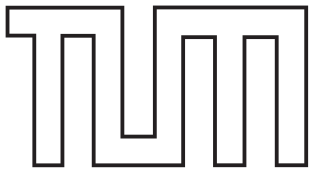
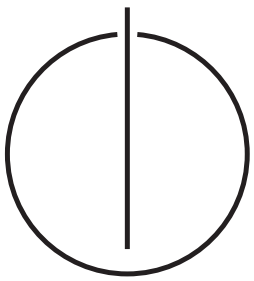


TECHNISCHE UNIVERSITÄT MÜNCHEN

LOAD-BALANCED MASSIVELY PARALLEL DISTRIBUTED DATA EXPLORATION

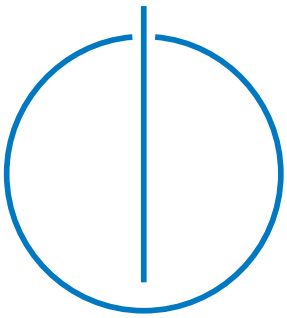
DIPLOM-INFORMATIKER UNIV. BENJAMIN GUFLER



TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK
LEHRSTUHL III — DATENBANKSYSTEME

LOAD-BALANCED MASSIVELY PARALLEL DISTRIBUTED DATA EXPLORATION

DIPLOM-INFORMATIKER UNIV. BENJAMIN GUFLER



Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Rüdiger Westermann

Prüfer der Dissertation:

1. Univ.-Prof. Alfons Kemper, Ph.D.
2. Prof. Dr. Johann Gamper, Freie Universität Bozen / Italien

Die Dissertation wurde am 27.09.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.02.2013 angenommen.

ABSTRACT

The volume of readily available data sets is growing at exponential rates. Besides large internet companies like Google, Microsoft or Facebook collecting, e. g., user data, this trend also applies to scientific data sets like observations or simulations. In both industry and science, there is a high interest in analysing these large data sets quickly. Observing the exponential data growth rates, centralised analysis will no longer be able to accommodate interactive data exploration. Distributed processing is the method of choice in such scenarios, as it allows to exploit the massive amount of corporate main memory and processing resources available in compute clusters.

Scientific data sets are often hierarchically structured. In contrast to flat data with tuples of roughly identical sizes, hierarchical structures may cause significant skew in the size of data elements. This aspect needs to be taken into account when exploring hierarchically structured data sets in a distributed manner in order to balance the workload reasonably well.

The MapReduce programming model gained much attention in the database community over the last years for offering a simple method of exploiting massive parallelism. In this thesis, we analyse the applicability of MapReduce style processing to scientific data exploration on tree structured data. Employing frequent subtree mining as a sample application, we propose the Pipelined MapReduce framework. This framework extends standard MapReduce in order to better support scientific data analysis. These extensions cover both large scale parallelisation in cluster environments, and small scale parallelisation on each of the cluster nodes.

We design TreeLatin, a high-level scripting language which allows to construct workflows for the Pipelined MapReduce framework in a user friendly manner. Similar to relational database management systems, we integrate an optimiser component in order to improve the efficiency of the workflows we compile TreeLatin scripts into.

The performance of MapReduce style systems strongly depends on a uniform data distribution to the cluster nodes. The inherent skew of many scientific data sets causes load imbalances which raise the processing time significantly. This imbalance is even amplified by the high runtime complexities of the scientific data exploration tasks. We propose an adaptive load balancing strategy for tackling this problem. A distributed monitoring component detects skewed data distributions. Taking into account the runtime complexity of the processing tasks, we distribute the data to the cluster nodes such that the workload is well balanced. Thereby, we reduce the time to complete the processing tasks.

ZUSAMMENFASSUNG

Die Menge weltweit vorgehaltener Daten steigt exponentiell an. Neben der Affinität großer Internet-Unternehmen wie Google, Microsoft und Facebook zur Sammlung von Daten betrifft dieser Trend auch wissenschaftliche Anwendungsbereiche. Die Möglichkeit, diese Datensets schnell analysieren zu können, ist dabei in beiden Gebieten essentiell. Zentralisierte Ansätze interaktiver Datenexploration sind dabei aufgrund der exponentiell wachsenden Datenmenge nicht länger plausibel. Stattdessen werden verteilte Verarbeitungsmethoden nötig, die die Ausnutzung der massiven Hauptspeicher- und Rechenressourcen von Rechnerverbünden erlauben.

Häufig sind wissenschaftliche Datensets hierarchisch strukturiert. Im Gegensatz zu „flachen“ Daten, deren Tupel annähernd gleich groß sind, können durch die hierarchische Struktur massive Größenunterschiede auftreten. Gerade bei verteilter Verarbeitung muss dieser Aspekt mit berücksichtigt werden, um die entstehende Arbeitslast möglichst gleichmäßig auf die einzelnen Rechner zu verteilen.

In den letzten Jahren erregte das MapReduce-Modell für massiv datenparallele Verarbeitung großes Aufsehen im Datenbank-Umfeld. Im Rahmen dieser Arbeit analysieren wir die Einsatzmöglichkeiten von MapReduce zur Exploration hierarchisch strukturierter wissenschaftlicher Daten. Anhand der Suche nach häufigen Teilbäumen als Beispiel einer wissenschaftlichen Anwendung schlagen wir das *Pipelined MapReduce Framework* als Erweiterung von MapReduce vor. Unsere Ergänzungen, die sowohl verteilte Verarbeitung in Rechnerverbünden als auch Parallelisierung auf einzelnen Rechnern abdecken, ermöglichen effizientere wissenschaftliche Datenanalysen.

Um Arbeitsabläufe für das Pipelined MapReduce Framework möglichst anwenderfreundlich erstellen zu können, entwerfen wir die Programmiersprache *TreeLatin*. Im Rahmen der Übersetzung von TreeLatin-Skripten in Arbeitsabläufe des zugrundeliegenden Frameworks wenden wir, ähnlich zu relationalen Datenbanksystemen, Optimierungen an, um einen möglichst effizienten Ausführungsplan zu erhalten.

Die Performance MapReduce-artiger Systeme hängt stark von einer gleichmäßigen Datenverteilung ab. Wissenschaftliche Datensets weisen jedoch häufig deutliche Schräglagen auf. Der negative Einfluss solcher Schräglagen auf die Verarbeitungsdauer wird durch die hohe Laufzeitkomplexität wissenschaftlicher Analyseanwendungen noch verstärkt. Wir entwerfen einen adaptiven Lastbalancierungsansatz zur Lösung dieses Problems. Daten-Schräglagen werden von einer verteilten Monitoring-Komponente erkannt und unter Miteinbeziehung der Laufzeitkomplexität der Anwendung durch eine geeignete Verteilung der Daten auf die einzelnen Rechner ausgeglichen, um die Verarbeitungszeit zu reduzieren.

ACKNOWLEDGMENTS

This thesis would not have been possible without the support of many people. I would like to take this opportunity and express my sincere gratitude to them.

I thank my advisor, Prof. Alfons Kemper, Ph.D., for giving me the opportunity to perform this work under his guidance, for his support and patience.

I owe sincere thankfulness to Dr. Angelika Reiser. The regular discussions we had often helped to keep my work focused on the essential aspects.

The work on the load balancing aspect was performed in collaboration with Dr. Nikolaus Augsten (Free University Bozen). I am thankful for his support and for the many fruitful discussions we had.

I thank the German Astrophysical Virtual Observatory (GAVO) project, especially Dr. Gerard Lemson, for providing me with insights on the Millennium simulation. Moreover, I thank GAVO and Prof. Alex Szalay, Ph.D., for the opportunity to participate in the International Virtual observatory Alliance.

Finally, I thank the members of the database group at TU München which shared part of way working on this thesis: Prof. Dr. Thomas Neumann, Prof. Dr. Torsten Grust, Martina Albutiu, Dr. Stefan Aulbach, Veneta Dobreva, Florian Funke, Dr. Daniel Gmach, Andrey Gubichev, Sebastian Hagen, Dr. Stefan Krompass, Dr. Richard Kuntschke, Alexander Lechner, Manuel Mayr, Henrik Mühe, Fabian Prasser, Dr. Jan Rittinger, Dr. Tobias Scholl, Dr. Andreas Scholz, Michael Seibold, Jessica Smejkal, Dr. Jens Teubner, Bernd Vögele and Dr. Martin Wimmer.

CONTENTS

1 INTRODUCTION 1
  1.1 E-Science 1
    1.1.1 Massive, Readily Available Data Volumes 2
    1.1.2 Complex Data Structures 4
    1.1.3 Complex Data Analysis Algorithms 5
  1.2 A Case Study: Astronomy 6
  1.3 Contributions and Outline 9
2 FREQUENT SUBTREE MINING 11
  2.1 Introduction 11
    2.1.1 Subtree Types 12
  2.2 Principles of Frequent Subtree Mining 15
    2.2.1 Candidate Generation 16
    2.2.2 Support Measures 18
  2.3 Distributing Frequent Subtree Mining 19
    2.3.1 Challenges 20
    2.3.2 A Distributed Tree Mining Workflow 21
  2.4 Sample Application Scenarios 23
    2.4.1 Sample Algorithms 24
    2.4.2 Astrophysical Applications 25
  2.5 Related Work 26
3 LARGE-SCALE PARALLELISATION 29
  3.1 Introduction 29
  3.2 Distributing the Data 29
    3.2.1 Full Replication 31
    3.2.2 Distributing by Attribute Value 31
    3.2.3 Random Tree Partitioning 32
  3.3 Massive Parallel Data Processing 33
  3.4 Frequent Subtree Mining with MapReduce 35
    3.4.1 Straight-Forward Approaches 37
    3.4.2 Partitioning Data on the Fly 38
    3.4.3 2-Step Mining with Persistent Partitions 39
    3.4.4 Discussion 41
  3.5 Operator Library 42

3.6 Pipelined MapReduce: E-Science Extensions to MapReduce 44
3.6.1 Multi-Step Jobs 44
3.6.2 Multi-Input and Multi-Output Tasks 45
3.6.3 Data Streaming 46
3.7 Towards Lower Communication and Synchronisation Overhead... 47
3.7.1 Reducing Communication Overhead 48
3.7.2 Reducing Synchronisation Overhead 49
3.8 Experimental Evaluation 49
3.8.1 Scaling the Data Set 51
3.8.2 Scaling the Cluster Size 52
3.8.3 Probabilistic Frequent Label Detection 53
3.9 Related Work 54
4 SMALL-SCALE PARALLELISATION 57
4.1 Introduction 57
4.2 Multi-Core Processors 57
4.3 Inter-Operator Parallelism 59
4.4 Cache Friendly Data Structures 62
4.5 Related Work 63
5 TREELATIN: A SCRIPTING APPROACH TO (DISTRIBUTED) TREE PROCESSING 67
5.1 Introduction 67
5.2 Pig Latin 67
5.3 Representing Tree Data with Pigs Data Model 71
5.3.1 Nested Tree Data Representation 71
5.3.2 Schema Modifications 73
5.4 Tree Processing Statements 73
5.4.1 Constructing Trees 74
5.4.2 Flattening Trees 79
5.4.3 Processing Trees 79
5.5 Multithreading Blocks 83
5.6 Compiling TreeLatin Scripts to MapReduce Execution Plans 86
5.6.1 Optimiser Overview 87
5.6.2 Group Refinement 87
5.6.3 Local Multithreading 90
5.6.4 Bilateral Projection Push-Down 91
5.7 Frequent Subtree Mining With TreeLatin 91
5.8 Experimental Evaluation 93

5.9 Related Work 94
6 HANDLING DATA SKEW 97
6.1 Introduction 97
6.2 Data Skew in MapReduce 98
6.3 The Partition Cost Model 99
6.3.1 Current Situation 100
6.3.2 Optimal Solution 101
6.3.3 Approximate Cost Estimation 104
6.4 TopCluster: A Distributed Monitoring Approach 104
6.4.1 Histogram Approximation Error 105
6.4.2 Uniformity-Based Monitoring 105
6.4.3 TopCluster Monitoring 107
6.4.4 TopCluster Approximation Guarantees 113
6.5 Extensions to TopCluster 115
6.5.1 Adaptive Local Thresholds 115
6.5.2 Approximate Local Histograms 116
6.5.3 Going Beyond Tuple Count 119
6.6 Load Balancing 120
6.6.1 Fine Partitioning 120
6.6.2 Dynamic Fragmentation 122
6.6.3 Incremental Assignment Calculation 124
6.7 Handling Large Clusters 125
6.8 Experimental Evaluation 127
6.8.1 Histogram Approximation Error 127
6.8.2 TopCluster: Approximation Quality vs. Head Size 129
6.8.3 Cost Estimation Error 132
6.8.4 Replication Overhead of Dynamic Fragmentation 132
6.8.5 Influence on Applications 134
6.9 Related Work 136
7 CONCLUSION AND OUTLOOK 139

BIBLIOGRAPHY 142

LIST OF FIGURES

Figure 1.1 Visualisation: Millennium Simulation [Springel et al. 2005] 3
Figure 1.2 Sample Trees from the Millennium Simulation 5

Figure 2.1 Sample Forest and Subtrees 13
Figure 2.2 Generating Candidate Subtrees: Pattern Growth 17
Figure 2.3 Generating Candidate Subtrees: À-Priori 17
Figure 2.4 Distributed Tree Mining Workflow 22
Figure 2.5 Distributed Path Join 25

Figure 3.1 Distributing the Data: Sample Trees 31
Figure 3.2 MapReduce Architecture 33
Figure 3.3 Symbols Used in the Workflow Realisation Figures 36
Figure 3.4 Straight-Forward Workflow Realisations 36
Figure 3.5 On-the-Fly Data Partitioning 38
Figure 3.6 2-Step Mining with Persistent Partitions 40
Figure 3.7 Modules of the Straight-Forward Workflow Realisation in Figure 3.4a 43
Figure 3.8 Pipelined MapReduce Workflow 44
Figure 3.9 Scaling the Data Set on a Cluster with 16 Hosts 52
Figure 3.10 Scaling the Cluster Size for the 8 GB Data Set 53
Figure 3.11 Approximation Quality of the Probabilistic Frequent Label Detection for the 16 GB Data Set on 16 Hosts 53
Figure 3.12 Execution Time of the Frequent Label Detection for the 16 GB Data Set on 16 Hosts 54

Figure 4.1 Multithreading Wrapper for Pipelined MapReduce 61
Figure 4.2 Operator Wrapper Details 62

Figure 5.1 Sample Trees 72
Figure 5.2 Tree Traversal Operator 82
Figure 5.3 Finding the Best Customers with Pig Latin 84
Figure 5.4 Finding the Best Customers with TreeLatin 85
Figure 5.5 Scaling the Data Set on a Cluster with 16 Hosts 94
Figure 5.6 Scaling the Cluster Size for the 8 GB Data Set 94

Figure 6.1 Partition Assignment in Current MapReduce Systems 100
Figure 6.2 Local and Exact Global Histograms. 103
Figure 6.3 Head of Local Histograms for $\tau_i = 14$. 109
Figure 6.4 Global Bounds 110
Figure 6.5 Histogram Aggregation for $\varepsilon = 10\%$ 117
Figure 6.6 Partitioned Data Distribution 122
Figure 6.7 Fragmented Data Distribution 123
Figure 6.8 Approximation Error for Varying Skew 128
Figure 6.9 Approximation Error for Varying ε 130
Figure 6.10 Histogram Head Size for Varying ε 131
Figure 6.11 Cost Estimation Error 132
Figure 6.12 Replication Overhead of Dynamic Fragmentation with p = 4r 133
Figure 6.13 Execution Time Reduction 135

LIST OF TABLES

Table 1.1 Sizes of Modern E-Science Data Sets 3

Table 3.1 Tree Distribution by Label 32
Table 3.2 Plain MapReduce Architecture Variant Overview 41
Table 3.3 Data Sets Used in the Evaluation 51

Table 4.1 Cache Structure of Modern Server Processors 59

Table 5.1 Tracked Schema Information for Algorithm 5.2 70
Table 5.2 Classification of TreeLatin Operators 88

1 INTRODUCTION

1.1 E-SCIENCE

Modern scientific research is based on large, comprehensive and often complex data sets. These data sets are typically available to entire research communities, or to the general public, on-line. Basing analyses on this kind of data sets has several advantageous effects on the complete scientific processing workflow from data discovery down to the incurred analysis results.

READILY AVAILABLE DATA SETS Relevant data sets are readily available. Scientists do not need to collect the base data for their experiments and analyses by themselves or request relevant data sets from other institutes through complicated, off-line processes. Thereby, they can save time which allows them to verify and publish their results faster than before.

LARGE DATA SETS Many of the publicly available scientific data sets are very comprehensive. By basing their analyses on large base data, scientists can effectively reduce the risk of obtaining results which apply only to a few special situations which are accidentally covered by a small base data set.

WELL-KNOWN DATA SETS Basing analyses on data sets which are available to an entire research community also increases the trust in the results obtained, as the base data is obviously not tailored to fit the desired result.

REPLICATED DATA SETS Eventually, these data sets are typically mirrored by several research institutes around the globe. This reduces the risk of data loss.

The term *e-science* was coined to describe this new type of science, where new knowledge is extracted from, or verified by, analysing massive, comprehensive, and complex data sets.

E-science also introduces several new challenges which need to be faced before the advantages described above can be fully exploited. Often, explorative data analysis methods play a central role in both identifying and analysing interesting aspects of the data sets. However, the size and complexity of the data sets and the complexity of

the analysis methods render providing the infrastructure required for interactive data exploration a challenging task. We will describe these three challenges in more detail in the following, and present approaches for tackling them throughout this thesis. In particular, we will emphasise that a centralised infrastructure will typically not be able to satisfy the constraints imposed by e-science environments.

1.1.1 *Massive, Readily Available Data Volumes*

Modern scientific research is based on, or supported by, the (often explorative) analysis of data sets collected through, e. g., experiments, observations, or simulations. As the analysis and evaluation methods improve over time and allow for continuously better and more precise results, the quality of the base data underlying the scientific processing becomes more and more important. Gathering base data which is sufficiently accurate to allow for meaningful analytical results, thereby, becomes a challenging and expensive task on its own. High costs are caused by the need for modern, highly precise, and powerful equipment like measurement instruments and data preprocessing devices on the one hand, and by the experts required to operate this equipment on the other hand. It becomes thus infeasible for every scientist to collect the base data required for her research project on her own. Instead, in modern e-science, the data acquisition and analysis steps are separated, and only a few institutes focus on the former.

With only a limited number of research institutes responsible for data acquisition and curation, the available human and financial resources can be dedicated to it more precisely and with greater benefit, e. g., by allowing larger data collectors to be built or larger experiments generating data to be performed. This obviously increases the amount of data available [Szalay and Gray 2006]. Thanks to the rapidly dropping costs of storage media over the last years, it is possible to preserve all this data. There is no necessity to store just aggregated data, or reduce the precision just for the sake of reducing the storage requirements. This trend is clearly reflected by the sizes of modern e-science data sets, as reported in Table 1.1 exemplary for the Astrophysics domain.

Moreover, with data curation expertise concentrated in a limited number of data centres, data cleansing and preprocessing is possible to a much higher extent and with higher precision than in a scenario where every single scientist has to care for

1 The numbers refer to data release 8, the largest and most recent version available at the time of writing this thesis.

2 The Pan-STARRS project is still running. The data volume is estimated.

Data Set	Publication Year	Size
Millennium Simulation (Figure 1.1)	2005	25 TB
Millennium II Simulation	2009	27 TB
Millennium XXL Simulation	2011	100 TB
Sloan Digital Sky Survey (SDSS)[1]	2011	49.6 TB
Pan-STARRS[2]	still running	100 TB

Table 1.1: Sizes of Modern E-Science Data Sets

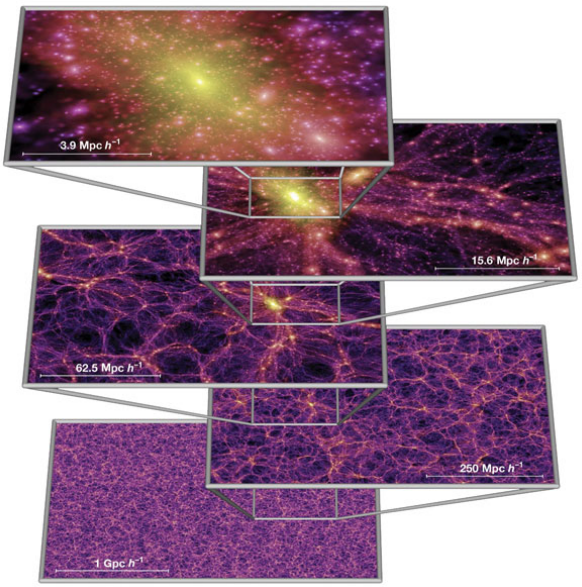


Figure 1.1: Visualisation: Millennium Simulation [Springel et al. 2005]

his own, small data set. Therefore, not just the amount of data, but also the quality of the available data sets increases.

The growing size and improving precision of the data sets impacts on the data analyses. Existing tools for scientific processing designed for smaller data sets often face a severe slow-down when the data sets grow larger and larger. In order to complete the analyses in reasonable time, more computational resources are required. The improved precision of the base data sets allows for more exact analyses, which again requires more computational resources.

Due to this increased demand of computational resources, scientific processing tools running on one single host are hitting their limits. In order to cope with the size and complexity of current and upcoming e-science data sets, a distributed infrastructure is required for performing analyses within reasonable time and with reasonable precision. In this thesis, we will devise a distributed framework allowing to run scientific data analyses in a massively parallel manner.

1.1.2 *Complex Data Structures*

Besides the sheer volume of scientific data sets, their structure is a second key aspect. Many data sets do not consist of "flat" relational data, but have an inherent tree or graph structure. This structure implicitly represents additional information like dependencies, containment, or a sort order. In scientific processing, this structure must be taken into account in order to ensure correct processing results. Often, one must process all the nodes and edges of a compound together in order to resolve the structure and honour this implicit information properly. If the number of nodes per compound varies, so will the processing time required for the single compounds. This raises the need for workload balancing strategies. Only with a well-balanced workload, the distributed resources can be fully utilised, and only a good utilisation of the infrastructure permits for a significant speed-up of the distributed processing as compared to a non-distributed variant.

Examples for structured e-science data sets are the protein database swissprot[3], DBLP[4] and, once more, the Millennium simulation. The latter exhibits a tree structure which represents temporal order. In describing the evolution of the universe over time, child nodes precede their parent in time. Thereby, the structure symbolises the influence of mass attraction over time. Due to simulation settings, the trees grow up

3 http://www.uniprot.org
4 http://dblp.uni-trier.de

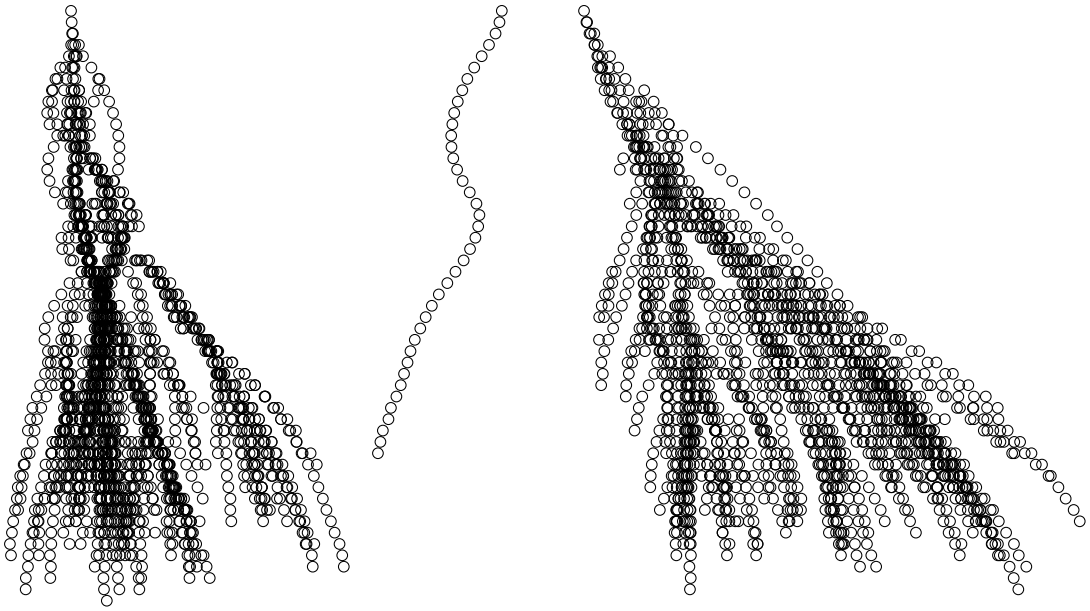


Figure 1.2: Sample Trees from the Millennium Simulation

to a height of 64 nodes. The largest tree in the simulation consists of almost 100 000 nodes, while on average, trees are composed of less than 50 nodes.

Example 1.1. *Figure 1.2 shows three trees from the Millennium simulation[5]. The left and right ones are rather complex and consist of about 1 800 nodes each. The middle tree, instead, is quite simple and consists of only 40 nodes. Assume a simple processing task consisting of a single pass over the nodes of a tree. Processing one of the large trees will require 45 times as much time as processing the small tree, even though each tree represents a single, "atomic" unit of processing.*

1.1.3 *Complex Data Analysis Algorithms*

Not only scientific data sets tend to be complex. Many scientific data analysis applications have a high – often superlinear – runtime complexity. This high complexity further complicates the task of balancing the workload in a distributed processing

5 Note that the Millennium trees are four-dimensional, with three spatial and one temporal dimension. The temporal dimension defines the parent-child relationship between the nodes. The plot uses the temporal dimension and one of the spatial dimensions.

environment. It is no longer sufficient to evenly distribute the data volume. Instead, the complexity of the algorithm must be taken into account.

Example 1.2. *Consider again the trees in Figure 1.2 and an algorithm with quadratic runtime complexity, e. g., comparing every pair of nodes within a tree to each other. We associate a cost of $40^2 = 1\,600$ with the simple tree in the middle, while the cost for each of the complex trees is $1\,800^2 = 3\,240\,000$. The cost hence differs by a factor of 2 025, as opposed to the factor of 45 with the linear processing algorithm of Example 1.1.*

These cost aspects must be taken into account in order to obtain a well-balanced workload in a distributed processing environment. Without reasonable load balancing, a distributed analysis tool can easily run as slow as a centralised variant, or even slower.

Efficiency is an essential aspect in e-science applications. The massive and complex data sets must still be processed in reasonable time. However, many e-science applications are developed by scientists of the respective domain which are rarely experienced in realising distributed applications. Therefore, the technical aspects of distribution must be handled by an underlying framework allowing the application developers to focus on the scientific aspects of their applications.

1.2 A CASE STUDY: ASTRONOMY

As a pioneering scientific discipline, astronomy puts a strong focus on supporting e-science. Due to this pioneering role and the free availability of astronomical data sets to the general public, we focus on astronomy in this section. Moreover, we will use astronomical data sets in experimental evaluations throughout this thesis.

Astronomy bundles its e-science support in the *Virtual Observatory*. In a globally joint effort, the *International Virtual Observatory Alliance (IVOA)*[6] was formed in 2002. Its goal is to

> facilitate the international coordination and collaboration necessary for the development and deployment of the tools, systems and organizational structures necessary to enable the international utilization of astronomical archives as an integrated and interoperating virtual observatory [IVOA 2010].

The IVOA is composed of national Virtual Observatory projects which collaboratively aim at standardising ways for representing, querying and transferring astronomical data sets. By 2011, 19 national virtual observatories from around the

6 http://www.ivoa.net

globe were participating in this joint effort. The Virtual Observatory infrastructure the IVOA is building consists of

- standardised data discovery mechanisms, data models, exchange protocols and data formats,
- data centres and institutions publishing their data according to these standards, and
- tools exploiting these standards in order to allow for easy integration of data from multiple sources.

In order to provide a solid foundation for this infrastructure, the IVOA defines standards for data formats and data access methods. The data formats include both data models (e. g., the *Simulation Data Model* [Lemson et al. 2011]), and data exchange formats (*VOTable* [Ochsenbein and Williams 2009]). The standards for data access range from data discovery (the *IVOA Registry* [Benson et al. 2009]) to data retrieval languages (the *Astronomical Data Query Language* [Ortiz et al. 2008]) and protocols (e. g., the *Table Access Protocol* [Dowler et al. 2010], *Simple Cone Search* [Williams et al. 2008], *Simple Image Access Protocol* [Tody and Plante 2009]). Some of the largest available data sets from both observational and theoretical astronomy and astrophysics, the aforementioned Sloan Digital Sky Survey and the Millennium Simulation are already published according to these standards. Upcoming data sets like Pan-STARRS[7] and the Millennium XXL simulation[8] will be available through VO compatible interfaces right from the beginning.

Virtual observatory-enabled tools like VODesktop [VODesktop 2010], Aladin [Aladin 2010] and TOPCAT [TOPCAT 2011] implement these standards. Thereby, they allow scientists to work with data published to the Virtual Observatory in a similar fashion as they are already used to work with data sets stored locally on their workstations. At the time of writing this thesis, the underlying workflow is typically as follows. Data sources relevant for a scientist's current research are identified via meta-data lookups in a Virtual Observatory Resource Registry, e. g., by searching for spacial coverage and interesting frequency bands. The identified data sources are then queried for relevant data, e. g., by sending a query formulated in the Astronomical Data Query Language over the Table Access Protocol. The resulting data sets are transferred, in VOTable format, either to the scientist's workstation, or to a personal remote storage location accessible through the *VOSpace* [Graham et al. 2010] protocol.

7 http://pan-starrs.ifa.hawaii.edu/public

8 http://www.mpa-garching.mpg.de/mpa/research/current_research/hl2011-9/hl2011-9-en.html

These data sets can then be combined with each other, and with other, locally available data, in order to perform scientific research. Thereby, the Virtual Observatory becomes a reliable and valuable source for scientific data sets, providing scientists with a simple way of finding and retrieving data relevant for their research. With more and more data sets published according to these standards, the value of the Virtual Observatory will rise even further.

With ever more and larger data sets available, however, running comprehensive experiments and analyses becomes a challenging task on its own. Performing all the processing on the scientist's workstation, as before with smaller data sets, is no longer feasible. The IVOA acknowledges this issue and defines the *Universal Worker Service* interface [Harrison and Rixon 2010] for remote job execution. Through this interface, well-known data analysis applications can be executed on remote sites which may provide more processing power than a scientist's local workstation does. The Universal Worker Service is similar to Grid-based remote application execution [Anjomshoaa et al. 2008] in that a processing task is scheduled on a remote system and processed asynchronously. In contrast to the very generic Grid job execution, the Universal Worker Service is tightly integrated with other Virtual Observatory components like the *IVOA authentication and authorisation mechanisms* [Rixon 2005] and VOSpace. The strongest restriction as compared to Grid-based jobs, however, is that the Universal Worker Service is limited to well-known scientific applications pre-installed on the worker nodes. There is no possibility of passing user-specific applications for execution. Only the application parameters of the pre-installed applications may be set. For scientists relying on non-standard analysis tools, e. g., applications developed by themselves, or more recent versions of a tool than the one offered by a Universal Worker Service site provider, this standard is, therefore, not sufficiently flexible. Moreover, distribution and parallelisation aspects are not treated in the Universal Worker Service. This makes it difficult to control applications running on top of the Universal Worker service in a distributed environment.

With growing data volumes, scalability (by means of parallelisation, distributed computing and exploitation of multi-core processors) becomes a central aspect of scientific processing. Applications need to be scalable up to hundreds or even thousands of processors in order to keep up with the processed data volumes while still providing reasonable response times. Developing a parallel application is a highly non-trivial task comprising communication, synchronisation, execution control, and load balancing aspects. A scientist developing a new research tool should be able to focus on the actual scientific aspect of his new application, and not be bothered with these technical aspects of distributed execution.

1.3 CONTRIBUTIONS AND OUTLINE

In this thesis, we propose a framework for scientific data processing which provides out-of-the-box scale-out and scale-up and a comfortable user interface abstracting from the technical aspects of distributed and parallel computing.

Throughout this thesis, we use frequent subtree mining as a sample application. Frequent subtree mining is an interesting data analysis application in many scientific disciplines. In theoretical astrophysics, e. g., it can be employed to find recurring evolution patterns in data sets like the Millennium simulation. In biology, frequent subtree mining can be used to analyse protein data sets like Swissprot, and in Phylogeny [Shasha et al. 2004]. We will describe frequent subtree mining in detail in Chapter 2, and discuss ideas for efficiently parallelising this task in a distributed environment.

The distributed execution framework we propose for scalable scientific data analysis is based on Google's MapReduce [Dean and Ghemawat 2008]. We introduce the *Pipelined MapReduce Framework* and describe the modifications to standard MapReduce we retain necessary for efficient scientific data analysis and for processing tree shaped data in Chapter 3.

Recent processors integrate multiple processing cores on a single chip. This trend of producing so-called *multi-core CPUs*, or *CMPs* (chip multi-processors) started with dual-core processors, and continues to evolve, increasing the number of cores per chip. Current commodity CPUs integrate up to 16 cores on a single chip. In contrast to the single-core processor speed-up according to Moore's Law [Moore 1965] over the past decades, exploiting this new kind of increased processing power requires modifications to the applications. We analyse to what extent multi-core processors can be exploited automatically for further speeding up scientific applications based on our distributed execution framework in Chapter 4.

A simple and user-friendly interface to the framework presented is a strict requirement in order to attract users. Technical aspects, especially concerning parallelisation, need to be hidden from the users such that they can focus on the scientific aspects of their applications. We propose a dedicated scripting language, *TreeLatin*, for writing inherently parallel applications on both relational and tree shaped data structures, in Chapter 5. We show how to automatically translate these scripts into highly optimised applications for the previously presented execution framework.

In distributed applications, load balancing is an important aspect to consider. With a poorly balanced workload, an application distributed on a cluster might perform just the same, or possibly even worse, than a stand-alone variant of the same application. Current MapReduce-style frameworks do not consider the load balancing

issue at all. We demonstrate the importance of load balancing especially in the context of distributed scientific data processing in Chapter 6. We introduce the *Partition Cost Model* and propose *TopCluster*, a distributed monitoring and load balancing approach tightly integrated with the MapReduce processing scheme, as a solution to this challenge.

Finally, we conclude this thesis by summarising the presented contributions and outlining future challenges in Chapter 7.

PREVIOUS PUBLICATIONS

Parts of this thesis have been previously published at CLOSER 2011 [Gufler et al. 2011], at ICDE 2012 [Gufler et al. 2012a] and in the Springer series "Service Science: Research and Innovations in the Service Economy" [Gufler et al. 2012b].

2 FREQUENT SUBTREE MINING

2.1 INTRODUCTION

Throughout this thesis, we will employ frequent subtree mining algorithms as an example of a scientific data analysis application. Analogously to frequent itemset mining [Agrawal and Srikant 1994], frequent subtree mining aims at identifying patterns exceeding a given number of minimum appearances within the base data. In frequent itemset mining, the analysed data set consists of sets (or transactions) of items. In frequent subtree mining, in contrast, the data to analyse is a set of trees. Based on [Chi et al. 2005], we formally define this data structure.

Definition 2.1 (Tree Structure)**.** *A* rooted, unordered tree $T = (V, E, r)$ *is a directed, connected, acyclic graph with vertices (or nodes)* $V = \{v_0, v_1, \ldots, v_n\}$*, edges* $E = \{(v, w) : v, w \in V\}$*,* $|E| = n$ *and a dedicated root node* $r \in V$*.*

A sequence $((v_0, v_1), (v_1, v_2), \ldots, (v_{m-2}, v_{m-1}), (v_{m-1}, v_m))$ *of length* m *of connected edges is a* path*. It may be denoted shorter as* $(v_0, v_1, v_2, \ldots, v_{m-1}, v_m)$*. For* $v_0 = r$*, the path is called a* root path*. All vertices reached by root paths of length* m *form the* m*-th* level *of* T*.*

For an edge $(v, w) \in E$*,* v *is called the* parent *of* w*, and* w *is a* child *of* v*. If there are edges* $(v, w_0), (v, w_1) \in E$ *with* $w_0 \neq w_1$*, then* w_0 *and* w_1 *are* siblings*. For each path* $(v, \ldots, z)$*,* v *is an* ancestor *of* z*, and* z *is a* descendant *of* v*.*

Given a relation $<\subset V \times V$ *inducing a total order between all sibling nodes of a rooted, unordered tree* $T' = (V, E, r)$*,* $T = (V, E, r, <)$ *is a* rooted, ordered tree*.*

A set $\mathcal{F} = \{T_1, T_2, \ldots, T_n\}$ *of trees is a* forest*.*

Analogously to frequent itemset mining, labels are used to identify matching fragments in multiple trees. We can assign a label to each node of a tree. Furthermore, we can also assign labels to edges — an aspect which exceeds the possibilities offered by frequent itemset mining. In a frequent subtree mining algorithm, edge labels, e. g., allow us to distinguish various types of edges, thereby putting a very strong emphasis on the structural aspect.

Definition 2.2 (Labelled Tree)**.** *Given a rooted, unordered tree* $T' = (V, E, r)$*, a set of labels* L *and a labelling function* $l : V \cup E \to L$ *assigning labels to vertices and edges, then* $T = (V, E, r, L, l)$ *is a* rooted, unordered, labelled tree*.*

If only vertices are labelled, i. e., if $l = l|_V$, T *is a* vertex-labelled tree. *If labels are only applied to edges, i. e.,* $l = l|_E$, *then* T *is an* edge-labelled tree.

A rooted, ordered, labelled tree *can be defined analogously.*

In this thesis, we focus on vertex-labelled trees. We will therefore use the terms *vertex label* and *label* as synonyms, and explicitly refer to edge labels in situations in which a label may also be assigned to an edge.

In frequent itemset mining, the goal is to identify all sets of items which exceed a user-defined *minimum support threshold*, i. e., that are contained in a sufficient number of transactions in the base data. Analogously, in frequent subtree mining, all subtrees with a sufficiently large support in the base forest are discovered. Due to the structural aspect of trees, which is not present in frequent itemset mining, *counting the occurrences of subtrees* is an ambiguous goal. In fact, there are multiple possibilities for defining subtrees, and also for counting their occurrences. We will discuss these two aspects in the following.

2.1.1 *Subtree Types*

First, we focus on possible definitions of *subtrees*. We introduce three types of subtrees often encountered in literature (see [Chi et al. 2005]), which represent three classes of difficulty for frequent subtree mining. While we present subtree definitions for unordered trees only, the definitions hold for ordered trees analogously.

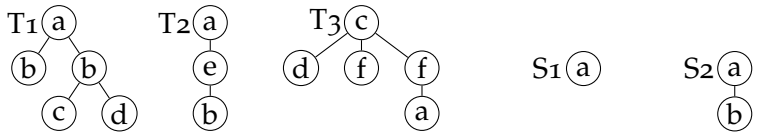
Definition 2.3 (Bottom-Up Subtree)**.** *A tree* $T_B = (V_B, E_B, r_B, L_B, l_B)$ *is called a* bottom-up subtree *of tree* $T = (V, E, r, L, l)$, *denoted as* $T_B \subset_B T$, *if and only if*

1. $V_B \subseteq V$
2. *if* $v \in V_B$, *then all descendants of* v *in* T *are also in* V_B
3. $(v, w) \in E \wedge v, w \in V_B \Rightarrow (v, w) \in E_B$, *and*
4. *both the vertex and edge labelling are preserved*

In order to emphasise the number of vertices $m = |V_B|$, T_B *may be called an* m-subtree.

Example 2.1. *Consider the three trees shown in Figure 2.1a. Tree S1 from Figure 2.1b is a bottom-up subtree of tree T3. However, it is no bottom-up subtree of trees T1 and T2, as the only vertex labelled a in both of these trees has child nodes which are not contained in S1.*

Tree S2 is not a bottom-up subtree of any of trees T1, T2 and T3.



(a) Sample Forest (b) Sample Subtrees

Figure 2.1: Sample Forest and Subtrees

Lemma 2.1. *A rooted, labelled tree* $T = (V, E, r, L, l)$ *has exactly* $|V|$ *bottom-up subtrees.*

Proof. A bottom-up subtree may be rooted in any vertex $v \in V$ and includes all the descendants of v in T along with the respective edges. Thus, a bottom-up subtree is uniquely identified by choosing its root node v, for which there are $|V|$ possibilities. □

Definition 2.4 (Induced Subtree)**.** *A tree* $T_\mathcal{J} = (V_\mathcal{J}, E_\mathcal{J}, r_\mathcal{J}, L_\mathcal{J}, l_\mathcal{J})$ *is called an* induced subtree *of tree* $T = (V, E, r, L, l)$*, denoted* $T_\mathcal{J} \subset_\mathcal{J} T$*, if and only if*

1. $V_\mathcal{J} \subseteq V$,
2. $v, w \in V_\mathcal{J} \wedge (v, w) \in E \Rightarrow (v, w) \in E_\mathcal{J}$*, and*
3. *both the edge and vertex labelling are preserved.*

As before, in order to emphasise the number of vertices $m = |V_\mathcal{J}|$*, the term* m-subtree *may be used.*

Example 2.2. *Tree S1 from Figure 2.1b is an induced subtree of all three trees in Figure 2.1a.*

Tree S2 is an induced subtree of tree T1 (picking the root vertex and either its left or its right child). However, it is not a subtree of T2 (the vertex labelled b is not a child of the vertex labelled a) and T3 (there is no node labelled b).

Lemma 2.2. *A rooted, labelled tree* $T = (V, E, r, L, l)$ *has up to* $|V| + 2^{|V|-1} - 1$ *induced subtrees.*

Proof. Every single vertex is a valid induced subtree of T. Hence, we have $|V|$ induced 1-subtrees. For k-subtrees with $k > 1$, the shape of the tree influences the number of subtrees. The maximum number is reached if T consists of two levels, i. e., r is the parent vertex of all nodes in $V \setminus \{r\}$. Then, we can choose for each of the nodes except r whether to include it in the subtree or not, resulting in $2^{|V|-1}$ possibilities. This includes the subtree consisting only of the root node, which we already counted as one of the $|V|$ 1-subtrees; we must thus subtract it once. □

Definition 2.5 (Embedded Subtree). *A tree* $T_\mathcal{E} = (V_\mathcal{E}, E_\mathcal{E}, r_\mathcal{E}, L_\mathcal{E}, l_\mathcal{E})$ *is called an* embedded subtree *of tree* $T = (V, E, r, L, l)$*, denoted* $T_\mathcal{E} \subset_\mathcal{E} T$*, if and only if*

1. $V_\mathcal{E} \subseteq V$,
2. $\forall (v, w) \in E_\mathcal{E} : \exists (v_0, v_1, \ldots, v_n) : v_0 = v \wedge v_n = w \wedge \forall 0 \leqslant i < n : (v_i, v_{i+1}) \in E$, *and*
3. *the vertex labelling is preserved, as is the edge labelling for edges contained in both* T *and* $T_\mathcal{E}$*. As* $T_\mathcal{E}$ *may contain edges not available in* T*,* $l_\mathcal{E}$ *might be extended to include edges in* $E_\mathcal{E} \setminus E$ *if we allow edges to be labelled.*

If special focus is laid on the number of vertices in the subtree, it might be referred to as a m-subtree*, where* $m = |V_\mathcal{E}|$.

Example 2.3. *Consider again the trees in Figure 2.1. Tree S1 is an embedded subtree of T1, T2, and T3.*

Tree S2 is an embedded subtree of T1 (as in the scenario for induced subtrees), and also of T2: the definition of embedded subtrees allows us to skip the vertex labelled e, and connect the vertices labelled a and b directly.

Lemma 2.3. *A rooted, labelled tree* $T = (V, E, r, L, l)$ *has up to* $2^{|V|} - 1$ *non-empty embedded subtrees.*

Proof. We obtain the maximum number of subtrees if we are free to choose, for every vertex $v \in V$, whether to include it in the subtree or not. This is the case when T consists of a single path. We can then remove any node without splitting the tree into multiple subtrees which are no longer connected. This results in $2^{|V|}$ possible choices including the empty subtree, or one less, if we eliminate the choice of not including any vertex in the constructed subtree. □

These three types of subtrees represent three levels of difficulty for frequent subtree mining. Bottom-up subtrees are the easiest to mine for. This is intuitively clear when looking at the number of possible subtrees, which grows exponentially in the number of vertices for both induced and embedded subtrees, but only linearly for bottom-up subtrees. Therefore, when mining for bottom-up subtrees, there are way less feasible patterns to take into account. In fact, mining for bottom-up subtrees does not even require the use of dedicated frequent subtree mining algorithms. Instead, it is possible to apply a *pre-order serialisation* (see below) to the trees. Then, frequent sub*string* mining algorithms can be employed to detect the frequent bottom-up subtrees [Chi

et al. 2005]. Due to their simplicity, we will not consider bottom-up subtrees further in this thesis.

Definition 2.6 (Pre-Order Serialisation)**.** *Let* $T = (V, E, r)$ *be a rooted tree. The* pre-order serialisation *of* T *is a string* S *composed of*

1. *a serialised representation of the root node* r,
2. *the pre-order serialisation of the bottom-up subtrees rooted in the child nodes of* r *in* T,
3. *and an end marker* $\perp$,

in this order.

Example 2.4. *Consider tree T1 in Figure 2.1. Assume the serialisation of a node consists of just the node label. Then, the pre-order serialisation of T1 is* $S = \mathtt{a}\mathtt{b}\perp\mathtt{b}\mathtt{c}\perp\mathtt{d}\perp\perp\perp$.

The number of both induced and embedded subtrees of a tree grows exponentially in its number of vertices. Algorithms enumerating all subtrees of these two types cannot, therefore, have less than exponential complexity. Despite both delivering an exponential number of choices in the worst case, however, the structural difference between induced and embedded subtrees makes it worthwhile to distinguish them, and provide dedicated algorithms for each of these two types.

Induced subtrees are easier to identify. Consider two matching vertices v and w from the complete tree T and the subtree S, respectively. Only immediate child nodes of v may be matched with children of w.

Embedded subtrees are the most difficult to search for, as parent-child relationships in the pattern tree may correspond to arbitrary ancestor-descendant relationships in the original tree. Consider, as above, the nodes v and w from trees T and S, respectively. All descendant nodes of v may be matched with immediate child nodes of w. This results in a much higher number of choices than with induced subtrees for all trees with more than two levels.

2.2 PRINCIPLES OF FREQUENT SUBTREE MINING

As already stated before, the goal of frequent subtree mining algorithms is to determine all subtrees with a sufficiently large support in the analysed forest. Two essential components of a frequent subtree mining algorithm are therefore candidate generation and support counting. *Candidate generation* describes the method employed for generating trees which are likely to be frequent in the base data and whose exact

support must hence be determined. *Support counting* determines how occurrences of a candidate subtree in a forest are counted. We will elaborate on these two aspects in the following.

2.2.1 *Candidate Generation*

Recall from above that, for both induced and embedded subtrees, the number of possible subtrees grows exponentially in the number of nodes of the base tree. This makes "blind" extraction of all possible subtrees an infeasible approach to frequent subtree mining. Instead, frequent subtree mining algorithms start detecting the frequent 1-subtrees, which are then extended to form larger and larger *candidates* whose support is then checked. Thereby, the frequent 1-subtrees play a special role. Frequent 1-subtrees are the building blocks for more complex patterns. The frequent 1-subtrees can be identified with one scan over the input data set by counting the number of occurrences of each node label and then discarding those labels whose count is below the minimum support threshold. Candidates consisting of more nodes are then derived from the already identified frequent subtrees with fewer nodes.

In order to restrict the number of generated candidate subtrees, frequent subtree mining algorithms rely on the *downward closure property*. Intuitively, a frequent subtree cannot contain nodes which are not frequent themselves. More formally, given a forest $\mathcal{F}$ and two candidate subtrees S and S' with S being a subtree of S', the downward closure property states that S' cannot have a larger support in $\mathcal{F}$ than S. Generating candidate subtrees, this property allows us to skip all those trees containing a subtree which we already know to be infrequent. In many practical situations, this allows for a significant reduction of the number of candidates whose occurrences must be counted.

Eventually, the actual support of the generated candidates is determined. The support calculation is an expensive operation, as it requires identifying the matches of the candidate subtree in the complete base data. Therefore, frequent subtree mining algorithms should generate as few false candidates, i. e., candidates with an effective support value below the minimum support threshold, as possible. Moreover, each candidate should only be generated once. On the other hand, of course, the algorithms must not miss any subtree which actually *is* frequent. There are two popular techniques for generating larger candidate subtrees from previously identified simpler ones which aim at reducing the amount of false candidates generated.

PATTERN GROWTH approaches extend frequent n-subtrees which were already identified by a single frequent node a time in order to form candidate $n+1$-subtrees

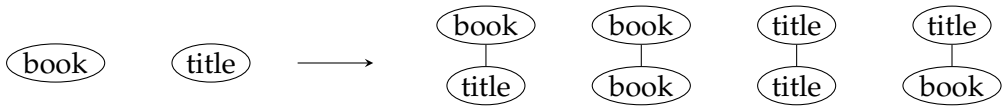


Figure 2.2: Generating Candidate Subtrees: Pattern Growth

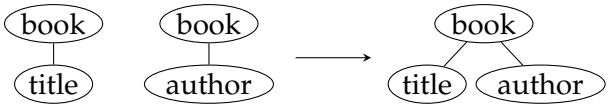


Figure 2.3: Generating Candidate Subtrees: À-Priori

whose frequency is then checked. By restricting the positions in the n-subtree where the new node can be attached it is possible to avoid duplicate candidate generation. Some of the algorithms using the pattern growth approach, especially those focussing on XML mining, additionally evaluate structural information (e. g., a DTD or XML schema) in order to avoid the creation of subtrees which can not occur in the input data due to the imposed structural constraints.

Example 2.5. *Consider an XML data set describing books according to the following DTD.*

```
<!ELEMENT book (title, author*, publisher?)>
<!ELEMENT title #PCDATA>
<!ELEMENT author #PCDATA>
<!ELEMENT publisher #PCDATA>
```

Assume we find both the `book` *and the* `title` *nodes to be frequent. Employing a pattern growth candidate generation which does not exploit structural information, we will create all four candidate 2-subtrees shown in Figure 2.2. Exploiting the information provided by the DTD, we will only create the leftmost of these four candidates. According to the DTD, the other three subtrees cannot be contained in the data set.*

À-PRIORI style algorithms proceed similarly to the à-priori algorithm for frequent itemset mining [Agrawal and Srikant 1994]. They combine frequent n-subtrees which differ in only one node in order to form new candidate $n+1$-subtrees.

Example 2.6. *Consider again the XML data set introduced in the previous example, and assume we found the two 2-subtrees on the left of Figure 2.3 to be frequent. An à-priori style candidate generator for unordered trees will combine these two subtrees to obtain the 3-subtree on the right of Figure 2.3 as a new candidate.*

Both these techniques have advantages. Pattern growth approaches allow the search space to be traversed in a depth-first manner. This boils down in low memory

consumption, as only very few of the generated subtrees must be kept in memory simultaneously. On the other hand, by expanding a frequent subtree with a frequent 1-subtree, the number of generated false candidates may be large.

À-priori approaches require two frequent n-subtrees to create a new candidate $n+1$-subtree. In order to provide this input, the search space must be traversed breadth-first. This is more memory-intensive than a depth-first traversal. All candidates of size n must be generated, keeping the frequent ones in memory, before starting to work on $n+1$-subtrees. However, by combining two frequent n-subtrees, the number of false candidates is likely to be lower than with pattern growth.

Both approaches, however, have in common that they start with frequent 1-subtrees which are then extended to form larger and larger patterns. If an infrequent subtree is found, no extensions of that pattern need to be considered due to the downward closure property.

2.2.2 *Support Measures*

Relying on the downward closure property to limit the number of candidates requires the employed occurrence counting algorithm to provide compatible counts. The number of occurrences of a candidate in the base data is termed its (absolute) *support*. Normalising this value by the number of trees in the base data, we obtain the relative support. The counting algorithm is called *support measure*.

Consider a forest $\mathcal{F}$ and two trees S and S' with $S \subseteq S'$. With a support measure compatible to the downward closure property, it must be impossible for S' to have a larger support than S. Informally, this defines an *admissible* [Vanetik et al. 2002] support measure. While this might seem a trivial requirement at first glance, not all support measures fulfil it.

Determining the support of a tree in a forest is an essential step in frequent subtree mining. It must be performed for every generated candidate in order to determine if it is frequent or not. In frequent itemset mining, it is clear how to count the occurrences of an itemset: it is the number of transactions containing that itemset as a subset. Due to the structural aspect of trees, counting occurrences of a subtree in a forest is not as straight-forward. Effectively, there are multiple ways by which we could count the occurrences of a tree in a forest, as the following example shows.

Example 2.7. *Consider the trees S2 and T1 from Figure 2.1. S2 is contained in T1 twice as an induced subtree: The root node with its left child, and the root node with its right child. We could count this as either a support of one ("S2 is contained in T1") or two ("there are two occurrences of S2 in T1").*

While both these support measures may be reasonable depending on the application scenario, only the former obeys the downward closure property.

Example 2.8. *Tree S1 from Figure 2.1 appears once in tree T1. Tree S2 is contained in T1 twice. However, S1 is a subtree of S2. Apparently, counting all occurrences of a subtree within a tree does not obey the downward closure property. It is thus not an admissible support measure.*

The support measure appearing most prominently in frequent subtree mining literature is the *transaction based support*, which is very similar to the support measure in frequent itemset mining.

Definition 2.7 (Transaction Based Support)**.** *Given a tree S and a forest $\mathcal{F}$, the* absolute transaction based support *of S in $\mathcal{F}$ is defined as*

$$\mathrm{supp}^{x}_{A}(S,\mathcal{F}) = |\{T \in \mathcal{F} : S \subset_x T\}| \qquad x \in \{\mathcal{I},\mathcal{E}\}\ ,$$

i.e., the number of trees in $\mathcal{F}$ containing S as a subtree according to the employed subtree definition (induced or embedded).
The relative transaction based support *of S in $\mathcal{F}$ is*

$$\mathrm{supp}^{x}_{R}(S,\mathcal{F}) = \frac{\mathrm{supp}^{x}_{A}(S,\mathcal{F})}{|\mathcal{F}|} \qquad x \in \{\mathcal{I},\mathcal{E}\}\ .$$

Example 2.9. *Consider once more the forest depicted in Figure 2.1a. The (absolute) transaction based support of tree S1 from Figure 2.1b is 3, as a node labelled a is contained in every tree of the forest. The support of tree S2 depends on the choice of the subtree definition. If we consider induced subtrees, S2 has a support of 1, as it only appears in tree T1 (see Example 2.2). If we look for embedded subtrees, the support of S2 is 2, as it is a subtree of both T1 and T2 (Example 2.3).*

In the remainder of this thesis, we will assume transaction based support in descriptions and examples. However, any other admissible support measure is applicable as well.

2.3 DISTRIBUTING FREQUENT SUBTREE MINING

Literature provides a vast variety of algorithms for mining frequent induced and embedded subtrees [Chi et al. 2005]. The essential differences between these algorithms are the way (and the order) in which they generate candidate subtrees and the realisation of the support counting. Nonetheless, all these algorithms share common aspects on which we will focus in the following.

Coarsely, frequent subtree mining algorithms proceed in two steps. First, the frequent labels — corresponding to frequent 1-subtrees — are detected. These are then used as building blocks for constructing more complex frequent subtrees. This pattern, which all frequent subtree mining algorithms follow, allows to derive a generic workflow to which all these algorithms can be mapped. We will identify the challenges arising from distributing the mining process in the following, and then derive the distributed workflow for frequent subtree mining.

2.3.1 *Challenges*

For candidate generation and support counting, frequent subtree mining algorithms typically do not scan the original input data set repeatedly. Doing so would be prohibitively expensive. Rather, they first extract the essential information on frequent nodes required for their work into a more compact data summary. The actual processing, then, takes place on this *intermediate structure* solely. The concrete realisation of this intermediate structure is specific to the mining algorithm. The structure may, e. g., be a forest like the *compressed trees* employed by PathJoin [Xiao et al. 2003], lists as the *scope lists* used by TreeMiner [Zaki 2005], or tree serialisations like TRIPS' [Tatikonda and Parthasarathy 2009] *Prüfer sequences*. There are, however, two important aspects common to all of these data structures:

SIZE As the intermediate data structure needs to represent every tree node of possible interest, its size grows at least linearly with the number of frequent nodes in the forest.

EXTRACTION OF STRUCTURE FRAGMENTS We can partition the intermediate structure by extracting fragments relevant for finding a certain subset of all frequent patterns. That is, for finding frequent subtrees with a given label at the root node, we only require fragments extracted from subtrees whose root node has this label. We will exploit this property for distributing the tree mining process to multiple hosts.

The intermediate structure summarises all aspects of the original input data relevant to the mining task. Once it is built, we can thus discard the input data. The intermediate structure is used to accelerate the scanning for frequent patterns in the input forest. Part of the support calculation is a subtree isomorphism test which is NP-hard for unordered trees. Special properties of the employed intermediate structure are often exploited in order to reduce the amount of required calculations under certain circumstances.

The intermediate structure is typically not accessed in a linear manner, but by random patterns. Thus, the performance of the mining algorithms degrades rapidly as soon as the intermediate structure no longer fits into the available main memory. The size of data sets which can be efficiently handled by an algorithm is thus limited by the lower bound on the intermediate structures' size mentioned above.

We aim at mining very large data sets whose intermediate structure will usually not fit into the main memory of a single server. Therefore we will distribute the mining process in cluster environments. Thereby, obviously, we must also distribute both the base data and the intermediate structure to allow for parallel processing throughout the entire frequent subtree mining workflow. This adds some new interesting challenges to the frequent subtree mining problem.

BASE DATA DISTRIBUTION Initially, the base data must be distributed to the involved hosts. This distribution must be such that the frequent subtree mining algorithm can begin extracting the intermediate structure on each host locally, without requiring communication (e. g., all nodes of a tree must typically be processed together on the same host).

INTERMEDIATE DATA DISTRIBUTION Once all hosts have extracted the fragments of the intermediate data structure from their share of input data, these fragments must be redistributed. Thereby, each fragment must be sent to all hosts that will require it during the final phase of the mining process.

INTERMEDIATE DATA MERGING On the receiver side, intermediate structure portions coming in from various hosts must be consolidated before the mining algorithm can use them.

Based on this analysis, we derive the distributed tree mining workflow presented in the following section.

2.3.2 *A Distributed Tree Mining Workflow*

Figure 2.4 shows a generic workflow for distributed frequent subtree mining. Processing steps are depicted as rectangles, arrows symbolise data flow between these steps (which is not necessarily data flow between different hosts). The possible parallelism is sketched by drawing two identically labelled processors for each phase. The actual number of parallel processors is, however, not limited to two. The workflow basically consists of the processing steps and the data exchange phases described above. We will explain the single steps in more detail in the following.

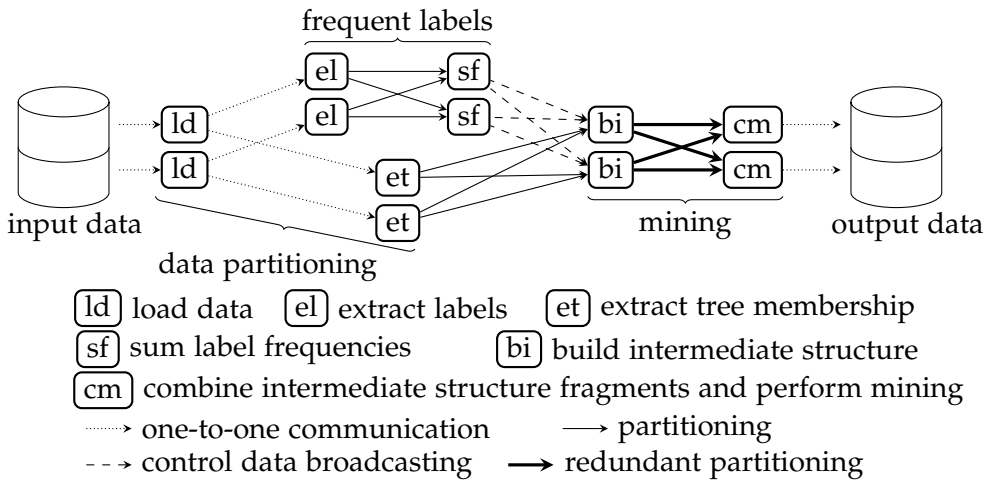


Figure 2.4: Distributed Tree Mining Workflow

LD: LOAD DATA The first step is to load the data into the system. The input data is either already partitioned (e. g., a set of files representing a partition each) or it is accessible in one big block (e. g., a single file or a database table) and we need to partition it in order to process the single partitions in parallel. We do not, however, assume any particular order on the data; it might be distributed arbitrarily to the involved hosts. While loading, we also discard attributes not relevant to the mining process, if there are such.

ET: EXTRACT TREE MEMBERSHIP Tree data sets are often stored one record per tree node. Before we can begin to work on the trees in a distributed setup, we thus need to ensure that all nodes belonging to the same tree will be processed on the same host, that is, the nodes are *treewise distributed*. The processors of this step receive the data loaded by one of the processors of the LD phase and determine the tree membership of each node, allowing to subsequently repartition the data in order to achieve a treewise distribution. Hence, this step ensures that the tree data is distributed to the processing hosts such that the requirements on the initial data distribution explained above are met.

EL: EXTRACT LABELS Logically independent from the ET phase, we determine the frequent labels in our data set. Therefore, in this phase we extract label occurrence lists, i. e., information on which labels occur in which trees. Note that, for this processing step, we do not need the data to be treewise distributed.

SF: SUM LABEL FREQUENCIES These label occurrence lists are distributed to the processors of the SF phase such that all information regarding the same label is collected on the same host. By aggregating the occurrence information, we can now determine the support of each label and hence decide whether it is frequent or not.

BI: BUILD INTERMEDIATE STRUCTURE FRAGMENTS We now combine the original tree data with the knowledge on the frequent labels, and extract the fragments of intermediate structure. Specifically, each processor receives a partition of the tree data from the processors of the DT phase. This is the first processing step where the data is treewise distributed, i. e., all nodes belonging to the same tree are guaranteed to be located on the same host. We combine the tree data with the complete list of frequent labels, which each host receives from the processors of the SF phase. Note that, for reasonable minimum support thresholds, the size of this list is independent of the size of the input data. The processors of the BI phase cannot begin their work until both the DT and SF phases have successfully completed, since all of their input is required. Each processor then creates the fragments of the intermediate structure for the trees in its partition, discarding information on nodes with infrequent labels. The need to work with the intermediate data structure makes BI the first processing step of this workflow whose concrete realisation depends on the actually employed frequent subtree mining algorithm.

CM: COMBINE FRAGMENTS AND PERFORM MINING Finally, we assign the intermediate structure fragments extracted in the previous step to processors of the CM phase according to the extraction property explained above and employing a *redundant partitioning* approach. By redundant partitioning, we refer to a distribution where parts of the data are replicated to multiple hosts, while other parts are available on one host only. We combine the received fragments to obtain the actual intermediate structure, and exploit this resulting structure to scan for frequent subtrees, which we store either in files or in a database. The implementation of this phases' processors is again specific to the employed frequent subtree mining algorithm, as it processes the intermediate structure.

2.4 SAMPLE APPLICATION SCENARIOS

Having introduced the abstract distributed frequent subtree mining workflow, we will now present sample application scenarios. First, we will present the concrete

mapping of two frequent subtree mining algorithms onto our workflow. Then, we will sketch real e-science application scenarios from the astrophysics domain which can exploit this workflow.

2.4.1 *Sample Algorithms*

As exemplary realisations of frequent subtree mining algorithms using our distributed workflow, we pick one algorithm mining for induced and one for embedded subtrees. Note that only the BI and CM phases' realisations depend on the actual frequent subtree mining algorithm. All preceding phases are independent of the algorithm. Therefore, we only describe the BI and CM phases of the two algorithms.

PATHJOIN [Xiao et al. 2003] mines for induced subtree in which no two sibling nodes share the same label. It employs an intermediate structure called *compressed forest* which summarises the appearances of frequent nodes in the base forest. Thereby, it annotates frequent nodes with positional information on each appearance and discards infrequent nodes. Figure 2.5 sketches the basic idea of distributed PathJoin with two hosts and two input trees, assuming node labels `a` and `b` are frequent. In the BI phase, we start building the compressed trees at each involved host from the data partition assigned to that host. The obtained tree fragments are then redistributed among the CM hosts such that all compressed trees sharing the same label at the root node are located on the same host. The CM workers merge the compressed tree fragments they obtain from the BI processors. Based on this information, the frequent subtrees rooted in a node with that label can be identified.

TREEMINER [Zaki 2005] was one of the first frequent subtree mining algorithms to be published. It mines for embedded subtrees. As intermediate structure, TreeMiner uses *scope lists*. For each frequent label, these lists store all the (identifiers of the) trees the label appears in, together with the *scope information* specifying the position the corresponding node appears at in that tree. The actual mining process of TreeMiner starts with the frequent 2-subtrees. In the BI phase, we therefore extract both the scope list fragments and candidates for frequent 2-subtrees. The candidate 2-subtrees are sent to the CM host responsible for the label of the root node of the candidate subtree. The scope list entries are sent to all CM hosts responsible for labels on the path from the node the scope list entry refers to the root node of the corresponding tree. The CM phase workers then combine the candidate 2-subtrees in order to identify the frequent 2-subtrees. By merging the obtained scope list fragments, all scope list

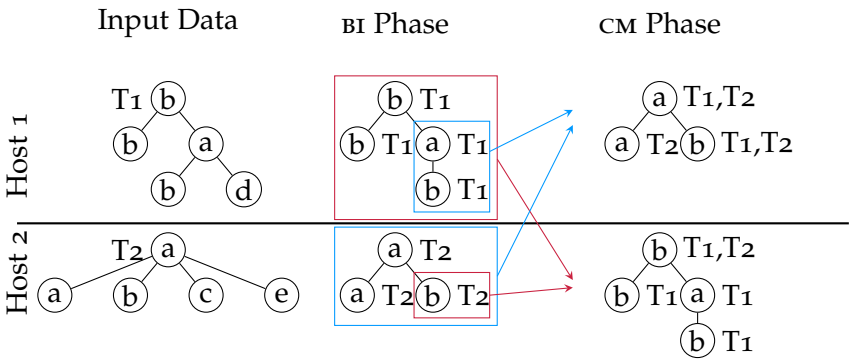


Figure 2.5: Distributed Path Join

entries relevant to that host are constructed. With this information, each host can begin the mining process on its partition of possible frequent subtrees.

2.4.2 *Astrophysical Applications*

Frequent subtree mining is an interesting data analysis application in both scientific and business environments. In theoretical astrophysics, for example, frequent subtree mining can be employed to analyse data sets like the Millennium simulation presented in the introductory chapter. Recall that the sheer size of the simulation results renders centralised frequent subtree mining infeasible. This is manifested by frequent subtree mining literature, where the data sets used for experimental evaluation are typically smaller than 1 GB.

A straight-forward application of frequent subtree mining to the trees of the Millennium simulation allows to identify common patterns in the evolution of the universe. Knowing the frequent evolution patterns allows scientists to restrict further analyses to either common subtrees, or outliers, depending on the goals of the research project.

As an example of a more complex scientific application, consider the Millennium and Millennium XXL simulations. Both these simulations describe the evolution of mass distribution in the universe. The Millennium XXL simulation covers a larger spatial area than the Millennium simulation. However, the Millennium simulation has a much higher level of detail than the Millennium XXL simulation, i. e., the Millennium trees contain nodes which were too small to be captured in Millennium XXL. Replacing coarse-grained trees from Millennium XXL by *corresponding* higher-

resolution trees from Millennium, the former simulation can be enhanced. *Corresponding trees* thereby refers to trees which are identical in the low resolution of the Millennium XXL simulation. In this task, frequent subtree mining can help identifying the trees most worthwhile to find replacements for.

2.5 RELATED WORK

Data mining offers a vast variety of interesting, complex data analysis methods. Especially since the seminal work presenting the À-priori algorithm [Agrawal and Srikant 1994], it has gained substantial attention in the research community.

There is not much prior work on distributed frequent subtree mining. Therefore, we will handle frequent subtree mining and distributed data mining separately.

Like for other areas, a noticeable number of algorithms were presented for frequent subtree mining, each proving its advantages for specific application scenarios or on data sets with specific properties. A comprehensive overview is given in [Chi et al. 2005].

TRIPS and TIDES [Tatikonda and Parthasarathy 2009, 2008] are two algorithms searching for embedded subtrees. Their intermediate structure is composed of string serialisations of the trees in the input forest. The authors proposed to exploit modern multi-core CPUs for speeding up these two algorithms. In their approach, the intermediate structure is not kept in memory, but is recalculated every time it has to be accessed, utilising all the processing cores available on that host. This approach requires continuous access to the entire data set, which may be expensive in a distributed environment. Moreover, their approach highly benefits from the shared-memory architecture it is supposed to run on. The cost of network messaging would be prohibitively high for their communication scheme, consisting of a high number of relatively small messages.

For a variety of data mining applications distributed algorithms were proposed. They can be classified into client/server and peer-to-peer approaches.

For client/server scenarios, [Januzaj et al. 2004] presents a distributed clustering algorithm. Each host extracts the candidate clusters from its data partition. Then, a central site combines this information to determine global clusters. Such an approach is feasible for applications in which the last processing step is rather light-weight. In frequent subtree mining, the last processing step — which is performed in the CM phase of our workflow — is the most complex. Hence, such an approach is not feasible for frequent subtree mining applications.

A basic component of many distributed data mining algorithms is the majority vote which allows to decide whether the sum of values held by each server is above a given threshold or not. [Datta et al. 2006] presents a majority vote for peer-to-peer environments where no global synchronisation is possible. While we can see the demand for algorithms with such a communication behaviour for volatile environments like P2P networks, they are not required in clusters, which we see as the typical environment for our workflow.

3 LARGE-SCALE PARALLELISATION

3.1 INTRODUCTION

E-science data sets are typically far too large to be analysed in a centralised manner on a single host. In this chapter, we will investigate means of distributing the processing of large, possibly tree structured data sets to a multitude of hosts. We propose the *Pipelined MapReduce Framework*, an enhancement of the popular MapReduce [Dean and Ghemawat 2008] programming model, which provides inherent parallelisation for data-intensive processing tasks. The key challenges in designing such a framework are

SCALABILITY In order to cope with the ever-growing sizes of e-science data sets, the framework must scale from small environments consisting of just a few scientists' workstations up to clusters in data centres composed of thousands of hosts.

EFFICIENCY The framework must permit the efficient processing of data sets of just a few gigabytes up to several petabytes which could not be handled on stand-alone hosts. Thereby, the framework must provide a significant speed-up of data-intensive applications as compared to stand-alone executions of the same application.

EASE OF USE In order to be widely accepted, the framework must be easy to use. It should clearly abstract from the technical aspects of distributed processing, such as communication with remote sites, data distribution, synchronisation, and the related error handling. This permits application developers to focus on the core aspects of their application, leaving error-prone technical distribution aspects to the framework.

3.2 DISTRIBUTING THE DATA

The first aspect to consider when planning distributed processing is data distribution. The decision on how to initially distribute the data influences the choices available

for the actual data processing. With some types of data distribution, additional communication phases may be necessary to provide every host in the cluster with the information it requires. Depending on the storage configuration, we distinguish two possible scenarios.

SHARED STORAGE The input data is stored on a file system which is directly accessible by all hosts participating in the distributed processing. On the one hand, this includes centrally provided storage systems like NAS or SAN. In these systems, the storage is typically provided by a dedicated server. All processing hosts thus need to access their data over the network. On the other hand, this scenario comprises distributed file systems in which each of the processing hosts shares some of its local disks. Examples for such systems are Lustre[1], the RedHat Global File System[2], but also special-purpose file systems like the Google File System [Ghemawat et al. 2003] and its free counterpart HDFS (Hadoop File System).

LOCAL STORAGE Every host participating in the distributed processing only has some dedicated storage — typically on a local disk — to store its portion of the input data. Other hosts may not be able to directly access the data residing on other hosts on file level. This storage configuration also includes environments in which partitions of the base data must not be brought together, e. g., due to privacy constraints (a possible scenario in medical sciences).

Even with shared storage configurations, typically, single hosts will not access the complete data set. Doing so would often result in every host performing the same data accesses and calculations – an unnecessary slow-down. It is typically better to distribute the processing to all hosts, and subsequently exchange the results. Therefore, shared storage settings can be interpreted as local storage settings in which the data distribution to the hosts can be defined on a per-job basis. We will focus on shared storage settings in the following.

For the initial data distribution, we consider the following possibilities. While we focus on distributed tree processing here, similar reasoning applies to other application scenarios in which the data items are processed in (not necessarily disjoint) groups as well. Such situations arise, e. g., in graph processing when all nodes and edges of a compound must be processed together, or in business intelligence applications where all items of an order must be processed together.

1 http://www.lustre.org/
2 http://www.redhat.com/gfs/

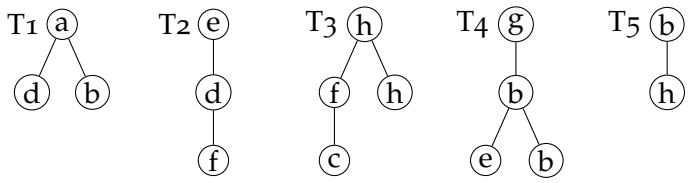


Figure 3.1: Distributing the Data: Sample Trees

Note that we only elaborate on the *initial* data distribution here. During processing, the data may be redistributed according to the requirements of the algorithms employed. A workload balancing data distribution plays an important role in this processing stage. We will focus on redistributing data in a workload balancing manner during processing in Chapter 6.

3.2.1 *Full Replication*

A very simplistic approach may require to have the entire input data accessible from all hosts. Consider the scenario of a distributed frequent subtree mining application, as introduced in Chapter 2. Instances of the algorithm running on each host can, in such a setup, agree on a partitioning of the node labels and then mine for frequent subtrees with "their" labels on the root node. Every host can run its share of the frequent subtree mining task independently from the other hosts.

Example 3.1. *Assume we have three hosts and the tree data set shown in Figure 3.1. With full replication and a local storage configuration, we store a copy of every tree on each of the hosts.*

Given the sheer size of e-science data sets and their expected exponential growth rate, it is clear that a replication approach would quickly reach its limits on storage requirement with local storage. With shared storage, we do not need to store a copy of the complete data set on every host. However, as already mentioned above, a scenario requiring full replication will most likely suffer from performance problems, as part of the calculation is repeated on every host. Therefore, we consider full replication no viable option.

3.2.2 *Distributing by Attribute Value*

For application scenarios where the workload can be partitioned according to some attribute of the data items (e. g., the node label for frequent subtree mining) by node

Host	Trees
Host 1	T1, T2, T4
Host 2	T1, T2, T3, T4, T5
Host 3	T2, T3

Table 3.1: Tree Distribution by Label

labels, we can assign each host a set of values of these workload partitioning attributes that host is responsible for. For tree data, this distribution still causes data replication if the partitioning attributes are node attributes. We then replicate each tree only to those hosts which are responsible for it. This way, the assignment of labels to their responsible hosts is defined when distributing the data. However, each host is still able to work on its trees without having to interact with others.

Example 3.2. *Consider the same scenario as in the preceding example. We assign the labels to the three hosts in a round-robin manner, i. e., labels a, d and g to the first host, labels b, e and h to the second, and finally labels c and f to the third host. Distributing the trees by label, we obtain the data distribution shown in Table 3.1.*

The effectiveness of such a distribution approach depends on the data distribution on the partitioning attributes. In the Millennium simulation, the 8 most frequent node labels (out of over 130 000) appear in more than 40% of the trees. Thus, even distributing the trees by node labels would lead to immoderately large data sets on the hosts responsible for those very frequent labels.

3.2.3 *Random Tree Partitioning*

In order to avoid the skewed data distribution observed, we consider assigning trees to hosts using a random (uniform) distribution. This allows for a better distribution of the trees among the hosts. However, the processing might need to be adapted to this scenario, and intermediate data exchange phases might be necessary. Our distributed frequent subtree mining workflow presented in the preceding chapter is already prepared for running in such an environment.

Example 3.3. *Consider once more the five trees in Figure 3.1. With random tree partitioning, we assign each tree to one host only. For example, we could store trees T1 and T3 host 1, trees T2 and T5 on host 2, and tree T4 on host 3.*

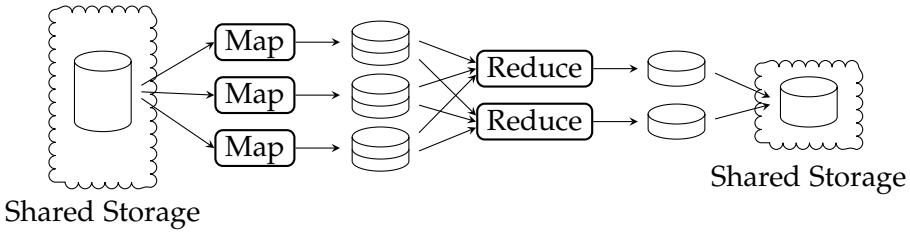


Figure 3.2: MapReduce Architecture

3.3 MASSIVE PARALLEL DATA PROCESSING

Processing data in parallel in a distributed environment has been a challenging subject in both research and industry for many years. Well-known parallelisation concepts range from multithreading APIs like PThreads to programming language constructs like OpenMP[3] to task queueing mechanisms like the Google Task Queue[4]. Especially the two former approaches, which introduce parallelisation on a very low abstraction level, are known to be complex and error-prone. The latter is already more abstract and typically hides the low-level thread handling and synchronisation aspects from the user. Other aspects like, e. g., data partitioning are, however, still left to the user.

With the publication of the MapReduce programming model [Dean and Ghemawat 2008] in 2004, Google started a new hype on distributed, data-parallel processing. The basic idea behind MapReduce is to partition the data and run side effect free functions, first on single data items (map) and then on groups of items (reduce), in parallel. This allows the execution of a filtering and aggregation operation on a data set in one MapReduce job, exploiting massive parallelism. Thereby, MapReduce scales from small setups with only a few hosts up to deployments in data centres with thousands of compute nodes.

MapReduce uses a shared storage setting to store both the input data and the results of MapReduce jobs. The data sets processed with MapReduce are sets of independent tuples. An abstract MapReduce job is shown in Figure 3.2. In the beginning of the MapReduce job, the input data is partitioned into chunks of fixed size – 500 MB to 1 GB in typical settings.

For the first processing phase, one data partition is assigned to each *mapper*. The mappers read their data partition from the shared storage and apply a map function

3 http://www.openmp.org/

4 https://developers.google.com/appengine/docs/java/taskqueue/

supplied by the user to every tuple. For every input tuple, the map function produces a set of intermediate (key,value) pairs. These tuples are then hash-partitioned according to their key and stored on the local disk of the host running the map task. All mappers create the same number of partitions and use the same hash function for partitioning the intermediate data. Therefore, intermediate pairs with the same key are assigned to the same partition, no matter which mapper they are created on. Note that the mappers of a MapReduce job do not need to all run simultaneously. As the map function must be side effect free and the intermediate results are stored on disk, every map task can run independently of all other map tasks of the job.

Example 3.4. *We want to calculate the absolute transaction based support (Section 2.2.2) of all labels in the tree data set in Figure 3.1. Assume the tree data set is stored one node a tuple. Each tuple contains the node identifier, the node label, a reference to the parent node, and the tree identifier, i. e., the input data set for tree T1 is* $\{(1, a, \bot, T1), (2, d, 1, T1), (3, b, 1, T1)\}$.

In order to calculate the transaction based support, we first need to group the data set by label. Then we can count the number of distinct tree identifiers in each group. In this scenario, the mapper must emit, for each tree node, a tuple containing the node label as key and the tree identifier as value. For tree T1, we generate the following three tuples: $(a, T1)$, $(d, T1)$, $(b, T1)$.

The second processing phase is the *reduce* phase. One reducer is run for every partition of intermediate data. The reducers retrieve their partition of intermediate data from all mappers. As all mappers employed the same partitioning, all intermediate pairs sharing the same key will be retrieved by the same reducer. Then the reduce function, which is again supplied by the user, is applied to all tuples sharing the same key in one step. The output generated by the reducers form the result of the MapReduce job, which is written to the shared storage once the reducer has completed its work.

Example 3.5. *We complete our sample MapReduce task calculating the absolute transaction based support of all labels in a tree data set. The reduce function is invoked once for each node label. It obtains all intermediate tuples emitted by the mappers for that label and can then count the number of distinct tree identifiers. For label* b*, we obtain the four tuples* $(b, T1)$, $(b, T4)$, $(b, T4)$, $(b, T5)$. *The number of distinct tree identifiers and thus the absolute transaction based support of label* b *is 3.*

Google's implementation of the MapReduce framework is not publicly released. However, there are numerous realisations of the framework which are freely available. The most prominent free MapReduce framework is *Hadoop*[5], provided by the Apache

5 http://hadoop.apache.org/

Software Foundation and implemented in Java, which we will use throughout this thesis as well.

MapReduce frameworks handle the distribution, communication, and fault tolerance aspects of the system transparently to the user. In large scale deployments of MapReduce frameworks, host failures are the most frequent problem encountered. This includes software and hardware failures on a host, and (often transient) network problems making a host unreachable. Such problems can be solved without having to recompute the complete job for failures on both mappers and reducers. For hosts running a map task, a problem arises if the host becomes unavailable before all reducers retrieved their partition of the intermediate data. In that case, only this single mapper must be run again. For hosts running a reduce task, a problem arises only if the host becomes unavailable during processing. The results of the reducer are only written to the shared storage once it has successfully completed its work. Therefore, a failed reducer can simply be started again, possibly on a different host. All this failure handling takes place in the MapReduce framework and is transparent to the application developer.

We can map our frequent subtree mining workflow presented in Chapter 2 to sequences of MapReduce jobs in multiple ways. In the following, we will discuss possible mappings. Then, we will propose the *Pipelined MapReduce Framework*, which provides extensions to the MapReduce framework allowing us to run the frequent subtree mining workflow more efficiently than on plain MapReduce.

3.4 FREQUENT SUBTREE MINING WITH MAPREDUCE

Our goal is to run a frequent subtree mining application on MapReduce. As the frequent subtree mining workflow we presented in Chapter 2 is quite complex, different ways of mapping it to a sequence of MapReduce jobs are possible. We will analyse three possible mappings which result in different numbers of MapReduce jobs. The key for all illustrations in this section is given in Figure 3.3. Note that in the figures showing the mappings of our frequent subtree mining workflow to the execution framework, we only show one processor for each map and reduce task. The focus of the figures is the mapping of the tree mining workflow to the underlying framework, not the multiplicity of single processing tasks.

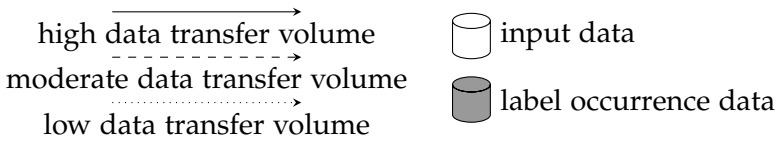
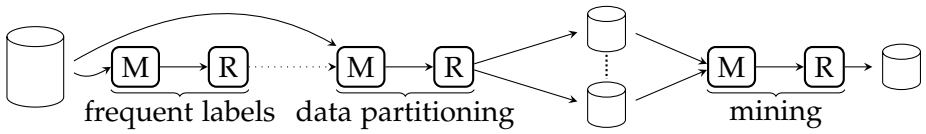
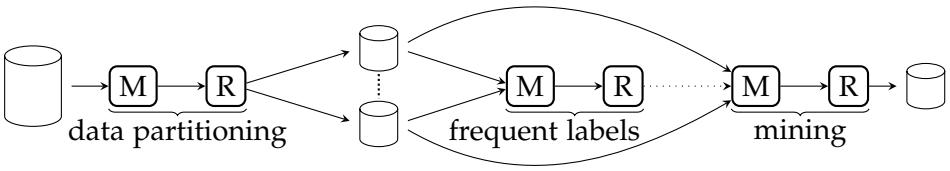


Figure 3.3: Symbols Used in the Workflow Realisation Figures



(a) Frequent Labels First



(b) Data Partitioning First

Figure 3.4: Straight-Forward Workflow Realisations

3.4.1 *Straight-Forward Approaches*

We derive the first, straight-forward workflow realisation as illustrated in Figure 3.4a by translating each of the phases of our distributed frequent subtree mining workflow to a separate MapReduce job and serialising the parallel phases such that the frequent labels are determined before partitioning the data.

We determine the frequent labels in one MapReduce job which extracts label occurrence information in its map phase. This information is distributed to the reducers using the label as the distribution key. In the reduce phase, we can then count the number of distinct trees a label occurs in. Note that this is the MapReduce job we constructed in Examples 3.4 and 3.5.

The second MapReduce job partitions the input forest into a configurable number of files, such that the data is treewise distributed. In the map phase of this partitioning job, we read the forest one node a time, parse it and hand it on to the reduce phase using the attribute (or attribute combination) indicating the tree the node belongs to as key. The reducer then needs to write all the values associated with the same key to the same output file. The files resulting from this MapReduce job may be reused by subsequent frequent subtree mining jobs as long as the original tree data does not change.

Eventually, in the third job, we start building the intermediate data structure employed by the actual frequent subtree mining algorithm in the map phase. The resulting structure fragments are distributed to the reducers, where we consolidate them in order to obtain the partitions of the intermediate representation each single reducer requires. Finally, we search for frequent subtrees exploiting the intermediate representation on the reducers.

Figure 3.4b shows an architecture variant consisting of the same three MapReduce jobs, but with a different serialisation of the parallel phases. Data partitioning is done before finding the frequent labels. Knowing the data is treewise distributed allows us to optimise the frequent label detection. As all of a trees' nodes are processed by the same mapper, we can eliminate duplicate label occurrences in the same tree already in the mapper. With this optimisation, we reduce the amount of data which needs to be transmitted from the mappers to the reducers in the MapReduce job determining the frequent labels.

While both of these straight-forward translations of our frequent subtree mining workflow to a sequence of MapReduce jobs show a very clear and comprehensible structure, the high number of jobs they are composed of introduces two severe drawbacks:

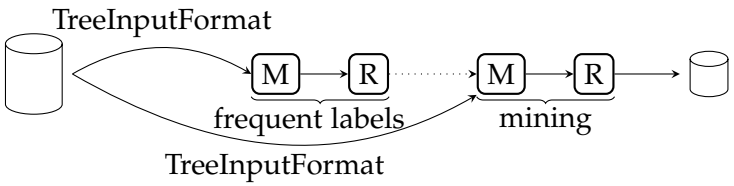


Figure 3.5: On-the-Fly Data Partitioning

1. We need to read the entire tree data set from disk as input for each of the three MapReduce jobs. Reading from disk is a quite expensive operation and may easily consume more time than the actual calculations for simple jobs like, e. g., the job extracting the frequent labels.

2. Especially in the latter of the two variants, where data distribution is done before frequent label detection, the second and third MapReduce jobs depend on the output produced by the job immediately preceding them, leading to two points of synchronisation. This will lead to a noticeable slow-down in processing.

However, since the data partitions are written to disk, they can be reused for subsequent mining tasks on the same data set, as long as the input data is not modified. If the partitioned data is already available, we can skip the partitioning phase, thus reducing the number of data reads by one.

An aspect to keep in mind is the number of files created by the data partitioning job. Obviously, we cannot create more files than there are trees in the data set, as a single tree must not be split over multiple files. A high number of files will lead to a large overhead spent on file system operations. Creating only a few files, on the other hand, limits the number of mappers we can create for subsequent jobs, as we assign entire files to mappers. Our experience shows that generating approximately ten times as many files as we expect the number of mappers to be is a good compromise.

3.4.2 *Partitioning Data on the Fly*

The straight-forward approaches presented above are able to reduce the data reading and synchronisation effort in the case of rarely changing data sets. Nonetheless, they remain expensive if applied to rapidly changing data or if a data set needs to be mined just once.

The architecture variant shown in Figure 3.5 reduces the number of MapReduce jobs to two even for data sets which have not yet been appropriately partitioned. However, this approach requires the input data set to be sorted such that all nodes belonging to the same tree are stored in an uninterrupted sequence. Then, we exploit the possibility offered by Hadoop to provide an *InputFormat* containing specialised instructions on how and where input files may be split. Using a customised InputFormat, the TreeInputFormat, we split the input file into partitions containing complete trees each. Thereby, we employ a binary search like approach to detect appropriate splitting positions in the input data. The two MapReduce jobs of this architecture variant are identical to the frequent labels and the mining job of the straight-forward architectures. The MapReduce job splitting the data in the straight-forward approaches is replaced by out TreeInputFormat and the condition that the tuples of the input data are sorted appropriately.

The goal of this variant is to reduce the number of MapReduce jobs and thus the number of times the tree data must be read from disk. Thanks to the binary search approach, the InputFormat only needs to read small parts of the data set. Using this architecture we hence need to read the data set twice completely, in the two map phases, and twice partially, in the TreeInputFormat. If the InputFormat caches its calculated splitting points, no tree data needs to be read when splitting the data for the second MapReduce job. Furthermore, as with the straight-forward architectures, cached results can be reused for subsequent invocations of the workflow – as long as neither the data nor the number of desired mappers changes. As opposed to the straight-forward variants, where partitioning was done in a MapReduce job, employing an InputFormat makes it impossible to speculatively create a higher number of partitions. Hadoop determines the effective number of mappers based on the number of partitions it receives from the employed InputFormat. Note also that Hadoop executes the InputFormat on one of the involved hosts only. As opposed to the straight-forward variants, the available massive parallelism is thus not exploited for partitioning the data set.

3.4.3 *2-Step Mining with Persistent Partitions*

The biggest advantage of the straight-forward workflow realisation variants over the variant with on-the-fly partitioning is the flexibility in the number of mappers employed for the actual mining MapReduce job. In order to regain this flexibility, we present the realisation depicted in Figure 3.6. We achieve a lower number of MapReduce jobs as compared to the straight-forward variants by reallocating the job deter-

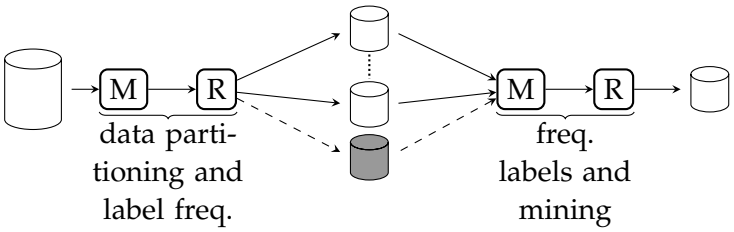


Figure 3.6: 2-Step Mining with Persistent Partitions

mining the frequent labels: Its map phase, which extracts the label occurrence lists from the data set, is integrated into the reduce phase of the data partitioning job. The reduce phase, aggregating the label occurrence lists in order to determine the frequent labels, is combined with the map phase of the last MapReduce job in the workflow, which extracts the intermediate structure fragments from the tree data.

Hence, in the first MapReduce job, besides partitioning the input data exactly as in the straight-forward variants, we determine label occurrences. As the reducers process entire trees at a time, we can easily extract all the distinct labels from a tree at that point. This label occurrence information is then pre-aggregated and stored in a separate file.

The second MapReduce job starts its map phase reading and aggregating the label information. It proceeds building the intermediate data representation and mining it, the same way as described for the previous architectures.

This variant reads the entire tree data set twice (once in each map phase). Additionally, at the beginning of the second MapReduce job, every mapper needs to read the pre-aggregated label occurrence list as well. As with the straight-forward approaches, once the data is partitioned, these partitions can be reused for subsequent data mining jobs unless the input data is modified. Reusing the label occurrence list is only possible as long as neither the input data nor the label definition (i. e., the set of attributes the label is composed of) changes. As opposed to the variants presented before, in this setting every mapper of the final MapReduce job needs to extract the frequent labels from the label occurrence lists. This is a very light-weight operation compared to the frequent subtree mining operations which are run in the remainder of this MapReduce job. Therefore, we consider this redundant calculation a viable solution.

Aspect	Architecture variant		
	straight forward (Figure 3.4)	on-the-fly (Figure 3.5)	2-step persistent (Figure 3.6)
data partitioning	once, distributed, persistent	twice, centralised	once, distributed, persistent
read tree data	three times entire set	twice entire set, twice partially	twice entire set
find frequent labels	once, distributed	once, distributed	on each mapper individually
synchronisation	twice	once	once
usage scenarios	fixed data set, changing label definition, changing cluster size	changing data set, changing label definition, fixed cluster size	fixed data set, fixed label definition, changing cluster size

Table 3.2: Plain MapReduce Architecture Variant Overview

3.4.4 *Discussion*

In this section, we presented possible realisations of our distributed frequent subtree mining workflow on top of MapReduce. All realisations consist of two to three MapReduce jobs. A realisation of the workflow as a single MapReduce job is not possible, as we need to exchange data twice: first for determining the frequent labels, then for distributing the intermediate structure fragments. However, within a single MapReduce job, we can exchange data only once.

Based on the preceding analysis, we can derive usage scenarios for each mapping to MapReduce jobs tailored to expected changes in the data set, the cluster size, and the label definition for our frequent subtree mining application. These are summarised along with other important aspects in Table 3.2. We will now further discuss the usage scenarios.

The straight-forward variant benefits from fixed data sets, as it reuses the data partitions which are stored to disk. If we create a reasonable number of partitions, we can flexibly react on changing cluster sizes without needing to repartition the data. The frequent labels are determined in a separate MapReduce task. If we want

to run multiple frequent subtree mining jobs using different minimum support values or different label definitions, we can reuse the data partitions.

The implementation using on-the-fly data partitioning needs to read only parts of the data set for extracting new partitions. This makes partitioning the data cheaper than for the other variants. However, it is necessary to recalculate the partitioning on both changes in the data set, and changes of the cluster size. Altering the minimum support or the label definition between subsequent mining jobs has no negative impact when using this workflow realisation.

The two-step variant with persistent partitioning again benefits from a fixed data set, as it reads the entire data for partitioning. Changes in the cluster size are flexibly handled if we create an appropriate number of partitions. As the label frequencies are determined together with the data partitioning, changing the label definition is expensive in this variant: it requires us to also recalculate the partitioning.

3.5 OPERATOR LIBRARY

Observing the four MapReduce translations presented in the previous section, we identify components appearing in more than one architecture. Effectively, some of them even appear multiple times within the same architecture (e. g., partitioning the input data using a dedicated InputFormat for tree data is done twice in the workflow realisation with on-the-fly data partitioning (Section 3.4.2)).

In plain MapReduce, the map and reduce functions are provided by the user and represent a black box to the framework. The same holds for the data deserialisation and serialisation routines, the InputFormat and OutputFormat, respectively, in Hadoop. Moreover, several components of the frequent subtree mining workflow, such as the mappers and reducers for splitting data or calculating frequent labels, or again the Input- and OutputFormat for treewise distributed data, are not specific to our application. We realise them in a generic, reusable manner and provide end-users with them in a scientific data processing library.

Note that the components of this library do not necessarily form a complete mapper or reducer each. Rather, they each represent a single data manipulation operation. An arbitrary number of such operators can then be combined to form a mapper or reducer. Consider, e. g., the BI phase of the distributed frequent subtree mining workflow. This phase consists of several logical steps. First, it combines the tree data with the label frequency information in order to retain only the frequent tree nodes. Next, it arranges the nodes according to the tree structure. Finally, it extracts the fragments of intermediate structure from these trees.

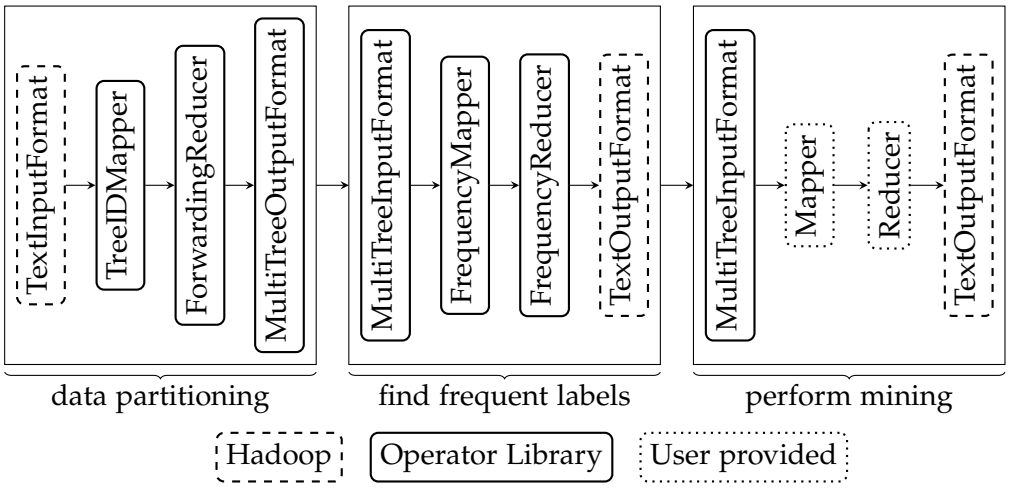


Figure 3.7: Modules of the Straight-Forward Workflow Realisation in Figure 3.4a

In our library, each of these steps is realised as a separate operator, similar to operators in relational database management systems. Each operator provides an elementary operation. By combining them, complex workflows can be constructed. In our example of the BI phase, the first step, i. e., combining the tree nodes with the label frequency information, is basically an equijoin of the two data sets on the label attribute.

The users of the operator library can hence focus on implementing just the core of the algorithm they want to run, plugging in ready-made components for common tasks. Figure 3.7 shows a possible module structure for the straight-forward workflow realisation depicted in Figure 3.4b. For all other workflow realisations we presented, similar module structures can be derived.

Assuming the input data to reside in text files and the results to be stored in text files as well, Hadoops out-of-the-box routines for reading and writing CSV files can be used for reading the input data to the first MapReduce job, and for storing the results of the last job which are also the results of the frequent subtree mining workflow. The *MultiTreeOutputFormat*, writing entire trees to a predefined number of files at the end of the data partitioning task, as well as the matching *MultiTreeInputFormat*, reading those files in subsequent tasks, are modules provided by our library. Likewise, the mappers and reducers for partitioning the tree data (the *TreeIDMapper* and the *ForwardingReducer*) and for determining the frequent labels (*FrequencyMapper* and *FrequencyReducer*) can be built using library components only. Both the *TreeIDMapper* and the *FrequencyMapper* just extract the key attribute for the subsequent data rear-

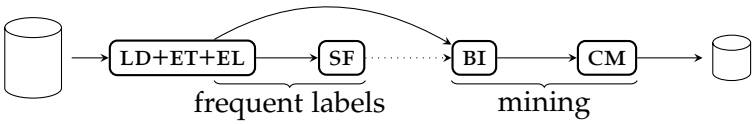


Figure 3.8: Pipelined MapReduce Workflow

rangement from each tuple. In case of the *TreeIDMapper*, the key is the tree identifier; for the *FrequencyMapper*, the node label is the key. Therefore, they will use the same operator from our library – the operator for setting the key of a (key,value) pair to some attribute from the value of that pair. The information on which attribute(s) to use form the configuration of the operator. For the third and final job, the reading and writing modules can again be provided by Hadoop or the mining library. Mapper and reducer implement the core of the mining algorithm employed and are thus the main user's contributions to the entire mining process. Nonetheless, some steps, like the join operation for combining the tree nodes with the label frequency information, can still be provided by the library.

3.6 PIPELINED MAPREDUCE: E-SCIENCE EXTENSIONS TO MAPREDUCE

We have seen that every mapping of the frequent subtree mining workflow to plain MapReduce has both its strengths and drawbacks. There is no mapping which suits all possible usage scenarios well. In the following, we will present three extensions to plain MapReduce which allow for an efficient mapping of the frequent subtree mining workflow for all usage scenarios considered. The extensions we propose allow for a more flexible number of processing steps within a job, and they permit to run multiple steps of a workflow overlapping each other. While we design these extensions with our frequent subtree mining workflow as a concrete application scenario in mind, they are not limited to this scenario. The goal of these extensions is to run the frequent subtree mining workflow as depicted in Figure 3.8.

3.6.1 *Multi-Step Jobs*

As major differences between our workflow and MapReduce, we identify the number and the ordering of the processing steps. While MapReduce consists of only two steps — map and reduce — with a strict sequential ordering, our workflow is composed of six steps (cf. Section 2.3.2), some of which could be executed in parallel if the

framework offered that possibility. As we have seen, it is possible to formulate our workflow as a series of MapReduce jobs. The performance of such a setup suffers, however, from the fact that the results of each single MapReduce job are written to the shared storage, even though all but the last job's outputs are only intermediate results which only need to be passed to the next processing step.

Jobs in the Pipelined MapReduce Framework may consist of more than the two processing steps permitted by plain MapReduce. This enables us to translate logically connected processing tasks to a single job instead of obeying the artificial limitation of having only two processing steps per job.

Example 3.6. *We realise the entire frequent subtree mining workflow as a single job on the Pipelined MapReduce Framework. This enables us to realise the workflow in a more efficient manner than with multiple standard MapReduce jobs. In particular, we do not need to store intermediate results to the distributed file system.*

Independent of the number of processing steps per job, we preserve the assumption of MapReduce that each processing step can be parallelised. The data items or item groups processed in each step are independent from each other. By partitioning the data and distributing it to all involved processing hosts, a notable speedup can be achieved.

3.6.2 *Multi-Input and Multi-Output Tasks*

As a consequence of allowing more than two processing steps, we have multiple intermediate result within a job. A processing task may thus be interested in gathering its input data from more than one of the preceding tasks. The Pipelined MapReduce Framework allows for such setups. If a processor is configured to have multiple data sources, each of the sources is presented to the processor as an iterator, as in the Sphere framework [Grossman and Gu 2008]. By selectively advancing or resetting these iterators, sequential, interleaved, or nested loop semantics can be achieved.

Analogously to a step consuming more than one input data set, it is possible for a processing step to generate more than one output data set. These data sets can then be used independently of each other, as if they were generated by different processing steps. We can thus reduce the number of processing steps by combining steps consuming the same inputs.

Example 3.7. *In our frequent subtree mining workflow, we merge the* ET *and* EL *steps, which are both not very CPU intensive. The label occurrences are thus counted in the same processing step as the tree membership of each node is identified. This reduces the number of data*

exchanges, the number of times the input data set must be read, and the number of processing steps of the job by one each and thereby allows us to solve the frequent subtree mining task more efficiently.

Note that MapReduce defines two slightly different programming interfaces for the map and reduce phases, as the mappers process single items, while the reducers work on item groups. Our framework does not distinguish between those phases. The interface of all processors is identical. Access to the input data sets is provided via a list of iterators, each representing one data set. If a processor is used like a mapper in MapReduce, its only input iterator provides access to a single data item. If used like a reducer, the iterator delivers all items of an item group. Analogously to the inputs, each processor has access to a list of data sinks for the generated results. For interaction with the framework, a reporting interface is provided.

Example 3.8. *Consider once more the* BI *phase of our distributed frequent subtree mining workflow. It obtains two inputs: the tree data set to analyse, and the label frequency information. These data sets are then joined in order to identify the frequent tree nodes. In standard MapReduce, such a multi-input task can only be realised using a trick. We need to add a new attribute to each tuple which indicates the data set that tuple originates from. According to this attribute, a* BI *phase processor can then separate the tuples of the two input data sets and process them accordingly. With Pipelined MapReduce, the two input data sets are two independent iterators to the worker, allowing for a clear and easy distinction between the different inputs.*

3.6.3 *Data Streaming*

A very important modification, compared to MapReduce, concerns the data exchange between subsequent processing steps. In MapReduce, the results of the map phase are stored to local disks. Writing intermediate results to disk severely impacts the performance of a processing step. In the Pipelined MapReduce Framework, we allow the pipelining of intermediate results to the subsequent processors as soon as they are available. This enables the receiver side processors to start their work as early as possible, thus speeding up the processing.

In case a processor cannot immediately digest the data it receives (e. g., if it requires two input data sets, but one of them is not yet available), the processor caches the data. Caching is done, if possible, using main memory. Only if the main memory available is too small, disks are used.

In the only data exchange phase defined by the MapReduce processing scheme, the data is distributed in a full mesh. Since the Pipelined MapReduce Framework allows

to have more processing phases, data can be exchanged more often within one job. It may not be necessary to have the data redistributed via a full mesh between all subsequent phases. In the case of computationally very intensive processing steps, e. g., one might decide to split this processing step to two (or more) processors, which are then sequentially concatenated. The data flow in such a scenario is one-to-one. A full mesh is, of course, the most general type of data exchange, and it can thus also be used for one-to-one communication or any other possible exchange pattern. However, this introduces unnecessary overhead, as we create a large number of communication channels most of which are not required. In order to avoid this overhead, the Pipelined MapReduce Framework provides a set of different communication patterns, including, e. g., one-to-one communication.

Example 3.9. *Consider the* SF *phase of the distributed frequent subtree mining workflow. In this step, we count the number of distinct tree identifiers for each label. We can start this aggregation for a label as soon as the first tuples with that label reach the worker, and then continuously aggregate the data as new tuples arrive.*

With the presented modifications to plain MapReduce, the Pipelined MapReduce Framework enables us to execute the distributed tree mining workflow as depicted in Figure 3.8, without having to write intermediate results to disk.

3.7 TOWARDS LOWER COMMUNICATION AND SYNCHRONISATION OVERHEAD: A PROBABILISTIC APPROACH

In order to further reduce the communication and synchronisation overhead incurred by executing our distributed frequent subtree mining workflow on MapReduce style frameworks, we consider replacing exact calculations by probabilistic ones. Our workflow contains two components which could be executed in a probabilistic manner: frequent label detection and the actual mining. The former component is independent of the actual frequent subtree mining algorithm, as all algorithms require a list of frequent labels in order to detect frequent subtrees. The latter component encapsulates the core of the actual frequent subtree mining algorithm. Replacing this component by a probabilistic approach would effectively result in a redesign of the frequent subtree mining algorithm.

We therefore pick the former component to investigate the impact of probabilistic approaches onto our distributed workflow. We are, however, aware that the communicated data volume in this component is quite low compared to the processed data sets. While we consider choosing this component viable for demonstrating the integration of probabilistic calculations into our workflow, we do not expect significant

savings in communication volume due to the choice of the component. We performed this work together with Anja Grünheid in terms of her bachelor's thesis [Grünheid 2009].

In its exact variant, the frequent label detection proceeds as follows. Every processor of the EL phase extracts label occurrence lists from its partition of the input data. Each list entry describes the occurrence of a specific label within a specific tree, i. e., it consists of the label and the tree identifier. The processors can remove duplicate list entries as, with transaction based support (see Section 2.2.2), multiple occurrences of a label within a tree have no impact on the support value. The label occurrence lists are then distributed to the processors of the SF phase using the label as the distribution key. Hence, all list entries for the same label are sent to the same SF phase processor. Counting the number of distinct tree identifiers per label, they can then determine the support of each label. Comparing the support values obtained to the minimum support threshold, which is an input parameter to the frequent label detection component, the frequent labels are identified and sent to the processors of the subsequent BI phase of our workflow.

The SF phase processors will typically receive a *relative* minimum support threshold from the user. Hence, we need to determine the number of trees in the data set in order to decide which labels are frequent. We achieve this by introducing a special label $\perp$ not occurring as a real label within the data set. For every tree identifier encountered, an EL phase processor emits a tuple with that tree identifier and $\perp$ as node label. This special label is then distributed to *all* SF phase processors. Counting the number of distinct tree identifiers for that label, the SF phase processors can then determine the number of trees in the data set.

We focus on two modifications to the frequent label detection, which aim at reducing the communicated data volume and synchronisation effort, respectively.

3.7.1 *Reducing Communication Overhead*

Consider the data communicated from the EL to the SF phase processors. Every processor of the EL phase sends out one data item for each label appearing in a tree. This information is required in order to determine the exact support of every label.

Nonetheless, labels appearing in only a few trees locally on one EL phase processor are less likely to be frequent than labels appearing in many trees. We exploit this fact, which is also used in distributed Top-K algorithms [Fagin et al. 2003], in order to reduce the communicated data volume. Every EL phase processor obtains a *local minimum support threshold* as an input parameter. During processing, we calculate

the local support of every label. We can perform this calculation efficiently in combination with the aforementioned duplicate elimination using a sort-based strategy. Occurrence information on labels which do not exceed the local minimum support threshold is not communicated to the SF phase processors.

3.7.2 *Reducing Synchronisation Overhead*

The second point we tackle is the cluster-wide synchronisation of the exact calculation required between the SF and the BI phase. The BI phase can only start when the SF phase is completed, as every BI processor requires the list of frequent labels to perform its work. Due to this synchronisation, many BI phase processors will begin their work almost simultaneously. Recall that, in the BI phase, the complete input data set is read — either from the ET phase processors in Pipelined MapReduce, or from disk in plain MapReduce. This may cause network congestion, as large parts of the data will typically be read from remote hosts.

We relax the synchronisation constraints in order to avoid this scenario. If the BI phase processors do not start (almost) simultaneously, reading the data is distributed over a longer time span and network congestion is less likely to occur. We achieve this goal by arranging the processors in groups, or *subsystems*. Each subsystem runs the frequent label detection independently from the others. This means synchronisation and communication is only necessary in between the processors of a subsystem. As soon as a subsystem completed the frequent label detection, its processors can start the BI phase.

In this approach, every subsystem only sees the share of input data assigned to its processors. The frequent label detection is thus based on incomplete data, and the list of frequent labels obtained may vary from subsystem to subsystem. According to the law of large numbers, the larger the subsystems are, the closer the frequent labels determined within the single subsystems will be to the correct result. This leads to a trade-off between exactness of the result and decoupling of the processors. With large subsystems, the frequent label lists will be close to the exact lists, but larger parts of the cluster need to be synchronised. With small subsystems, we have smaller groups of processors which need to synchronise, at the cost of less precise frequent label lists.

3.8 EXPERIMENTAL EVALUATION

In the following, we present an experimental comparison of realisations of our distributed tree mining workflow on the Pipelined MapReduce Framework and on plain

MapReduce to a "traditional", centralised implementation of the same algorithm. We analyse the scalability regarding both the data set size and the size of the compute cluster available for processing. Moreover, we evaluate the probabilistic frequent label detection.

In our analysis, we compare the performance of an implementation of PathJoin [Xiao et al. 2003] (cf. Chapter 2) running as a job on our Pipelined MapReduce framework and as a series of MapReduce tasks on Apache Hadoop to a stand-alone execution of the same algorithm. All the variants share the implementation of the core algorithm. Our PathJoin implementation is not meant to compete with the one of the original authors; we aim at comparing the scalability of the different distribution approaches presented in this chapter. For such an evaluation, a common code base for all variants is essential. Only this way, it is possible to isolate the distribution aspect and obtain measurement results which can be compared to each other reasonably.

The stand-alone execution is used as a baseline throughout the measurements. In order to handle the data sets which do not fit into main memory of one host, we process the data in partitions of roughly 2 GB in the stand-alone implementation. This approach reads the tree data set from the same distributed file system as the other approaches, and stores its results there. Temporary files are created on the local disk, like the intermediate results in MapReduce. Evaluating this approach, we note an interesting effect. Accessing the distributed file system for reading a complete data set from a single host gets slower with an increasing number of hosts the file system is distributed to. This effect, which is clearly visible in Figure 3.10, seems to be compensated by distributing the reading of the data to many hosts in typical MapReduce applications.

The workflow realisation on the Pipelined MapReduce Framework implements our distributed frequent subtree mining application as shown in Figure 3.8.

The implementation running directly on MapReduce realises the architecture introduced in Section 3.4.2, consisting of a series of two MapReduce jobs and using a specialised InputFormat for partitioning the data appropriately. We choose this architecture as we do not want our measurements to be influenced by persisted data partitionings or label frequency information.

The environment we conduct our tests on consists of a cluster of 16 identically configured virtual servers, each equipped with an Intel Core 2 Quad CPU running at 2.6 GHz, 7 GB of main memory and a dedicated 250 GB SATA II hard disk for storing the data sets. The servers run the 64-bit edition of RedHat Enterprise Linux 5.2. All servers are connected via 1 Gbps ethernet links to a central switch. We use Sun Java 1.6 Update 10. For Hadoop, we run a development snapshot version (subversion revision 711 482) as the performance of the Hadoop Distributed File System was im-

				fanout	
Size	trees	nodes	labels	avg	max
2 GB	38 261	4 983 332	6 179	1.07	7
8 GB	151 718	19 734 424	8 080	1.07	8
16 GB	303 436	39 468 848	8 080	1.07	8

Table 3.3: Data Sets Used in the Evaluation

proved by patches not contained in release versions at the time the experiments were conducted.

The data set we use in our measurements is a subset extracted from the Millennium simulation. The extracted subset consists of all trees whose root nodes' *np* (number of particles) attribute ranges between 1 000 and 1 500, occupying roughly 8 GB in CSV format. Starting from this data set, we created a larger one by duplicating each tree, and a smaller one consisting of 25% of the trees (chosen by a uniform random distribution). By duplicating each tree, we double the amount of data to process while keeping all other parameters constant. This allows us to measure the scalability of the of the distributed workflows with minimal interference from data set related parameters. The scaling of the data sets is chosen such that, for the largest data set the ratio of main memory to data set size (regarding the main memory available in the entire cluster) approximately reflects that of the Hadoop TeraSort [O'Malley 2008] setup. Table 3.3 summarises the relevant properties of the data sets.

In our experiments, we vary the data set size and the cluster size, in order to evaluate scaling along those two dimensions. We keep the minimum support fixed at 50% for the results we report on here. Modifying this parameter would show the scaling abilities of the algorithm employed, not of the frameworks. Measurements we conducted with lower minimum support values showed comparable scaling of the different workflow realisations.

3.8.1 *Scaling the Data Set*

In our first experiment, we evaluate the behaviour of the different workflow realisations on changing data set size. Figure 3.9 shows the total execution time of the different workflow implementations on our cluster of 16 hosts. The results for a cluster of 8 nodes are very similar.

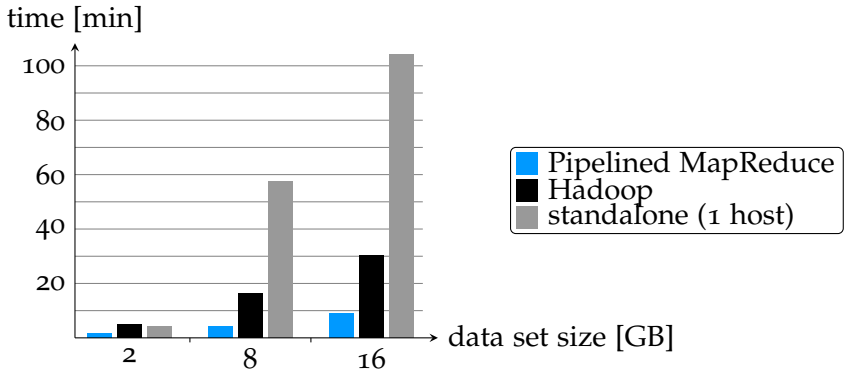


Figure 3.9: Scaling the Data Set on a Cluster with 16 Hosts

When comparing the results of the Hadoop variant and the Pipelined MapReduce variant, we can clearly see the benefits of the pipelined execution. The pipelined framework needs to read the data set only once, in the very beginning of the processing, whereas the Hadoop variant reads it twice completely and twice partially in order to calculate the splitting positions in the input file.

For the smallest data set, where the intermediate structure still fits into the main memory of one cluster node, the traditional approach is 40 seconds faster than the Hadoop variant. The Pipelined MapReduce realisation outperforms the stand-alone implementation even in this scenario, as reading the data set can be done in parallel on all 16 hosts on Pipelined MapReduce, whereas the traditional approach can only read sequentially on its only execution host.

3.8.2 *Scaling the Cluster Size*

In our second experiment, we analyse the impact of scaling the cluster size. Figure 3.10 shows the measured running time of our workflow realisations when mining the 8 GB data set on a cluster of 4, 8, and 16 hosts, respectively.

As expected, both the distributed approaches benefit from a higher number of available hosts. For the step from 4 to 8 hosts, we observe superlinear scaling of the Pipelined MapReduce framework, dropping from 12.3 minutes down to 5.5, whereas the Hadoop-based setup scales only sublinearly. We see two reasons why the Pipelined MapReduce framework outperforms Hadoop. On the one hand, the Pipelined MapReduce framework is able to keep the entire data in main memory, whereas Hadoop writes it to disk after each processing phase. On the other hand, the

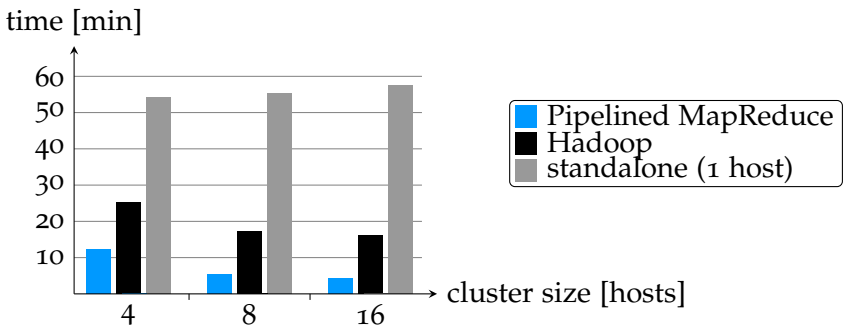


Figure 3.10: Scaling the Cluster Size for the 8 GB Data Set

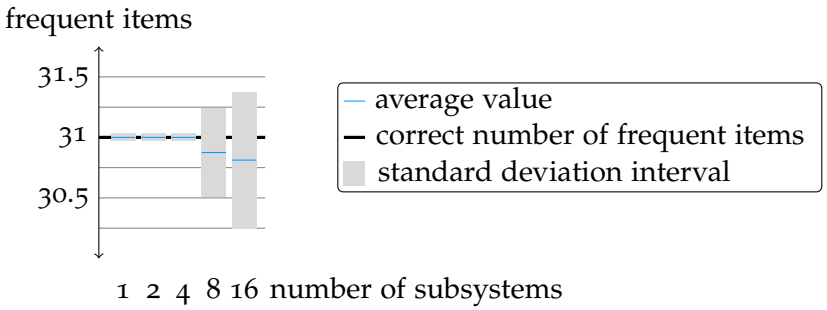


Figure 3.11: Approximation Quality of the Probabilistic Frequent Label Detection for the 16 GB Data Set on 16 Hosts

performance is affected by Hadoop choosing to run only one reducer in each MapReduce job, even though the maximum number allowed was configured as 1.75 times the number of hosts, as suggested by the Hadoop manual. The Pipelined MapReduce Framework uses all available hosts throughout the entire workflow.

3.8.3 *Probabilistic Frequent Label Detection*

Finally, we evaluate the impact of probabilistic frequent label detection. For the local filtering, we reuse the relative minimum support, i. e., every processor only transmits information on labels which appear in more than 50% of the trees encountered in its partition of the input data. This filtering allows a significant reduction of the communicated data volume. For the exact calculation, the communicated data volume is 60 kB. For the probabilistic approach, it is less than 1 kB in all evaluated settings.

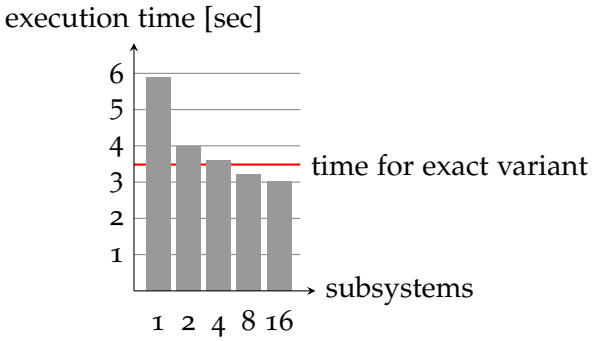


Figure 3.12: Execution Time of the Frequent Label Detection for the 16 GB Data Set on 16 Hosts

The impact of the probabilistic calculation on the exactness of the result is shown in Figure 3.11 for a varying number of subsystems on 16 hosts, using the 16 GB data set. Up to 4 subsystems, the probabilistic calculation introduces no error, i. e., even though a subsystem only sees down to 25% of the input data, the frequent labels are detected correctly. For a higher number of subsystems, some subsystems miss one of the frequent labels. Nonetheless, we feel the probabilistic results are close enough to the correct results to take the probabilistic frequent label detection as a viable alternative to the correct calculation.

Figure 3.12 shows the execution time of the probabilistic frequent label detection. As expected, the required time decreases with an increasing number of subsystems, because the number of hosts which must synchronise to each other decreases. The execution time especially for the setup with only one subsystem is larger than the time required for the exact variant. This shows the overhead introduced by the probabilistic approach, i. e., the subsystem handling and the local minimum support calculation.

3.9 RELATED WORK

Since the publication of the MapReduce programming model [Dean and Ghemawat 2008] by Google, numerous extensions to the framework were proposed.

Similar to the multiple processing steps in our Pipelined MapReduce Framework (Section 3.6.1), Nephele [Battré et al. 2010] and Dryad [Isard et al. 2007] abolish the strict two-step pattern of MapReduce. They allow arbitrary directed acyclic graphs for specifying the data flow in their applications. Besides file-based data exchange,

they allow for direct network channels (like our pipelining presented in Section 3.6.3) and in-memory communication.

Map-Reduce-Merge [Yang et al. 2008] extends MapReduce by a *merge* phase in which two data sets can be joined. The merge phase is similar to a processor in our framework receiving two input data sets. As opposed to Pipelined MapReduce, the ordering of the phases in Map-Reduce-Merge is fixed: first, both data sets pass their mappers, followed by the reducers, and only then they are merged. A workflow requiring the merging to be done first and running the map and reduce tasks on the joined data set must be realised by two subsequent Map-Reduce-Merge tasks, including three unnecessary processing phases. Moreover, in Map-Reduce-Merge, it is not possible to change the key of data items between the reduce and the merge phase. In Pipelined MapReduce, the keys may be changed in every processing phase, allowing for more flexibility in the composition of workflows.

HaLoop [Bu et al. 2010] extends the MapReduce model by the possibility to loop over a sequence of map and reduce phases until a fixpoint is reached. An iterative processing scheme could also be realised with plain MapReduce by writing an appropriate driver application. By integrating the loop control into the framework, however, data locality can be exploited better, and the termination condition can be checked more efficiently.

Sector and Sphere [Grossman and Gu 2008] form a MapReduce-like framework for data mining applications. In contrast to most other MapReduce-style frameworks which are targeted at cluster systems, they primarily focus on wide-area clouds.

Both MapReduce and database management systems (DBMSs) are used for large scale data analysis. Comparisons between the two systems [Stonebraker et al. 2010, Pavlo et al. 2009, Loebman et al. 2009] show that MapReduce systems excel in simplicity of installation and usage even in large-scale deployments. Distributed DBMSs, however, are often able to deliver better performance due to their highly optimised data access and processing patterns.

In order to improve over the drawbacks of each of the systems, some projects consider combinations of DBMSs and MapReduce. The Hadoop++ project [Dittrich et al. 2010] proposes indexing support for MapReduce. They introduce a data loading phase, in which indexes are built. Moreover, in this phase, the data can be pre-partitioned to allow for more efficient join operations.

HadoopDB [Abouzeid et al. 2009] replaces the distributed file system storage backend of MapReduce by local DBMSs installed on every host. This configuration overcomes the difficulty of properly installing distributed DBMSs. Moreover, HadoopDB allows to push initial calculations of a workflow into the DBMS backend. Thereby,

the advantages of DBMSs like, e. g., indexing support can be exploited for these calculations.

SQL/MapReduce [Friedman et al. 2009] integrates map- and reduce-style functions as new types of user-defined functions in distributed DBMSs. Data is handed to these new functions partition by partition. As the partitions can be processed independently of each other, the processing can easily be parallelised. Queries to the system are formulated in SQL. The new functions can be used like tables in the queries, allowing for an easy integration of MapReduce-style processing in DBMSs.

Several projects focus on optimising MapReduce applications. [Herodotou and Babu 2011] perform automatic tuning of Hadoop's job and system configuration parameters to suit a given application best. Manimal [Jahani et al. 2011] aims at optimising MapReduce applications using code analysis. They try to recognise the operations performed in the map and reduce functions and extract hints for preprocessing the input data set in order to speed up the actual processing.

4 SMALL-SCALE PARALLELISATION

4.1 INTRODUCTION

In Chapter 3, we presented the Pipelined MapReduce Framework as an approach to large scale parallelisation of (tree) processing. This framework allows users to build highly parallelised applications in a simple way. In this chapter, we will focus on parallel processing on a single host. Over the recent years, *multi-core CPUs*, combining multiple processing cores on one dice, became the state-of-the-art. Hence, modern commodity hardware provides the inherent capability of running multiple applications or threads in parallel.

Current MapReduce systems exploit these capabilities only in a very limited manner. If the framework is configured properly, one map or reduce instance is executed on each available core. While this is a straight-forward way of exploiting the available hardware's parallelism, it still leaves substantial room for improvements. We will discuss an alternative approach allowing better exploitation of the available resources in detail in this chapter.

4.2 MULTI-CORE PROCESSORS

For more than 30 years, Moore's Law [Moore 1965] captured the constantly increasing processing power of computer processors surprisingly well. With a growing number of transistors per chip, the clock speed could rise continuously. Only recently, this trend stopped, hitting most prominently the problem of heat emission.

Chip manufacturers thus switched to a new approach in order to further increase the processing capabilities of processors. Instead of raising the capacity of a single processing core, they started increasing the number of cores on a chip. While first processors in this area contained two processing cores, current commodity chips already integrate up to 16.

By increasing the number of cores per chip, the overall processing power obviously increases. However, from an application's point of view, there are significant differences between increasing the speed of a single core, and increasing the number of cores per chip. With the former approach, applications automatically profit from the

enhanced capabilities of a new chip. Due to the increased clock rate, commands take less time to complete, and the applications run faster.

With more cores available, however, applications will not automatically run faster, as most applications are designed to use just one processing core. In order to be able to run their code in a parallel fashion, they need to be modified manually. Only then, the increased number of processing cores can be exploited. Parallelising an application, i. e., modifying it accordingly, is a difficult and error-prone task. Therefore, many applications do not immediately profit from multi-core processors.

Additionally, processing cores are not the only component of modern processor chips. Very important additional components are two to three levels of caches, whose goal is to significantly speed up main memory access. With single-core processors, these on-chip caches were at the sole disposition of the one processing core on the chip. With multi-core processors, multiple cores share the same cache. The actual assignment of caches to cores depends on the processor design and ranges from dedicated caches for each core to caches shared by all cores on the chip.

Ideally, all the data a processing core is working on (its *working set*) should reside in cache. Otherwise, waiting times of several 100 cycles — so-called *cache stalls* — will arise, in which a core is waiting for data it requires to continue its work. However, the caches are quite small compared to typical modern main memory sizes. Table 4.1 exemplary shows the available cache levels and sizes of two modern multi-core processors. In order to fully exploit the processing capabilities of these processors, it is essential to have the working set readily available in the cache. The cache is filled asynchronously by a cache prefetcher. The prefetcher tries to detect patterns in the memory access of applications. It then places the items it expects the application to access next in the cache. The prefetcher must decide quickly which data items to load in order to keep up with the data processing speed. Therefore, it is only able to detect simple access patterns like, e. g., sequential access. It is thus essential for applications to access their data in a predictable and detectable way for the prefetching mechanism to accelerate the execution. This fact is emphasised also in [Albutiu et al. 2012]. They give three "commandments" for scalable multi-core processing, stating that main memory should be written (C1) and read (C2) sequentially, and that fine-grained locking should be avoided (C3). Commandment C2 underlines the importance of reading data sequentially such that the prefetcher can hide memory access latencies.

With the processing scheme of current MapReduce systems, the caches are not optimally exploited. Hadoop, e. g., can only be configured to execute one worker, i. e., a mapper or a reducer, per processing core. With this strategy, every core processes a different part of the data set. Thus, every core has a different working set. Due to

	Intel Xeon E3 1280	AMD Opteron 6160 SE
Cores	4	12
L1 Cache	64 kB per core	128 kB per core
L2 Cache	512 kB per core	512 kB per core
L3 Cache	8 MB shared by all cores	6 MB shared by 6 cores

Table 4.1: Cache Structure of Modern Server Processors

the limited cache sizes, the cores will swamp out each other's data from the caches, causing each other an increased number of cache stalls.

In the following, we will propose an alternative approach for exploiting multi-core processors for MapReduce style processing. This alternative takes into account the cache architecture of modern processors and is thus able to provide significantly better performance than current MapReduce systems. Thereby, it obeys the three commandments for scalable multi-core processing stated in [Albutiu et al. 2012].

4.3 INTER-OPERATOR PARALLELISM

MapReduce style frameworks offer automatic parallelisation of data-intensive applications on clusters. The data is partitioned and different partitions are processed by different workers, typically on different hosts of the cluster. Thereby, every worker is executed in a dedicated process.

The same approach is applicable to multi-core machines as well. One worker is executed on each processing core available, and each of them processes a different chunk of data. This solution is chosen, e. g., by Hadoop. If configured accordingly, it runs one worker process per processing core. Thereby, all available cores are exploited.

Recall that, in current MapReduce systems, both the map and the reduce algorithm are provided by the user as a pre-compiled binary, e. g., a Java class or a shared library. This binary is a black box to the execution framework. Running multiple workers in parallel is thus the only feasible way for the framework to introduce multithreading assuming the user provided code is single-threaded. As multi-core processors with caches shared between processing cores become ubiquitous, however, it may become beneficial to run a single worker in a multithreaded manner to exploit modern processor architectures best.

If the substeps of a worker are decoupled, each of them can be executed on a dedicated processing core. Thereby, we form a processing pipeline on the worker, with the cores sharing one working set. This intra-worker pipelining complements the inter-worker pipelining we introduced in the preceding chapter. The issue remaining to solve is how to detect the substeps of a worker and split the code appropriately.

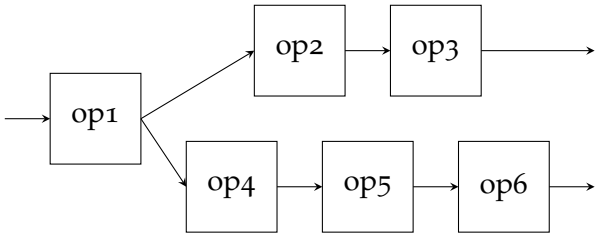
The operator library introduced in Chapter 3 exhibits an additional benefit here. The operators represent logically independent processing units. They employ tuple streams as their inputs and outputs, and can be orchestrated by linking one operator's output to the subsequent operator's input. Thereby, they naturally constitute processing units which can be executed in a multithreaded fashion in an operator pipeline within the worker. Single operators, or sub-chains of a workers complete operator chain, can then run in dedicated threads.

To support such execution, we introduce the *local multithreading operator*. This operator wraps the operator chain which runs within one worker. Each wrapped operator (or operator sub-chain) will then run in a dedicated thread within the wrapper. Subsequent wrappers are loosely linked via queues.

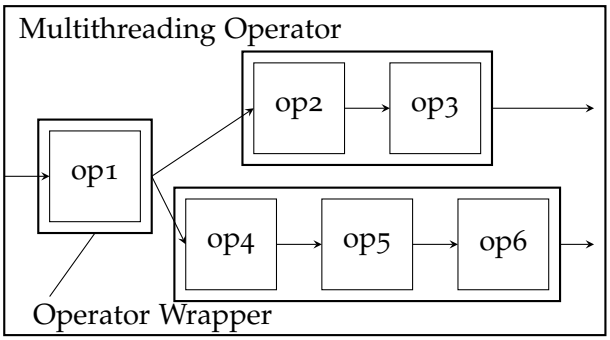
Example 4.1. *Consider the two variants of the same workflow sketched in Figure 4.1 which consists of six operators. This is the workflow for the first reducer of our frequent subtree mining application variant presented in Section 3.4.3. The first operator loads the data obtained from the mappers. The upper one of the two parallel sub-chains arranges the obtained nodes according to the tree structure (op 2) and stores these trees (op 3). On the lower sub-chain, we count (op 4), pre-aggregate (op 5), and store (op 6) the label frequency information.*

The workflow variant in Figure 4.1a runs in a single-threaded manner, with operators directly connected to each other. Figure 4.1b shows the same workflow, wrapped within the local multithreading operator. The entire workflow is surrounded by a single "container" operator. Within this container, the operators of the replaced workflow run in three threads.

The wrapper for a single operator is shown in detail in Figure 4.2. The wrapped operator (sub-chain) runs in a dedicated thread. We assume pull-based operators. Hence, the operator wrapper actively pulls tuples from the wrapped operator and puts them into the output queue. The wrapped operator, in turn, pulls its input from the preceding operator(s). Except for the operators loading data from disk, every operator has such a predecessor. It resides either within the local multithreading operator, or immediately precedes it. The data loading operators simply read data from disk a tuple a time. After processing an input tuple, the wrapped operator places the result tuple(s) in its output queue. The only exception to this scheme are the operators storing the results of a workflow to disk. As these operators do not



(a) Single-Threaded Workflow



(b) Multithreaded Workflow

Figure 4.1: Multithreading Wrapper for Pipelined MapReduce

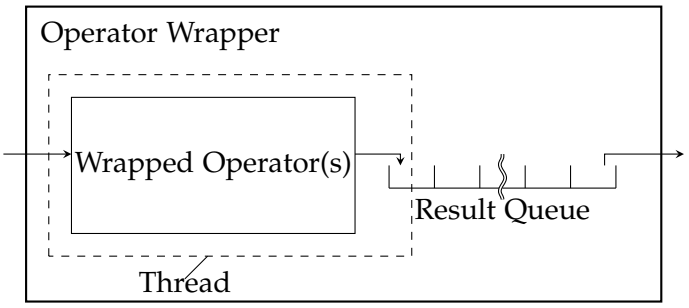


Figure 4.2: Operator Wrapper Details

produce output tuples for subsequent operators, the operator wrappers around these operators do not need to pull data.

We follow the three commandments for scalable multi-core processing in [Albutiu et al. 2012] for determining implementation details and beneficial usage scenarios for the local multithreading operator. We realise the queues linking the wrappers as arrays. This permits us to obey both commandments C1 and C2, as data is read and written in a strictly sequential manner when exchanged between threads. For the usage scenarios, we envision to run independent (i. e., parallel) operator sub-chains in dedicated threads. Thereby, we follow commandment C3. As the parallel sub-chains do not interfere with each other, no fine-grained synchronisation is required.

4.4 CACHE FRIENDLY DATA STRUCTURES

With the local multithreading operator introduced above, we can efficiently use the available processing cores and caches of modern multi-core processors building operator pipelines. Data items are loaded into the cache as they enter the pipeline, and ideally remain within the cache until they passed the entire pipeline.

For efficient processing on multi-core processors, it is also important to avoid cache stalls as new data items enter the working set. As already explained in Section 4.2, the cache prefetcher aims at detecting memory access patterns of the application and loads the data it expects the application to access next into the cache. The application can then read its data directly from the cache, avoiding waiting times of several 100 processing cycles while the data is transferred from main memory to the cache. With the clock rates of modern processors, the time the prefetcher can spend on the detection of memory access patterns is very limited. Therefore, the prefetcher is only able to detect simple access patterns.

Data in the caches is accessed a *cache line* a time. On modern processors, cache lines typically have a size of 64 bytes. At this same granularity, the cache prefetcher tries to detect access patterns for the prefetching.

In order for the prefetching to obtain the best results, the data must be accessed in patterns which are easy to detect on cache line granularity, e. g., sequentially. The data passed between operators and, even more important, between workers, should therefore be serialised appropriately. This is especially important for tree data. Object structures representing the trees one object a node offer comfortable access to the data from a user's perspective. Due to the heavy use of pointer structures in typical realisations of such structures, the low-level data access patterns produced by such structures can often not be detected by the prefetcher. They thus result in rather poor performance as compared to optimised low-level data structures. This is again emphasised by commandments C1 and C2 for scalable multi-core processing in [Albutiu et al. 2012], as well as by the micro-benchmarks motivating these commandments.

In our frequent subtree mining application, we represent the trees using a pre-order serialisation stored in a byte array. This allows us to sequentially read the tree data when building the intermediate structure in the BI phase of our workflow. Together with the local multithreading operator employed, e. g., in the situation described in Example 4.1, exploiting modern multi-core processors best possible.

4.5 RELATED WORK

Over the last years, analyses of the behaviour of MapReduce style frameworks on different platforms, including several multi-core systems, were presented.

The Phoenix project [Ranger et al. 2007] investigates MapReduce on single multi-core and multi-processor machines. In contrast to our work, they do not consider pipelining over the cores of a multi-core processor for avoiding cache conflicts. Every core runs an independent worker thread. They solve the caching issue by sizing working sets such that the data processed by all cores of a chip fits into the caches of that chip. With current cache sizes, this leads to working sets which are clearly smaller than 1 MB. Accordingly, for large data sets, the number of map and reduce tasks increases drastically, compared to standard MapReduce systems where the working sets of mappers typically range from 256 MB to 1 GB. Running each of these tasks in a dedicated thread would incur a huge thread management overhead. Phoenix diminishes this problem by running continuous worker threads and assigning the tasks to worker threads dynamically. Nonetheless, the work required for sorting and merging both the results of the job, and the intermediate (key,value) pairs, increases with the

number of map and reduce tasks. For the data sets used for evaluating the Phoenix system, which scale up to 1.5 GB, this overhead is still tolerable. For e-science scale data sets, however, we consider such small working sets infeasible.

The follow-up project, Phoenix Rebirth [Yoo et al. 2009], considers large-scale shared-memory systems with non-uniform memory access (NUMA) characteristics. In order to minimise the number of distant memory accesses, they create locality groups, which aim at keeping the processing close to the data. A work stealing approach is employed to balance the workload in case the locality groups would cause imbalanced system utilisation. Moreover, they adapt the internal data structures to suit the large number of concurrent threads in a NUMA system better. In contrast to our approach, which is targeted at a cluster of worker nodes, Phoenix Rebirth is tailored to one large system.

A number of publications discusses MapReduce on systems with mixed processor architectures, such as systems equipped with accelerator boards. As with multi-core processors, a central aspect in these systems is memory management.

The Mars framework [He et al. 2008] realises a MapReduce framework on graphics processors (GPUs). Modern GPUs are equipped with about 1 GB of memory which is connected at very high bandwidth to the processing cores. However, they have no automatic memory management, and no support for direct I/O to hard disks or network. Therefore, the Mars framework has to manually manage the memory of the GPUs. As a drawback of this solution, the size of the outputs of map and reduce tasks must be known beforehand in order to reserve a sufficient amount of memory for the results, and to avoid write conflicts between the processing cores of a GPU. The output sizes must be determined by the user of the framework. In a worst-case scenario, this could require to run each mapper and each reducer twice: first for calculating the result volume, and then for emitting the actual results.

Merge [Linderman et al. 2005] aims at distributing a MapReduce workload over processors of different architectures, e. g., GPUs and multi-core general-purpose processors. They employ a recursive divide-and-conquer approach for determining the size of working sets. Data sets are repeatedly split into smaller subsets until they can be handled by the available processing cores.

Besides GPUs, the Cell Broadband Engine (Cell/B.E.) has gained attention as an accelerator board for MapReduce applications. IBM proposes a MapReduce framework for their QS20 blade servers which comprise Cell/B.E. processors [de Kruijf and Sankaralingam 2009]. While the Cell/B.E. processors provide a more general instruction set than GPU cores, the memory must still be managed by software. In order to provide the Cell/B.E. processors with a continuous stream of data to process, IBM employs a double buffering approach for copying data between the system's

main memory and the memory on the accelerator. In their evaluation, they classify MapReduce applications according to the computational intensity of the phases of a job. Regrettably, they consider reduce-intensive applications — the common scenario in e-science — a rarely occurring configuration and do not evaluate them further.

An other project [Rafique et al. 2009] considers clusters of Cell systems. The considered environment consists of a powerful cluster head-node, orchestrating a set of worker nodes with possibly only small amounts of memory. They propose a management infrastructure of two layers, where the cluster head-node can delegate some management tasks to intermediate manager instances co-located with every worker node. Memory management is tackled by an adaptive partitioning approach. They assign working sets of varying sizes to the first mappers and reducers to start, and subsequently adapt the working set size in order to minimise the processing time per data element. In contrast to our local multithreading operator which allows to run a single worker using multiple threads, they run an independent worker on each processing core of the Cell system.

5

TREELATIN: A SCRIPTING APPROACH TO (DISTRIBUTED) TREE PROCESSING

5.1 INTRODUCTION

In the preceding two chapters, we described a pipelined distributed data processing framework with explicit support for multi-core worker nodes. This framework allows us to efficiently execute data-intensive applications processing both "flat" and tree structured data.

So far, the user interface to the framework consists of a Java API. The user provides classes containing the actual data processing instructions to execute on the workers. This represents a feasible low-level interface. However, it is not really comfortable especially for non-expert users, as they need programming experience in order to use the system.

The operator library we introduced in Chapter 3 is a first step towards a higher level interface. The users chain predefined operators in order to obtain the desired processing pipeline. Nonetheless, the users are still required to provide at least some glue logic to connect the operators. This becomes challenging especially in combination with the multithreading wrapper operator presented in Chapter 4, where fragments of the constructed workflow must be nested within the wrapper. Additionally, the operator arrangement within the constructed workflows must still be optimised manually.

We address these shortcomings in the following. We will define a scripting language, TreeLatin, with built-in support for processing tree structured data sets. With this language, we achieve two goals. First, it represents a user-friendly interface to the underlying execution framework. Second, it allows for automatic optimisation of the data processing pipeline while compiling the script into an operator chain.

5.2 PIG LATIN

Dedicated scripting languages for distributed execution have recently gathered substantial popularity. Typically, these languages offer a set of data manipulation instructions similar to that of relational database management systems, including pro-

jections, selections, grouping and joins. All these instructions are side effect free. With this precondition, the instruction set allows to only write applications which can automatically be executed in a massively parallel fashion. The simplicity of these languages which hide all parallelisation and synchronisation aspects from the user has made them a very popular data analysis tool of choice especially for many "Web 2.0" enterprises like Facebook, Google, Microsoft or Yahoo!. Among the first of these languages, there was Pig Latin [Olston et al. 2008, Gates et al. 2009], a project of the Apache Software Foundation originating from Yahoo!.

Pig, the framework around the Pig Latin language, includes a data model which helps users to keep track of their data along the processing steps. The data model comprises three basic data types.

ATOMIC VALUES Atomic values are the primitive data elements in Pig. They include integers, floating point values, single characters, and character arrays.
Examples: 1, 5.7, 'a', 'hello'

TUPLES Similar to entries in relations in a relational database management system, data items can be arranged in tuples. We will denote tuples by surrounding their elements with square brackets.
Examples: [1, 5, 'b'], [5.3, 'abcde']

BAGS Multiple data items can be placed in a bag or multiset. This roughly corresponds to tables in the relational model [Codd 1970]. In contrast to relational database management systems, the data items in a bag do not need to adhere to the same schema. It is possible to place tuples of different structure, possibly even mixed with plain atomic values, into one bag. We will denote bags by surrounding their elements by curly brackets.
Examples: { 1, 4 }, { 2.7, ['a', 'nested tuple'], { 'nested bag' } }

The schema of a data set is specified at load time, i. e., in the beginning of a Pig Latin script. Pig then transforms the schema as operations are applied to the data set, aiding the users to keep track of their data.

Pig Latin provides statements for basic operations, such as loading and storing data and performing projections, selections and grouping, as well as inner and outer equijoins.

Example 5.1. *Consider a sales data set with schema*

$$\{[customer, day, month, year, country, totalprice]\} \ .$$

For a statistical analysis, we are interested in the country and price of all orders from 2011. The Pig Latin script in Algorithm 5.1 extracts this data from the base data set.

Algorithm 5.1 Selection and Projection in Pig Latin

```
1: sales = LOAD '/sales' AS (customer,day,month,year,country,totalprice);
2: sales11 = FILTER sales BY year == 2011;
3: sales11ct = FOREACH sales11 GENERATE country, totalprice;
4: STORE sales11ct INTO '/sales11ct';
```

For this Algorithm, Pig tracks the following schema information. In the `LOAD` statement (line 1), the user provides the initial schema of the data set. The selection applied in line 2 does not modify the schema. In line 3, we apply a projection discarding all but two attributes. The resulting data set `sales11ct` thus has schema

$$\{[country, totalprice]\} \ .$$

Finally, the `STORE` operation does not create a new data set and hence no schema is derived.

For complex operations, user-defined functions (UDFs) can be included in almost any statement. Similar to relational database systems, UDFs can be used for calculations on single attribute values, or for computing aggregates. Moreover, UDFs in Pig Latin can return complex structures like tuples, bags, or arbitrary nestings thereof. Finally, UDFs can be used in the statements for reading and writing data from respectively to disk in order to support user specific file formats. By nesting bags of items within each other, we can build hierarchical data structures.

Pig allows to arbitrarily nest data items within each other. Not only is it possible to place arbitrary items in bags. It is also allowed to nest tuples and bags within tuples. We will exploit this possibility for representing the structure of tree data sets. This aspect of Pigs data model roughly corresponds to NF^2 models in databases [Schek and Pistor 1982].

Pig Latin's `GROUP` operation places all the tuples belonging to the same group within a bag. Thereby, the `GROUP` statement has different semantics than a `GROUP BY` operation in SQL. In SQL, the result of a `GROUP BY` operation contains one tuple for each combination of values of the attributes on which the grouping is performed. All attributes not present in the `GROUP BY` clause may only be used in aggregate operations. The single values are no longer accessible after the group operation was applied. Pig creates one tuple per group attribute combination as well. Each of these tuples gets a new, additional attribute besides the group attributes. This attribute is a bag holding

data set	schema
`sales11ct`	{[country, totalprice]}
`bycountry`	{[group, sales11ct: {[country, totalprice]}]}
`res`	{[country, volume]}

Table 5.1: Tracked Schema Information for Algorithm 5.2

all the input tuples belonging to that group. Not only is it henceforth possible to apply aggregation operations per group (as it is in SQL) in subsequent statements of a Pig Latin script, but every single tuple is preserved and may be further processed individually. Moreover, by preserving each single tuple, it is possible to revert grouping operations. We can do so by applying the `FLATTEN` operator. Applied to a bag of tuples, it removes this bag container, thus cancelling the grouping, and returns each contained tuple individually. As a side effect of this semantics of the group operation, in contrast to SQL, grouping data in Pig does not reduce the data volume as all tuples are preserved.

Example 5.2. *In Example 5.1, we prepared sales data for an analysis. Now, we are interested in the total order volume per country for the year 2011. In order to calculate the values of interest using Pig Latin, we first need to group the data by country. Then, we can sum up the prices of the single orders in each group. The Pig Latin script in Algorithm 5.2 calculates these values from the data set we generated in Example 5.1.*

Algorithm 5.2 Grouping and Aggregation in Pig Latin

```
1: sales11ct = LOAD '/sales11ct' AS (country,totalprice);
2: bycountry = GROUP sales11ct BY country;
3: res = FOREACH bycountry
       GENERATE group AS country, SUM(sales11ct.totalprice) AS volume;
4: STORE res INTO '/sales11bycountry';
```

The tracked schema information for this script is shown in Table 5.1. Note that the `GROUP` operation implicitly names the group attribute `group`. The name of the bag containing the grouped tuples is inherited from the input data set to the `GROUP` operation.

5.3 REPRESENTING TREE DATA WITH PIGS DATA MODEL

Pig Latin provides a comfortable way for defining a workflow for massively parallel execution using high-level procedural statements. The expressiveness of the readily available processing statements is perfectly suitable for scenarios like large scale log file analysis, where data is filtered, projected and aggregated. Scientific processing of tree structured data poses additional requirements on a language to allow for simple and comprehensible scripts. As this thesis focuses on processing tree structured data, we aim at processing this type of data in an equally comfortable way. There are two approaches for representing tree data in Pig in order to allow for reasonably simple processing.

1. We can group all the nodes belonging to the same tree using Pig's `GROUP` operation. Every subsequent operation then gets a bag of nodes representing a complete tree as input. Operations to which the tree structure is not relevant can then simply iterate over all tree nodes. Such operations comprise, e. g., counting the nodes of a tree, or summing up values of all nodes in a structure-independent manner. If the operations, however, need to work on the actual tree structure, they must arrange the tuples accordingly on their own. Summing up values for each bottom-up subtree is an example for such an operation.

2. We can explicitly build the actual tree structure exploiting Pig's nested data model. Then, subsequent operations do not need to arrange the nodes by themselves, but can simply traverse the tree exploiting the data structure provided as input. We choose this option for two reasons. First, it clearly separates the operations building the tree structure from the tree processing. Second, it allows for simpler realisations of tree processing operators which take into account the tree structure. We expect this to be the prevalent type of operators in tree processing applications.

5.3.1 *Nested Tree Data Representation*

Employing only the statements Pig provides out of the box, it is rather difficult to create hierarchical data structures of arbitrary depth out of "flat" ones, and to process them.

We represent tree nodes as tuples. All elements of such a tuple but the last one contain the actual node data. The last element of the tuple is a bag containing the child nodes of the current node. For leaf nodes, the bag is empty. This way, we obtain

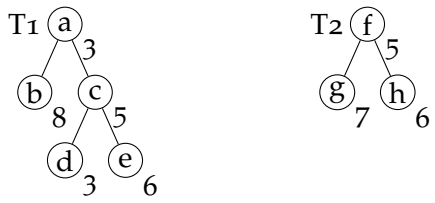


Figure 5.1: Sample Trees

a nested representation for subtrees. We represent an entire tree as a 2-tuple, with the tree identifier in the first component, and the tree's nodes, arranged as described above, in the second component. We name this representation of tree data *nested tree representation*.

Example 5.3. *Consider the two trees in Figure 5.1. The letter within each node is the node identifier. The value attached to the node is its weight (i. e., an additional attribute besides the node identifier available for each node). We represent these trees in nested tree representation as*

$$\left\{ \begin{bmatrix} T1, \begin{bmatrix} a, 3, \left\{ \begin{bmatrix} [b, 8, \{\}] \\ c, 5, \left\{ \begin{matrix} [d, 3, \{\}] \\ [e, 6, \{\}] \end{matrix} \right\} \end{bmatrix} \right\} \end{bmatrix} \end{bmatrix} \\ \begin{bmatrix} T2, \begin{bmatrix} f, 5, \left\{ \begin{matrix} [g, 7, \{\}] \\ [h, 6, \{\}] \end{matrix} \right\} \end{bmatrix} \end{bmatrix} \end{bmatrix} \right\} .$$

The nested representation explicitly reflects the tree structure. It allows us to directly map operations on the tree structure (e. g., finding all child nodes of the current node) to operations in the data model. This allows us to write intuitive and easily understandable tree processing statements.

The nested data representation also allows us to reduce the data volume. Parent-child relations are implicitly represented by nesting the child nodes within the parent node. Hence, there is no need to store references to parent and/or child nodes, as, e. g., in relational representations of tree structured data. Moreover, attributes which are constant throughout a tree need not be stored with every node, but only once per tree. An example for such an attribute is the tree identifier associated with every tree node.

5.3.2 *Schema Modifications*

Recall that Pig keeps track of the schema of data sets throughout a script. With our nested tree representation, this would require the schema information to be nested as deep as the trees.

Example 5.4. *Consider the trees in Example 5.3. Using Pig's metadata model, the schema of the tree data set is*

$$
\begin{array}{l}
\{[\mathit{treeid}, \mathit{nodegroup} : [\mathit{nodeid}, \mathit{weight}, \\
\quad \mathit{nodegroup} : \{[\mathit{nodeid}, \mathit{weight}, \\
\quad\quad \mathit{nodegroup} : \{[\mathit{nodeid}, \mathit{weight}, \\
\quad\quad\quad \mathit{nodegroup} : \{[\mathit{nodeid}, \mathit{weight}]\} \\
\quad\quad]\} \\
\quad]\} \\
]]\} \ .
\end{array}
$$

We see that the schema information becomes rather complex even for the small trees in this example. Moreover, the same tuple schema is repeated again and again. In order to avoid this nesting of metadata, we extend the tuple schema of Pig by a boolean flag to indicate nested tree representation. This flag is set on the tuple schema of the root node of a tree data set. It indicates the tuple contains, in its last element, a bag in which tuples of the same structure can be nested to arbitrary depth. We denote this flag by appending a T subscript to the tuple schema if and only if the flag is set. The element of the tuple containing the nested tuples is omitted when denoting the schema.

Example 5.5. *With this schema extension, the schema of the tree data set in Example 5.3 is*

$$\{[\mathit{treeid}, \mathit{nodegroup} : [\mathit{nodeid}, \mathit{weight}]_{\mathsf{T}}]\} \ .$$

5.4 TREE PROCESSING STATEMENTS

Although we can now intuitively represent trees using a slightly adapted version of Pig's data model, Pig Latin does not provide suitable statements for both bringing data into this format efficiently, and processing trees in this format node by node following the tree structure. Grouping and flattening operations add or remove only one level of nesting. There is no suitable loop statement for iterating over the nodes

of a tree, and no possibility of writing recursive scripts. Therefore, trees of previously unknown height cannot be handled using plain Pig Latin. In the following, we provide a set of new statements extending Pig Latin, which are focused on – but not limited to – handling trees. Due to its main focus, we call the resulting new language *TreeLatin*. The new statements are presented in the following.

5.4.1 *Constructing Trees*

For tree structured data, two storage layouts are common. The first is to store an entire tree within a single data item. The most representative example for this representation is XML, where each document is a tree. The second storage layout is to store each tree node separately. This layout is often used when trees are stored within a relational database management system. Each node typically contains, in addition to its payload, either a reference to its parent node, or a reference to the first child node and to the last descendant node (the *nested sets representation*).

One Tuple Per Tree

When loading tree data stored one tree per tuple, we can employ a UDF to transform the data into the representation introduced above. This UDF can be used either directly within the `LOAD` statement, or in a projection statement inserted between loading the data and processing the trees. The latter method can be advantageous especially if preprocessing steps can operate directly on the encoded representation.

Example 5.6. *Consider the tree data set shown in Figure 5.1. Suppose we have an XML document containing this data set, one tree per tuple, according to the following DTD.*

```
<!ELEMENT node (id, weight, node*)>
<!ELEMENT id #PCDATA>
<!ELEMENT weight #PCDATA>
```

We are interested in the number of child nodes each root node in our data set has. Algorithm 5.3 extracts this information from the data set in nested tree representation.

Algorithm 5.3 Filtering Data After Deserialisation

```
1: tree = LOAD '/xmltrees' USING XmlToTreeLoader;
2: rootchildcount = FOREACH tree
       GENERATE node.id, COUNT(node.node) AS childcount;
```

In a second application, we are only interested in trees containing nodes with weight 7. We can recognise these trees by string matching on the serialised data representation. Thereby, we avoid deserialising all trees we are not interested in. The script in Algorithm 5.4 solves this problem.

Algorithm 5.4 Filtering Data Before Deserialisation

```
1: treexml = LOAD '/xmltrees' AS (xmldata);
2: weight7 = FILTER treexml BY Contains(xmldata, "<weight>7<");
3: tree = FOREACH weight7 GENERATE XmlToTree(xmldata);
```

One Tuple Per Tree Node

If data is stored one node per tuple, we have multiple possibilities for transforming it into nested tree representation.

First, for a data set where the maximum tree height is known beforehand, we can use a series of Pig Latin's `GROUP` statements, constructing the trees level by level, as the following example shows.

Example 5.7. *Consider the trees in Figure 5.1. With node schema*

$$[nodeid, parentid, treeid, weight]$$

the data set containing all nodes from both trees is

$$\begin{aligned} nodes = \{ &[a, \perp, T1, 3], [b, a, T1, 8], [c, a, T1, 5], [d, c, T1, 3], \\ &[e, c, T1, 6], [f, \perp, T2, 5], [g, f, T2, 7], [h, f, T2, 6]\} \end{aligned}$$

where $\perp$ symbolises the empty parent node reference of the root nodes. The Pig Latin script in Algorithm 5.5 transforms this data set into nested tree representation (however, every node still contains the `parentid` attribute).

In line 1, we load the node data set. As we do not want to include the `treeid` attribute in every node, we create a copy of the node data set not containing this attribute in line 2. We group all nodes sharing the same parent node in line 3. Next, we join these node groups with the node data set bringing the groups together with their parent node (line 4). Using a left outer join, we also preserve nodes without child nodes. In line 5, we adapt the attribute names to the definition of the nested tree representation. The data set `cleaned1` contains all subtrees of height 1 and 2.

In lines 6 to 8, we repeat the statements from lines 3 to 5 in order to obtain subtrees of heights up to 2. For every additional level, we need to repeat those three lines of code again.

Algorithm 5.5 Transforming a Data Set in Nested Tree Representation using Pig Latin

```
1: fullnode = LOAD '/nodes' AS (nodeid,parentid,treeid,weight);
2: node = FOREACH fullnode GENERATE nodeid, parentid, weight;
3: nodegroup1 = GROUP node BY parentid;
4: level1 = JOIN node BY nodeid LEFT OUTER , nodegroup1 BY group;
5: cleaned1 = FOREACH level1
       GENERATE node::nodeid AS nodeid, node::parentid AS parentid,
           node::weight AS weight, nodegroup1::node AS node;
6: nodegroup2 = GROUP cleaned1 BY parentid;
7: level2 = JOIN node BY nodeid LEFT OUTER , nodegroup2 BY group;
8: cleaned2 = FOREACH level2
       GENERATE node::nodeid AS nodeid, node::parentid AS parentid,
           node::weight AS weight, nodegroup2::cleaned1 AS node;
9: complete = FILTER cleaned2 BY parentid == ⊥;
10: tree = COGROUP fullnode BY nodeid, complete BY nodeid;
11: nested = FOREACH tree
       GENERATE FLATTEN(fullnode.treeid) AS treeid, cleaned2 AS node;
```

With n iterations, we obtain all subtrees of heights up to $n+1$. Note that, in each iteration, we need to use different data set names, as every name might be used only once within a Pig Latin script.

The data set `cleaned2` which we obtain after two iterations contains all subtrees of heights 1, 2 and 3, i.e., both the trees T1 and T2, and all of their bottom-up subtrees. Filtering on the parent id of the root node in line 9, we preserve only the full trees T1 and T2. Finally, in lines 10 and 11, we add the `treeid` to each tree.

Besides the required prior knowledge of the maximum tree height, this approach suffers from the fact that Pig's current optimiser would generate one MapReduce task for each `(CO)GROUP` and `JOIN` statement. Hence, this approach will perform rather poorly.

Example 5.8. *Pig compiles the script in Algorithm 5.5 to a sequence of 5 MapReduce tasks, as it contains two `GROUP` statements, two `JOIN` statements, and one `COGROUP` statement. Every additional iteration, i.e., every additional level of the trees, adds two MapReduce tasks.*

A MapReduce system enhanced by a loop control mechanism, *HaLoop*, was presented in [Bu et al. 2010]. The HaLoop system allows to re-execute a sequence of MapReduce tasks until their results converge to a given fixed point, or for a predefined number of times. Moreover, data sets can be *pinned* to reducers for the duration

of a loop execution. A pinned data set is distributed to a set of hosts by partitioning it according to some attribute values, just like typical data redistribution between mappers and reducers. The pinned data set is then cached on these hosts, such that it needs to be distributed during the first loop iteration only. Later iterations prefer to use the same hosts, thereby avoiding to re-distribute the pinned data set. We can exploit this system for building our nested tree representation in a loop. Using a fixpoint as loop termination condition, we could even build trees of previously unknown height using HaLoop. Nonetheless, we still require one iteration per tree level, which is very expensive. A single, very high tree in the data set may cause an excessive number of iterations just to build the tree data structure.

Second, we can employ a single `GROUP` statement for bringing together all nodes belonging to the same tree, followed by a projection calling a UDF within the reducer. This UDF transforms a group of tuples into a tree in nested tree representation. We would thus avoid the problem of having one MapReduce task per tree level, as before. Depending on the implementation of the UDF, it is also possible to handle trees of arbitrary height with plain MapReduce. However, developing UDFs is more complex and more error-prone than using available language constructs. Moreover, we would need to generate both the result schema, and the actual nested data items manually within the UDF. Both these tasks are non-trivial and should not be burdened to the user.

We introduce TreeLatin's `FOREACH...NEST` statement as the third way for creating a nested tree data set. The statement takes a grouped data set as its input. Each group contains the nodes of one tree. We can create such a grouped data set using Pig's `GROUP` statement. The result of the `FOREACH...NEST` statement is a data set containing the trees in nested set representation. We define the syntax of the `FOREACH...NEST` statement as

<res> = **FOREACH** <ds> **NEST** <ng>
BY CONNECTING OUTER <att1> **WITH INNER** <att2>
[**TREES IDENTIFIED BY** <att3>]

Thereby, `att1` is the node identifier, `att2` holds the reference to the parent node, and `att3` is the tree identifier. This statement (including the optional `TREES` clause) transforms a data set

$$\text{ds} : \{[\text{group} : [\text{group attributes}], \text{ng} : \{[\text{att1}, \text{att2}, \text{att3}, \dots]\}]\}$$

(i. e., the result of a preceding `GROUP` operation) into the nested tree representation

$$\text{res} : \{[\text{att3}, \text{ng} : [\text{att1}, \dots]_T]\}$$

where "..." stands for an arbitrary number of attributes on each tree node in the input data set `ds` (i. e., the payload of each tree node) which are not touched by the nesting operation. The `ds` attribute of the resulting data set `res` is the root node of a tree in nested tree representation. The attribute `att2` is not part of the result. This attribute contained the "reference" to the parent node, which is now implicitly expressed by the data set structure. Note that, as we stick to Pig's naming scheme and name inheritance rules, each subtree in the `res` set will be named `ng`.

The optional `TREES` clause can be employed to extract attributes which have constant values within a tree, e. g., a unique id of the tree. The `res` data set shown above assumes the clause was present. If we processed the same data set omitting the clause, the result would have been

$$\text{res} : \{[[], \text{ng} : [\text{att1}, \text{att3}, \ldots]_T]\} \ ,$$

i. e., the attribute `att3` would still be contained in every tree node.

Example 5.9. *Consider again the trees in Figure 5.1 and the corresponding node data set* `nodes` *introduced in Example 5.7. In order to transform this data set into nested tree representation, we first need to group the nodes such that all nodes belonging to the same tree are contained in the same bag:*

nodegroup = ***GROUP*** *nodes* ***BY*** *treeid;*

We obtain the data set

$$\textit{nodegroup} = \left\{ \left[T1, \left\{ \begin{array}{l} [\text{a}, \bot, T1, 3] \\ [\text{b}, \text{a}, T1, 8] \\ [\text{c}, \text{a}, T1, 5] \\ [\text{d}, \text{c}, T1, 3] \\ [\text{e}, \text{c}, T1, 6] \end{array} \right\} \right], \left[T2, \left\{ \begin{array}{l} [\text{f}, \bot, T2, 5] \\ [\text{g}, \text{f}, T2, 7] \\ [\text{h}, \text{f}, T2, 6] \end{array} \right\} \right] \right\}$$

with schema

$$\textit{nodegroup} : \{[\textit{group}, \textit{nodes} : \{[\textit{nodeid}, \textit{parentid}, \textit{treeid}, \textit{weight}]\}]\} \ .$$

Employing the `FOREACH...NEST` *statement, we then transform the data into nested tree representation.*

ts = ***FOREACH*** *nodegroup* ***NEST*** *nodes*
BY CONNECTING OUTER *nodeid* ***WITH INNER*** *parentid*
TREES IDENTIFIED BY *treeid;*

The result of this operation is a data set named `ts`*. Its contents are shown in Example 5.3. They adhere to the schema we derived in Example 5.5.*

5.4.2 *Flattening Trees*

In some situations, it may become necessary to convert a data set from nested tree representation back to a flat representation. This might be the case, e. g., when a TreeLatin script is used to prepare data to be loaded into a database. We introduce the `FOREACH...UNNEST` statement to perform this task. It converts a data set from our nested tree representation back to a simple, grouped data set. The syntax is

<res> = **FOREACH** <ds> **UNNEST** <ng> **NODE ID IS** <att1>

With this statement, we could thus revert the data sets `res` from above back to `ds`.

Example 5.10. *Consider the data set* `ts` *in nested tree representation, shown in Example 5.3, we constructed in Example 5.9. By applying the operation*

*unnest = **FOREACH** ts **UNNEST** nodegroup **NODE ID IS** nodeid;*

to this data set, we obtain a new data set `unnest` *with the same contents and schema as the data set* `nodegroup` *from Example 5.9.*

5.4.3 *Processing Trees*

When working with trees, there are three frequent processing patterns.

1. A function is applied to an entire tree. In Pig Latin, this operation can be realised using the `FOREACH...GENERATE` statement on a data set in nested tree representation. As each tree is a single data item, we obtain the desired result.
2. A few nodes at well-known positions within the tree are accessed. With simple access patterns (e. g., all children of the root node, as in Algorithm 5.3), this is again possible using Pig Latin.
3. A function is applied to each single node of a tree, or the nodes to which a function must be applied can not be determined easily by position. We will focus on this scenario in the following.

If the processing is independent of the actual tree structure, processing trees node by node is also possible using Pig's `FOREACH...GENERATE`. We can then simply `UNNEST` the trees beforehand in order to obtain a data set containing all tree nodes, grouped by tree, and apply the processing on the unnested data set. If the tree structure is relevant to the processing — which we assume to be the common case for TreeLatin

applications —, this approach would ignore the benefits of the nested tree representation. This representation allows to intuitively traverse a tree. We exploit this property and introduce a loop statement and a corresponding operator for iterating over all nodes of a set of trees each represented in the nested data set structure.

<res> = **FOREACH** <ts> **FOREACH** <ng> [**TRAVERSE** <order>] **GENERATE** ...

This statement processes a tree data set `ts`, visiting each node of each tree in the order specified in the `TRAVERSE` clause. For every node, the `GENERATE` list is processed in order to obtain a result tuple, exactly like the "ordinary" `FOREACH...GENERATE` statement does for flat data sets. Depending on the performed task, different tree traversals might be necessary or preferable.

Example 5.11. *Consider an application which wants to annotate each tree node with its number of descendants. A bottom-up traversal is best in this scenario. We start with the leaf nodes, which have 0 descendants each. For each inner node, we can sum up the number of children and their already computed number of descendants.*

On the other hand, an application calculating the distance of each node to the root node of its tree will be best served with a top-down traversal. We start at the root node, which has distance 0, and proceed towards the leaves, calculating each node's distance as the distance of its already processed parent node plus 1.

The traversal can be specified using the optional `TRAVERSE` clause in the tree traversal statement. The nodes of every tree are then processed in the specified order. As can be seen in the preceding example, access to the results of previously processed nodes can be helpful. Therefore, we provide them as *context information* when processing a tree node. Depending on the traversal, different context information is available. The traversals predefined by TreeLatin are:

TOP-DOWN Using this traversal, processing starts at the root node of each tree and proceeds towards the leaves. No node is processed before its parent node has been processed. For sibling nodes, no processing order is defined. Nodes in disjoint subtrees of a single tree may even be processed in parallel, as long as their common ancestor nodes have been processed before. The context information for this traversal consists of the parent node and its processing result.

BOTTOM-UP This traversal starts at the leaf nodes of each tree and proceeds towards the root. A node is processed only when all its child nodes were already handled. Again, nodes in disjoint subtrees may be processed in arbitrary order, but they have all to be processed before processing their common ancestors. The

context information consists of all the current node's immediate child nodes and their processing results.

DEPTH-FIRST This traversal starts at the root node of each tree, and proceeds by traversing it in a depth-first manner. In contrast to the above traversals, the nodes of a tree are guaranteed to be visited sequentially. Sibling nodes are visited in the order they appear in the data representation, i. e., a sort order introduced by the user is respected by the traversal. (Note that, similar to the rows of a table in a relational database management system, the elements of a bag may be sorted.) Each node of a tree is visited twice: once when passing it on the way from the root towards the leaves below that node, and once on the way back. The context in this traversal consists of the `pass` value which indicates whether a node is visited for the first (`pass = 1`) or for the second (`pass = 2`) time, and the processing results already generated for the tree.

Additional traversals may be implemented by the user as necessary. The traversal handler obtains a tree and places single nodes in a processing queue as the preconditions for processing them are met.

If no `TRAVERSE` clause is given, the tree nodes are not visited in any particular order. It behaves thus similarly to processing each node of a tree independently, as described before. Using the `FOREACH...FOREACH...GENERATE` statement in such a scenario can, however, improve on processing performance even in this scenario, as there is no need to explicitly decompose the trees to single nodes beforehand.

For UDFs employed in TreeLatin's tree traversal statement, we extended Pig's UDF interface by an additional argument: a data bag providing context information.

From the `GENERATE` list, the context is accessible using the keyword `context`, analogously to accessing the current node, as the following example shows.

Example 5.12. *Consider the data set* `ts` *defined in Example 5.9. The following statement calculates the distance of each node to the root node of its tree. We start at the root node of each tree with distance -1+1=0 and increase the distance by 1 with every step towards the leaves.*

r1 = ***FOREACH*** *ts* ***FOREACH*** *nodegroup*
TRAVERSE *top-down*
GENERATE *nodeid, treeid, COALESCE(context.distance,-1) + 1 as distance;*

The resulting data set `r1` *is*

$$r1 = \{[a, T1, 0], [b, T1, 1], [c, T1, 1], [d, T1, 2], [e, T1, 2], [f, T2, 0], [g, T2, 1], [h, T2, 1]\}$$

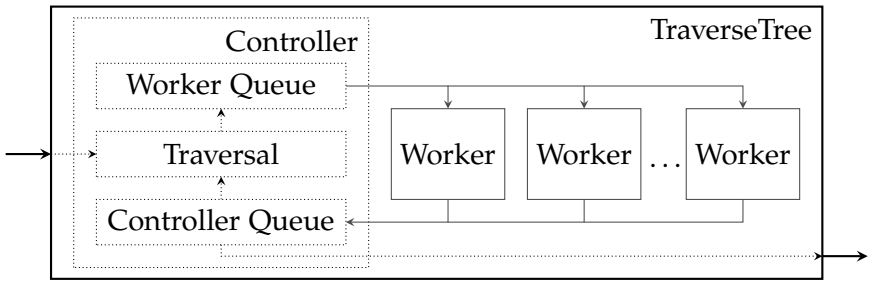


Figure 5.2: Tree Traversal Operator

From the same data set ts*, we now want to calculate the weight of every bottom-up subtree of each tree. For the leaf nodes, the subtrees' weight is just their own weight. Then, proceeding towards the root node, each subtrees' weight is its root nodes' weight plus the weight of all bottom-up subtrees rooted in the children of its root node.*

r2 = ***FOREACH*** *ts* ***FOREACH*** *nodegroup*
TRAVERSE *bottom-up*
GENERATE *nodeid, treeid, nodegroup.weight + SUM(context.weight) AS weight;*

$$r2 = \{[e, T1, 6], [d, T1, 3], [c, T1, 14], [b, T1, 8], [a, T1, 25], [h, T2, 6], [g, T2, 7], [f, T2, 18]\}$$

The operator implementing the tree traversal operation is sketched in Figure 5.2. It consumes one tree at a time, and outputs the processing results for each tree node.

The operator consists of one controller and a flexible number of workers. The traversal, representing the core part of the controller, obtains the tree to process. It schedules the nodes for processing as soon as the dependency constraints of the traversal allow to. Nodes are scheduled for processing by placing them, along with their context information, in the worker queue. The workers fetch items from this queue and process them. The processing results are then stored in the controller queue. From this queue, the controller learns which tree nodes completed processing, allowing it to schedule the next nodes for processing as their dependencies are met. Moreover, the contents of the controller queue form the processing result of the tree traversal operator.

Complementing the inter-operator multithreading presented in Section 4.3, the tree traversal operator enables intra-operator parallelisation. *Independent* nodes of the same tree can be processed in parallel by different worker threads. The notion of *independence* thereby depends on the tree traversal employed. For the bottom-up and

top-down traversals, bottom-up subtrees rooted in sibling nodes are independent. For the depth-first traversal, no two nodes of a tree are independent of each other.

Example 5.13. *Consider tree T1 in Figure 5.1 being processed using a top-down traversal. The first node to be scheduled for processing is the root node* a*. As soon as this node is processed, both its child nodes,* b *and* c*, are scheduled for processing. The bottom-up subtrees rooted in these two nodes can be processed independent of each other.*

5.5 MULTITHREADING BLOCKS

In Section 4.3, we introduced the inter-operator multithreading operator. This operator allows us to run chained operators within a worker using multiple threads. TreeLatin operators are similar to the library operators in that they are composed in operator chains in a workflow. Therefore, as with the library operators, we consider running the TreeLatin operators from a chain in dedicated threads each.

The multithreading operator is a container within which other operators are placed. Therefore, we denote it as a block containing the operator chain which is to be executed in a multithreaded manner.

```
<var> = {
  <statements>;
  RETURN <statement>;
};
```

Within such a *multithreading block*, all TreeLatin statements can be used as usual. There is, however, no data exchange between the hosts involved, but only between the threads of a single host. The last statement of a multithreading block is a `RETURN` statement instead of an assignment. The result of this last statement is stored in the variable the entire block is assigned to.

Example 5.14. *From the sales data set introduced in Example 5.1, we want to find our most valuable customers, i.e., customers being responsible for 5% or more of a month's revenue, along with the number of months in which they exceeded this threshold. The plain Pig Latin script solving this task is given in Algorithm 5.6. The resulting workflow is shown in Figure 5.3. We first calculate the total monthly revenue by partitioning the data set by month (line 2) and then summing up all the prices per month (line 3). Next, we partition the initial data set by customer and month in order to calculate each customer's monthly spending (lines 4 and 5). Then, we join the two resulting data sets bringing together the monthly revenue of each single customer with the total revenue of the corresponding month (line 6).*

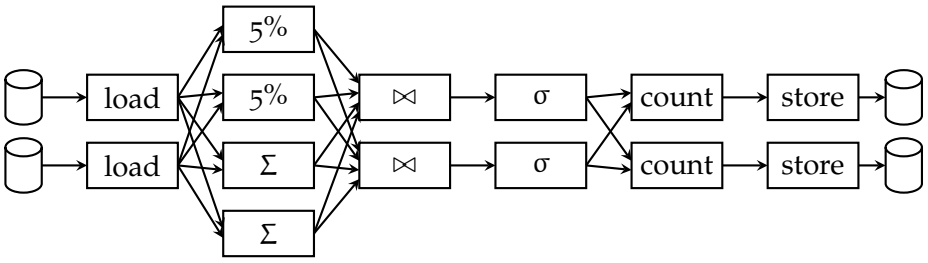


Figure 5.3: Finding the Best Customers with Pig Latin

Eventually, we select the relevant customers (line 7) and count the occurrences of each customer (line 8).

Algorithm 5.6 Finding the Best Customers with Pig Latin

```
1: sales = LOAD '/sales' AS (customer,day,month,year,country,totalprice);
2: a = GROUP sales BY month;
3: b = FOREACH a
       GENERATE group AS month, SUM(sales.totalprice)/20 AS pp;
4: c = GROUP sales BY (month, customer);
5: d = FOREACH c
       GENERATE group.month, group.customer, SUM(sales.totalprice) AS tp;
6: e = JOIN d BY month, b BY month;
7: f = FILTER e BY tp ⩾ pp;
8: g = GROUP f BY customer;
9: h = FOREACH g GENERATE group AS customer, COUNT(f.b::month);
10: STORE h INTO '/bestcustomers';
```

Note that data is partitioned five times in this scenario. The full data set is rearranged twice (lines 2 and 4). The join partitions both involved data sets on the join attributes (line 6). For counting the occurrences of each customer we need to repartition the selected customers (line 8).

With TreeLatin, we can solve this problem partitioning the data only twice. Thereby, we eliminate even one of the two partitionings of the full data set, which are the most expensive ones. The TreeLatin script is shown in Algorithm 5.7, and the resulting workflow is depicted in Figure 5.4. Once more, we begin by grouping our sales data by month (line 2). Then, we start a block for thread-level parallelisation. We calculate the total monthly revenue as before, but within the multithreading block (line 4). Then, instead of partitioning the input data again by customer and month, we exploit the partitioning by month we previously generated.

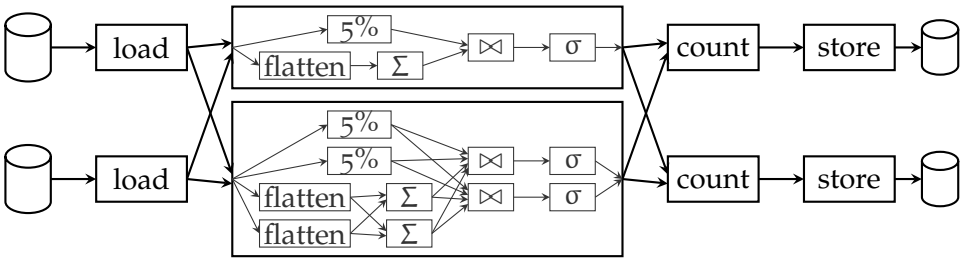


Figure 5.4: Finding the Best Customers with TreeLatin

As the grouping by month is more coarse-grained than a grouping by customer and month, we can refine the grouping without having to move any data between hosts (lines 5 and 6). Recall that within the multithreading block, no data is exchanged between hosts, so the `GROUP` *operator in line 6 groups locally available data only. Identical reasoning applies to the join. As the data is partitioned by month, which is our join attribute as well, the join can be executed without data exchange between hosts. Only for counting the occurrences of each customer, we need to leave the multithreading block and repartition the data by customer.*

Algorithm 5.7 Finding the Best Customers with TreeLatin

```
1:  sales = LOAD '/sales' AS (customer,day,month,year,country,totalprice);
2:  a = GROUP sales BY month;
3:  b = {
4:    c = FOREACH a
          GENERATE group AS month, SUM(sales.totalprice)/20 AS pp;
5:    d = FOREACH a GENERATE FLATTEN(sales);
6:    e = GROUP d BY (month, customer);
7:    f = FOREACH e
          GENERATE group.month, group.customer, SUM(d.totalprice) AS tp;
8:    g = JOIN f BY month, c BY month;
9:    RETURN FILTER g BY tp ⩾ pp;
    }
10: h = GROUP b BY customer;
11: i = FOREACH h GENERATE group AS customer, COUNT(b.c::month);
12: STORE i INTO '/bestcustomers';
```

5.6 COMPILING TREELATIN SCRIPTS TO MAPREDUCE EXECUTION PLANS

The main advantages of TreeLatin discussed so far are improved efficiency, as compared to PigLatin, introduced by our new statements, and user-friendliness. By using TreeLatin, users are enabled to accomplish many common tree processing tasks without writing a single line of Java or C code, as would be necessary with plain MapReduce. An additional, very important advantage is that, by using a higher level language as user interface, the system gains insight on the processing taking place. The code executed on the mappers and reducers is no longer a black box, but a sequence of well-known operators (possibly embedding some user-provided code fragments in form of UDFs for highly specialised operations). This enables the system to apply optimisations to the workflow while translating it into the actual operator chain. The optimising compiler for TreeLatin was realised together with two students. Ludwig Nägele realised the TreeLatin parser and schema derivation in terms of his bachelor's thesis [Nägele 2010]. The optimiser was implemented as a master's thesis [Hien 2011] by Michaela Hien.

In some aspects, optimising TreeLatin is similar to the optimisations relational database management systems apply to SQL statements. Many of the operations available in TreeLatin have corresponding SQL operators, e. g., selections (`FILTER`), projections (`FOREACH..GENERATE`), or equijoins (`JOIN`, `COGROUP`). Accordingly, similar optimisation techniques are applicable. For instance, operator re-orderings such as projection pushdown or techniques like eager aggregation [Yan and Larson 1995] can be used to reduce the data volume that passes through the operator chain.

Depending on the environment and application, relational database management systems may follow different optimisation goals. Common such goals are short overall execution time (in OLTP scenarios), low resource consumption (e. g., for databases running on embedded devices), or short time until obtaining the first results (e. g., for streaming applications). For TreeLatin, we will focus on short overall execution time as optimisation goal in this thesis. This is reasonable, as scientists running data analysis tasks on TreeLatin will typically want to obtain their results quickly, while, e. g., resource consumption on the workers is less relevant. Obtaining first results quickly is not possible with standard MapReduce due to the global synchronisation performed on every transition from map to reduce phase or vice versa, and due to the fault tolerance realisation of the reducers, making results available only when a worker completed successfully.

5.6.1 *Optimiser Overview*

In compiling TreeLatin scripts to optimised execution plans, we follow the rule based optimisation approach of relational database management systems [Graefe and DeWitt 1987]. In the first step, we translate a given TreeLatin script statement by statement into a canonical data flow graph. This graph corresponds to the unoptimised logical execution plan in database management systems with rule based optimiser. Then, the optimiser applies transformation rules to the graph. These rules may rearrange operators within the graph, introduce additional operators, or replace operator subchains by other, logically equivalent subchains which promise a more efficient execution according to the chosen optimisation goals.

In relational database management systems, the optimiser typically bases its optimisation decisions on comprehensive statistics over the processed data sets. Thereby, the system gracefully adapts its behaviour to the characteristics of the data it processes by ordering the operators appropriately, and by choosing from different implementations of semantically equivalent operators. Current MapReduce systems, in contrast, provide no statistics on the processed data. This makes it more difficult for the optimiser to take appropriate choices, as it has no information to base complex decisions on. The optimiser is thus limited to "best practice" and heuristic approaches. However, it still leaves a large space of optimisation possibilities.

5.6.2 *Group Refinement*

A very expensive operation in MapReduce is to switch from the map phase to the reduce phase, as this transition requires global synchronisation. A good execution plan should therefore contain as few operations as possible requiring such phase transitions. We classify the available operators into two groups: operators running locally, within a worker (i. e., a mapper or a reducer), and operators requiring global data rearrangement, thus causing a transition from map to reduce phase. This classification is given in Table 5.2. In order to achieve a short job execution time, our primary optimisation goal is to reduce the number of operations causing phase transitions within the data flow graph. Besides this primary goal, of course, the data volume passed through the operator chain should be as low as possible. This is our secondary optimisation goal.

All operators causing phase transitions rearrange the processed data set such that all tuples holding the same value in some attribute (combination) are sent to the same reducer. Therefore, we term these operators *data rearrangement operators*, and

Operators Running Locally	Operators Rearranging Data
`FILTER`	`COGROUP`
`FOREACH...FOREACH...GENERATE`	`CROSS`
`FOREACH...GENERATE`	`DISTINCT`
`FOREACH...NEST`	`GROUP`
`FOREACH...UNNEST`	`JOIN`
`LOAD`	
`STORE`	

Table 5.2: Classification of TreeLatin Operators

the attributes according to which the rearrangement is performed *data rearrangement attributes*. We will use these attributes to identify and remove data rearrangement operators which are not required as the data distribution at the point the operator is executed already fulfils the new partitioning criteria.

Example 5.15. *For a marketing campaign, we are planning to send out gifts to some of our customers. We devise different gifts, depending on the volume of orders the customers placed. Moreover, the gifts vary by country. The script in Algorithm 5.8 calculates how many of which gift items we need for each country. The Pig optimiser installs two data rearrangement operators to realise the join in line 3 and the grouping operation in line 5. The join partitions both the joined data sets by* `country`*. The grouping operation rearranges the filtered join result by* `country` *and* `gift`*. TreeLatin recognises the latter of these two operators is unnecessary. The available partitioning by* `country` *permits the creation of groups by* `country` *and* `gift` *without further data rearrangement. It is sufficient to group the items locally according to the group attributes. Therefore, with TreeLatin we can replace the data rearrangement operator for the grouping operation on* `country` *and* `gift` *by a local grouping operator (e. g., using hash partitioning, or a sort-based strategy).*

Now consider a more complex scenario. In a slightly modified version of Algorithm 5.6, we are interested in the number of customers exceeding our 5% threshold per month. This result is calculated by Algorithm 5.9. Instead of grouping by `customer` *in line 8, we now group the data by* `month` *and the monthly threshold value,* `pp`*. Semantically, this is equivalent to grouping the data set just by* `month`*, as the threshold value* `pp` *is calculated per month. We decided to include* `pp` *in the grouping attributes to allow for easier access to the value in line 9.*

Due to the `JOIN` *performed in line 6, the data is already partitioned appropriately and the group operation could be performed without data rearrangement. However, the grouping is not*

Algorithm 5.8 Recognising Group Refinements: A Simple Scenario

```
1: sales = LOAD '/sales' AS (customer,day,month,year,country,totalprice);
2: gifts = LOAD '/gifts' AS (country,gift,minprice,maxprice);
3: a = JOIN sales BY country, gifts BY country;
4: b = FILTER a BY totalprice ⩾ minprice and totalprice ⩽ maxprice;
5: c = GROUP b BY (sales::country, gift);
6: d = FOREACH c GENERATE group.country, group.gift, COUNT(b.customer);
```

performed on just the attributes used in the join. Only by taking into account the calculation of the `pp` *attribute and its way through the data flow graph, we can correctly recognise the data is already partitioned appropriately.*

Algorithm 5.9 Recognising Group Refinements: A Complex Scenario

```
1: sales = LOAD '/sales' AS (customer,day,month,year,country,totalprice);
2: a = GROUP sales BY month;
3: b = FOREACH a
       GENERATE group AS month, SUM(sales.totalprice)/20 AS pp;
4: c = GROUP sales BY (month, customer);
5: d = FOREACH c
       GENERATE group.month, group.customer, SUM(sales.totalprice) AS tp;
6: e = JOIN d BY month, b BY month;
7: f = FILTER e BY tp ⩾ pp;
8: g = GROUP f BY b::month, pp;
9: h = FOREACH g GENERATE group.month, group.pp, COUNT(f.customer);
10: STORE h INTO '/goodcustbymonth';
```

Example 5.15 shows two situations in which we want to recognise unnecessary data rearrangement operators which can be replaced by less expensive, local operations. The second example emphasises that just observing the data rearrangement attributes is not sufficient to detect all situations in which data rearrangements are not necessary. Therefore, we keep track of functional dependencies introduced by the operations along the data flow graph, and exploit this information for identifying data rearrangement operations which can be safely replaced.

We proceed as follows. Starting at the `LOAD` statements, we traverse the data flow graph capturing the functional dependencies introduced by operations like selections, calculation of attributes (as in the second part of Example 5.15), groupings, and joins.

We check for each data rearrangement operator if it can be replaced by a semantically equivalent operator chain not causing a phase transition. Replacing the operator is possible if the data set to be partitioned was already partitioned before, and all the partitioning attributes of this existing partitioning are implied by the partitioning attributes of the operator under consideration.

Example 5.16. *Consider again the script in Algorithm 5.9. The* `GROUP` *operation in line 2 introduces a functional dependency indicating the data partitioning is defined by the attribute* `month`*, and one indicating the* `group` *attribute determines the* `sales` *bag in each tuple. Line 3 renames the* `group` *attribute to* `month`*. Moreover, we learn that the* `pp` *attribute is determined by* `month` *as it is calculated from a bag of values all determined by* `month`*.*

The join in line 6 is performed on the `month` *attribute of that data set, so the partitioning is not modified. The functional dependency indicating* `pp` *is determined by* `month` *remains valid. We exploit this dependency in line 8 and recognise that the data is distributed appropriately for executing the* `GROUP` *operation without data exchange.*

By keeping track of functional dependencies, we are able to recognise all unnecessary data rearrangement operations performed on attributes whose functional dependencies become sufficiently clear from the preceding part of the workflow. Due to the reduction of data rearrangement operations, the processing speed of TreeLatin applications can be significantly improved.

Note that the parametrisation of the local multithreading operators, e. g., the choice of the actual number of threads to spread the operator to, is not performed by the optimiser. Instead, it is postponed to execution time. Thereby, we allow every worker to adapt the actual processing parameters to the locally available resources without requiring the optimiser to have knowledge on the worker nodes.

5.6.3 *Local Multithreading*

Moreover, reducing data rearrangement operations has advantageous side effects on the further optimisation. The TreeLatin optimiser wraps worker subchains in local multithreading operators. By replacing data rearrangement operators, global synchronisation points are removed. Thereby, the operator chains per worker grow longer, allowing to wrap longer workflow fragments in multithreading operators to exploit multi-core processing as described in the preceding chapter.

Moreover, users can explicitly introduce multithreading blocks using the syntax described in Section 5.5. Recall that, within a multithreading block, no data is exchanged between hosts. Placing data rearrangement operators in multithreading blocks, users

can thus explicitly forbid global data rearrangement even in situations where the TreeLatin optimiser cannot detect that data rearrangement is not necessary. Such situations may arise, e. g., when functional dependencies are implicitly contained in the data sets, but not expressed in the TreeLatin script. This behaviour allows the user to gain the advantage of local re-groupings even in situations where the system cannot automatically recognise their applicability.

5.6.4 *Bilateral Projection Push-Down*

Recall from above that our secondary optimisation goal was to reduce the intermediate data volume. The Pig optimiser applies several optimisation rules known from database management systems striving for this goal. These optimisations include transformations like selection and projection push-down. We apply these optimisations in TreeLatin as well, enhancing the selection push-down as follows.

The Pig optimiser inspects selections in `FILTER` statements and pushes them down as far as possible on the path in the data flow graph from the operator currently being processed back to the loading of the respective data set. In TreeLatin, in this situation, we exploit once more the functional dependencies captured for recognising replaceable data rearrangement operations. At `JOIN` and `COGROUP` operations, the functional dependencies provide information we can use to replicate selections and push them down on multiple input paths of the respective operator.

Example 5.17. *Consider again the script in Algorithm 5.8. Assume we are only interested in the number of gifts required in Germany. We therefore add an appropriate* `FILTER` *on the* `country` *attribute at the end of the script. In line 3, the two input data sets are joined on the* `country` *attribute. The functional dependencies introduced by this join allow TreeLatin to duplicate the* `FILTER` *and push it down on both inputs of the join operator.*

Once the optimisations are performed, the resulting data flow graph is translated into chains of physical operators to be run on the single workers, and submitted to the underlying framework for execution.

5.7 FREQUENT SUBTREE MINING WITH TREELATIN

TreeLatin enables us to realise tree processing applications on massive data sets in a clear and comprehensible manner. As an example, we use it to implement the frequent subtree mining workflow introduced in Chapter 2. The resulting TreLatin script is shown in Algorithm 5.10.

Algorithm 5.10 Frequent Subtree Mining with TreeLatin

Input: input tree data set, stored in files
minsupp: absolute minimum support for a label to be frequent
Output: frequent patterns found in input data set are stored on disk

```
1: node = LOAD '/nodes' AS (treeid, nodeid, label, parentid);
2: labelgroup = GROUP node BY label;
3: labelfreq = FOREACH labelgroup
       GENERATE group AS flabel, COUNTDISTINCT(nodes.treeid) AS cnt;
4: freqlb = FILTER labelfreq BY cnt ⩾ minsupp;
5: anode = JOIN node BY label, freqlb BY flabel USING 'replicated';
6: nodegroup = GROUP anode BY treeid;
7: tree = FOREACH anodegroup NEST anode
       BY CONNECTING OUTER nodeid WITH INNER parentid
       TREES IDENTIFIED BY treeid;
8: intermediate = FOREACH tree FOREACH nodegroup
       TRAVERSE bottom-up
       GENERATE user.IntermediateStructures(nodegroup) AS (key,structure);
9: intermediategr = GROUP intermediate BY key;
10: res = FOREACH intermediategr
       GENERATE user.CombineAndMine(intermediate, minsupp);
11: STORE res to '/result';
```

This script reflects the processing steps of our distributed frequent subtree mining workflow introduced in Section 2.3.2 (page 21). We load the data (LD phase) in line 1, extract the frequent labels in lines 2 to 4 (EL and SF phases). We combine the frequent label information and the tree nodes in line 5 and determine the tree membership of each node in line 6 (DT phase). Then, we extract the intermediate structure fragments and redistribute them appropriately (BI phase, lines 7 to 9). Eventually, lines 10 and 11 constitute the CM phase.

The tree traversal to choose for building the intermediate structure (line 8) depends on the frequent subtree mining algorithm employed. For PathJoin and TreeMiner (Section 2.4.1), we choose a bottom-up traversal and a depth-first traversal, respectively. For PathJoin, we have the fragments of the *compressed forest* built from the child nodes of a node readily available in the context information when processing a node. The *scope lists* of TreeMiner resemble the nested sets representation of trees. With a depth-first traversal, we can assign the left id when we encounter a node on the way down from the root towards the leaves, and the right id on the way back up towards the root.

Note how the abstract structure of the frequent subtree mining workflow is completely expressed with TreeLatin statements. The script in Algorithm 5.10 is suitable for all algorithms mining for frequent induced subtrees. For frequent embedded subtrees, only one detail needs to be changed. As infrequent inner nodes can be skipped in embedded subtrees, we must ensure the complete trees are constructed in line 7. Therefore, we must use a left outer join in line 5, as infrequent inner nodes must be preserved.

For both induced and embedded subtrees, only the fragments specific to the actually employed frequent subtree mining algorithm are encapsulated in user defined functions. Recall from Chapter 2 that the data structures used for the intermediate data representation are specific to the actual frequent subtree mining algorithm. Therefore, expressing these parts of the algorithm in a generic manner is not possible.

5.8 EXPERIMENTAL EVALUATION

In order to evaluate the impact of the TreeLatin optimisations on our frequent subtree mining application, we reran the experiments of Chapter 3, comparing TreeLatin to Pig.

The TreeLatin script presented in Section 5.7 is compiled to the pipelined MapReduce workflow we already benchmarked in Chapter 3. For Pig, we wrote a comparable script using user defined functions instead of the tree handling statements of

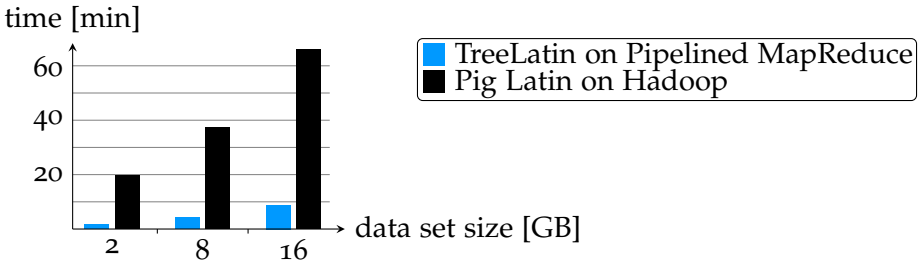


Figure 5.5: Scaling the Data Set on a Cluster with 16 Hosts

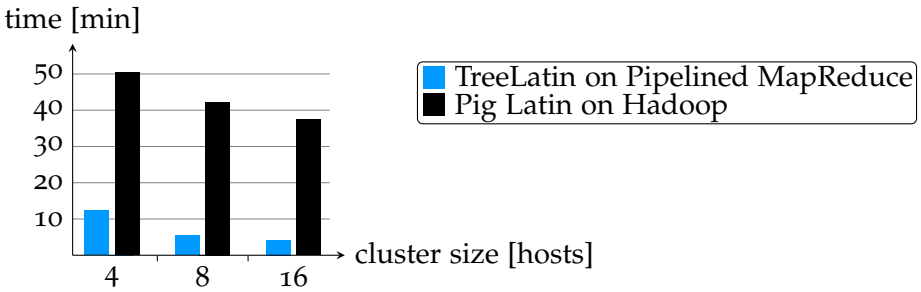


Figure 5.6: Scaling the Cluster Size for the 8 GB Data Set

TreeLatin. This script compiles to the workflow we presented in Section 3.4.1 (Figure 3.4a).

For this evaluation, we used the same infrastructure and data sets as for the measurements presented in Chapter 3. For Pig, we chose a recent snapshot version at the time we ran the experiments (SVN revision 709206).

Both TreeLatin and Pig show the expected scaling behaviour with respect to both data set size (Figure 5.5) and the number of hosts (Figure 5.6). The processing times of TreeLatin are, however, significantly lower than those of Pig. Besides the benefits of pipelining, this difference arises from the optimisations TreeLatin employs to reduce the communication and synchronisation overhead.

5.9 RELATED WORK

Accompanying the advent of MapReduce style frameworks (Chapter 3), several scripting languages for general-purpose data parallel processing on top of these frameworks have been proposed over the last years. We have already discussed Pig [Olston

et al. 2008, Gates et al. 2009], which provides the foundation for TreeLatin, throughout this chapter.

Similar to TreeLatin, optimisation in the Nephele/PACTS [Battré et al. 2010] system aims at reducing the number of data rearrangement operations. The approach they follow uses annotations on data rearrangement operators indicating whether the current data partitioning already fulfils the requirements of the operator or not. If all operators have complete and correct annotations, Nephele/PACTs can recognise all situations in which data rearrangement operations can be replaced. Our automatic detection of data rearrangement operations which can safely be replaced by local operations corresponds to deriving these annotations automatically.

Hive [Thusoo et al. 2010] builds a data warehouse on top of Hadoop and includes a query language, HiveQL, which resembles SQL. Similar to Pig, several optimisations are applied translating queries to execution plans. Besides the optimisations applied by Pig as well, the Hive optimiser can reorder join operations such that the largest involved tables are streamed through the operators. This allows a reduction of the memory consumption as the largest tables are not materialised within an operator to perform, e. g., hash lookups.

Google presented Sawzall [Pike et al. 2005] and Tenzing [Chattopadhyay et al. 2011], both building on top of their MapReduce implementation. The former provides a rather low-level language which clearly exhibits the underlying MapReduce processing framework. Tenzing, in contrast, offers a comprehensive SQL interface. Queries are translated to sequences of MapReduce jobs applying several optimisations which focus especially on data locality and low memory consumption. Similar to the preceding systems there are two types of optimisations. Some, e. g., projection push-down, are applied blindly as they will improve the resulting execution plan in most situations. Other optimisations, like the choice between different join implementations, are explicitly requested by the users by including hints in the queries.

SCOPE [Chaiken et al. 2008, Zhou et al. 2010] provides an SQL-like interface to the Dryad [Isard et al. 2007] framework. Workflows are specified as sequences of SQL queries, where each query may use outputs of preceding queries as its input. Moreover, functions written in C# can be embedded directly within a SCOPE script. Optimisations applied when translating SCOPE scripts into Dryad workflows primarily focus on reducing the number of data partitioning operations.

6 HANDLING DATA SKEW

6.1 INTRODUCTION

In the preceding chapters, we presented approaches for parallelising and distributing data-intensive and computationally intensive tasks in cluster environments. An important question in parallelisation is how to distribute the data to the processing hosts, i. e., which subsets of the data are processed where. The inherent data skew of many scientific data sets, combined with the complex analysis algorithms, makes this an interesting and challenging task, which we will discuss in this chapter.

Data skew in e-science data sets arises from physical properties of the observed objects (e. g., the height of patients in medical studies), from research interests focussing on subsets of the entire domain (e. g., areas with active volcanoes in geosciences), or from properties of the instruments and software employed to gather the data. In the Millennium simulation, each tree node has a mass. The mass distribution is highly skewed, with the 7 most frequent values appearing over 20 million times each, while almost 75% of the values appear no more than 10 times.

In the map phase, MapReduce systems generate (key,value) pairs from the input data. A cluster is the subset of all (key,value) pairs, or tuples, sharing the same key. Standard systems like Hadoop use hashing to distribute the clusters to the reducers. Thereby every reducer gets approximately the same number of clusters. For skewed data, this approach is not good enough since clusters may vary considerably in size. When the runtime complexity of the reducer is superlinear — a very common scenario in e-science —, the problem is even worse. Sets of clusters (called *partitions* subsequently) with the same overall number of tuples can have very different execution times since the non-linear reduce function is evaluated for each cluster. The processing time for a small number of large clusters is much higher than the processing time for many small clusters.

Example 6.1. *Assume a set of clusters consisting of nine tuples. The cost of a cluster is the number of tuples squared. If the nine tuples belong to one cluster, its cost is* $9^2 = 81$*. If the set consists of three clusters with three tuples each, the cost is only* $3 \cdot 3^2 = 27$*.*

If the cluster cost is exponential in the number of tuples, the difference grows even larger. For the set consisting of one large cluster, we obtain $2^9 = 512$*, while the set consisting of three clusters only costs* $3 \cdot 2^3 = 24$*.*

In this chapter, we design a new cost model, the *Partition Cost Model*, which takes into account non-linear reducer functions and skewed data distributions. Instead of considering only the size of the data partition that is assigned to each reducer, we estimate its processing cost. This is a challenging problem since a single cluster may be produced by different mappers in a distributed manner. Computing detailed statistics for every cluster is too expensive since the number of clusters may be proportional to the data size in the worst case. We propose *TopCluster*, a distributed monitoring technique, for collecting statistics on which we can base cardinality estimation and, consequently, cost estimation.

We design two new algorithms that use our cost model to distribute the work load to reducers. The first algorithm, *fine partitioning*, splits the input data into a fixed number of partitions. Contrasting to standard MapReduce, we choose the number of partitions to be larger than the number of reducers. The goal is to distribute the partitions such that the execution times for all reducers are similar. Fine partitioning does not control the cost of the partitions while they are created, but achieves balanced loads by distributing expensive partitions to different reducers. In our second approach, *dynamic fragmentation*, expensive partitions are split locally by each mapper while they are created, and tuples are replicated if necessary. As a result, the cost of the partitions is more uniform and a good load balancing is easier to achieve for highly skewed distributions.

6.2 DATA SKEW IN MAPREDUCE

From a data-centric perspective, a MapReduce system works as follows. m mappers transform the input to a MapReduce job into a bag of (key,value) pairs, the *intermediate result* $I \subseteq \mathbb{K} \times \mathbb{V}$. Thereby, $\mathbb{K}$ and $\mathbb{V}$ represent the key and value domains of the intermediate result, respectively. The sub-bag of I containing all (key,value) pairs with a specific key k is a *cluster*

$$C(k) = \{(k, v) \in I\} \qquad k \in \mathbb{K} \ .$$

The intermediate result is then split into p *partitions*. The partition for an intermediate tuple is determined by applying a partitioning function

$$\pi : \mathbb{K} \to \{1, \ldots, p\} \qquad k \mapsto \pi(k) = j$$

to the key of the tuple. This way, all tuples belonging to the same cluster, i. e., sharing the same key, are placed into the same partition. A partition is thus a "container", or bucket, for one or more clusters. We denote a partition with index j as

$$P(j) = \biguplus_{k \in \mathbb{K} : \pi(k) = j} C(k) \ .$$

Finally, the partitions are distributed to r reducers which produce the output of the MapReduce job. All partitions assigned to the same reducer form a *partition bundle*.

A workload balancing data distribution algorithm tries to assign the clusters such that all reducers will require roughly the same time for processing. There are two aspects which need to be considered.

NUMBER OF CLUSTERS Some reducers might get more clusters than others, leading to larger partition bundles and longer execution times.

DIFFICULTY OF CLUSTERS The execution times may vary from cluster to cluster. Reducers with "difficult" clusters might take much longer to complete than reducers with "easy" clusters, even if the number of clusters in the partition bundles is the same.

The first of these two points can be solved by using an appropriate hash function for partitioning the data. The second point describes a challenge which cannot be handled by optimal hashing: The "difficulty" of clusters can vary, e. g., due to a varying number of tuples per cluster. This is the aspect of load balancing in MapReduce systems we will focus on in this chapter.

6.3 THE PARTITION COST MODEL

Whenever achievable, all reducers of a MapReduce job should require roughly the same amount of time for processing their share of data. Distributing the workload evenly over all reducers maximises resource utilisation, as no reducers remain idle, waiting for some other, overloaded reducers to complete. Moreover, well-balanced execution times minimise the time until job completion, because parallel processing is exploited best possible. Finally, similar execution times are a common (and often implicit) assumption in both scheduling and failure detection detection strategies proposed for MapReduce [Dean and Ghemawat 2008, Zaharia et al. 2008].

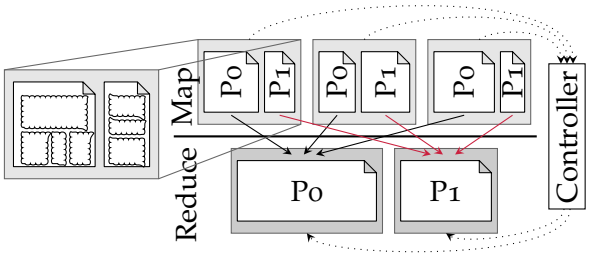


Figure 6.1: Partition Assignment in Current MapReduce Systems

6.3.1 *Current Situation*

In state of the art MapReduce systems, like Hadoop, every mapper partitions the share of intermediate results it creates into r partitions (i. e., $p = r$ in the partitioning function π defined above, and all partition bundles consist of a single partition only). As all mappers use the same partitioning function, intermediate tuples belonging to the same cluster are all placed into the same partition no matter which mapper they were produced on.

The partitioning strategy of current MapReduce systems is visualised in Figure 6.1. In this example, we have two reducers. Therefore, every mapper creates two partitions. The partitions of the first mapper are shown in more detail on the left of the figure. The first partition contains four clusters, the second one holds three.

Typically, a hash function is used for partitioning. Assuming a reasonably good hash function, the keys are uniformly distributed to the partitions, i. e., every partition contains roughly the same number of clusters. Every partition is then assigned to a dedicated reducer for further processing. This is visualised again in Figure 6.1. Partition P_0 of every mapper is assigned to the reducer on the left, P_1 to the one on the right.

This approach is perfectly suitable in situations where the key frequencies are (almost) uniformly distributed, and the amount of work a reducer spends per cluster does not vary strongly. In many other situations, however, distributing the keys uniformly is suboptimal. The most prominent problems are:

SKEWED KEY FREQUENCIES If some keys in the intermediate result are more frequent than others, the clusters will vary in the number of tuples they contain. The resulting partition assignment to the reducers may thus be skewed in the number of tuples even though the (distinct) keys are uniformly distributed.

SKEWED EXECUTION TIMES Finally, even if the partition assignment to the reducers is well balanced in the total number of tuples, complex calculations within the reducer may lead to skewed execution times. The processing of a single, large cluster may take much more time than processing a higher number of small clusters, even though they may sum up to the same total size.

Example 6.2. *Consider a partition consisting of nine tuples distributed to three clusters. The reducer does a pairwise comparison of the tuples within each cluster, i. e,., $n(n-1)/2$ comparisons for a cluster of n tuples. If the three clusters contain three tuples each, the incurred cost is $3 \cdot 3 = 9$. If we have one cluster containing seven tuples while the remaining two clusters only contain one tuple each, the cost is $2 \cdot 0 + 21 = 21$.*

Skew is symbolised by smaller and larger partition icons and reducer boxes in Figure 6.1. In this example, partition P0 is much larger than partition P1 on two of the mappers. The reducer on the left thus gets a much larger share of data than the one on the right.

6.3.2 *Optimal Solution*

In order to balance the workload on the reducers, we need to know the amount of work to spend on every cluster. Often, the work per cluster depends on the number of tuples in the cluster. Therefore, while creating the clusters, we monitor for every cluster $C(k)$ the number of tuples $|C(k)|$ it contains. Based on the complexity of the reducer algorithm, we can then calculate the amount of work, or weight, $w(|C(k)|)$ for each cluster k as a function of the tuple count.

Example 6.3. *Recall the reducer algorithm described in Example 6.2, comparing all tuples within a cluster to each other. As the reducer's complexity is quadratic in the number of tuples per cluster, we estimate the weight of a cluster in this scenario as $w(t) = t(t-1)/2$.*

We are aware that, for some applications, monitoring only the tuple count may not be sufficient for estimating the processing cost accurately. We will present appropriate extensions of our monitoring algorithms in Section 6.5.

MapReduce systems process the data in a distributed manner. The bag I holding all intermediate tuples is not materialised on a single host. Therefore, we need to collect our monitoring data in a distributed manner on the mappers as they process the data, and then consolidate it after the map phase. We denote by I_i the bag of intermediate (key,value) pairs generated by mapper i, $1 \leqslant i \leqslant m$, i. e., $\{I_1, I_2, \ldots, I_m\}$ is a partitioning of I. On every mapper, we create a local histogram.

Definition 6.1 (Local Histogram). *Let* I_i *be the bag of all intermediate (key,value) pairs produced by mapper* i*. The* local histogram $L_i(j)$ *is defined as a set of pairs* (k, v)*, where* $k \in \{x : \exists y((x, y) \in I_i) \wedge \pi(x) = j\}$ *is a key in* I_i *assigned to partition* j *and* v *is the number of tuples in* I_i *with key* k*.*

Aggregating these histograms on a central controller, we construct the global histogram. Note that we do not need to introduce a new centralised component in MapReduce, which would represent a new single point of failure, in order to aggregate the local histograms. There is already a centralised controller for task scheduling, which we exploit for load balancing.

Definition 6.2 (Global Histogram). *Given* m *local histograms* $L_i(j)$*,* $1 \leqslant i \leqslant m$*, of pairs* (k, v)*, where* k *is the key and* v *is the associated cardinality, the* global histogram $G(j)$ *is the set* $\{(k, v)\}$ *with*

(i) $\exists v((k, v) \in G(j)) \Leftrightarrow \exists i, v'((k, v') \in L_i(j))$

(ii) $\forall (k, v) \in G(j) : v = \sum_{\substack{1 \leqslant i \leqslant m \\ (k,v') \in L_i(j)}} v'$.

The global histogram maps all keys in the intermediate data I to the cardinality of the respective cluster. It is a sum aggregate over all local histograms. Since a global cluster can consist of 1 to m local clusters, the cardinality of the exact global histogram is bounded by

$$\max_{1 \leqslant i \leqslant m} |L_i(j)| \leqslant |G(j)| \leqslant \sum_{i=1}^{m} |L_i(j)| \quad .$$

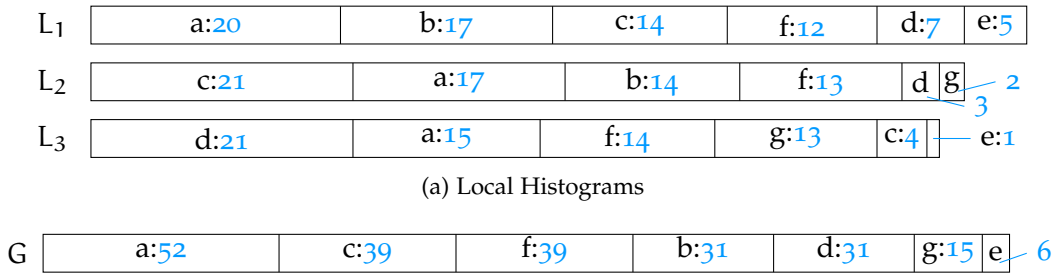
Example 6.4. *We compute the global histogram for partition* j *from the local histograms for the corresponding partition in a scenario with* $m = 3$ *mappers:*

$$L_1(j) = \{(a, 20), (b, 17), (c, 14), (f, 12), (d, 7), (e, 5)\}$$
$$L_2(j) = \{(c, 21), (a, 17), (b, 14), (f, 13), (d, 3), (g, 2)\}$$
$$L_3(j) = \{(d, 21), (a, 15), (f, 14), (g, 13), (c, 4), (e, 1)\}$$

The exact global histogram (see Figure 6.2) is

$$G(j) = \{(a, 52), (c, 39), (f, 39), (b, 31), (d, 31), (g, 15), (e, 6)\} \quad .$$

Lemma 6.1. *Consider a scenario with* m *mappers, which each produce* $O(|I|/m)$ *tuples for the intermediate result* I*. The computation of the local histogram requires* $O(|I|/m \log(|I|/m))$ *time and* $O(|I|/m)$ *space; the computation of the exact global histogram requires* $O(|I| \log |I|)$ *time and* $O(|I|)$ *space.*



(a) Local Histograms

(b) Global Histogram

Figure 6.2: Local and Exact Global Histograms.

Proof. In the worst case, all keys are unique such that the number of clusters is equal to the number of tuples. The local histogram for $O(|I|/m)$ clusters requires $O(|I|/m)$ counters. We store and update the counters in a binary search tree, which requires $O(|I|/m \log(|I|/m))$ time and $O(|I|/m)$ space. Since the keys of all mappers can be different, the global histogram can be of size $O(|I|)$. The global histogram is aggregated from the local histograms using an m-way merge in $O(|I| \log |I|)$ time. □

Based on the monitoring data, we can determine the weight of the clusters. By solving the associated bin packing problem, we can then calculate the optimal assignment of clusters to reducers.

In practice, this optimal solution is not feasible for two reasons.

1. In a worst-case scenario, the monitoring data grows linearly in the size of the intermediate data I. Such a situation arises, e. g., when joining two tables on their primary key columns. Every key value can appear only once per table. The clusters over the combined data of both tables can thus contain at most two elements. Consequently, the number of clusters must grow linearly in the number of tuples in the input data sets.

2. The bin packing problem is NP hard. Hence, even for a moderate number of clusters, calculating the optimal assignment of clusters to reducers can become more expensive than the actual execution of the reducers.

We will address these two problems in Sections 6.4 and 6.6, respectively, and develop heuristics for approximately solving the load balancing problem.

6.3.3 *Approximate Cost Estimation*

The first problem with the optimal solution is the size of the data monitored. In the worst case, the number of clusters, $|\mathbb{K}|$, grows linearly with the number of intermediate data tuples. With MapReduce being a system designed for processing terabyte scale data sets, we can therefore not afford to monitor every cluster individually. We approximate the exact monitoring data on partition level instead, i. e., the histogram $G(j)$ *approximates* the actual monitoring data.

This introduces additional challenges. Each mapper sees only a small fraction of the data and has only partial information about the cluster sizes. In particular, the mapper cannot know what fraction of a specific cluster it sees. Capturing the local histograms exactly on each mapper is feasible, as the number of map tasks is typically chosen based on the data volume, i. e., the amount of data per mapper is constant.

Sending all partial cluster sizes to the controller and summing up the costs for all clusters centrally is not feasible since the number of clusters can be in the order of the data size. The controller must therefore base its cost estimation on small summaries.

In addition, not all mappers do necessarily run at the same time. Thus the controller cannot incrementally retrieve information as is done, e. g., in distributed top-k scenarios, where the distributed rankings are incrementally consumed until the central ranking is accurate enough.

Recall from above that we calculate the weights, or processing costs, per cluster. The histogram $G(j)$ allows us to extract the required information, i. e., the tuple count, on a per-cluster basis. We can then estimate the processing cost $W(j)$ of partition j by summing up the weights of all clusters contained in that partition

$$W(j) = \sum_{(k,v)\in G(j)} w(v) \quad .$$

Finally, we use these partition costs to balance the load on the reducers.

In the following sections, we will first present concrete realisations of histograms allowing us to estimate the processing costs according to the Partition Cost Model. Then, we will show how to exploit the calculated partition costs in order to balance the reducer workload.

6.4 TOPCLUSTER: A DISTRIBUTED MONITORING APPROACH

Due to the distributed nature of MapReduce, we need to collect the monitoring data required for load balancing in a distributed manner and then aggregate it. In this

section, we will present two techniques for creating appropriate histograms of monitoring data. The first, Uniformity-Based Monitoring, assumes uniform distribution of cluster cardinalities within each partition. This assumption allows for a very compact representation of the monitoring data, but limits the effectiveness of the load balancing algorithm when the data are heavily skewed. The second algorithm, TopCluster, allows us to compute good partition cost estimations also when the data are highly skewed, at the cost of an increased volume of monitoring data.

To simplify the discussion, in the following we describe the computation of the global histogram for a single partition. We will denote the local histograms as L_i instead of $L_i(j)$ and the global histogram as G instead of $G(j)$. The same procedure is repeated for every partition j.

6.4.1 *Histogram Approximation Error*

Both Uniformity-Based Monitoring and TopCluster approximate the global histogram. As the approximation algorithms run on the controller, they must be efficient and the complexity of the algorithms should be significantly smaller than $|I|$.

We measure the error of an approximation to the exact global histogram as the percentage of tuples that the approximated histogram assigns to a different cluster than the exact histogram. The goal is to have a small approximation error. For the error computation, we do not identify the clusters by their key, but we order the clusters by their size and compare clusters with the same ordinal number in sort order, i. e., we compare the largest clusters from the exact and approximated histograms, then the second-largest clusters, etc. This is reasonable since the processing cost of a cluster in the partition cost model is independent of its key.

Example 6.5. *Consider the exact histogram* $G = \{(a, 20), (b, 16), (c, 14)\}$ *and the approximated histogram* $\widetilde{G} = \{(a, 20), (c, 17), (b, 13)\}$. *The difference between the largest clusters in both these histograms is 0. For the second largest,* b *in* G *and* c *in* $\widetilde{G}$*, we note a difference of 1, the same as for the smallest clusters. So, in total we have a difference of* $0 + 1 + 1 = 2$ *tuples. Both the histograms contain information on 50 tuples. As every tuple assigned to a wrong cluster is counted twice (once for the cluster it is missing in, and once for the cluster it is assigned to), we obtain an approximation error of* $1/50 = 2\%$.

6.4.2 *Uniformity-Based Monitoring*

The first monitoring approach we present is Uniformity-Based Monitoring. This approach stands out for very low memory requirements on both the mappers and the

controller, and by very simple aggregation of the local histograms on the controller. Therefore, Uniformity-Based Monitoring is a valuable monitoring approach for environments where the management overhead for load balancing must be kept at a strict minimum.

With Uniformity-Based Monitoring, we only capture the tuple count t and the cluster count c for each partition, i. e., $G = (c, t)$. For the cost calculation, we need the tuple count per cluster. By assuming uniform distribution of the cluster sizes within a partition, we can estimate the tuple count per cluster as

$$\overline{t_c} = \frac{t}{c} \ .$$

As in the optimal case, we can now determine the processing cost per cluster, $w(\overline{t_c})$ using the tuple count estimates. We then sum up all processing costs belonging to the same partition in order to obtain the partition costs W. As we assume uniform distribution of clusters within each partition, the partition cost is

$$W = c \cdot w(\overline{t_c}) \ .$$

Assuming uniform distribution within each partition, the estimated cost might well deviate from the actual cost when the cluster sizes within a partition are skewed. The approximation error of Uniformity-Based Monitoring will increase with the data skew getting heavier. TopCluster will circumvent this issue by capturing the data distribution within the partitions more precisely.

Distributed Monitoring

Recall that we have to collect the monitoring data in a distributed manner. Counting the number of tuples per partition in a distributed manner is easy to achieve. On each mapper, we monitor in the local histogram L_i the number of tuples t_i assigned to the corresponding partition for the share of data that mapper processes. Summing up the values received from all mappers on the controller, we obtain the total number of tuples in the partition:

$$t = \sum_{i=1}^{m} t_i \ .$$

For the number of clusters, the same approach is not applicable, as clusters are typically distributed over multiple mappers. By simply summing up the cluster counts

from all mappers, we would thus obtain too large a value in most cases. Estimating the cluster count correctly is crucial for the calculation of the partition cost, as the following example shows.

Example 6.6. *Assume we have a partition consisting of six tuples. The cost of a cluster is the square of its tuple count. If we estimate the partition to contain three clusters, we get* $3 \cdot 2^2 = 12$ *as the partition cost. If we estimate the partition to contain only two clusters, the partition cost is* $2 \cdot 3^2 = 18$*.*

Ideally, we capture the presence of clusters on every mapper exactly. Besides counting the tuples, we would create a *presence indicator* p_i such that $p_i(k)$ is true if and only if a cluster with key k exists in L_i:

$$p_i(k) = \begin{cases} \text{true} & \text{if } \exists v((k, v) \in L_i) \\ \text{false} & \text{otherwise.} \end{cases}$$

Capturing the presence indicator exactly is, however, not feasible since we had to capture each single cluster, and the number of clusters may be $O(I)$. Hence, we create an approximation of the presence indicator, $\widetilde{p_i}$, which is implemented as a bit vector of fixed length. We hash the keys to a position in the bit vector and set the bit for each key in I_i. In order to estimate the total number of clusters in the partition, we calculate the disjunction of all bit vectors for a partition. Linear Counting [Whang et al. 1990] then allows us to estimate the number of clusters based on the bit vector length and the ratio of reset bits, also taking into account the probability of hash collisions.

With increasing skew in the cluster distribution, the quality of the Uniformity-Based Monitoring approximation will decrease, as we ignore skew within a partition. In this situation, a more sophisticated approach is required, which captures the clusters most relevant for cost estimation as precisely as possible. TopCluster, presented in the following, fulfils these requirements.

6.4.3 *TopCluster Monitoring*

In the optimal solution presented in Section 6.3.2, we based the processing cost estimation on exact cardinality information of all clusters, extracted from the exact global histogram G. We can reconstruct this global histogram G from local histograms L_i collected on every mapper.

The local histograms L_i contain exact cardinality information for all clusters in the corresponding partition on mapper i ($1 \leqslant i \leqslant m$). With TopCluster, we capture the local histograms exactly. (For situations in which this is not feasible, we will present an appropriate extension to TopCluster in Section 6.5.2.) Each mapper i then sends only the *head of the local histogram*, which contains only the largest clusters, to the controller. The controller uses the local histogram heads to approximate the global histogram.

Definition 6.3 (Local Histogram Head)**.** *Given a local histogram* L_i *(see Definition 6.1) and a local threshold* τ_i*, the head of the histogram,* $L_i^{\tau_i}$*, is defined as a subset of* L_i *such that the cardinalities of all clusters is at least* τ_i*; if there is no cluster of size* τ_i *or larger, the next smallest cluster(s) is (are) also in the head* $L_i^{\tau_i}$*:*

$$L_i^{\tau_i} = \{(k, v) \in L_i \mid v \geqslant \tau_i \vee (v < \tau_i \wedge \nexists (k', v') \in L_i (v < v'))\} \ .$$

The local threshold τ_i can be different for all local histograms; the sum of all local thresholds is the global cluster threshold τ which is the input parameter for TopCluster. In the basic TopCluster algorithm we choose the local threshold to be $\tau_i = \tau/m$. In Section 6.5 we discuss an extension of TopCluster that adapts τ_i for each histogram based on the skew of the data. The following discussion holds independently of the choice of the local threshold τ_i.

Lower and Upper Bound Histograms

We define the upper and lower bound histograms, which are used to calculate the global histogram approximation. As we will show in Section 6.4.4, the cardinalities of all clusters in the lower/upper bound histograms are lower/upper bounds of the exact cardinality values of the respective clusters.

Definition 6.4 (Upper and Lower Bound Histogram)**.** *Given the head of* m *local histograms* $L_i^{\tau_i}$, $1 \leqslant i \leqslant m$*, the* lower bound histogram G_l *is defined as follows:*

(i) $\exists v : (k, v) \in G_l \Leftrightarrow \exists i, v' : (k, v') \in L_i^{\tau_i}$

(ii) $\forall (k, v) \in G_l : v = \sum_{\substack{1 \leqslant i \leqslant m \\ (k, v') \in L_i^{\tau_i}}} v' \ .$

Let p_i *be the (exact) presence indicator for* L_i*, and let* v_i *be the smallest value in* $L_i^{\tau_i}$*, i.e.*

$$v_i = \min_v \left\{(k, v) \in L_i^{\tau_i}\right\} \qquad 1 \leqslant i \leqslant m \ .$$

The upper bound histogram G_u *is defined as*

(i) $\exists v : (k, v) \in G_u \Leftrightarrow \exists i, v' : (k, v') \in L_i^{\tau_i}$

L_1^{14}	a:20	b:17	c:14	... (24)
L_2^{14}	c:21	a:17	b:14	... (18)
L_3^{14}	d:21	a:15	f:14	... (18)

Figure 6.3: Head of Local Histograms for $\tau_i = 14$.

(ii) $\forall (k, v) \in G_u : v = \sum_{1 \leqslant i \leqslant m} \text{val}(k, i)$ *with*

$$\text{val}(k, i) = \begin{cases} v' & \text{if } (k, v') \in L_i^{\tau_i} \\ v_i & \text{if } p_i(k) \wedge \nexists v' : (k, v') \in L_i^{\tau_i} \\ 0 & \text{otherwise.} \end{cases}$$

Note that both G_l and G_u contain values for exactly those clusters that appear in at least one of the local histogram heads, so $|G_l| = |G_u|$, and the set of keys of both histograms is identical. The number of items in G_l and G_u is bounded by the largest local histogram head on the lower end (if this largest local histogram head contains all keys which appear in any of the other local histogram heads), and the sum over all local histogram heads (if all keys in the local histogram heads are distinct):

$$\max_{1 \leqslant i \leqslant m} \left|L_i^t\right| \leqslant |G_l| = |G_u| \leqslant \sum_{i=1}^{m} \left|L_i^t\right| \quad .$$

Example 6.7. *The head of the local histograms* $L_i^{\tau_i}$ *extracted from the local histograms introduced in Example 6.4 for* $\tau_i = 14$ *are shown in Figure 6.3.*

Key a *is contained in all three local histogram heads. Therefore, its exact value is known, and the upper and lower bounds coincide:* $20 + 17 + 15 = 52$.
Key c *is not contained in* L_3^{14}. *The corresponding lower bound is thus* $14 + 21 + 0 = 35$. *From* $p_3(c) = \text{true}$, *we know* c *occurred in* L_3. *As* $v_3 = 14$, *the upper bound for* c *is* $14 + 21 + 14 = 49$.
Key b *is not contained in* L_3^{14} *as well. The lower bound is* $17 + 14 + 0 = 31$. *As* $p_3(b) = \text{false}$, *we know* b *was not contained in* L_3 *and we can add* 0 *in the calculation of the upper bound of* b *for local histogram 3:* $17 + 14 + 0 = 31$.
The bounds for the remaining keys are calculated analogously. We obtain the following bounds, which are also visualised in Figure 6.4 (the orange dots are explained in the next example).

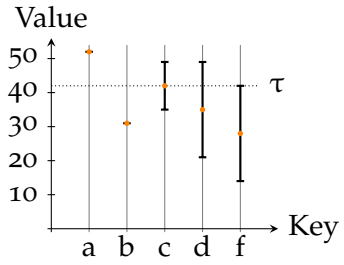


Figure 6.4: Global Bounds

$$G_l = \{(a, 52), (c, 35), (b, 31), (d, 21), (f, 14)\}$$
$$G_u = \{(a, 52), (c, 49), (d, 49), (f, 42), (b, 31)\}$$

The approximated global histogram has a *named* and an *anonymous* part. The named part is a histogram which stores cluster cardinalities for specific key values. The goal is to have the largest clusters in the named part of the global histogram. The cardinalities of all other keys are covered by the anonymous part. The clusters in the anonymous part have no name and we do not store values for each cluster; we only know the number of clusters and their average size. This is sufficient for cost estimation. Since the largest clusters are in the named part, the error introduced by the assumption of a uniform distribution on the anonymous part is small.

Named Histogram Part

We define two approximations for the global histogram. The *complete* histogram stores a cardinality for all keys that appear in at least one local histogram head, the *restrictive* histogram is the subset of the complete histogram in which all cardinalities are at least τ.

Definition 6.5 (Global Histogram Approximation)**.** *Let* G_l *and* G_u *be the pair of lower and upper bound histograms in a given setting. The* complete global histogram *is defined as*

$$\widetilde{G} = \left\{ \left(k, \frac{v_u + v_l}{2} \right) : (k, v_u) \in G_u \wedge (k, v_l) \in G_l \right\} \ ,$$

the restrictive global histogram *is defined as*

$$\widetilde{G_r} = \left\{ (k, v) : (k, v) \in \widetilde{G} \wedge v \geqslant \tau \right\} \ .$$

Example 6.8. *We approximate the global histogram with the upper and lower bounds computed in Example 6.7. For the complete global histogram approximation, we obtain*

$$\widetilde{G} = \{(a, 52), (c, 42), (d, 35), (b, 31), (f, 28)\} \ .$$

These values are visualised as orange dots in Figure 6.4. The restrictive global histogram approximation ($\tau = 42$) *is*

$$\widetilde{G_r} = \{(a, 52), (c, 42)\} \ .$$

Anonymous Histogram Part

The named part of the global histogram approximation contains cardinality values only for the largest clusters. In order to compute the partition cost, however, we need to consider *all* clusters in the partition. We assume uniform distribution on the small clusters that are not in the named histogram part. To compute the average cluster size, we determine the global sum of cluster cardinalities and the number of different clusters. We calculate these values like we did with Uniformity-Based Monitoring, based on the local tuple cardinalities and the presence indicators.

Example 6.9. *The sum of cluster cardinalities for the three local histograms in Example 6.4 is* $75 + 70 + 68 = 213$. *Assume a global cluster count of 7. The cardinality sum of the restrictive approximation,* $\widetilde{G_r}$, *in Example 6.8 is* $52 + 42 = 94$. *Thus, for each of the four anonymous clusters we estimate a value of* $(213 - 94)/(7 - 2) = 23.8$ *tuples. We compute the approximation error as introduced in Section 6.4.1. The absolute difference between the exact and the approximated clusters is* $0 + 3 + 15.2 + 7.2 + 7.2 + 8.8 + 17.8 = 59.2$, *thus* $59.2/2 = 29.6$ *tuples were assigned to the wrong cluster. Even though the approximate global histogram provides no information on 5 out of 7 clusters (more than 70%), less than 14% of the tuples were assigned to the wrong cluster. For a reducer with* n^2 *complexity, we obtain an estimated cost of 7300.2, as opposed to the exact cost of 7929; an error of less than 8%.*

Although the restrictive histogram approximation has a longer anonymous part in which uniform distribution is assumed, it outperforms the complete histogram, in particular when the data skew is small. This is explained as follows.

An approximation error is only introduced if a cluster exists in a local histogram, but is not in the head. With heavily skewed data, the clusters contained in the local histogram heads will have clearly larger frequencies than the clusters merged in the histogram tails. For almost uniform distributions, however, a very small difference in the tuple count may decide on whether a cluster is contained in a local histogram head or not. For a key k that is not in the head of a local histogram, $L_i^{\tau_i}$, but $p_i(k) = \text{true}$,

we add ν_i (the smallest cardinality in $L_i^{\tau_i}$) to the upper bound, and 0 to the lower bound. Using the arithmetic mean as a cardinality estimate, we assume cardinality $\nu_i/2$ for that cluster on mapper i. With an almost uniform data distribution, the real cardinality is however likely to be close to ν_i. The estimated global cardinality of such a cluster is typically slightly larger than $\tau/2$ but smaller than τ. Therefore, these clusters are not contained in $\widetilde{G_r}$.

Example 6.10. *Consider the cardinality estimation for the cluster with key* f *in Examples 6.7 and 6.8. Although key* f *exists in all three local histograms, it is not in the heads of* L_1 *and* L_2*. With* $\nu_1 = \nu_2 = 14$*, we approximate the cardinality of cluster* f *with* $14/2 = 7$ *for both local histograms. Combined with the exact value for cluster* f *from* L_3^{14}*,* 14*, we obtain a value of* 28 *in the global histogram approximation. The correct value is* 39*. Cluster* f *is not included in the restrictive histogram since its estimated cardinality is below* $\tau = 42$*.*

Approximate Presence Indicators

As with Uniformity-Based Monitoring, we cannot employ exact presence indicators as their size may be $O(|I|)$. Instead, we create again a bit vector $\widetilde{p_i}$ of fixed length. Employing Linear Counting [Whang et al. 1990], as before, we can estimate the number of clusters in the partition and hence also the number of clusters in the anonymous part of the histogram. Additionally, we use this bit vector like a Bloom filter [Bloom 1970] on the controller in order to check for the presence of clusters whose keys were reported by other mappers when calculating the upper and lower bound histograms. $\widetilde{p_i}$ may introduce false positives, but cannot introduce false negatives.

False positives impact the quality of the approximation of the global histogram. Recall that we calculate the approximated value of an item as the arithmetic mean of its upper and lower bound. The lower bound, G_l, is not affected by our approximation of p_i, as we do not employ p_i in the respective calculation. The upper bound, G_u, however, may change. Consider the specification of $\text{val}(k, i)$ in Definition 6.4. For a key k that is not contained in the local histogram L_i, we must add 0. In case of a false positive on $p_i(k)$ we add ν_i (the smallest value in the local histogram head $L_i^{\tau_i}$) to the upper bound instead, thereby overestimating the upper bound. As false negatives are impossible, we will never underestimate the bound. Hence, the upper bound remains in place, but it may become looser in case of false positives.

Regarding the presented variants of the approximated global histogram, the influence of approximating p_i is as follows. If we overestimate the upper bound of a cluster, the estimated cluster cardinality raises as well. Therefore, the actual values in the approximated global histogram may change as a consequence of approximating

p_i. As the complete approximation contains all items occurring in any local histogram head, approximating p_i has no influence on which items are included in the complete approximation. The restrictive approximation chooses items based on their average values. If we overestimate the upper bound of an item, its average value rises as well. Hence, items may be included in the restrictive approximation of the global histogram which would not have been included with exact presence indicators.

Example 6.11. *We approximate* p_i *with a bit vector of length 3, based on a hash function* h *mapping character keys* a *to 0,* b *to 1 etc. (mod 3). For the histogram heads with* $\tau_i = 14$ *from Example 6.7, we will have a false positive for key* b *on local histogram* L_3*, as* $p_3(b) =$ false*, but* $\widetilde{p_3}(b) =$ true *(*$h(b) = h(e)$*, and key* e *is contained in local histogram* L_3*). We therefore calculate the upper bound for* b *as* $17 + 14 + 14 = 45$ *instead of* $17 + 14 + 0 = 31$*. In consequence, the estimated value for* b *in* $\widetilde{G}$ *increases from* 31 *to* 38*. In this example, the overestimation has no impact on the restrictive global histogram approximation, as the overestimated cardinality of* 38 *is still below the cluster threshold* $\tau = 42$*.*

6.4.4 *TopCluster Approximation Guarantees*

Since only the head of each local histogram is sent to the controller, it is not possible to compute the exact global histogram, G, at the controller. The approximate global TopCluster histograms presented above take values between the lower and upper bound histograms, G_l and G_u. We show that G_l and G_u are in fact respectively lower and upper bounds for the exact global histogram G.

Theorem 6.1. G_l *is a lower bound on* G*:*

$$\forall (k, v) \in G_l : \exists (k, v') \in G \wedge v \leqslant v' \ .$$

Proof. As $(k, v) \in G_l \Leftrightarrow \exists i, w : (k, w) \in L_i^{\tau_i}$, and $L_i^{\tau_i} \subseteq L_i$, it is clear that $\forall (k, v) \in G_l : \exists (k, v') \in G$.
Choose $(k, v) \in G_l$ and $(k, v') \in G$. Let

$$K' = \left\{ i \in \{1, \ldots, m\} : \exists (k, v'') \in L_i^{\tau_i} \right\} \text{ and}$$
$$K = \left\{ i \in \{1, \ldots, m\} : \exists (k, v'') \in L_i \right\} \ .$$

From $L_i^{\tau_i} \subseteq L_i$ follows $K' \subseteq K$. As the cluster cardinalities cannot be negative, i. e., $v'' > 0$, we obtain $v \leqslant v'$. Moreover, $v = v' \Leftrightarrow K' = K$, i. e., in that case the bound is tight. □

Note that $L_i^{\tau_i}$ does not need to contain the t *largest* elements of L_i for G_l to be a lower bound on G. The theorem is valid for any subsets $S_i \subseteq L_i$. For the following theorem to hold, however, $L_i^{\tau_i}$ must consist of the largest elements of L_i.

Theorem 6.2. G_u *is an upper bound on* G*:*

$$\forall (k, v) \in G_u : \exists (k, v') \in G \wedge v \geqslant v' \ .$$

Proof. Analogously to the proof of Theorem 6.1, $\forall (k, v) \in G_u : \exists (k, v') \in G$. Choose $(k, v) \in G_u$ and $(k, v') \in G$. Let

$$K' = \left\{ i \in \{1, \ldots, m\} : \exists (k, v'') \in L_i^{\tau_i} \right\} \text{ and}$$
$$K = \left\{ i \in \{1, \ldots, m\} : \exists (k, v'') \in L_i \right\} \ .$$

Then, $K' \subseteq K$. For local histograms L_i with $i \in K'$, the same, exact value is added to both v and v'. For local histograms L_i with $i \in \{1, \ldots, m\} \setminus K$, $p_i(k) = \text{false}$, and we add 0 to both v and v'. Finally, for local histograms L_i with $i \in K \setminus K'$, we add v_i to the value of the upper bound, v, but v_e, where $(k, v_e) \in L_i$, to the value of the exact global histogram, v'. As $i \notin K'$, $(k, v_e) \notin L_i^{\tau_i}$. $L_i^{\tau_i}$ contains the elements from L_i with the largest values. Therefore, $v_i \geqslant v_e$, and subsequently also $v \geqslant v'$. Moreover, $v = v' \Leftrightarrow K' = K$, i. e., in that case the bound is tight. □

As an error estimation, we can derive an upper bound on the cardinality of the clusters that we might have missed in the approximated global histogram.

Theorem 6.3. *Let* L_i *be local histograms,* $1 \leqslant i \leqslant m$, G *the corresponding exact global histogram, and* τ *a cluster threshold. Then the complete TopCluster histogram approximation* $\widetilde{G}$ *has the following properties:*

- Completeness: *All clusters of the exact histogram* G *with cardinality at least* τ *are in the approximated histogram:* $\forall k(\exists v((k, v) \in G \wedge v \geqslant \tau) \Rightarrow \exists v'((k, v') \in \widetilde{G}))$.
- Error Guarantee: *The error for the cluster cardinalities in the approximated histogram is at most* $\tau/2$*:* $\forall k((k, v) \in G \wedge (k, v') \in \widetilde{G} \Rightarrow |v - v'| < \tau/2)$.

For the restrictive histogram approximation $\widetilde{G_r}$*, the error guarantee holds as well, but not completeness.*

Proof. Choose $(k, v) \in \widetilde{G}$ and $(k, v') \in G$.

- *Completeness:* $v' \geqslant \tau = \tau_1 + \ldots + \tau_m$. There must be at least one local histogram L_i with $(k, v'') \in L_i$ and $v'' \geqslant \tau_i$. Then, $(k, v'') \in L_i^{\tau_i}$ and the cluster with key

k is contained in $\widetilde{G}$. As we may underestimate the cardinality of cluster k for local histograms L_i which do not contain k in their head, the cluster might not be contained in $\widetilde{G_r}$.

- *Error Guarantee:* Let

$$K' = \left\{ i \in \{1, \ldots, m\} : \exists (k, v'') \in L_i^{\tau_i} \right\} \text{ and}$$
$$K = \{ i \in \{1, \ldots, m\} : p_i(k) = \text{true} \} \ .$$

Then, $K' \subset K$. We only make estimation errors for local histograms $i \in K \setminus K'$. Recall from Definition 6.4 that we v_i is the smallest value contained in $L_i^{\tau_i}$. By using the arithmetic mean as the estimated cardinality, the largest possible error we make on each of these histograms is $v_i/2$. According to the definition of $L_i^{\tau_i}$, we know $v_i \leqslant \tau_i$. As $(k, v) \in \widetilde{G}$, $K' \neq \emptyset$ and there is at least one local histogram for which we know the exact cluster cardinality. Hence, the global error is at most

$$\sum_{i \in K \setminus K'} \frac{v_i}{2} \leqslant \sum_{i \in K \setminus K'} \frac{\tau_i}{2} < \sum_{i=1}^{m} \frac{\tau_i}{2} = \frac{\tau}{2} \ .$$

□

6.5 EXTENSIONS TO TOPCLUSTER

Next, we discuss three possible extensions to TopCluster. First, we show how the parameter τ can be determined automatically in a distributed manner. Then, we reconsider our assumption that exact monitoring is feasible on all mappers, and analyse the implications on the approximated global histogram in situations where this assumption no longer holds. Finally, we discuss cost functions that depend on other parameters than cluster cardinality.

6.5.1 *Adaptive Local Thresholds*

So far, we assumed the parameter τ to be supplied by the user. Finding a suitable value for τ before starting a MapReduce job is challenging. Therefore, the system should be able to determine a suitable τ automatically. As explained before, communication between all mappers is impossible. We devise a strategy in which every mapper determines the *relevant* items in its local histogram autonomously, and sends

only those items to the controller. As we assume uniform distribution on the items not captured in the named part of the global histogram approximation, the clusters that depart most prominently from uniform distribution should be transmitted.

We base the decision on which items to transmit on the local data distribution, and only send the items with values exceeding the local average cardinality on mapper i, μ_i, by a factor of ε, where ε is a user-supplied error ratio. This allows us to keep the local error on every mapper within well-known bounds. The largest item that we possibly miss in the named global histogram is within $\tau = (1+\varepsilon)\sum_{i=1}^{m}\mu_i$.

With these settings, our approach works well with both uniform, and skewed data distributions. If the data is skewed, only a small number of items with strong impact on the partition cost will exceed the local error threshold of $(1+\varepsilon)\mu_i$. We are therefore able to capture the partition cost reasonably well while keeping the communication volume for monitoring low. If the data is distributed evenly, on the other hand, our assumption of uniform distribution on the items that are not communicated to the controller is accurate, and we obtain good cost estimates as well.

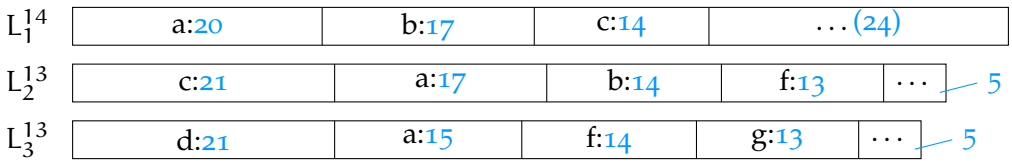
Example 6.12. *Continuing our running example, from the monitored tuple and cluster counts (see Example 6.4), we calculate* $\mu_1 = 77/7 = 11$, $\mu_2 = 70/7 = 10$, *and* $\mu_3 = 68/6 = 43/3$, *each on the corresponding mapper. We allow an error of* $\varepsilon = 10\%$. *The thresholds for the local item counts are thus* 12.1, 11, *and* 12.47, *respectively. The resulting local histograms are shown in Figure 6.5a).*
The restrictive global approximation based on this input is

$$\widetilde{G_r} = \{(a, 52), (c, 41.5)\}$$

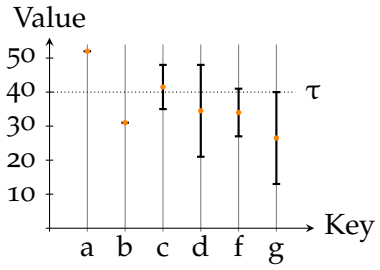
and thus very close to the approximation we obtained in Example 6.8 using the user-supplied value $\tau = 42$ *(resulting in* $\tau_i = 14$*).*

6.5.2 *Approximate Local Histograms*

Current MapReduce systems choose the number of mappers for each MapReduce job based on a compromise. High parallelism and the reduced amount of work which has to be repeated in case of a node failure are arguments favouring many mappers processing a small amount of data each. On the other side, the startup and management costs for each mapper plea for a low number of mappers and thus a higher data volume per mapper. Collecting monitoring data on every mapper introduces additional arguments for processing more data per mapper. As every mapper sees a larger share of base data, the quality of the local histograms is likely to improve. We thus propose to increase the amount of data processed by every mapper. As this reduces



(a) Local Histograms



(b) Bounds for the Global Histograms

Figure 6.5: Histogram Aggregation for $\varepsilon = 10\%$

the total number of mapper instances, we have the additional benefit of reducing the burden on the controller, as fewer local histograms must be aggregated.

In a worst case scenario, the number of clusters generated on a mapper could grow linearly with the amount of data processed. As we locally monitor every cluster, the monitoring data volume would grow at the same rate as well, possibly occupying resources required for the actual processing. We therefore need to provision a way of limiting the amount of monitoring data every mapper keeps. In case the exact monitoring data would exceed this imposed limit, we can switch to approximate ranking algorithms, e. g. Space Saving [Metwally et al. 2006], which allow to limit the amount of memory used. Space Saving was originally designed for finding the top-k items over data streams. At any point in time, it keeps the most frequent items encountered so far in a cache of fixed size. A new item not yet contained in the cache replaces the least frequent item in the cache. Thereby, Space Saving guarantees that no item whose actual frequency is higher than the reported frequencies is missing in the obtained ranking.

Obviously, using approximate instead of exact local monitoring impacts the bounds of clusters we derive on the controller.

Theorem 6.4. *For local histograms approximated using the Space Saving algorithm [Metwally et al. 2006], both the global lower and upper bound can be overestimated. Underestimation is only possible for the global lower bound.*

Proof. Let $\widetilde{L}_i^{\tau_i}$ be the head of an approximate local histogram on mapper i calculated using Space Saving, and let L_i be the corresponding exact local histogram. Let k be a key in $\widetilde{L}_i^{\tau_i}$ with estimated occurrence count $\widetilde{v}_k$ and actual occurrence count v_k, and let l be the least frequent item in $\widetilde{L}_i^{\tau_i}$ with estimated occurrence count $\widetilde{v}_l$ and actual occurrence count v_l. Then, $\widetilde{v}_k \geqslant v_k$ and $\widetilde{v}_l \geqslant v_l$ according to [Metwally et al. 2006] (Lemma 3.4).

- *Overestimation:*
  1. *Global Lower Bound:* We overestimate the global lower bound for the item with key k if $\widetilde{v}_k > v_k$.
  2. *Global Upper Bound:* Again, we overestimate the global upper bound of the item with k if $\widetilde{v}_k > v_k$. For an item with key k' not contained in $\widetilde{L}_i^{\tau_i}$, but appearing in L_i according to p_i, we overestimate the global upper bound if $\widetilde{v}_l > v_l$.
- *Underestimation:* The only scenario in which underestimation may occur is as follows. Consider an item with key k' with actual occurrence count $v_{k'}$, which is not contained in $\widetilde{L}_i^{\tau_i}$, but would be contained in $L_i^{\tau_i}$.
  1. *Global Lower Bound:* As k' is not contained in $\widetilde{L}_i^{\tau_i}$, we add 0 to the lower bound, thus underestimating it.
  2. *Global Upper Bound:* As k' is apparently contained in L_i, we will add $\widetilde{v}_l$ to the global upper bound. According to [Metwally et al. 2006] (Theorem 3.5), $\widetilde{v}_l \geqslant v_{k'}$. Therefore, we do not underestimate the global upper bound.

□

From Theorem 6.4 follows that the upper bound calculated as described in Definition 6.4 remains valid if it is (completely or partially) based on local histograms calculated using Space Saving. For the lower bound, this does not hold: due to the possible overestimation, it might no longer be in place. In consequence, we could overestimate the corresponding cluster's size. In order to keep the lower bound in place, we therefore decide to not increase it at all for mappers using Space Saving. A flag indicating the usage of Space Saving can be included in the communication between every mapper and the controller at the cost of one bit per mapper.

When using adaptive thresholds (Section 6.5.1), approximating the local histograms can also interfere with the choice on how many items to transmit to the controller. Recall that we base the decision on which items to transmit on the average local cluster cardinality. In order to calculate the average cluster cardinality, we need to keep

track of the sum of all cluster cardinalities, and the cluster count. When monitoring all clusters exactly, we obtain both these values as a side product. With approximate monitoring, this is no longer the case. The global cardinality is still easy to track with approximate local monitoring employing a dedicated counter. For the cluster count, we reuse once more the bit vectors $\widetilde{p}_i$ created for approximating the presence indicator and apply Linear Counting in order to obtain an estimation. Based on the approximate average cluster cardinality derived from this information, we can then pick the relevant items from our monitoring data to send to the controller. In an extreme case, we might not have monitored all clusters which should be transmitted, i. e., even the cardinality of the smallest clusters monitored is larger than the threshold. If that situation arises, all we can do is inform the user on the actual error margin we are able to provide on this mapper, and propose to increase the memory available for monitoring if better cost estimation is required.

Switching from exact local histograms to Space Saving is possible at runtime if the monitoring data volume exceeds a predefined threshold. The bit vector $\widetilde{p}_i$ is not affected by switching to Space Saving. For the total cluster cardinality on that mapper, we can initialise the counter with the sum of all cluster cardinalities counted so far. Then, we can discard the monitoring data on the clusters with the lowest cardinalities observed in order to reduce the consumed memory. The remaining cluster information is the initial state with which we can continue the monitoring process using Space Saving.

6.5.3 *Going Beyond Tuple Count*

So far, we considered cluster cardinality being the only parameter for the cost estimation. In some applications additional parameters might be desirable. For example, if serialized objects (which are a collection of items each) are passed as tuples, the data volume per cluster could be an appropriate additional parameter of the cost function. Neither Uniformity-Based Monitoring nor TopCluster are specific to monitoring cardinalities. The same technique is also applicable to other parameters like data volume — either instead of, or in addition to, cluster cardinality. When monitoring multiple parameters per cluster, correlations between the parameters can be important for an accurate cost estimation, i. e., we need to know both cardinality and data volume of a specific cluster. TopCluster allows the reconstruction of these correlations for clusters in the named histogram part on the controller using the cluster keys.

Example 6.13. *Consider an application scenario where every tuple contains an array of values. The reducer's task is to find the median of these values per cluster over all contained*

tuples. The amount of work to spend is thus independent of the number of tuples within the cluster: five tuples containing 10 values each cause the same work as one tuple containing 50 values. The weight depends on the combined size of all arrays within a cluster instead. With array sizes proportional to the tuple sizes, we will therefore base the work estimation on the data volume of the clusters. Finding the median element of an array of length n *is possible in* $n \log n$ *time by first sorting the array, then picking the middle element. Therefore, we estimate the work to spend on a cluster as* $s \log s$ *with* s *being the data volume.*

The nested tree representation we introduced in Chapter 5 also falls into this category. The more nodes a tree consists of, the larger the tuple representing the tree grows. The amount of work per tree, for calculations on each node of the tree (e. g., operations using the tree traversal operator as in Example 5.12), thus grows with the tuple size.

6.6 LOAD BALANCING

So far, we defined the Partition Cost Model that takes into account skewed data distributions and non-linear reducer tasks, and we defined two monitoring approaches suitable for providing the cost model with the required information. Next, we define two load balancing algorithms based on this cost model, Fine Partitioning and Dynamic Fragmentation, which aim to balance the load on the reducers. Fine Partitioning achieves this goal by assigning the partitions to reducers such that all reducers get roughly the same share of the estimated total workload. Dynamic Fragmentation allows to split partitions in order to obtain an even better balanced distribution of the workload. However, the splitting may introduce additional costs which we must take into account.

6.6.1 *Fine Partitioning*

By creating more partitions than there are reducers (i. e., by choosing $p > r$, in contrast to current MapReduce systems where $p = r$), we can retain some degree of freedom for balancing the load on the reducers. The range p should be chosen from is obviously bounded by the number of reducers, r, on the lower end, and the number of clusters, $|\mathbb{K}|$, on the upper end. With $p < r$, some reducers would not obtain any input. With $p > |\mathbb{K}|$, some partitions will remain empty.

The number of partitions, p, influences the quality of the load balancing obtained. The higher we choose p, the more possibilities the controller has to balance the load. On the other hand, the larger p is, the higher the incurred management overhead becomes. This overhead impacts on the execution of the MapReduce job twice. First,

we need to collect and process more monitoring data, as the partitions represent the histogram buckets. This contradicts our effort to reduce the amount of monitoring data. For very high values of p, handling the monitoring data could thus become a bottleneck in the job execution. Second, partitions are the units of data transfer (i. e., files) from the mappers to the reducers. Transferring a small number of large files is faster and results in less overhead than transferring a large number of small files. We need to be aware of this trade-off when choosing p.

We assign the partitions to reducers in a load balancing manner. As already explained in Section 6.3.2, ideally we would employ a bin packing algorithm for calculating the partition bundles. Unfortunately, bin packing is NP hard. We thus propose a greedy heuristic to determine the partition bundles. This heuristic is sketched in Algorithm 6.1. In every iteration, we pick the most expensive partition not yet assigned to a reducer (lines 4 and 5). If we have not initialised r reducers yet, we assign the partition to a new reducer (lines 6 and 7). Otherwise, we assign the partition to the reducer which has the least total load at that point. As the load of a reducer, we use the sum of the cost of all partitions already assigned to that reducer (lines 9 and 10). We repeat these steps until all partitions have been assigned. The set R we obtain contains the partition bundles to assign to the reducers.

Algorithm 6.1 Assign Partitions to Reducers

Input: $W : \{1, \ldots, p\} \to \mathbb{R}^+$
Output: R: a set of partition bundles

1: $R \leftarrow \emptyset$
2: $P = \{1, \ldots, p\}$
3: **while** $P \neq \emptyset$ **do**
4: $\quad q = \arg\max_{j \in P} W(j)$
5: $\quad P \leftarrow P \setminus \{q\}$
6: $\quad$ **if** $|R| < r$ **then**
7: $\quad\quad R \leftarrow R \cup \{\{q\}\}$
8: $\quad$ **else**
9: $\quad\quad s = \arg\min_{l \in R} \sum_{j \in l} W(j)$
10: $\quad\quad R \leftarrow (R \setminus \{s\}) \cup \{s \cup \{q\}\}$
11: $\quad$ **end if**
12: **end while**
13: **return** R

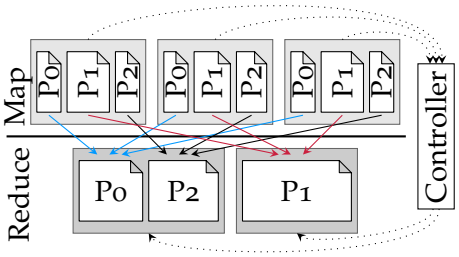


Figure 6.6: Partitioned Data Distribution

Note that we calculate the partition bundles only when all mappers are completed. This impacts on the *reducer slow-start* optimisation of Hadoop. We will discuss this aspect in Section 6.6.3.

Example 6.14. *Consider the scenario in Figure 6.6. Even though we only have two reducers, every mapper creates three partitions. Based on the monitoring data obtained from the mappers, the controller determines the assignment of partitions to reducers. As partition P1 is most expensive on every mapper (symbolised by the larger partition boxes in the figure), it is assigned to a dedicated reducer. Po and P2, which are both less expensive, share the other reducer.*

6.6.2 *Dynamic Fragmentation*

Even with fine partitioning, in some situations data might not be balanced appropriately and single partitions may grow excessively large. We therefore devise a strategy which dynamically splits very large partitions into smaller fragments. We define a partition to be very large if it exceeds the average partition size on that mapper as calculated up to that point by a predefined factor. Similar to partitions, *fragments* are containers for multiple clusters, and each fragment represents a histogram bucket for monitoring. The number of partitions is, however, fixed, and identical on all mappers. As for fragmentation, every mapper can decide individually whether to fragment partitions, and if so, how many of them.

As before, every mapper starts creating its output partitions according to the partitioning function π. If a partition gains excessively more weight than the others, the mapper splits this partition into fragments. We choose the number of fragments, f, to be the smallest integer greater than 1 s. t. $p \not\equiv 0 \mod f$ to avoid empty fragments caused by the hash strategy. This is shown in Figure 6.7. The leftmost mapper splits

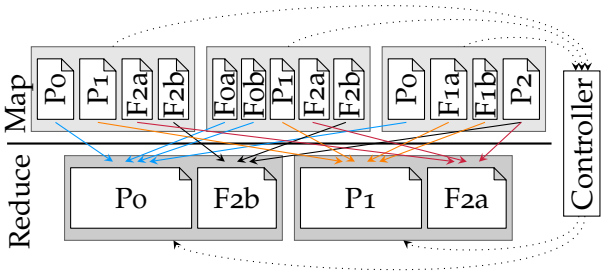


Figure 6.7: Fragmented Data Distribution

partition P2, which would have grown to almost twice the weight of the other partitions, into two fragments ($3 \not\equiv 0 \mod 2$). The mapper in the middle splits partitions P0 and P2, while on the rightmost mapper partition P1 grew too large. When a mapper splits a partition into fragments, it iterates over the intermediate tuples already in the partition, assigning them to the newly created fragments. Thereby, the monitoring data is captured on fragment level.

Upon completion, each mapper sends a list of partitions it has split into fragments, and all its fragment and partition monitoring data to the controller.

For each partition which has been fragmented on at least one mapper, the controller considers both exploiting, and ignoring the fragments. We achieve this by calculating the set R according to Algorithm 6.1 for each possible combination and then picking the best one. When exploiting fragmentation, data from mappers which have not fragmented that partition needs to be replicated to all reducers which get assigned one of the fragments. In Figure 6.7, fragment F2a is assigned to the reducer on the right, whereas fragment F2b is assigned to the left one. Partition P2 from the rightmost mapper must be copied to both reducers, as it might contain data belonging to both fragments. A filtering step is inserted at reducer side, immediately after receiving the file, in order to eliminate data items not belonging to the fragments of that reducer.

For choosing the best assignment of partitions and fragments to reducers, we employ a cost based strategy. The cost function needs to consider two aspects.

BALANCEDNESS The central aspect for load balancing is, of course, to distribute the load best possible over the reducers. We use the standard deviation σ of the weight of the bundles R each reducer gets assigned to express this aspect in our cost function. The lower the standard deviation, the better the load is balanced.

REPLICATION OVERHEAD The second point to take into account is the amount of replication involved. For this aspect, we use the average weight $\overline{w}$ of the bundles R in the cost function. We want to keep $\overline{w}$, and thus the amount of replicated data, as low as possible.

We thus define the cost of an assignment R as

$$\mathcal{C}(R) = \overline{w}(R) \cdot (1 + \sigma(R))^e$$

and strive for an assignment with low cost. We use the parameter e to adjust the influence of the balancing over the degree of replication. A low value of e implies a stronger influence of the average weight of a partition bundle. Therefore, assignments requiring replication have low cost only if they are balanced much better. A higher value of e, on the other hand, increases the influence of the load balancing aspect within the cost function. Replication is thus accepted even if the assignment obtained is only slightly better balanced. We should therefore choose e depending on the complexity of the reducer side algorithm.

Example 6.15. *In the situation depicted in Figure 6.7, the benefit of assigning fragments F2a and F2b to different reducers outweighs the increased cost resulting from the required replication of partition P2 to both reducers. Partition P1, on the other hand, was only fragmented on the rightmost mapper. Placing its fragments on different reducers would require to replicate partition P1 from both the other mappers to both reducers, which in this example would have incurred in too high expenses.*

6.6.3 *Incremental Assignment Calculation*

Both fine partitioning and dynamic fragmentation rely on a controller calculating the assignment of partitions (and fragments) to reducers. In order to calculate this assignment, the weights of all partitions (and fragments) need to be known. This, in turn, implies that we can only determine the assignment when all mappers completed their processing and sent their local histograms to the controller.

In current MapReduce the first reducers are already launched when a given percentage of mappers (default used by Hadoop: 5%) is done. During this so-called *slow-start phase*, reducers fetch their inputs from mappers which are already finished, and start sorting and merging this data. This way, some processing time can be saved later on. The actual processing of data using the user-supplied reduce function, however, can only start once all mappers are completed, and the reducers have fetched and merged all the data. The reason for this is that the reducers cannot know which

clusters are already complete and could be processed unless they retrieved their input from all mappers. Moreover, some systems guarantee to process the clusters in ascending key order.

The main focus of our work are MapReduce applications with complex reduce algorithms. In these scenarios, the time spent for preparing (i. e., retrieving, sorting, and merging) the input data to the reducers is negligible compared to the reducer processing time, and so is, therefore, the impact of omitting the slow-start phase.

Nevertheless, a MapReduce system equipped with our partitioning techniques should not unnecessarily slow down simple MapReduce tasks. For such situations, we devise an incremental calculation of the assignment function. We calculate, in these cases, the assignment of partitions to reducers based on aggregated histogram data from those mappers which are already completed. As in current MapReduce, the actual percentage of mappers after which to start this incremental calculation can be defined by the user. By setting this value to 100%, incremental assignment calculation is disabled.

As more and more mappers complete their processing and send their local histograms to the controller, the incrementally calculated assignment converges towards the assignment which would balance the partition weights best. We observe the difference between the assignment currently in use, and the last calculated assignment. If the difference grows too large, we replace the current assignment by the latest one in such a way that the least possible amount of data needs to be transferred to different reducers. The actual difference threshold triggering such a replacement depends on the complexity of the reducer side algorithm. The simpler the algorithm, the less impact on execution times a minor imbalance has. Therefore, the less complex the reducer side algorithm, the larger the tolerated difference can be. Configuring this threshold, one should however ensure that the impact of a possibly suboptimal assignment of clusters to reducers does not outweigh the benefits of the slow-start.

6.7 HANDLING LARGE CLUSTERS

The techniques presented so far aim at distributing clusters to reducers such that the resulting load on all reducers is balanced best possible. There are, however, situations in which good load balancing is not possible. Such situations arise, e. g., when we have less clusters than reducers ($|\mathbb{K}| < r$) and thus, some reducers will remain idle, or when the cluster costs are so heavily skewed that very few of the clusters make up for most of the total cost.

According to the MapReduce processing model, a single cluster must not be distributed to multiple reducers for processing. Therefore, the possibilities for the framework to react on expensive clusters are very limited. In some situations, techniques like eager aggregation [Yan and Larson 1995] can help to shift some of the workload from the reducers to the mappers. In other situations, however, such techniques may not be applicable, or they may not provide a significant benefit. Consider, e. g., a reduce task calculating the median value of every cluster. It is not possible to shift (part of) the median calculation to the mappers.

We propose to provide an optional extension to the interface, allowing the framework to notify the user code if expensive clusters are encountered. This extension consists of an additional method, `reduceHeavy`. The method's signature is identical to that of the `reduce` method. The user can then decide how to react depending on which method was called by the framework. Possible reactions could include, e. g., to process the cluster using multiple threads (if the reducer is running on a multi-core machine with a sufficient amount of free resources), or to calculate an approximate result only.

What remains for the framework to do is to recognise expensive clusters. We define a cluster to be expensive if its cost exceeds the average cluster cost by a given percentage (which can be supplied by the user, on a per-job basis).

Calculating the global average cost of all clusters is a by-product of the bin packing heuristic we employ for assigning the partitions to reducers. There, we need to determine the cost of every partition (line 4 in Algorithm 6.1). The additional overhead for summing up these weights and dividing them by the number of partitions is negligible, as is the communication for distributing the obtained value to the reducers.

The cost for each individual cluster is calculated on the reducer processing the cluster. After retrieving all relevant partitions from the mappers, the reducers sort and merge these partitions in order to bring together fragments of the same cluster produced by different mappers. It is easily possible to monitor the relevant parameters for cost estimation during this process. By collecting this information during the last merge pass, and pipelining a cluster to the user's reduce function immediately after merging it, we only need to keep track of size and tuple count of a single cluster a time. The additional memory required on every reducer for detecting expensive clusters is thus independent the number of clusters it processes.

6.8 EXPERIMENTAL EVALUATION

We experimentally evaluate the skew handling techniques presented in this chapter on both synthetic data with different distributions and skew, and on real world e-science data.

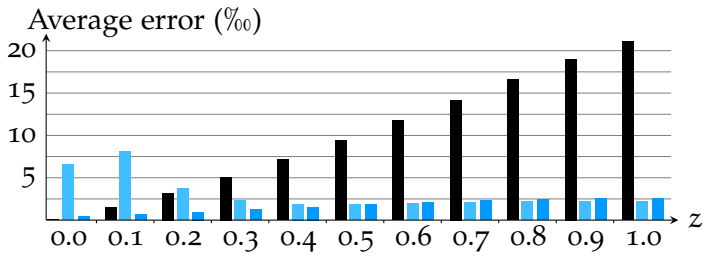
All experiments are run on a simulator. The simulator generates or loads the input data and distributes it into partitions the same way standard MapReduce systems do. The simulator allows us to generate synthetic input data with controlled skew in the key values. Further, the simulator emulates the runtime of the reducers, which provides us with the ground truth for our cost estimation.

We use the following parameters in our evaluation. The synthetic data sets follow Zipf distributions with varying z parameters. Many real world data sets, for example, word distributions in natural languages, follow a Zipf distribution. The skew is controlled with the parameter z; higher z values mean heavier skew. For the synthetic data sets we run 400 mappers that generate 1.3 million output tuples each. The total of 2,000 clusters is distributed to 40 partitions with a hash function; we found the number of 40 partitions being a typical setting for the MapReduce environments used in scientific processing. As real e-science data, we use once more the merger tree data set from the Millennium simulation [Springel et al. 2005]. We distribute the data to the mappers the same way Hadoop does, resulting in 389 mappers that each process 1.3 million tuples. We partition the data by the mass attribute, obtaining 32 000 clusters, and create 40 partitions. We repeat each experiment 10 times and report averages.

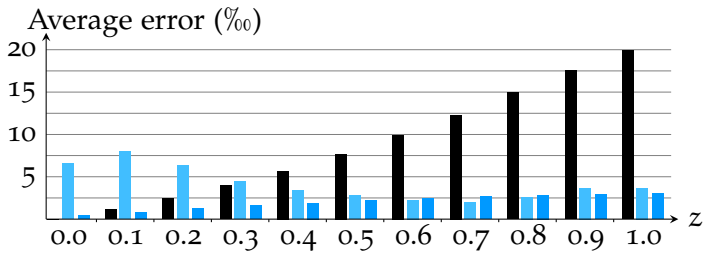
6.8.1 *Histogram Approximation Error*

We analyse the quality of the approximation obtained with both restrictive and complete TopCluster (with adaptive local thresholds) and compare it to Uniformity-Based Monitoring. We compute the approximation error as defined in Section 6.4.1.

The results for Zipf distributions with varying z parameter are shown in Figure 6.8a ($\varepsilon = 1\%$). TopCluster restrictive outperforms the other approximations in almost all settings, and the approximation error is very small (below 3‰). As expected, for heavily skewed data, TopCluster complete achieves similar or even slightly better results than TopCluster restrictive. Uniformity-Based Monitoring performs slightly better than TopCluster restrictive if the data is perfectly balanced ($z = 0$). With increasing skew, this behaviour changes, and TopCluster restrictive widely outperforms Uniformity-Based Monitoring.



(a) Zipf Distributed Data



(b) Zipf Distributed Data with Trend

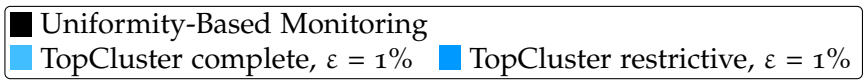


Figure 6.8: Approximation Error for Varying Skew

The difference between the restrictive and the complete variant of TopCluster is explained as follows. If the data is heavily skewed, the skew is visible on all mappers and a similar set of clusters is in the heads of all local histograms. Thus we get exact values for many clusters in the named part of the global histogram, leading to a small approximation error. There is little or no benefit in omitting clusters of size smaller than τ like TopCluster restrictive does. For moderate skew, on the other hand, the restrictive variant is beneficial since the clusters with higher approximation error are omitted. For $z = 0.1$, for instance, the approximation error is reduced by more than an order of magnitude with respect to the complete variant.

We repeat the experiments using a data distribution which simulates a trend over time (Figure 6.8b). Such trends may appear in scientific data sets, e. g., due to shifting research interests. In order to simulate a trend, we fix two Zipf distributions. For every value drawn by a mapper i, the mapper follows the first distribution with a probability of i/m, and the second distribution with a probability of $(m - i)/m$, where m is the total number of mappers. In this setting, the benefits of the restrictive TopCluster variant over the complete one become more evident, as it results in a lower error even for heavily skewed data.

6.8.2 *TopCluster: Approximation Quality vs. Head Size*

For TopCluster monitoring with adaptive local thresholds, the parameter ε controls the length of the local histogram heads, i. e., the number of cluster cardinalities that the mappers send to the controller. We evaluate the cluster quality and the size of the histogram heads depending on ε.

Approximation Quality

The results for a Zipf distribution and a distribution with trend, both with $z = 0.3$, i. e., rather moderate skew, are shown in Figure 6.9a and Figure 6.9b, respectively. The results for the heavily skewed Millennium data set are depicted in Figure 6.9c. For the complete approximation, we note an interesting effect: the error decreases for small ε values before growing again for larger values of ε. This is explained as follows. The error is minimal for ε values that allow the skewed clusters to be in the head, but the clusters with uniform distribution, which introduce the approximation error, are ignored. Very low values of ε allow too many non-skewed clusters, very high values of ε miss part of the skew. The restrictive TopCluster approximation removes poorly approximated clusters and is robust to this effect. The approximation error grows with increasing ε, i. e., the shorter the histogram head, the larger the error.

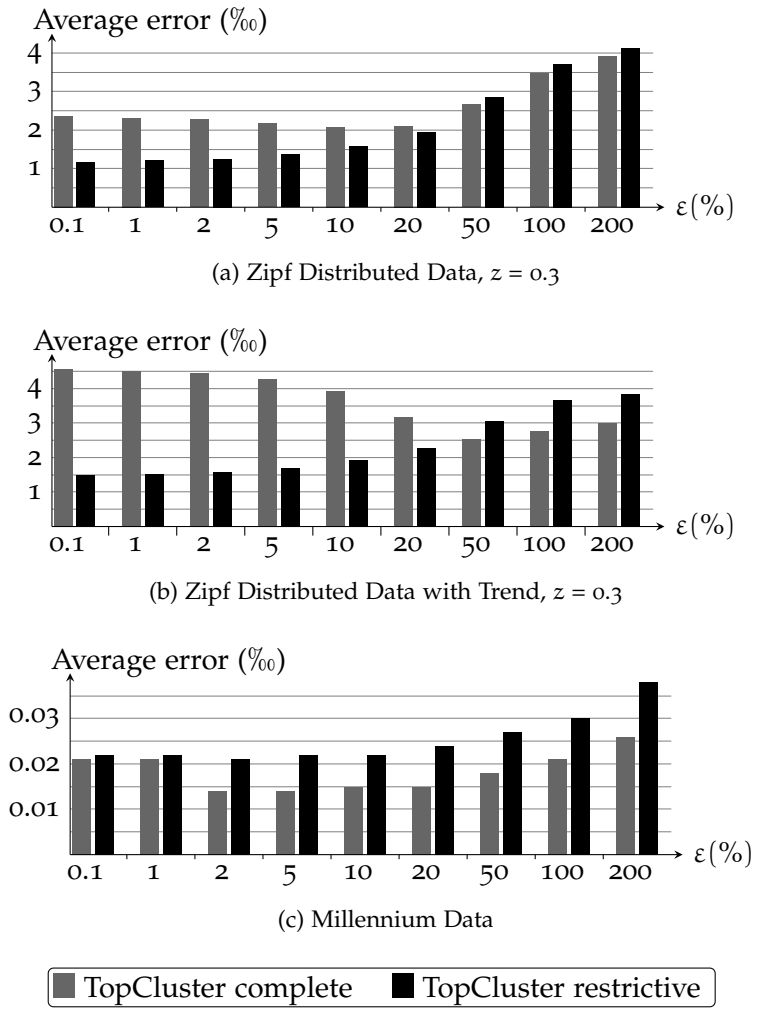


Figure 6.9: Approximation Error for Varying ε

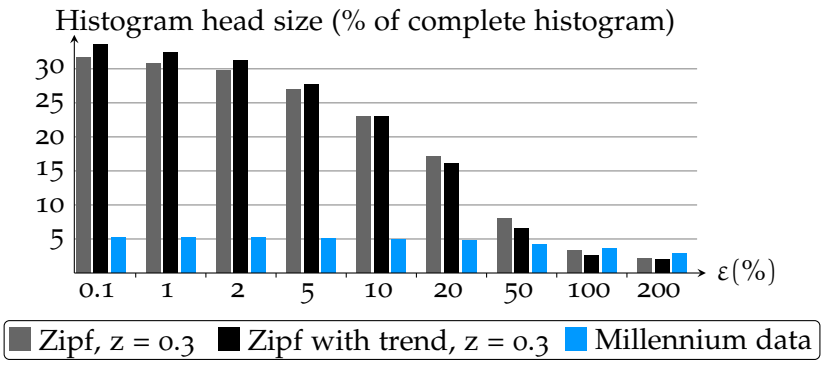


Figure 6.10: Histogram Head Size for Varying ε

For the heavily skewed Millennium data, the approximation error of both TopCluster variants is even smaller.

Histogram Head Size

We measure the size of the local histogram heads with respect to the full local histogram. Recall that the size of the histogram head is independent of the chosen TopCluster variant. Only the heads of the local histograms are sent from the mappers to the controller. Short histogram heads imply low network traffic and a small amount of calculations required on the controller. The experimental results are shown in Figure 6.10. For the synthetic data set with moderate skew ($z = 0.3$) the head size decreases to $1/3$ of the full local histogram even for a very low error margin of $\varepsilon = 0.1\%$. By increasing the error margin to $\varepsilon = 200\%$ the head size is further decreased by an order of magnitude to 2%. Note that the approximation error is below 1% also for the highest error margin in our setting (see Figure 6.9). For the heavily skewed Millennium data the histogram head is only about 5% the size of the full local histogram also for small ε values.

Overall, the results are very encouraging. Due to the high approximation quality of TopCluster, the error margin can be increased to get head sizes that are 20 to 50 times smaller than the full local histogram. Thus TopCluster scales to large data sets with good approximation quality.

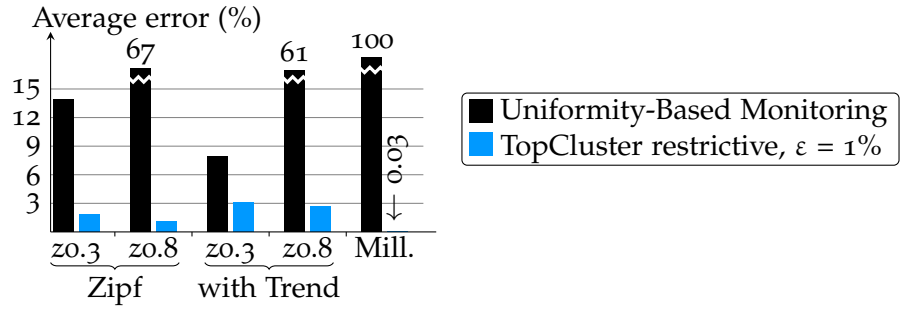


Figure 6.11: Cost Estimation Error

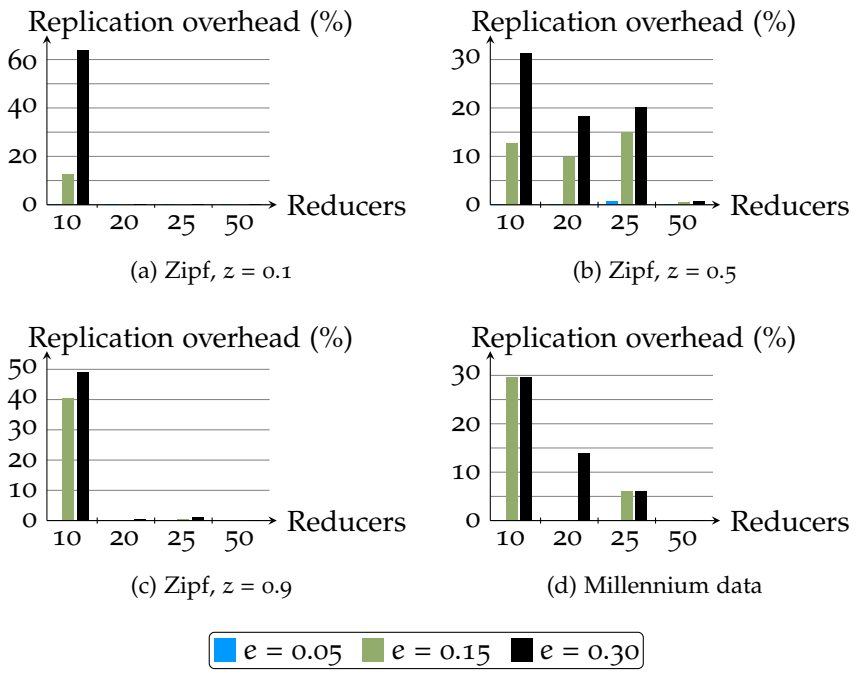
6.8.3 *Cost Estimation Error*

We measure the quality of the cost estimation for reducers with quadratic runtime and compare Uniformity-Based Monitoring to the restrictive TopCluster approximation. We use the histogram approximations to compute the expected partition cost and compare the result with the exact cost. Figure 6.11 shows the average error over all partitions. TopCluster outperforms Uniformity-Based Monitoring in all settings. As expected, the advantage of TopCluster increases with the data skew since Uniformity-Based Monitoring assumes uniform distribution of the cluster sizes within each partition. Note how the approximation error of the histogram (see Figure 6.8) is amplified by the non-linear reducer task. Recall however that current MapReduce systems do not estimate the reducer processing costs at all. Even cost estimates with the error of Uniformity-Based Monitoring result in a significantly better load balancing compared to current systems, as we will see in Section 6.8.5.

6.8.4 *Replication Overhead of Dynamic Fragmentation*

We analyse the replication overhead introduced by dynamic fragmentation for varying exponents e (0.05, 0.15, 0.3) in the cost function. We chose the number of partitions, p, to be four times the number of reducers. With this choice, we obtain a sufficient number of partitions to balance the load quite well, while not exceeding the number of clusters.

We show the results obtained for varying numbers of reducers in Figure 6.12. Dynamic fragmentation has the highest impact in the scenario with moderate skew (Figure 6.12b) and with moderate reducer count. The remaining situations are explained as follows. For very low skew (Figure 6.12a), except for the scenario with 10 reducers,



(a) Zipf, $z = 0.1$

(b) Zipf, $z = 0.5$

(c) Zipf, $z = 0.9$

(d) Millennium data

Figure 6.12: Replication Overhead of Dynamic Fragmentation with $p = 4r$

no partition grows noticeably larger than the others. Therefore, no fragments are created at all, and dynamic fragmentation cannot introduce replication. For very high skew (Figure 6.12c), the partition(s) with very expensive clusters are fragmented. The expensive clusters, however, cannot be split. Therefore, the possible gain in balancedness is low, and fragments – if considered at all – are exploited only for high values of ε. An exception is the scenario with 10 reducers. Due to the very low number of reducers, splitting the partition with the most expensive cluster allows for a significantly better load balancing. Therefore, even a high fragmentation overhead is tolerated.

We note that ε is a reasonable parameter for configuring the "aggressiveness" of the approach, i. e., the amount of replication tolerated in order to achieve better data balancing. For $\varepsilon = 0.05$, only solutions requiring a very low amount of replication are accepted. Therefore, the corresponding bars in Figure 6.12 are hardly visible. With increasing ε, more and more replication is accepted if it allows to obtain a better balancing. It is thus a reasonable approach to choose ε depending on the expected execution time of the reducers. For fast reducers, slightly skewed execution times are typically acceptable. For long-running reducers, on the other hand, the replication overhead will still be outweighed by better balancing the reducer execution times.

6.8.5 *Influence on Applications*

Finally, we evaluate the influence of our load balancing approaches on the execution times of a MapReduce application. To that end, we assume a reducer with quadratic complexity in the number of input tuples, e. g., an algorithm doing a pairwise comparison of all tuples within a cluster. We create 40 partitions on each mapper and assign them to 10 reducers. We calculate the synthetic execution time according to the algorithm complexity, based on exact cluster sizes.

Assuming that all reducers run in parallel, the slowest reducer determines the job execution time. In Figure 6.13 we compare the standard load balancing of MapReduce to both Uniformity-Based Monitoring and TopCluster restrictive. The percentage in the figure is the execution time reduction over standard MapReduce; higher bars mean shorter processing times. Both load balancing algorithms clearly outperform the standard load balancing of MapReduce. TopCluster is as good as Uniformity-Based Monitoring in the settings in which the latter is almost optimal, and better in the other setting.

For data with moderate skew ($z = 0.3$), the job execution time primarily depends on the number of clusters a reducer must process. In this configuration, even the

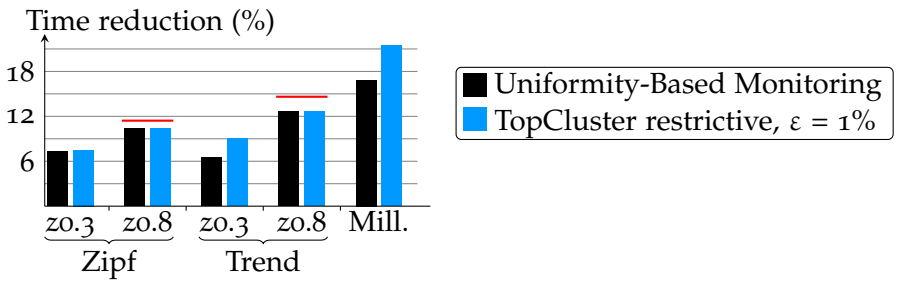


Figure 6.13: Execution Time Reduction

simple hash partitioning of current MapReduce systems results in a reasonably well balanced workload. Both our load balancing algorithms are, however, able to obtain a better balanced workload even here. Moreover, for data sets that exhibit a trend over time, TopCluster outperforms Uniformity-Based Monitoring as it is able to capture the trend within a partition.

For the data sets with $z = 0.8$, the job execution is dominated by the time required to process the largest clusters. A good load balancing algorithm should assign less partitions to reducers with large clusters. We show the highest achievable reduction of execution time as red lines in the diagram. This limit is dictated by the share of processing time required for the largest cluster in the data set. Even if this largest cluster is assigned to a dedicated reducer, the execution time cannot be reduced by more than indicated by this red line. As expected, the impact of load balancing on execution time grows with increasing skew. We note that even Uniformity-Based Monitoring is able to obtain near-optimal load balancing in these scenarios. The reason is that, for these configurations, it is sufficient to recognize partitions with expensive clusters, and a good cost approximation for the smaller clusters is less relevant.

For the heavily skewed Millennium data set distinguishing expensive from cheap partitions is not sufficient. Rather, the actual cost differences become important. Partitions with very large clusters must be assigned to a dedicated reducer, while partitions containing only moderately large clusters may share a reducer with other partitions. Assuming uniform distribution within a partition that contains a very large cluster leads to an underestimation of the partition cost. TopCluster captures the largest clusters explicitly. This has not only a significant impact on the estimated partition cost, as shown above. It also allows for a much better load balancing, as indicated by the execution time reduction for the Millennium data set in Figure 6.13.

6.9 RELATED WORK

Despite the popularity of MapReduce systems, which have been one of the centres of distributed systems research over the last years, skew handling has gained attention only very recently.

SkewReduce [Kwon et al. 2010] is a hierarchical decomposition based approach to skew handling in MapReduce systems. The default partitioning scheme of MapReduce (random partitioning for the Map phase, hash partitioning for the Reduce phase) is replaced by a hierarchical decomposition of a multidimensional space into hypercubes. The number of decomposition steps, and the actual splitting of hypercubes is determined based on user-specified cost functions. These cost functions need to reproduce the runtime behaviour of the user-defined functions employed. Moreover, they are required to fulfil several mathematical constraints. The SkewReduce approach is applicable to all situations in which splitting the input data according to attribute values allows to reduce the data volume within each partition. If too many data items are identical in all dimensions, partitioning will not be able to obtain a balanced data distribution. Moreover, the burden of designing valid cost functions is shifted to the user.

A load balancing approach for entity resolution is presented in [Kolb et al. 2012]. They introduced an additional MapReduce task for collecting the monitoring data exploited for load balancing. As the monitoring data is processed in the same distributed manner as the actual data, they can afford exact monitoring. With a complex data processing task like entity resolution on skewed data, the cost of an additional MapReduce task for load balancing is outweighed by the better balanced workload. For simple reduce tasks or well-balanced data, however, it might deteriorate the overall performance. Hence, users must decide whether to run the additional task or not. Our load balancing approaches, in contrast, are integrated into the MapReduce task processing the actual data. They are designed to cause low overhead and will therefore have no significant performance impact even for simple and well-balanced tasks. Moreover, the entity resolution task does not require clusters to be processed on a single reducer. Similar to distributed joins in databases, they only need to ensure that every pair of possibly matching tuples is processed somewhere. This allows for more flexibility in the load balancing, which is exploited in [Kolb et al. 2012].

SkewTune [Kwon et al. 2012] monitors the resource utilisation in a MapReduce cluster. If processing slots become available while a job is running, they distribute the remaining work of the worker with the largest expected time to completion such that it utilises all the free slots. Thereby, they need to stop the running worker, determine the distribution of the data which was not yet processed, and distribute that data to

the cluster hosts with free capacities. Our approach, in contrast, avoids the overhead for re-distributing data. We anticipate skewed workload distributions and adapt the initial data distribution to the reducers in a load balancing manner.

When processing joins on MapReduce systems, data skew might arise as well. A recent publication [Afrati and Ullman 2010] shows how to use Symmetric Fragment-Replicate Joins [Stamos and Young 1993] on MapReduce systems best in order to minimise communication. Based on the input relation sizes, the presented system determines the optimal degree of replication for all relations. Our work is orthogonal to this approach. Skewed join attribute distribution can lead to load imbalance on the reducers, which is tackled by the techniques we presented in this chapter.

An improved scheduling algorithm for MapReduce in heterogeneous environments was presented in [Zaharia et al. 2008]. They show that an improved scheduling strategy can effectively decrease the response time of Hadoop. The scheduling strategy determines invocation time and hosts for the single reduce tasks, but not the assignment of clusters to reducers. Their approach can thus be combined with our load balancing techniques in order to further reduce the response time.

Distributed database literature offers much prior work on handling skewed data distributions. The dynamic fragmentation approach we presented in this chapter was inspired by distributed hash join processing [Zeller and Gray 1990], extending it such that multiple mappers can contribute as data sources.

Data skew was also tackled within the Gamma project [DeWitt et al. 1992]. Some of the techniques developed there are also applicable to MapReduce. The fine partitioning approach, e. g., is similar to the *Virtual Processor Partitioning* in Gamma. Other techniques are very specific to distributed join processing and cannot be directly transferred to our scenario. An example is the *Subset-Replicate* approach. Similar to the Fragment-Replicate Join, this approach allows to distribute one cluster over multiple sites. Such a technique is not applicable to arbitrary distributed grouping/aggregation tasks, which we need for load balancing in MapReduce.

Scarlett [Ananthanarayanan et al. 2011] considers skewed popularity of data sets hosted in a MapReduce cluster. They propose to adapt the number of replica according to the popularity of a data set, thereby avoiding resource contention when accessing popular data sets. Skewed popularity of data sets is orthogonal to skewed workloads. Therefore, their approach can easily be combined with our workload balancing solution.

7 CONCLUSION AND OUTLOOK

In this thesis, we investigated means of load-balanced massively parallel distributed data exploration. We presented frequent subtree mining on e-science data from the astrophysics domain as a motivating example for complex scientific data processing on large, hierarchically structured data sets.

The volume of the data sets of interest is growing at exponential rates. Centralised data analyses are therefore no longer feasible. We analysed the possibilities of running scientific data exploration applications in a massively parallel manner, exploiting both inter-host and intra-host parallelism. We based these analyses on the popular MapReduce framework. In order to improve the performance of our distributed frequent subtree mining workflow, we proposed three extensions to MapReduce which we combined to form the Pipelined MapReduce framework. First we abolished the artificial limitation to two processing tasks per job. This allows us to translate logically connected processing steps to a single job. Thereby, we avoid storing intermediate results in the distributed file system, which severely impacts the performance. Second, we allowed every processing step to have multiple input and output data sets. Tasks in a multi-step job can thus consume the results of more than one preceding step, realising, e. g., join operations in a clear and legible manner. Third, we permitted the pipelining of intermediate results between subsequent processing tasks. Later tasks can thus start their processing before their predecessor task(s) have completed. These three extensions permit the execution of complex multi-step processing workflows like, e. g., our frequent subtree mining application, significantly faster than on plain MapReduce. We devised a local multi-threading operator for MapReduce style frameworks which allows to run sub-steps of a task in parallel on a single host. Together with cache-aware data structures, this operator allows us to exploit the capabilities provided by modern multi-core processors.

We analysed the impact of probabilistic calculations reducing the communication and synchronisation overhead. Exemplarily, we evaluated this possibility on the frequent label detection component of our frequent subtree mining workflow. Thereby, we were able to significantly reduce the communicated data volume while observing only a slight loss in precision. However, we expect such techniques to have a higher impact if applied to the processing steps which consume the largest share of processing time. For our example workflow, these would be the BI and CM phases (which are

specific to the frequent subtree mining algorithm employed). Future research should investigate this aspect.

We designed TreeLatin, a scripting language based on Pig Latin, which is tailored to distributed tree processing. TreeLatin includes statements for building, traversing and flattening tree structures. An optimising compiler translates TreeLatin scripts to workflows for the underlying MapReduce style platform. In this translation step, we apply several optimisations to the workflow in order to obtain an efficient execution plan. These optimisations include tailored techniques, like group refinement, and well-known techniques applied in relational database management systems, like projection and selection push-down.

As a next step, cost based optimisation aspects could be included in the optimiser. Cost based optimisers base their decisions on statistics on the processed data sets. Similar to relational database management systems, the statistics on the base data could be collected before processing, e. g., during a loading phase. Moreover, we also envision approaches collecting statistics on intermediate results during processing. We can then optimise a workflow in multiple steps, basing the optimisation of later steps of the workflow on the statistics collected during earlier steps. We expect such an approach to be advantageous especially with complex workflows, where the statistics on the base data are not able to provide reasonably accurate information on intermediate results.

Eventually, we analysed the problem of proper workload balancing in MapReduce style frameworks. We proposed the Partition Cost Model for balancing the workload based on estimated processing costs. We estimated the processing costs using statistical information on the input data of the task, taking into account the algorithmic complexity of the task. TopCluster, the distributed monitoring approach we designed, provides us with the required statistics on the processed data.

A possible extension to the load balancing aspect to consider in future work is the scenario of reducers combining multiple input data sets like, e. g., join operations. To that end, we need to collect multiple statistics in the map phase – one for each data set – and combine them in the cost estimation in order to properly estimate the partition costs. Moreover, our prototype implementing the Partition Cost Model currently assumes a homogeneous environment. However, the cluster running the MapReduce framework might consist of hosts with varying capabilities (e. g., different processors or varying amounts of main memory). In such an environment, balancing the workload evenly will not minimise the processing time. Instead, the capabilities of the hosts must be taken into account when distributing the workload.

In summary, in this thesis, we presented several enhancements for MapReduce style processing frameworks. Categorising these extensions very coarsely, each of them belongs to one of the following two points.

1. They simplify the development of MapReduce applications from a user's perspective by allowing the specification of the required workflow in an abstract manner.

2. They increase the responsiveness of the framework by speeding up the application processing, e. g., by means of optimisation and workload balancing.

MapReduce attributes much of its popularity to the fact that technical aspects like the parallel execution and fault tolerance handling are completely handled by the framework and thus hidden from the user. The extensions to MapReduce we proposed in this thesis shift the burden of handling more low-level aspects from the user to the framework. Therefore, we are convinced they are able to further increase the popularity of MapReduce and bring the benefits of this style of massively parallel processing to a broader audience, especially in non-IT domains.

BIBLIOGRAPHY

Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D. J., Silberschatz, A., and Rasin, A.: HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, August 2009.

Afrati, F. N. and Ullman, J. D.: Optimizing Joins in a Map-Reduce Environment. In: Manolescu, I., Spaccapietra, S., Teubner, J., Kitsuregawa, M., Léger, A., Naumann, F., Ailamaki, A., and Özcan, F. (eds.), *Proceedings of the 13th International Conference on Extending Database Technology (EDBT), ACM International Conference Proceeding Series*, vol. 426, pp. 99–110, ACM, Lausanne, Switzerland, March 2010, ISBN 978-1-60558-945-9.

Agrawal, R. and Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases. In: Bocca, J. B., Jarke, M., and Zaniolo, C. (eds.), *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pp. 487–499, Morgan Kaufmann, Santiago de Chile, Chile, September 1994, ISBN 1-55860-153-8.

Aladin: `http://aladin.u-strasbg.fr`, November 2010, UDS/CNRS.

Albutiu, M.-C., Kemper, A., and Neumann, T.: Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.

Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A. G., Stoica, I., Harlan, D., and Harris, E.: Scarlett: Coping with Skewed content Popularity in MapReduce Clusters. In: Kirsch, C. M. and Heiser, G. (eds.), *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, pp. 287–300, ACM, Salzburg, Austria, February 2011, ISBN 978-1-4503-0634-8.

Anjomshoaa, A., Brisard, F., Drescher, M., Fellows, D., Ly, A., McGough, S., Pulsipher, D., and Savva, A.: *Job Submission Description Language (JSLD) Specification v1.0*. Open Grid Forum, July 2008.

Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., and Warneke, D.: Nephele/-PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In: [Hellerstein et al. 2010], pp. 119–130.

Benson, K., Plante, R., Auden, E., Graham, M., Greene, G., Hill, M., Linde, T., Morris, D., O'Mullane, W., Rixon, G., Stébé, A., and Andrews, K.: *IVOA Registry Interfaces Version 1.0*. International Virtual Observatory Alliance, November 2009.

Bloom, B. H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

Bu, Y., Howe, B., Balazinska, M., and Ernst, M.: HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the VLDB Endowment*, 3(1):285–296, September 2010.

Chaiken, R., Jekins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J.: SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

Chattopadhyay, B., Lin, L., Liu, W., Mittal, S., Aragonda, P., Lychagina, V., Kwon, Y., and Wong, M.: Tenzing A SQL Implementation On The MapReduce Framework. *Proceedings of the VLDB Endowment*, 4(12):1318–1327, 2011.

Chi, Y., Muntz, R. R., Nijssen, S., and Kok, J. N.: Frequent Subtree Mining - An Overview. *Fundamenta Informaticae*, 66(1-2):161–198, 2005.

Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

Datta, S., Bhaduri, K., Giannella, C., Wolff, R., and Kargupta, H.: Distributed Data Mining in Peer-to-Peer Networks. *IEEE Internet Computing*, 10(4):18–26, 2006.

de Kruijf, M. and Sankaralingam, K.: MapReduce for the Cell Broadband Engine Architecture. *IBM Journal of Research and Development*, 53(5):10:1–10:12, 2009.

Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

DeWitt, D. J., Naughton, J. F., Schneider, D. A., and Seshadri, S.: Practical Skew Handling in Parallel Joins. In: Yuan, L.-Y. (ed.), *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB)*, pp. 27–40, Morgan Kaufmann, Vancouver, BC, Canada, August 1992, ISBN 1-55860-151-1.

Dittrich, J., Quiané-Ruiz, J.-A., Jindal, A., Kargin, Y., Setty, V., and Schad, J.: Hadoop++: Making a Yellow Elephant Run Like a Cheetah. *Proceedings of the VLDB Endowment*, 3(1):518–529, September 2010.

Dowler, P., Rixon, G., and Tody, D.: *Table Access Protocol v1.0*. International Virtual Observatory Alliance, March 2010.

Fagin, R., Lotem, A., and Naor, M.: Optimal Aggregation Algorithms for Middleware. *Journal of Computer an System Sciences*, 66(4):614–656, 2003.

Friedman, E., Pawlowski, P., and Cieslewicz, J.: SQL/MapReduce: A practical approach to self-describing, polymorphic, an parallelizable user-defined functions. *Proceedings of the VLDB Endowment*, 2(1):1402–1413, August 2009.

Gates, A. F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S. M., Olston, C., Reed, B., Srinivasan, S., and Srivastava, U.: Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. *Proceedings of the VLDB Endowment*, 2(1):1414–1425, August 2009.

Ghemawat, S., Gobioff, H., and Leung, S.-T.: The Google File System. In: Scott, M. L. and Peterson, L. L. (eds.), *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 29–43, ACM, Bolton Landing, NY, USA, October 2003, ISBN 1-58113-757-5.

Graefe, G. and DeWitt, D. J.: The EXODUS Optimizer Generator. In: Dayal, U. and Traiger, I. L. (eds.), *Proceedings of the 13th ACM International Conference on Management of Data (SIGMOD)*, pp. 160–172, ACM, San Francisco, CA, USA, May 1987.

Graham, M., Morris, D., Rixon, G., Dowler, P., Schaaff, A., and Tody, D.: *VOSpace specification v2.0*. International Virtual Observatory Alliance, November 2010, working Draft.

Grossman, R. and Gu, Y.: Data Mining Using High Performance Data Clouds: Experimental Studies Using Sector and Sphere. In: Li, Y., Liu, B., and Sarawagi, S. (eds.), *Proceedings of the 14th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pp. 920–927, ACM, Las Vegas, NV, USA, August 2008, ISBN 978-1-60558-193-4.

Grünheid, A.: *Probabilistic Frequent 1-Itemset Mining*. Bachelor's Thesis, Technische Universität München, 2009, Supervisor: Benjamin Gufler.

Gufler, B., Augsten, N., Reiser, A., and Kemper, A.: Handing Data Skew in MapReduce. In: Leymann, F., Ivanov, I., van Sinderen, M., and Shishkov, B. (eds.), *Proceedings of the 1st International Conference on Cloud Computing and Services Science*

(CLOSER), pp. 574–583, SciTePress, Noordwijkerhout, The Netherlands, May 2011, ISBN 978-989-8425-52-2.

Gufler, B., Augsten, N., Reiser, A., and Kemper, A.: Load Balancing in MapReduce Based on Scalable Cardinality Estimates. In: [Kementsietsidis and Salles 2012], pp. 522–533.

Gufler, B., Augsten, N., Reiser, A., and Kemper, A.: The Partition Cost Model for Load Balancing in MapReduce. In: Ivanov, I., van Sinderen, M., and Shishkov, B. (eds.), *Cloud Computing and Services Science*, Service Science: Research and Innovations in the Service Economy, Springer, 2012b, ISBN 978-1-4614-2325-6.

Harrison, P. and Rixon, G.: *Universal Worker Service Pattern v1.0*. International Virtual Observatory Alliance, October 2010.

He, B., Fang, W., Luo, Q., Govindaraju, N. K., and Wang, T.: Mars: A MapReduce Framework on Graphics Processors. In: Moshovos, A., Tarditi, D., and Olukotun, K. (eds.), *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 260–269, ACM, Toronto, Ontario, Canada, October 2008, ISBN 978-1-60558-282-5.

Hellerstein, J. M., Chaudhuri, S., and Rosenblum, M. (eds.): *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, ACM, Indianapolis, IN, USA, June 2010, ISBN 978-1-4503-0036-0.

Herodotou, H. and Babu, S.: Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, August 2011.

Hien, M.: *Optimierte Übersetzung einer Skriptsprache für ein auf MapReduce basierendes verteiltes Anfragesystem semistrukturierter Daten*. Master's Thesis, Technische Universität München, 2011, Supervisor: Benjamin Gufler.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D.: Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

IVOA: International Virtual Observatory Alliance: About IVOA. `http://www.ivoa.net/pub/info/`, May 2010.

Jahani, E., Cafarella, M. J., and Ré, C.: Automatic Optimization for MapReduce Programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, March 2011.

Januzaj, E., Kriegel, H.-P., and Pfeifle, M.: Scalable Density-Based Distributed Clustering. In: Boulicaut, J.-F., Esposito, F., Giannotti, F., and Pedreschi, D. (eds.), *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, *Lecture Notes in Computer Science*, vol. 3202, pp. 231–244, Springer, Pisa, Italy, September 2004, ISBN 3-540-23108-0.

Kementsietsidis, A. and Salles, M. A. V. (eds.): *Proceedings of the 28th IEEE International Conference on Data Engineering (ICDE)*, IEEE Computer Society, Washington, DC, USA, April 2012, ISBN 978-0-7685-4747-3.

Kolb, L., Thor, A., and Rahm, E.: Load Balancing for MapReduce-based Entity Resolution. In: [Kementsietsidis and Salles 2012], pp. 618–629.

Kwon, Y., Balazinska, M., Howe, B., and Rolia, J.: SkewTune: Mitigating Skew in MapReduce Applications. In: Candan, K. S., Chen, Y., Snodgrass, R. T., Gravano, L., and Fuxman, A. (eds.), *Proceedings of the 38th ACM International Conference on Management of Data (SIGMOD)*, pp. 25–36, ACM, Scottsdale, AZ, USA, May 2012, ISBN 978-1-4503-1247-9.

Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. A.: Skew-resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In: [Hellerstein et al. 2010], pp. 75–86.

Lemson, G., Bourges, L., Cervino, M., Gheller, C., Gray, N., LePetit, F., Louys, M., Ooghe, B., Wagner, R., and Wozniak, H.: *Simulation Data Model v1.0*. International Virtual Observatory Alliance, May 2011.

Li, F., Moro, M. M., Ghandeharizadeh, S., Haritsa, J. R., Weikum, G., Carey, M. J., Casati, F., Chang, E. Y., Manolescu, I., Mehrotra, S., Dayal, U., and Tsotras, V. J. (eds.): *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE)*, IEEE Computer Society, Long Beach, CA, USA, March 2010, ISBN 978-1-4244-5444-0.

Linderman, M. D., Collins, J. D., Wang, H., and Meng, T. H.: Merge: A Programming Model for Heterogeneous Multi-core Systems. In: Eggers, S. J. and Larus, J. R. (eds.), *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 287–296, ACM, Seattle, WA, USA, March 2005, ISBN 978-1-59593-958-6.

Loebman, S., Nunley, D., Kwon, Y., Howe, B., Balazinska, M., and Gardner, J. P.: Analyzing Massive Astrophysical Datasets: Can Pig/Hadoop or a Relational DBMS

Help? In: *Proceedings of the 11th IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 1–10, IEEE Computer Society, New Orleans, LA, USA, September 2009, ISBN 978-1-4244-5012-1.

Metwally, A., Agrawal, D., and Abbadi, A. E.: An Integrated Efficient Solution for Computing Frequent and Top-*k* Elements in Data Streams. *ACM Transactions on Database Systems (TODS)*, 31(3):1095–1133, 2006.

Moore, G. E.: Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

Nägele, L.: *Entwicklung eines Analysewerkzeugs für geschachtelte Datenkonstrukte einer prozeduralen Datenbanksprache*. Bachelor's Thesis, Technische Universität München, 2010, Supervisor: Benjamin Gufler.

Ochsenbein, F. and Williams, R.: *VOTable Format Definition v1.2*. International Virtual Observatory Alliance, November 2009.

Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A.: Pig Latin: A Not-So-Foreign Language for Data Processing. In: Wang, J. T.-L. (ed.), *Proceedings of the 34th ACM International Conference on Management of Data (SIGMOD)*, pp. 1099–1110, ACM, Vancouver, BC, Canada, June 2008, ISBN 978-1-60558-102-6.

O'Malley, O.: TeraByte Sort on Apache Hadoop. `http://www.hpl.hp.com/hosted/sortbenchmark/YahooHadoop.pdf`, May 2008.

Ortiz, I., Lusted, J., Dowler, P., Szalay, A., Shirasaki, Y., Nieto-Santisteban, M. A., Ohishi, M., O'Mullane, W., Osuna, P., the VOQL-TEG, and the VOQL Working Group: *Astronomical Data Query Language v2.0*. International Virtual Observatory Alliance, October 2008.

Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., and Stonebraker, M.: A Comparison of Approaches to Large-Scale Data Analysis. In: Çetintemel, U., Zdonik, S. B., Kossmann, D., and Tatbul, N. (eds.), *Proceedings of the 35th ACM International Conference on Management of Data (SIGMOD)*, pp. 165–178, ACM, Providence, RI, USA, July 2009, ISBN 978-1-60558-551-2.

Pike, R., Dorward, S., Griesemer, R., and Quinlan, S.: Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

Rafique, M. M., Rose, B., Butt, A. R., and Nikolopoulos, D. S.: Supporting MapReduce on large-scale asymmetric multi-core clusters. *SIGOPS Operating Systems Review*, 43(2):25–34, 2009.

Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: *Proceedings of the 13th International Conference on High-Performance Computer Architecture (HPCA)*, pp. 13–24, IEEE Computer Society, Phoenix, AZ, USA, February 2007.

Rixon, G.: *Single-Sign-On Authentication for the IVO: introduction and description of principles v1.0*. International Virtual Observatory Alliance, October 2005, note.

Schek, H.-J. and Pistor, P.: Data Structures for an Integrated Data Base Management and Information Retrieval System. In: *Proceedings of the 8th International Conference on Very Large Data Bases (VLDB)*, pp. 197–207, Morgan Kaufmann, Mexico City, Mexico, September 1982, ISBN 0-934613-14-1.

Shasha, D., Wang, J. T.-L., and Zhang, S.: Unordered Tree Mining with Applications to Phylogeny. In: Özsoyoglu, Z. M. and Zdonik, S. B. (eds.), *Proceedings of the 20th IEEE International Conference on Data Engineering (ICDE)*, pp. 708–719, IEEE Computer Society, Boston, MA, USA, April 2004, ISBN 0-7695-2065-0.

Springel, V., White, S. D. M., Jenkins, A., Frenk, C. S., Yoshida, N., Gao, L., Navarro, J., Thacker, R., Croton, D., Helly, J., Peacock, J. A., Cole, S., Thomas, P., Couchman, H., Evrard, A., Colberg, J., and Pearce, F.: Simulating the Joint Evolution of Quasars, Galaxies and their Large-Scale Distribution. *Nature*, 435:629–636, June 2005.

Stamos, J. W. and Young, H. C.: A Symmetric Fragment and Replicate Algorithm for Distributed Joins. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1345–1354, 1993.

Stonebraker, M., Abadi, D. J., DeWitt, D. J., Madden, S., Paulson, E., Pavlo, A., and Rasin, A.: MapReduce and Parallel DBMSs: Friends or Foes? *Communications of the ACM*, 53(1):64–71, 2010.

Szalay, A. and Gray, J.: Science in an Exponential World. *Nature*, 440:412–414, March 2006.

Tatikonda, S. and Parthasarathy, S.: An Adaptive Memory Conscious Approach for Mining Frequent Trees: Implications for Multi-Core Architectures. In: Chatterjee, S. and Scott, M. L. (eds.), *Proceedings of the 13th ACM SIGPLAN Symposium*

on Principles and Practice of Parallel Programming (PPOPP), pp. 263–264, ACM, Salt Lake City, UT, USA, February 2008, ISBN 978-1-59593-795-7.

Tatikonda, S. and Parthasarathy, S.: Mining Tree-Structured Data on Multicore Systems. *Proceedings of the VLDB Endowment*, 2(1):694–705, August 2009.

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., and Murthy, R.: Hive - A Petabyte Scale Data Warehouse Using Hadoop. In: [Li et al. 2010], pp. 996–1005.

Tody, D. and Plante, R.: *Simple Image Access Specification v1.0*. International Virtual Observatory Alliance, November 2009.

TOPCAT: `http://www.star.bristol.ac.uk/~mbt/topcat`, May 2011, Mark Taylor.

Vanetik, N., Gudes, E., and Shimony, S. E.: Computing Frequent Graph Patterns from Semistructured Data. In: Kumar, V., Tsurnoto, S., Zhong, N., Yu, P. S., and Wu, X. (eds.), *Proceedings of the 2nd IEEE International Conference on Data Mining (ICDM)*, pp. 458–465, IEEE Computer Society, Maebashi City, Japan, December 2002, ISBN 0-7695-1754-4.

VODesktop: `http://www.astrogrid.org/wiki/Help/IntroVODesktop`, January 2010, The AstroGrid Project.

Whang, K.-Y., Vander-Zanden, B. T., and Taylor, H. M.: A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

Williams, R., Hanisch, R., Szalay, A., and Plante, R.: *Simple Cone Search v1.03*. International Virtual Observatory Alliance, February 2008.

Xiao, Y., Yao, J.-F., Li, Z., and Dunham, M. H.: Efficient Data Mining for Maximal Frequent Subtrees. In: Wu, X., Tuzhilin, A., and Shavlik, J. (eds.), *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM)*, pp. 379–386, IEEE Computer Society, Melbourne, FL, USA, December 2003, ISBN 0-7695-1978-4.

Yan, W. P. and Larson, P.-Å.: Eager Aggregation and Lazy Aggregation. In: Dayal, U., Gray, P. M. D., and Nishio, S. (eds.), *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB)*, pp. 345–357, Morgan Kaufmann, Zurich, Switzerland, September 1995, ISBN 1-55860-379-4.

YANG, H.-C., DASDAN, A., HSIAO, R.-L., AND PARKER, D. S.: Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In: CHAN, C. Y., OOI, B. C., AND ZHOU, A. (eds.), *Proceedings of the 33rd ACM International Conference on Management of Data (SIGMOD)*, pp. 1029–1040, ACM, Beijing, China, June 2008, ISBN 978-1-59593-686-8.

YOO, R. M., ROMANO, A., AND KOZYRAKIS, C.: Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System. In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 198–207, IEEE Computer Society, Austin, TX, USA, October 2009, ISBN 978-1-4244-5156-2.

ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I.: Improving MapReduce Performance in Heterogeneous Environments. In: DRAVES, R. AND VAN RENESSE, R. (eds.), *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 29–42, USENIX Association, San Diego, CA, USA, December 2008, ISBN 978-1-931971-65-2.

ZAKI, M. J.: Efficiently Mining Frequent Trees in a Forest: Algorithms and Applications. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(8):1021–1035, 2005.

ZELLER, H. AND GRAY, J.: An Adaptive Hash Join Algorithm for Multiuser Environments. In: MCLEOD, D., SACKS-DAVIS, R., AND SCHEK, H.-J. (eds.), *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB)*, pp. 186–197, Morgan Kaufmann, Brisbane, Queensland, Australia, August 1990, ISBN 0-55860-149-X.

ZHOU, J., LARSON, P.-Å., AND CHAIKEN, R.: Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. In: [Li et al. 2010], pp. 1060–1071.