5-, 10- and 20-fold cross-validation on a dense $100,000 \times 10,000$ matrix with 1 billion entries as well as on a sparse $500,000 \times 20,000$ matrix with 1 billion non-zeros. For every case, we first generate the training sets and test sets directly with the naïve approach. We compare this to the generation of the training sets and test sets via k-folds with 'dynamic sample matrix composition' activated in our algorithm. Figures 5.12(a) and 5.12(b) depict the results of this experiment for the dense and sparse matrices. The plots on the left show the execution times of both approaches. We see that the time of the naïve method grows linearly with increasing k, while the execution time of the fold-based approach stays constant. This is expected because the naïve approach has to create k copies of the data, while the fold-based method only has to create the folds (summing to a single copy of the input data) plus some extra blocks to allow for the dynamic composition of sample matrices. In the case of 5-fold cross-validation on the sparse matrix, the naïve approach is faster as it only runs a single pass over the input data, while the fold-based approach has to additionally process the partially filled blocks of the sample matrices after generation, for creating the extra blocks for dynamic composition. For $k > 5$, the fold-based approach outperforms the naïve approach in all cases by a factor between 1.6 and 3.3. The plots on the right side of Figure 5.12 show the ratio of output size (the sample matrices forming the training sets and test sets) to input size. As expected, the naïve approach produces an output that is k-times as large as the input size. On the contrary, the fold-based approach produces a much smaller output: it creates k folds, which, in combination, have the same size as the

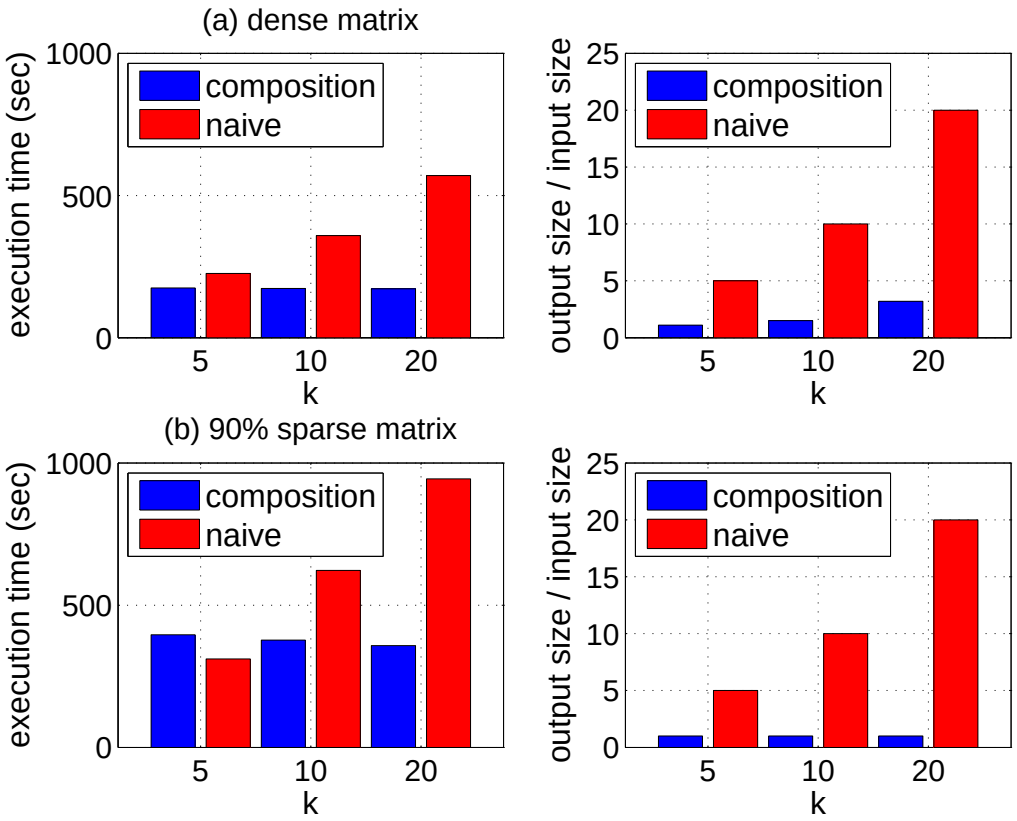


Figure 5.12: Runtime and output sizes when generating training sets and test sets for k-fold cross-validation with and without 'dynamic sample matrix composition'.

input. Additionally, the fold-based approach has to create extra blocks per training set from the non-fully filled blocks of the folds (c.f., Section 5.9.2). In the case of the dense matrix, the additional output size for the extra blocks amounts to a factor between 0.1 and 2.2 of the input size (which is still small compared to the sizes produced by the naïve approach). In the case of sparse matrices, the additional output size for the extra blocks is tiny (as we represent only the non-zeros of blocks) and ranges from a factor of 0.003 to 0.008. In summary, the fold-based approach provides a runtime constant in k (instead of linear in k for the naïve approach) and produces output sizes that are up to an order of magnitude smaller.

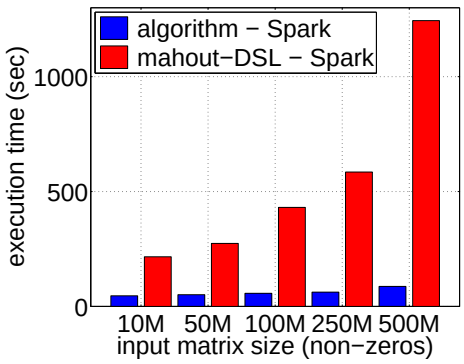
5.10.3 Comparison to Existing Techniques

In the following, we compare our proposed algorithm to two existing approaches for distributed sample generation.

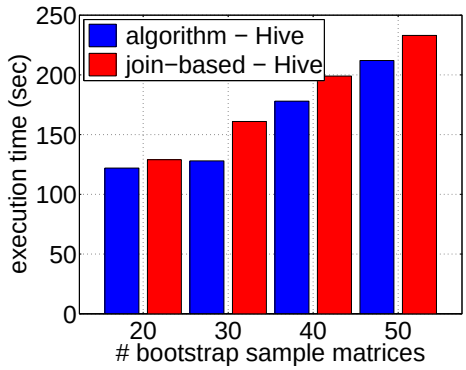
Comparison against Matrix-based Sample Generation: This experiment demonstrates that our proposed algorithm provides a huge performance benefit over implementing sampling with matrix operations. Many large-scale ML systems (e.g., [13, 76, 89, 99]) offer operators to conduct linear algebraic operations on large matrices, including ones required for sample generation, which is performed using linear algebra as follows. Assume we want to generate sample matrices with a total of p rows from an $m \times n$ input matrix M. For that, we have to create a binary, sparse 'selection' matrix S of size $p \times m$. This selection matrix contains only a single non-zero entry per row and is multiplied to the left with the input data matrix: $SM = A$. The resulting matrix A contains all p rows selected for the sample matrices. Setting the value (i, j) in the selection matrix S makes the j-th row of the input matrix M appear at the i-th position of A. We obtain individual sample matrices by slicing A. We compare our algorithm to sample generation with existing distributed matrix operations. The latest version of Apache Mahout [13] provides a Scala-based domain specific language (DSL) for distributed linear algebraic operations on row-partitioned matrices. Programs written in this DSL are automatically parallelized and executed on Apache Spark. We implement matrix-based sample generation as follows in Mahout's DSL. First, we load the input matrix M from the DFS into memory on our cluster. Next, we programatically create the sparse selection matrix S in-memory. We multiply the distributed data matrix M on the left by the sparse selection matrix S to obtain A. Since A is a concatenation of all requested sample matrices, we finally read the individual sample matrices in parallel as slices of A.

```
// load distributed input data matrix
val M = loadMatrixFromHDFS(...)
// programmatically create in-memory selection matrix
val S = createSelectionMatrix(input, numSamples, ...)
// distributed multiplication of selection matrix S and input matrix M
val A = S %*% M
// read individual sample matrices as slices of A in parallel
(0 until numSamples).par.foreach { ... }
```

We compare the performance of the matrix-based sample generation with Mahout's DSL on Apache Spark against a Spark implementation of our algorithm. Since Mahout is optimized for handling sparse data, we use matrices with 90% sparsity in this evaluation. We generate 20 hold-out samples from several large matrices with up to $500,000,000$ non-zeros and measure the execution time. Both approaches read the input matrix from the DFS and materialize the individual sample matrices in-memory. The results shown



(a) Comparison of our Spark implementation to a matrix-based approach with Mahout's Spark DSL.



(b) Comparison of our Hive implementation to join-based sample generation in Apache Hive.

Figure 5.13: Comparisons against existing techniques.

in Figure 5.13a show our algorithm handles the sample generation in very short time (only 87 seconds for the largest matrix) with its runtime only moderately affected by the input size. On the contrary, the matrix operations in Mahout need 5 to 20 times longer for the sample generation. Furthermore, the matrix-based approach does not scale well, as a linear increase of the data size leads to super-linear increases in the runtime. These results indicate that distributed matrix operations do not fit the sample generation problem well and large-scale ML systems benefit from a dedicated sample generation machinery.

Comparison against Join-based Sample Generation: Our final experiment compares a SQL-based bootstrap sampling technique [45] to our approach. This technique assumes the input matrix is stored in a relational table and executes in two steps. First, a temporary table describing assignments of rows to sample matrices is created. Second, this temporary assignments table is joined with the input matrix table to form the sample matrices. This approach has several drawbacks for our ML use case. First, the non-blocked matrix representation, where every cell is individually represented, has huge performance penalties for many operations [76]. Secondly, while many ML toolkits and systems offer primitives for linear algebraic and statistical operations, many lack join primitives [13, 76, 89, 99]. Nevertheless, we perform this experiment to illustrate the benefits of our approach which eliminates both the need for the temporary assignment table and the join. We adapt the join-based approach to use a blocked matrix representation for dense matrices using double arrays. We restrict our experiments to the dense case, as it is unclear to us how an elegant representation of blocked sparse matrices would look like in a SQL-based system. We implement the join-based approach in the distributed

query processor Apache Hive using HiveQL[3]. The input is held in the table `input` with the schema `(h_index INT, v_index INT, block ARRAY<ARRAY<DOUBLE>>)`, where `h_index` and `v_index` refer to the horizontal and vertical index of a block and `block` represents its contents as a two-dimensional array. Again, we employ a block size of 1000×1000. The temporary table is called `assignments` with the schema `(sm_id INT, row_id INT, sm_pos INT)` where `sm_id` denotes the sample matrix, `row_id` the row to select and `sm_pos` the position for the row in the sample matrix to create. We implement the user-defined aggregation function `reblock`, which forms blocks from a collection of rows (c.f., 'streaming re-block' in Section 5.7). Unfortunately, we do not find an elegant way to create and fill the assignment table during query time in Hive, so we decide to create and fill it upfront and omit the time required for that in the evaluation. After creation of that table, we execute the sample generation in a single query: we extract row partitions from the blocks of the input matrix, join these row partitions with the `assignments` table, group the result and materialize the blocks with `reblock`:

```
-- Creation and filling of assigments table omitted
SELECT a.sm_id, floor(a.sm_pos / 1000), r.v_index, reblock(r.row, a.sm_pos) FROM (
  SELECT h_index * 1000 + offset AS row_id, v_index, row AS r FROM input
  LATERAL VIEW explode(block) AS offset, row)
JOIN assignments a ON r.row_id = a.row_id
GROUP BY a.sm_id, r.v_index, floor(a.sm_pos / 1000);
```

Next, we implement our algorithm in HiveQL, with the sampling-UDF implemented as a UDTF. Analogous to the join-based approach, the sample generation requires only a single query: we extract row partitions from the blocks, apply the sampling function, group the result and form blocks with `reblock`.

```
SELECT sm_id, sm_h_index, v_index, sm_block FROM (
 SELECT sm_id, v_index, collect(row, rel_pos) AS col
 FROM (
  SELECT sm_id, v_index, rel_pos, row FROM input
   LATERAL VIEW explode(block) AS offset, row
   LATERAL VIEW sampling_UDF(h_index * 1000 + offset, params) AS sm_id, rel_pos)
 GROUP BY sm_id, v_index)
LATERAL VIEW reblock(col) AS sm_h_index, sm_block
```

Figure 5.13b shows the execution times for generating 20, 30, 40, and 50 bootstrap sample matrices with $10,000$ rows each from a dense $500,000 \times 10,000$ matrix. We omit the time for creating and filling the assignments table in the join-based approach, as we had to conduct this manually before running the experiment. Both approaches only require a single MapReduce job. Hive selects a broadcast hash-join for executing the

[3]LATERAL VIEW is HiveQL's statement for applying table-generating functions. We omit aliases for lateral view statements to improve readability.

join-based approach. The results illustrate that our proposed approach outperforms the join-based approach in all experiments, although the latter one has competitive performance. However, our approach has the additional benefit of eliminating both materialization of a temporary assignments table and the required join of this table with the input data. From a relational data processing perspective, the 'local-sampling'-phase of our algorithm can be imagined as distributed hash-join: The row partitions in the matrix blocks form the probe side and we generate the sample assignments in the build-side on the fly using the sampling-UDF.

5.11 Related Work

Non-distributed languages and libraries for machine learning offer extensive meta learning functionality. R [166] has support for cross-validation in the 'Design', 'DAAG' and 'Boot' packages, scikit-learn [142] offers the 'sklearn.cross_validation' package and furthermore includes models with built-in cross-validation. MATLAB provides a 'model building and assessment' module in its statistics toolbox.

In recent years, many systems and libraries have been proposed for scalable, distibuted ML; however, these lack support for a comprehensive meta learning layer. A comprehensive vision for meta learning on large ML datasets is presented as part of MLBase [108, 126]; however, the proposed optimizer has not been implemented yet. SystemML [76] executes programs written in an R-like language called DML in parallel on MapReduce. SystemML requires manual expression of sample generation and model evaluation in terms of matrix operations in DML [26]. Cumulon [89] executes matrix-based data analysis programs in the cloud, but presents no meta learning functionality. Apache Mahout [13] offers implementations of ML algorithms in MapReduce and recently switched to building upon an R-like DSL that executes in parallel on Apache Spark. Mahout provides limited meta learning functionality for individual algorithms (e.g., hold-out testing for recommendation models), but lacks a principled meta learning layer. GraphLab [117] trains ML models using an asynchronous, graph-parallel abstraction, and presents a Python API supporting evaluation for recommenders and regression models. However, we are unaware of any description for mapping meta learning onto GraphLab's vertex centric data model. Apache Spark [187], which forms the runtime for MLBase, provides rudimentary sample generation functionality. Similar to our proposed sampling-UDF, a customizable random sampler is applied to large datasets. At the time of this writing, this sampling functionality cannot handle block-partitioned matrices with partitioned training instances and does not offer the generation of a large number of samples in a single pass over the data. [81] proposes extending the MapReduce execution model to efficiently implement predicate-based sampling so that required cluster resources for sample computation are a function of sample size instead of input data.

Cohen et al. [45] present a SQL-based approach for distributed generation of bootstrap samples. In contrast to our work, this approach is not designed for block-partitioned data and requires the generation of a temporary assignment table as well as join primitives (c.f., Section 5.10.3). Spark produces bootstrap samples from distributed datasets via a so-called 'takeSample' action. This action uses Poisson sampling, which will not guarantee a fixed sample size and therefore requires to draw a much larger sample than requested. Furthermore, the sampling must potentially be repeated if the Poisson sample is too small and the sampled data must fit into the memory of the driver machine. Kleiner et al. present an alternative to the classical bootstrap called 'bag of little bootstraps' which combines the results of bootstrapping multiple small subsets of a larger original dataset [104].

5.12 Conclusion

In this chapter, we propose a general, single-pass algorithm for generating sample matrices from large, block-partitioned matrices stored in a distributed filesystem. Embedded in the algorithm is a user-defined sampling function which allows adaption to a variety of meta learning techniques with minimal code modification. We discuss implementing common meta learning techniques using the sampling UDF, particularly distributed sampling with replacement and generation of training sets for k-fold cross-validation. We implemented our algorithm in three different systems: Apache Hadoop, Apache Spark, and Apache Hive. Through extensive empirical evaluation, we found our algorithm scales linearly for both input data size and number of samples. Our algorithm outperforms matrix-based sample generation techniques by more than an order of magnitude and is faster than an approach using a distributed hash-join.

6 Conclusion

We revisit the thesis statement in Section 6.1 and the research conducted in its context. Afterwards, we briefly discuss results and future research directions in Section 6.2 and look at both the negative and positive real-world impact of scalable data mining technology in Sections 6.3 and 6.4.

6.1 Retrospective

This thesis lays ground work for enabling scalable data mining in massively parallel dataflow systems on the large datasets collected nowadays. The discussion of Principal Component Analysis in Chapter 1 exemplarily illustrated common fallacies with respect to scalable data mining. It is in no way sufficient to naively implement textbook algorithms on parallel systems; bottlenecks on all layers of the stack prevent the scalability of such naive implementations. For example, the runtime of mathematical operations in the algorithm might be superlinearly dependent on the input cardinality, the parallel processing system might not be able to efficiently execute a certain computational workload such as iterations or tasks specific to data mining like hyperparameter search might not be well supported. Acknowledging these fallacies lead us to the thesis statement:

Scalability in data mining is a multi-leveled problem and must therefore be tackled on the interplay of algorithms, systems, and applications. None of these layers is sufficient to provide scalability on its own. Therefore, this thesis presents work on scaling a selection of problems in these layers. For every selected problem, we show how to execute the resulting computation automatically in a parallel and scalable manner. Subsequently, we isolate user-facing functionality into a simple user-defined function, to enable usage without a background in systems programming.

Achieving scalability in data mining is a huge research area and a thesis can obviously only address a selection of problems from that area. In accordance to the thesis statement, we chose a multi-leveled approach, and presented research on the level of algorithms, systems and applications. We started by investigating *algorithm-specific scalability aspects* in Chapter 3. We put our focus on the family of collaborative filtering algorithms for computing recommendations, a popular data mining use case with many industry deployments. We showed how to efficiently execute the two most common approaches to collaborative filtering, namely neighborhood methods and latent factor models on

MapReduce. We evaluated our implementations on datasets with billions of datapoints and highlighted deficiencies of the MapReduce model. Furthermore, we investigated a specialized architecture for scaling collaborative filtering to extremely large datasets that occur in the use cases of world-leading internet companies with hundreds of millions of users. Lastly, we presented research in the area of distributed graph processing, with the focus on network centrality. We highlighted scalability bottlenecks in dataflow systems for this specific scenario and discussed short-comings of the vertex-centric programming model, on which current specialized graph processing systems build upon. We turned to *system-specific scalability aspects* in Chapter 4. We concentrated on a special class of algorithms, namely fixpoint algorithms, which are common in machine learning and graph processing tasks. We improved system performance during the execution of these iterative algorithms by drastically reducing the system overhead required for guaranteeing fault tolerance. We proposed a novel optimistic approach to fault-tolerance which exploits the robust convergence properties of a large class of fixpoint algorithms and does not incur measurable overhead in failure-free cases. Finally, we conducted work on an example of *application-specific scalability aspects* of data mining in Chapter 5. This area is often overlooked in database research, which mostly focuses on efficient model training. A common problem when deploying machine learning applications in real-world scenarios is that the prediction quality of the ML models heavily depends on hyperparameters that have to be chosen in advance. Finding these hyperparameters very often involves searching through a huge number of possible combinations, which is a resource-intensive and tedious task that greatly benefits from parallelism. We picked out one aspect of this problem, namely the efficient generation of samples from large, block-partitioned matrices. We developed an algorithmic framework for efficiently generating samples from these matrices in parallel and described how to implement sampling with replacement as well as common cross validation techniques. The currently observable trend of research on scalable data mining slowly moving up the stack shows the viability of our approach of tackling the scalability problem on a multitude of layers.

Another theme in the thesis statement is enabling data scientists without a background in systems programming to conduct scalable data mining. We tackle this challenge by isolating user-facing functionality into simple user-defined functions. We model these UDFs such that users can easily modify algorithm implementations without having to understand the underlying distribution and parallelization strategies. These UDFs typically only require a few lines of code and basic mathematical knowledge about the data mining approach. We introduced a huge variety of UDFs throughout the work on different levels of scalability:

- abstraction of similarity measures for parallel similarity computation in neighborhood-based collaborative filtering (Section 3.3.1)
- abstraction for solving the linear systems in the Alternating Least Squares algorithm

for latent factor models (Section 3.3.3)

- abstraction for stochastic gradient descent updates in latent factor models for easy adaption of the objective function (Section 3.4.3)
- abstraction for the compensation function to enable optimistic recovery for the distributed execution of fixpoint algorithms (Section 4.4.1)
- abstraction of the sampling technique for efficiently implementing sample generation from distributed, block-partitioned matrices (Section 5.6)

The generality of this UDF isolation technique lets us conclude that this approach forms an important pattern for building scalable and easy-to-use data mining libraries. We already implemented the UDF isolation in our open-source contributions to Apache Mahout and currently see further adoption of this pattern in younger libraries such as Spark MLlib.

6.2 Discussion of Results and Future Research Directions

This thesis investigated scalable machine learning tasks on a variety of systems: MapReduce-based systems, vertex-centric graph processing systems, general dataflow systems and parameter server architectures. This gives rise to the question which systems are going to dominate the field and industry deployments in the future. During our work on scaling latent factor models on MapReduce in Section 3.3, we found that while we could efficiently scale the computation, it required a lot of effort in our join implementation as well as manual tuning and changes in the map task execution model. Modern systems like Apache Flink automate these tasks, e.g. Flink's optimizer selects a well working join implementation automatically, chains joins with subsequent map tasks and furthermore makes Flink efficiently cache iteration-invariant data. Due to the limitations of MapReduce for the iterative algorithms and its lack of general operators, we think that the current trend of moving a large class of scalable machine learning tasks to general dataflow systems such as Apache Spark and Apache Flink will continue. The short-comings we encountered during our investigation on scaling centrality computation on distributed graph processing systems Section 3.5, lead us to think that specialized systems like Apache Giraph and Pregel will also soon be subsumed by general dataflow systems. General data flow systems are able to efficiently run vertex-centric programs, and additionally provide the machinery for conducting relational operations on the graph data. While recent work shows that single-machine solutions easily outperform distributed systems on the dataset sizes used in academia [29, 112, 119, 123], we think that these single machine solutions will stay constrained to academic use cases. In industry scenarios, the value of distributed graph processing on general dataflow systems comes not from the performance

of the algorithms, but from the ability to create complex pipelines mixing ETL, machine learning and graph-processing tasks, using a single system. Furthermore, the data is typically collected and stored in distributed filesystems, which is again beneficial for systems tailored to such architectures. We think that specialized asynchronous systems for machine learning, such as our parameter server Factorbird (c.f., Section 3.4) will stay a niche phenomenon. Such architectures only make sense in use cases with extremely large datasets (which only a few companies in the world encounter) and require highly skilled technical staff to develop and maintain such a system, as the inherent asynchronicity makes the system (and development issues like debugging) very complex. For all the reasons mentioned, we think that the majority of scalable machine learning tasks in the coming years will run on general data flow systems like Apache Spark and Apache Flink.

While our mechanism for optimistic recovery with compensation functions from Section 4 has great theoretical benefits like optimal failure-free performance, we acknowledge that a major disadvantage of our proposed approach is that it incurs high implementation complexity to both system as well as library implementors. Furthermore, the method is only applicable for a small class of workloads. In our research, a lot of this complexity stems from Flink's optimizer-integrated iterations abstraction though. In the future, an integration into simpler dataflow systems such as Spark, which already provide lineage-based recovery could be worth looking into. Compensation functions could replace the lineage computations in iterative parts of the workload, and thereby leverage the already existing re-computation functionality. We think that there are still many interesting academic questions open with regards to this line of research; e.g., whether the compensation function can be automatically derived from invariants present in the algorithm definition, or whether a compensation function can provide an upper bound on the additional work incurred. It would also be interesting to find compensation functions for an even broader class of problems. Finally, it might be beneficial to leverage memory-efficient probabilistic data structures for storing statistics to create better compensation functions.

We think that our work on efficient sample generation from block-partitioned matrices in Section 5 is going to be part of a large body of work on scalable meta learning that will be conducted in the coming years. In industry use cases, meta learning is such a tedious and resource-intensive task that improvements in this field will have a huge real-world impact. We expect to see systems for distributed ML with a comprehensive meta learning layer in the future [108, 126]. This layer will automate hyperparameter selection and ensemble learning and will offer a wide variety of cross-validation techniques. A major advantage of such an automation is that the amount of systems programming knowledge required for data scientists using these ML engines will be minimized. This will make large-scale ML systems accessible for much broader audiences (e.g., trained data analysts and statisticians). Distributed block-partitioned matrices are becoming the dominant

physical representation for large-scale ML; we currently observe their widespread adoption in many systems (e.g., in Apache Mahout [13], the ML library of Apache Flink [9] and Apache Spark MLlib [126]). Sample generation from such matrices, as presented in this chapter, will form an integral part of a future meta learning layer, as it is instrumental for automating cross-validation. We expect to see a tighter integration of the meta learning layer with automatic optimizers present in many systems. A promising research direction is to provide algebraic rewrites that allow optimizers to exploit specific data partitionings (e.g., the overlapping partitioning required for k-fold cross-validation).

6.3 Negative Implications of Scalable Data Mining Technology

It is the responsibility of scientists to reflect about potential real-world impact of their field, therefore we feel obliged to issue our concerns about some of the current political and economical developments that are enabled by technological developments in scalable data collection and analysis. Nowadays, scaling data mining to datasets containing data in the order of the whole human population is technically possible and already practiced in cases such as Facebook, which operates a social network comprised of a sixth of the world's population. Furthermore, scalable data mining is constantly getting cheaper and becoming increasingly commoditized through the availabilty of cloud-based data centers and recent advances of data mining technology, e.g., as discussed in this thesis.

These technological advances are accompanied by a massive increase of data collection on private individuals, conducted by both commercial and state actors. In the commercial sphere, this data collection is motivated by the aim to improve the conversion rates in personalized advertising, the prime source of revenue for many internet-based companies. As the ML models used for selecting the ads to display improve with increasing amount of profile data about the targeted persons, a technological infrastructure has been created that allows for tracking the content consumed by individuals browsing the general web. This tracking is realized with ads, like-buttons and tracking pixels that are embedded in many websites but are always directly loaded from the tracking companies' servers, which allows the tracking companies to record the browsing behavior of visitors of the tracked websites. The data recorded by this tracking infrastructure has been reported to contain large portions of the online news consumption [171] as well as intimate, health-related personal data [55]. The ability to consume news and form a political opinion in an independent and unwatched manner as well as the privacy of personal health-related data are vital for an open society, and should not be subject to commercially motivated data collection. We feel that public pressure for regulation of tracking on the internet is necessary and ultimately hope for the development of new business models that make tracking obsolete.

The most controversial data collection and analysis nowadays is performed by state actors. We encounter an unprecedented bulk collection (of which many details are unclear) of phone and internet communication data, that is performed by secret services on a massive scale [135, 168]. Furthermore, this data collection seems to be accompanied by a concerning lack of parliamentary supervision [57, 134]. This bulk surveillance not only violates constitutionally assured privacy rights and contradicts the principle of presumption of innocence, but also poses a dystopian potential for misuse in the future. An even more alarming development of which the public has recently become aware is the military use of scalable data mining technology, which, according to many commentators, is happening outside of limits imposed by the rule of law. Examples for this are phone and location metadata analysis which reportedly put a journalist on a terror watchlist [160], and the identification of unnamed targets for lethal drone strikes [133]. In the face of these hard-to-deal-with learnings, we feel that it is the duty of our generation to oppose these developments, and make sure that our fundamental rights are preserved in our societies, despite of the many technological advances that are abused to restrict these fundamental rights.

6.4 Benefits of Scalable Data Mining Technology to Other Scientific Fields

We close the thesis with a positive outlook. Scalable data processing technology has a huge potential to transform and advance a wide field of sciences. We observe this development already for subfields of computer science. One such field is natural language processing, where the current abundance of available textual data and computational processing power enables many advances. Examples are the mining of parallel text from public available web corpora for many domains previously unavailable in curated datasets [162], and the development of simpler and at the same time more powerful models for machine translation systems [83]. In web science, scalable data mining allows researchers to study the shape of the webgraph formed by billions of websites [127], and enables breakthroughs such as the rejection of Stanley Milgram's famous assumption of 'six degrees of separation' among humans through experiments on social networks with hundreds of millions of users [16]. A non-computer science field that will be profoundly transformed by scalable data collection and processing technology are the social sciences. Currently there is a transition undergoing from methods of the humanities towards methods of the sciences, accompanied by a breakdown of the division between quantitative and qualitative social science [102]. Though in its early phase, this transformation is already producing astonishing scientific outcomes such as a large-scale study of nation-state censorship mechanisms [103].

Bibliography

[1] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *Journal of Machine Learning Research*, 15(1), pp. 1111–1133, 2014.

[2] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *Journal on Very Large Data Bases*, 23(6), pp. 939–964, 2014.

[3] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Mapreduce and pact-comparing data parallel programming models. *Fachtagung Datenbanksysteme für Business, Technologie und Web*, pp. 25–44, 2011.

[4] Alexander Alexandrov, Felix Schüler, Tobias Herb, Andreas Kunft, Lauritz Thamsen, Asterios Katsifodimos, Odej Kao, and Volker Markl. Implicit parallelism through deep language embedding. *ACM SIGMOD International Conference on Management of Data*, pp. 47–61, 2015.

[5] Alexander Alexandrov, Kostas Tzoumas, and Volker Markl. Myriad: scalable and expressive data generation. *Proceedings of the VLDB Endowment*, 5(12), pp. 1890–1893, 2012.

[6] Kamal Ali and Wijnand van Stam. Tivo: Making show recommendations using a distributed collaborative filtering architecture. *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 394–401, 2004.

[7] Sattam Alsubaiee1 Yasser Altowim1 Hotham Altwaijry, Alexander Behm, Vinayak Borkar1 Yingyi Bu1 Michael Carey, Inci Cetindil1 Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova1 Raman Grover1 Zachary Heilbron, Pouria Pirzadeh1 Vassilis Tsotras7 Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source BDMS. *Proceedings of the VLDB Endowment*, 7(14), pp. 1905–1916, 2014.

[8] Xavier Amatriain. Tutorial for building industrial-scale real-world recommender systems, http://www.slideshare.net/xamat/building-largescale-realworld-recommender-systems-recsys2012-tutorial.

[9] Apache Flink ML, https://flink.apache.org.

[10] Apache Giraph, https://giraph.apache.org.

[11] Apache Hadoop, https://hadoop.apache.org.

[12] Apache HBase, https://hbase.apache.org.

[13] Apache Mahout, https://mahout.apache.org.

[14] Apache ZooKeeper, https://mahout.apache.org.

[15] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. *ACM SIGMOD International Conference on Management of Data*, pp. 1383–1394, 2015.

[16] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. *ACM Web Science Conference*, pages 33–42, 2012.

[17] Chaitanya K. Baru, Gilles Fecteau, Ambuj Goyal, H Hsiao, Anant Jhingran, Sriram Padmanabhan, George P. Copeland, and Walter G. Wilson. Db2 parallel edition. *IBM Systems Journal*, 34(2), pp. 292–322, 1995.

[18] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/Pacts: a programming model and execution framework for web-scale analytical processing. *ACM Symposium on Cloud Computing*, pp. 119–130, 2010.

[19] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. *ACM International Conference on the World Wide Web*, pp. 131–140, 2007.

[20] Robert M. Bell and Yehuda Koren. Lessons from the netflix prize challenge. *ACM SIGKDD Explorations Newsletter* 9(2), pp. 75–79, 2007.

[21] James Bennett and Stan Lanning. The netflix prize. *Proceedings of KDD cup and workshop*, 2007.

[22] Dimitri Bertsekas and John Tsitsiklis. *Parallel and Distributed Computation.* Athena Scientific, 1997.

[23] Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A scripting language for large scale semistructured data analysis. *Proceedings of VLDB Endowment*, 2011.

[24] Christopher M Bishop. *Pattern recognition and machine learning*, Volume 4. Springer, 2006.

[25] Spyros Blanas, Jignesh M Patel, Vuk Ercegovac, Jun Rao, Eugene J Shekita, and Yuanyuan Tian. A Comparison of Join Algorithms for Log Processing in Mapreduce. *ACM SIGMOD International Conference on Management of Data*, pp. 975–986, 2010.

[26] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, YuanYaun Tian, Douglas Burdick, and Shivakumar Vaithyanathan. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *Proceedings of the VLDB Endowment*, 7(7), pp. 553–564, 2014.

[27] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. HyperANF: Approximating the neighbourhood function of very large graphs on a budget. *ACM International Conference on the World Wide Web*, pp. 625–634, 2011.

[28] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. *ACM International Conference on the World Wide Web*, pp. 595–602, 2004.

[29] Paolo Boldi and Sebastiano Vigna. In-core computation of geometric centralities with hyperball: A hundred billion nodes and beyond. *Data Mining Workshops, IEEE International Conference on Data Mining*, pp. 621–628, 2013.

[30] Paolo Boldi and Sebastiano Vigna. Axioms for Centrality. *Internet Mathematics*, 10(3-4), pp. 222–262, 2014.

[31] Phillip Bonacich. Power and Centrality: A Family of Measures. *American Journal of Sociology*, pp. 1170–1182, 1987.

[32] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. *IEEE International Conference on Data Engineering*, pp. 1151–1162, 2011.

[33] Vinayak Borkar, Michael J Carey, and Chen Li. Inside big data management: ogres, onions, or parfaits? *ACM International Conference on Extending Database Technology*, pp. 3–14, 2012.

[34] Leo Breiman. Bagging predictors. *Machine learning*, 24(2), pp. 123–140, 1996.

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

[36] Yingyi Bu, Vinayak Borkar, Jianfeng Jia, Michael J Carey, and Tyson Condie. Pregelix: Big (ger) graph analytics on a dataflow engine. *Proceedings of the VLDB Endowment*, 8(2), pp. 161–172, 2014.

[37] Prabir Burman. A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods. *Biometrika*, 76(3), pp. 503–514, 1989.

[38] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. *USENIX Symposium on Operating Systems Design and Implementation*, pp. 335–350, 2006.

[39] Meeyoung Cha, Hamed Haddadi, Fabrício Benevenuto, and P. Krishna Gummadi. Measuring user influence in twitter: The million follower fallacy. *AAAI International Conference on Weblogs and Social Media*, pp. 10–17, 2010.

[40] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2), pp. 1265–1276, 2008.

[41] Janani Chakkaradhari. Large Scale Centrality Measures in Apache Flink and Apache Giraph. 2014.

[42] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), 2008.

[43] Clueweb09 hyperlink graph, http://boston.lti.cs.cmu.edu/clueweb09/wiki/tiki-index.php?page=Web+Graph.

[44] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), pp. 377–387, 1970.

[45] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M Hellerstein, and Caleb Welton. Mad Skills: new analysis practices for big data. *Proceedings of the VLDB Endowment*, 2(2), pp. 1481–1492, 2009.

[46] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: scalable online collaborative filtering. *ACM International Conference on the World Wide Web*, pp. 271–280, 2007.

[47] James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, and Dasarathi Sampath. The youtube video recommendation system. *ACM International Conference on Recommender Systems*, pp. 293–296, 2010.

[48] Charles S Davis. The computer generation of multinomial random variates. *Computational Statistics & Data Analysis*, 16(2), pp. 205–217, 1993.

[49] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communication of the ACM*, 51, pp. 107–113, 2008.

[50] Jeffrey Dean and Sanjay Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1), pp. 72–77, 2010.

[51] David J DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), pp. 44–62, 1990.

[52] Sergey Dudoladov, Sebastian Schelter, Chen Xu, Asterios Katsifodimos, Stephan Ewen, Kostas Tzoumas, and Volker Markl. Optimistic Recovery for Iterative Dataflows in Action. *ACM SIGMOD International Conference on Management of Data (demo track)*, pp. 1439-1443, 2015.

[53] Ted Dunning. Accurate Methods for the Statistics of Surprise and Coincidence. *Computional Linguistics*, 19, pp. 61–74, 1993.

[54] Ted Dunning and Ellen Friedman. *Practical Machine Learning: Innovations in Recommendation.* O'Reilly, 2014.

[55] HealthCare.gov Sends Personal Data to Dozens of Tracking Websites, https://www.eff.org/de/deeplinks/2015/01/healthcare.gov-sends-personal-data.

[56] Bradley Efron and Robert Tibshirani. Improvements on cross-validation: the 632+ bootstrap method. *Journal of the American Statistical Association*, 92(438), pp. 548–560, 1997.

[57] Codewort Eikonal - der Albtraum der Bundesregierung, http://www.sueddeutsche.de/politik/geheimdienste-codewort-eikonal-der-albtraum-der-bundesregierung-1.2157432.

[58] Michael D. Ekstrand, Michael Ludwig, Joseph A. Konstan, and John T. Riedl. Rethinking the recommender research ecosystem: reproducibility, openness, and lenskit. *ACM Conference on Recommender Systems*, pp. 133–140, 2011.

[59] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3), pp. 375–408, 2002.

[60] Marius Eriksen. Your server as a function. *ACM Workshop on Programming Languages and Operating Systems*, pp 5, 2013.

[61] Stephan Ewen, Sebastian Schelter, Kostas Tzoumas, Daniel Warneke, and Volker Markl. Iterative parallel data processing with stratosphere: An inside look. *ACM SIGMOD International Conference on Management of Data (demo track)*, pp. pp. 1053-1056, 2013.

[62] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment*, 5(11), pages 1268–1279, 2012.

[63] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 124, pp. 5, 2004.

[64] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *International Conference on the Analysis of Algorithms*. 2007

[65] Iterative Computations in Flink, https://flink.apache.org/docs/0.7-incubating/iterations.html.

[66] Flixster, http://www.cs.sfu.ca/ sja25/personal/datasets/.

[67] Building a recommendation engine, foursquare style, http://engineering.foursquare.com/2011/03/22/building-a-recommendation-engine-foursquare-style.

[68] Simon Funk. Netflix update: Try this at home, http://sifter.org/ simon/journal/20061211.html, 2006.

[69] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. An overview of the system software of a parallel relational database machine grace. *Proceedings of the VLDB Endowment*, 86, pp. 209–219, 1986.

[70] Zeno Gantner, Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. Mymedialite: a free recommender system library. *ACM Conference on Recommender Systems*, pp. 305–308, 2011.

[71] Hector Garcia-Molina and Kenneth Salem. Sagas. *ACM SIGMOD International Conference on Management of Data*, pp. 249–259, 1987.

[72] Seymour Geisser. The predictive sample reuse method with applications. *Journal of the American Statistical Association*, 70(350), pp. 320–328, 1975.

[73] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 69–77, 2011.

[74] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. *ACM SIGMOD International Conference on Management of Data*, pp. 1197–1208, 2013.

[75] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37, pp. 29–43, 2003.

[76] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. SystemML: Declarative machine learning on MapReduce. *IEEE International Conference on Data Engineering*, pages 231–242, 2011.

[77] David Goldberg, David Nichols, Brian M Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35, pp. 61–70, 1992.

[78] Gene H Golub and Charles F Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[79] Michel Gondran and Michel Minoux. *Graphs, Dioids and Semirings - New Models and Algorithms*. Springer, 2008.

[80] Goetz Graefe. Volcano-an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), pp. 120–135, 1994.

[81] Raman Grover and Michael J Carey. Extending map-reduce for efficient predicate-based sampling. *IEEE International Conference on Data Engineering*, pp. 486–497, 2012.

[82] H20, http://0xdata.com.

[83] Alon Halevy, Peter Norvig, and Fernando Pereira. The Unreasonable Effectiveness of Data. *IEEE Intelligent Systems*, 24(2), pp. 8–12, 2009.

[84] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2), pp. 217–288, 2011.

[85] Trevor Hastie, Rahul Mazumder, Jason Lee, and Reza Zadeh. Matrix completion and low-rank svd via fast alternating least squares. *arXiv preprint arXiv:1410.2596*, 2014.

[86] Arvid Heise, Astrid Rheinländer, Marcus Leich, Ulf Leser, and Felix Naumann. Meteor/Sopremo: an extensible query language and operator model. *Workshop on End-to-end Management of Big Data*, 2012.

[87] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. *USENIX Symposium on Networked Systems Design and Implementation*, 11, pp. 22–22, 2011.

[88] Yifan Hu, Yehuda Koren, and Chris Volinsky. Collaborative Filtering for Implicit Feedback Datasets. *IEEE International Conference on Data Mining*, pp. 263–272, 2008.

[89] Botong Huang, Shivnath Babu, and Jun Yang. Cumulon: optimizing statistical data analysis in the cloud. *ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2013.

[90] Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and J Freytag. Peeking into the Optimization of Data Flow Programs with Mapreduce-style UDFs. *IEEE International Conference on Data Engineering, (demo track)*, pp. 1292–1295, 2013.

[91] Fabian Hueske, Mathias Peters, Matthias J Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the Black Boxes in Data Flow Optimization. *Proceedings of the VLDB Endowment*, 5(11), pp. 1256–1267, 2012.

[92] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41, pp. 59–72, 2007.

[93] Jing Jiang, Jie Lu, Guangquan Zhang, and Guodong Long. Scaling-up item-based collaborative filtering recommendation algorithm based on hadoop. *SERVICES '11*, pp. 490–497, 2011.

[94] Vasiliki Kalavri, Stephan Ewen, Kostas Tzoumas, Vladimir Vlassov, Volker Markl, and Seif Haridi. Asymmetry in large-scale graph analysis, explained. *Workshop on GRAph Data management Experiences and Systems*, pp. 1–7, 2014.

[95] Vasiliki Kalavri, Vladimir Vlassov, and Per Brand. Ponic: Using stratosphere to speed up pig analytics. *Euro-Par Parallel Processing*, pp. 279–290. Springer, 2013.

[96] Krishna Kamath, Aneesh Sharma, Dong Wang, and Zhijun Yin. RealGraph: User Interaction Prediction at Twitter. *User Engagement Optimization Workshop, ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2014.

[97] U Kang, Spiros Papadimitriou, Jimeng Sun, TJ Watson, and Hanghang Tong. Centralities in large networks: Algorithms and observations. *SIAM International Conference on Data Mining*, p. 119, 2011

[98] U Kang, Charalampos Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. HADI: Fast diameter estimation and mining in massive graphs with Hadoop Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2008.

[99] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. *IEEE International Conference on Data Mining*, pp. 229–238, 2009.

[100] Efthalia Karydi and Konstantinos G Margaritis. Parallel and distributed collaborative filtering: A survey. *arXiv preprint arXiv:1409.2762*, 2014.

[101] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1), pp. 39–43, 1953.

[102] Gary King. Restructuring the Social Sciences: Reflections from Harvard's Institute for Quantitative Social Science. *Political Science & Politics*, 47(01), pp. 165–172, 2014.

[103] Gary King, Jennifer Pan, and Margaret E Roberts. How censorship in china allows government criticism but silences collective expression. *American Political Science Review*, 107(02), pp. 326–343, 2013.

[104] Ariel Kleiner, Ameet Talwalkar, Purnamrita Sarkar, and Michael I Jordan. A scalable bootstrap for massive data. *Journal of the Royal Statistical Society: Series B*, 76(4), pp. 795–816, 2014.

[105] Yehuda Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM Transactions on Knowledge Discovery from Data*, 4(1), 2010.

[106] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42, pp. 30–37, 2009.

[107] Kornakapi, https://github.com/plista/kornakapi.

[108] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. Mlbase: A distributed machine-learning system. *Conference on Innovative Data Systems Research*, 2013.

[109] W.J. Krzanowski. Cross-validation in Principal Component Analysis. *Biometrics*, 1987.

[110] Jérôme Kunegis and Andreas Lommatzsch. Learning Spectral Graph Transformations for Link Prediction. *ACM International Conference on Machine Learning*, pp. 561–568, 2009.

[111] Jérôme Kunegis, Andreas Lommatzsch, and Christian Bauckhage. The slashdot zoo: mining a social network with negative edges. *ACM International Conference on the World Wide Web*, pp. 741–750, 2009.

[112] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale Graph Computation on just a PC. *Operating Systems Design and Implementation*, pp. 31–46, 2012

[113] Silvio Lattanzi and Vahab Mirrokni. Distributed Graph Algorithmics: Theory and Practice. *ACM International Conference on Web Search and Data Mining*, pp. 419–420, 2015.

[114] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. *Operating Systems Design and Implementation*, 2014.

[115] Jimmy Lin and Alek Kolcz. Large-scale Machine Learning at Twitter. *ACM SIGMOD International Conference on Management of Data*, pp. 793-804, 2012.

[116] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1), pp. 76–80, 2003.

[117] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed Graphlab: A framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment*, 5(8), pp. 716–727, 2012.

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

[119] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. *LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays*. Doctoral Dissertation. Harvard University, 2014.

[120] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. *ACM SIGMOD International Conference on Management of Data*, pp. 135–146, 2010.

[121] Volker Markl. Breaking the Chains: On Declarative Data Analysis and Data Independence in the Big Data Era. *Proceedings of the VLDB Endowment*, 7(13), pp. 1730–1733, 2014.

[122] John Markoff. Google cars drive themselves, in traffic. *New York Times*, 9, 2010.

[123] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what COST? *USENIX Workshop on Hot Topics in Operating Systems*, 2015.

[124] Frank McSherry, Derek G Murray, Rebecca Isaacs, and Michael Isard. Differential Dataflow. *Conference on Innovative Data Systems Research*, 2013.

[125] Scientific article recommendation with Mahout, http://www.slideshare.net/krisjack/scientific-article-recommendation-with-mahout.

[126] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in Apache Spark. *arXiv preprint arXiv:1505.06807*, 2015.

[127] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Graph structure in the web—revisited: a trick of the heavy tail. *ACM International Conference on the World Wide Web*, pp. 427–432, 2014.

[128] Svilen R. Mihaylov, Zachary G. Ives, and Sudipto Guha. Rex: Recursive, Delta-based Data-Centric Computation. *Proceedings of the VLDB Endowment*, 5(11), pp. 1280–1291, 2012.

[129] Movielens1M, http://www.grouplens.org/node/73.

[130] Seth A Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. Information network or social network?: The structure of the Twitter follow graph. *ACM International Conference on the World Wide Web*, pp. 493–498, 2014.

[131] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review*, 74(3), 2006.

[132] Mark Newman. *Networks: an Introduction.* Oxford University Press, 2010.

[133] The NSA's Secret Role in the U.S. Assassination Program, https://firstlook.org/theintercept/2014/02/10/the-nsas-secret-role/.

[134] NSA's Bulk Collection of Phone Records Is Illegal, Appeals Court Says, https://firstlook.org/theintercept/2015/05/07/appellate-court-rules-nsas-bulk-collection-phone-records-illegal/.

[135] NSA collecting phone records of millions of Verizon customers daily, http://www.theguardian.com/world/2013/jun/06/nsa-phone-records-verizon-court-order.

[136] Takashi Onoda, Gunnar Ratsch, and Klaus-R. Müller. Survey of Model Selection Criteria for Large Margin Classifiers. *Central Research Institute of Electric Power Research Report*, pp. 1-19, 2002.

[137] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. *ACM SIGMOD International Conference on Management of Data*, pp. 1099–1110, 2008.

[138] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. *Stanford InfoLab*, 1999.

[139] Christopher R Palmer, Phillip B Gibbons, and Christos Faloutsos. ANF: A fast and scalable tool for data mining in massive graphs. *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 81–90, 2002.

[140] Rong Pan, Yunhong Zhou, Bin Cao, Nathan Nan Liu, Rajan Lukose, Martin Scholz, and Qiang Yang. One-class collaborative filtering. *IEEE International Conference on Data Mining*, pp. 502–511, 2008.

[141] Biswanath Panda, Joshua S Herbach, Sugato Basu, and Roberto J Bayardo. Planet: massively parallel learning of tree ensembles with mapreduce. *Proceedings of the VLDB Endowment*, 2(2), pp. 1426–1437, 2009.

[142] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12, pp. 2825–2830, 2011.

[143] Kornakapi recommender platform by plista.

[144] PredictionIO, http://prediction.io.

[145] Benjamin Recht and Christopher Ré. Parallel stochastic gradient algorithms for large-scale matrix completion. *Mathematical Programming Computation*, 5(2), pp. 201–226, 2013.

[146] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. *Advances in Neural Information Processing Systems*, pp. 693–701, 2011.

[147] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. *Conference on Uncertainty in Artificial Intelligence*, pp. 452–461, 2009.

[148] Researchgate uses Mahout, http://s.apache.org/tkz.

[149] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: an open architecture for collaborative filtering of netnews. *ACM Conference on Computer Supported Cooperative Work*, pp. 175–186, 1994.

[150] Andreas Reuter and Friedemann Schwenkreis. Contracts - a low-level mechanism for building general-purpose workflow management-systems. *IEEE Data Engineering Bullettin*, 18(1), pp. 4–10, 1995.

[151] Francesco Ricci, Lior Rokach, Bracha Shapira, and Paul B. Kantor. *Recommender Systems Handbook*. Springer, 2011.

[152] Semih Salihoglu and Jennifer Widom. GPS: A graph processing system. *ACM International Conference on Scientific and Statistical Database Management*, p. 22, 2013.

[153] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *Proceedings of the VLDB Endowment*, 7(7), pp. 577–588, 2014.

[154] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. *ACM International Conference on the World Wide Web*, pp. 285–295, 2001.

[155] Joseph L Schafer. *Analysis of incomplete multivariate data*. CRC press, 1997.

[156] Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. All Roads Lead to Rome: Optimistic Recovery for Distributed Iterative Data Processing. *ACM Conference on Information and Knowledge Management*, pp. 1919–1928, 2013.

[157] Sebastian Schelter, Venu Satuluri, and Reza Zadeh. Factorbird - a Parameter Server Approach to Distributed Matrix Factorization. *Distributed Machine Learning and Matrix Computations workshop, Advances in Neural Information Processing Systems*, 2014.

[158] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. *ACM SIGMOD International Conference on Management of Data*, pp. 23–34, 1979.

[159] Jack Shemer and Phil Neches. The genesis of a database computer. *Computer*, 17(11), pp. 42–56, 1984.

[160] U.S. Government Designated Prominent Al Jazeera Journalist as 'Member of Al Qaeda', https://firstlook.org/theintercept/2015/05/08/u-s-government-designated-prominent-al-jazeera-journalist-al-qaeda-member-put-watch-list/.

[161] SlashdotZoo, http://konect.uni-koblenz.de/networks/slashdot-zoo.

[162] Jason R Smith, Herve Saint-Amand, Magdalena Plamada, Philipp Koehn, Chris Callison-Burch, and Adam Lopez. Dirt cheap web-scale parallel text from the common crawl. *Association for Computional Linguistics*, pp. 1374–1383, 2013.

[163] Evan R Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph Gonzalez, Michael J Franklin, Michael I Jordan, and Tim Kraska. MLI: An API for Distributed Machine Learning. *IEEE International Conference on Data Mining'13*, pp. 1187–1192, 2013.

[164] Ellen Spertus, Mehran Sahami, and Orkut Buyukkokten. Evaluating similarity measures: a large-scale study in the orkut social network. *ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pp. 678–684, 2005.

[165] Michael Stonebraker, Daniel Abadi, David J DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and Parallel DBMSs: friends or foes? *Communications of the ACM*, 53(1), pp. 64–71, 2010.

[166] RDevelopment Core Team et al. R: A language and environment for statistical computing. *R foundation for Statistical Computing*, 2005.

[167] Christina Teflioudi, Faraz Makari, and Rainer Gemulla. Distributed matrix completion. *IEEE International Conference on Data Mining*, pp. 655–664, 2012.

[168] GCHQ taps fibre-optic cables for secret access to world's communications, http://www.theguardian.com/uk/2013/jun/21/gchq-cables-secret-world-communications-nsa.

[169] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2), pp. 1626–1629, 2009.

[170] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. *IEEE International Conference on Data Engineering*, pp. 996–1005, 2010.

[171] Trackography - Meet the Trackers, https://myshadow.org/trackography-meet-the-trackers.

[172] TwitterICWSM, http://twitter.mpi-sws.org/data-icwsm2010.htm.

[173] Madeleine Udell, Corinne Horn, Reza Zadeh, and Stephen Boyd. Generalized low rank models. *arXiv preprint arXiv:1410.0342*, 2014.

[174] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), pp. 103–111, 1990.

[175] Daniel Warneke and Odej Kao. Nephele: efficient parallel data processing in the cloud. *ACM Workshop on many-task computing on grids and supercomputers*, p. 8, 2009.

[176] Webbase2001, http://law.di.unimi.it/webdata/webbase-2001/.

[177] Markus Weimer, Tyson Condie, Raghu Ramakrishnan, et al. Machine Learning in Scalops, a higher order cloud computing language. *Biglearn Workshop on Parallel and Large-scale Machine Learning, Advances in Neural Information Processing Systems*, 9, pp. 389–396, 2011.

[178] Markus Weimer, Alexandros Karatzoglou, Quoc Viet Le, and Alex Smola. Maximum margin matrix factorization for collaborative ranking. *Advances in Neural Information Processing Systems*, 2007.

[179] Tom White. Hadoop–the definite guide, 2009.

[180] Reynold S Xin, Daniel Crankshaw, Ankur Dave, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. *arXiv preprint arXiv:1402.2394*, 2014.

[181] YahooMusic, http://webscope.sandbox.yahoo.com/catalog.php?datatype=r.

[182] R2 - Yahoo! Music User Ratings of Songs with Artist, Album, and Genre Meta Information, v. 1.0.

[183] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *USENIX Symposium on Operating Systems Design and Implementation*, 8, pp. 1–14, 2008.

[184] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, SVN Vishwanathan, and Inderjit Dhillon. NOMAD: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *Proceedings of the VLDB Endowment*, 7(11), pp. 975–986, 2014.

[185] Reza Bosagh Zadeh and Gunnar Carlsson. Dimension independent matrix square using mapreduce. *arXiv preprint arXiv:1304.1467*, 2013.

[186] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *USENIX Conference on Networked Systems Design and Implementation*, pp. 2–2., 2012.

[187] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *USENIX conference on Hot topics in cloud computing*, pp. 10–10, 2010.

[188] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. *USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, 2012.

[189] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. *Algorithmic Aspects in Information and Management*, pages 337–348, Springer 2008.