Performance Optimization Techniques and Tools for Data-Intensive Computation Platforms

An Overview of Performance Limitations in Big Data Systems and Proposed Optimizations

VASILIKI KALAVRI

Licentiate Thesis
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden
Abstract

Big data processing has recently gained a lot of attention both from academia and industry. The term refers to tools, methods, techniques and frameworks built to collect, store, process and analyze massive amounts of data. Big data can be structured, unstructured or semi-structured. Data is generated from various different sources and can arrive in the system at various rates. In order to process these large amounts of heterogeneous data in an inexpensive and efficient way, massive parallelism is often used. The common architecture of a big data processing system consists of a shared-nothing cluster of commodity machines. However, even in such a highly parallel setting, processing is often very time-consuming. Applications may take up to hours or even days to produce useful results, making interactive analysis and debugging cumbersome.

One of the main problems is that good performance requires both good data locality and good resource utilization. A characteristic of big data analytics is that the amount of data that is processed is typically large in comparison with the amount of computation done on it. In this case, processing can benefit from data locality, which can be achieved by moving the computation close to the data, rather than vice versa. Good utilization of resources means that the data processing is done with maximal parallelization. Both locality and resource utilization are aspects of the programming framework’s runtime system. Requiring the programmer to work explicitly with parallel process creation and process placement is not desirable. Thus, specifying good optimization that would relieve the programmer from low-level, error-prone instrumentation to achieve good performance is essential.

The main goal of this thesis is to study, design and implement performance optimizations for big data frameworks. This work contributes methods and techniques to build tools for easy and efficient processing of very large data sets. It describes ways to make systems faster, by inventing ways to shorten job completion times. Another major goal is to facilitate the application development in distributed data-intensive computation platforms and make big-data analytics accessible to non-experts, so that users with limited programming experience can benefit from analyzing enormous datasets.

The thesis provides results from a study of existing optimizations in MapReduce and Hadoop related systems. The study presents a comparison and classification of existing systems, based on their main contribution. It then summarizes the current state of the research field and identifies trends and open issues, while also providing our vision on future directions.

Next, this thesis presents a set of performance optimization techniques and corresponding tools for data-intensive computing platforms;

PonIC, a project that ports the high-level dataflow framework Pig, on top of the data-parallel computing framework Stratosphere. The results of this work show that Pig can highly benefit from using Stratosphere as the backend system and gain performance, without any loss of expressiveness. The work also identifies the features of Pig that negatively impact execution time and presents a way of integrating Pig with different backends.

HOP-S, a system that uses in-memory random sampling to return approximate, yet accurate query answers. It uses a simple, yet efficient random sampling technique implementation, which significantly improves the accuracy of online aggregation.

An optimization that exploits computation redundancy in analysis programs and m2r2, a system that stores intermediate results and uses plan matching and rewriting in order to reuse results in future queries. Our prototype on top of the Pig framework demonstrates significantly reduced query response times.

Finally, an optimization framework for iterative fixed points, which exploits asymmetry in large-scale graph analysis. The frameworks uses a mathematical model to explain several optimizations and to formally specify the conditions under which, optimized iterative algorithms are equivalent to the general solution.
Acknowledgments

I am deeply thankful to the following people, without the help and support of whom, I would not have managed to complete this work:

  My primary supervisor Vladimir Vlassov, for his guidance, constant feedback and encouragement throughout this work.
  My secondary advisors, Peter Van Roy and Christian Schulte, for their valuable advice and insights.
  Per Brand, Jim Dowling and Seif Haridi, for reviewing my thesis proposal and for suggesting several improvements.
  Anne Håkansson for agreeing to serve as the internal reviewer of this thesis. Her experience and invaluable feedback significantly helped improve the final version of this document.
  My co-authors, Hui Shang, Vaidas Brundza, Stephan Ewen, Kostas Tzoumas and Volker Markl. Working with them has been a great pleasure and indispensable learning experience.
  The anonymous reviewers of the accepted papers included in this thesis, for providing observant and constructive feedback.
  My emjd-dc colleagues and colleagues at KTH and UCL, especially Ying Liu, Manuel Bravo, Zhongmiao Li, Paris Carbone, Vamis Xhagjika, Ruma Paul and Hooman Peiro Sajjad, for always being available to discuss ideas, for being supportive and for the good times we spent hanging out together, in “out-of-office” contexts.
  Lalith Suresh, for serving as a “human RSS reader”, sending me comics and funny links every morning, reminding me to read the Naruto manga chapter every Wednesday and urging me to never give up.
  My family, for their continuous support, their faith in me, their patience and immeasurable understanding.

Stockholm, 11 June 2014

Vasiliki Kalavri
# Contents

## List of Figures

### I Thesis Overview
1

#### 1 Introduction
3
  1.1 Research Objectives .............................................. 4
  1.2 Research Methodology ........................................... 5
    1.2.1 General Approach ............................................ 6
    1.2.2 Challenges ................................................... 6
    1.2.3 Experimental Evaluation ................................. 7
  1.3 Thesis Contributions ........................................... 7
  1.4 Thesis Organization ............................................ 7

#### 2 Background
9
  2.1 MapReduce/Hadoop ............................................... 10
    2.1.1 The MapReduce Programming Model ..................... 10
    2.1.2 The Hadoop ecosystem ..................................... 11
    2.1.3 YARN ....................................................... 12
  2.2 Pig .............................................................. 13
  2.3 MapReduce Online ............................................... 13
  2.4 Stratosphere .................................................... 14
    2.4.1 The Nephele Execution Engine ......................... 14
    2.4.2 Stratosphere’s Programming Model ................... 15

#### 3 Performance Optimization for Data-Intensive Computation Platforms
17
  3.1 Optimization Types and Methods .............................. 17
  3.2 Summary of Proposed Optimizations .......................... 19

#### 4 Thesis Contribution
21
  4.1 List of Publications ............................................ 21
  4.2 State-of-the-Art Survey on MapReduce ....................... 21
  4.3 Integration of Pig and Stratosphere ....................... 22
  4.4 Approximate Results in MapReduce .......................... 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Results Materialization and Reuse</td>
<td>24</td>
</tr>
<tr>
<td>4.6</td>
<td>Asymmetry in Large-Scale Graph Analysis</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>Conclusions and Future Work</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Summary of Results</td>
<td>27</td>
</tr>
<tr>
<td>5.2</td>
<td>Evaluation of Results</td>
<td>28</td>
</tr>
<tr>
<td>5.3</td>
<td>Known Limitations</td>
<td>30</td>
</tr>
<tr>
<td>5.4</td>
<td>Future Work</td>
<td>31</td>
</tr>
</tbody>
</table>

Bibliography | 33   |

II  Research Papers | 37   |

6  MapReduce: Limitations, Optimizations and Open Issues | 39   |
| 6.1 | Introduction | 40   |
| 6.2 | Background | 41   |
| 6.2.1 | The MapReduce Programming Model | 41   |
| 6.2.2 | HDFS | 42   |
| 6.2.3 | YARN | 42   |
| 6.3 | Hadoop/MapReduce Limitations | 43   |
| 6.3.1 | Performance Issues | 43   |
| 6.3.2 | Programming Model Issues | 43   |
| 6.3.3 | Configuration and Automation Issues | 44   |
| 6.4 | Hadoop/MapReduce Variations | 44   |
| 6.4.1 | Performance Optimizations | 44   |
| 6.4.2 | Programming model extensions | 47   |
| 6.4.3 | Automatic tuning | 48   |
| 6.5 | Discussion | 49   |
| 6.6 | Conclusions | 51   |

Bibliography | 53   |

7  PonIC: Using Stratosphere to Speed Up Pig Analytics | 57   |
| 7.1 | Introduction | 58   |
| 7.2 | Background | 60   |
| 7.2.1 | The MapReduce Programming Model | 60   |
| 7.2.2 | Pig | 60   |
| 7.2.3 | Stratosphere | 61   |
| 7.3 | Plan Compilation | 61   |
| 7.3.1 | Plan Compilation in Pig | 61   |
| 7.3.2 | Pig to PACT Plan Translation | 63   |
| 7.3.3 | Discussion | 64   |
| 7.4 | Implementation | 64   |
| 7.5 | Evaluation | 65   |
# List of Figures

2.1 Typical Architecture of a Big Data Framework ........................................... 10  
2.2 The stages of a MapReduce job ................................................................. 11  
2.3 YARN (Hadoop v.2) Architecture ............................................................. 12  
2.4 Execution Flow of a Pig Query ................................................................. 13  
2.5 Data Pipelining between Operators in MapReduce Online. ......................... 14  
2.6 Architecture of the Nephele Execution Engine ........................................... 15  

7.1 Pig Plans for Example Query 1 ................................................................. 62  
7.2 PACT Plans for Example Query 1 ............................................................. 64  
7.3 Evaluation Results: Overhead ................................................................. 66  
7.4 Evaluation Results: Execution Time Comparison ......................................... 67  

8.1 Popularity Variation Keyword Searches over a 3-month Period ....................... 73  
8.2 The Block Sampling Technique Steps ....................................................... 77  
8.3 Evaluation Results: Performance ............................................................ 80  
8.4 Evaluation Results: Error Variation over Sampling Rate ............................. 82  
8.5 Evaluation Results: Error Range of Average Temperature Estimations .......... 82  

9.1 Flow Diagram of the Match and ReWrite Algorithm .................................... 96  
9.2 Implementation of the Reuse Framework on Pig ......................................... 98  
9.3 Bottom-Up Logical Plan Traversal ........................................................... 99  
9.4 Storing the Result of a Sub-Job ............................................................... 100  
9.5 Comparison of Query Execution Times on Pig without Materialization and Query Execution Times with Materialization. When the materialization mechanism is enabled, materialized results are produced but not exploited. .................. 101  
9.6 Comparison of Query Execution Times With and Without Using Sub-Job Materialized Results .......................................................... 102  
9.7 Speedup When Using Results of Whole Jobs ............................................. 103  

10.1 An Example Execution of SSSP .............................................................. 113  
10.2 Graph processing techniques as data flow plans ....................................... 115  
10.3 Updated Elements per Iteration. CC: Connected Components, PR: PageRank . 121  
10.4 Connected Components Execution Time per Iteration .............................. 121  
10.5 PageRank Execution Time per Iteration ................................................. 122
Part I

Thesis Overview
Chapter 1

Introduction

Recent advances in technology and the web, as well as and the decreasing cost of data storage, have motivated all kinds of organizations to capture, store and analyze vast amounts of data. The term “Big Data” is often used to describe this phenomenon, meaning information that cannot be processed or analyzed using traditional processes or tools [44], such as relational databases and data warehouses. Big data differs from traditional, relational data in several characteristics, also known as the “3 V’s of Big Data”, which are Volume, Variety and Velocity [27].

- **Volume.** The amount of data that is being collected and analyzed today is incomparable to that of few years back [40, 30]. Data surely does not fit in memory and often does not fit in the disk of a single machine either, and, therefore needs to be partitioned and distributed in several places [10, 38, 5].

- **Variety.** Data is being collected from new sources, such as sensor networks and the web. Data that was once neglected or dropped, is now stored with the hope that its analysis and combination with existing data will yield business value. Most of the data collected from the web, user interactions, social media, sensors and review sites is semi-structured or unstructured [21, 4, 14]. We need to deal with new complex types of data being represented in different ways, such as raw text, xml, json, streams of clicks, logs, graphics, audio, video and more.

- **Velocity.** The third main difference from traditional data is the rate at which data is generated. Data is nowadays generated and captured at rates similar to which we had never seen before [6, 43, 34]. For example, it is reported that 100 hours video data are being uploaded to YouTube every minute [3].

In order to minimize storage and processing costs, shared-nothing architectures of commodity machines have been favored for big data frameworks [13, 32, 10]. In these architectures, the data is partitioned and distributed in several machines, usually using

---

1 Veracity, the trustworthiness of the data source, is often considered to be the 4th V of big data. Since this work is focused on performance issues of big data frameworks, veracity is not analyzed here.
utilities of a distributed file system [9, 15, 28]. The processing system is also deployed on
the same cluster of machines, so that data locality can be leveraged.

The most popular system built based on these ideas is Google’s MapReduce [13] and
its open-source implementation, Hadoop [1]. MapReduce offers a simple programming
model for parallel data analysis. It simplifies parallel data processing by abstracting the
details of data partitioning, node communication and fault tolerance. MapReduce is a pow-
nerful model for simple analysis tasks, such as text processing and web data processing tasks.
On the other hand, it soon proved to be a poor fit for more complex tasks [38, 28, 22], such
as machine learning [19, 29], iterative [11, 16, 42] and graph processing [31, 29, 37]. As a
result, several other data-intensive frameworks were developed, offering more flexible and
efficient runtimes, while maintaining the simplicity and abstraction level of MapReduce.
MapReduce Online [12] implements data pipelining between operators and a simple form
arbitrary DAG to describe the application’s communication patterns and express data trans-
port mechanisms, Stratosphere [7] offers an extension of the MapReduce programming
model with a relational taste and runtime optimizer, ASTERIX [8] offers a storage and
computing platform for semi-structured data, HaLoop [11], Twister [16] and CIEL [33] are
optimized for iterative computations, while Pregel [31] and GraphLab [29] are specialized
systems for graph processing.

Data-intensive computation platforms, like the ones mentioned here, face several chal-
enges that, if not dealt with carefully, result in inefficiencies and poor performance [38, 10,
5]. In order to make the power of big data frameworks accessible to non-technical users and
relieve the programmer from error-prone, low-level tuning, automatic optimization must be
performed [41, 35]. In this thesis, we examine how to provide performance optimizations
for big data frameworks. Our goal is to make data analysis faster, while minimizing manual
tuning and user intervention.

1.1 Research Objectives

The main goal of this work is to research methods and techniques, as well as to build tools
for easy and efficient processing of very large data sets. We aim at making systems faster,
by inventing ways to shorten job completion times. We are working towards facilitating the
application development in distributed data-intensive computation platforms. Our vision is
to make big-data analytics accessible to non-experts, so that users with limited program-
ing experience and understanding of distributed architectures can benefit from analyzing
enormous datasets. In this direction, we specify optimizations with the objective of reliev-
ing the programmer from low-level, error-prone instrumentation.

In order to speed up big-data analysis, we design and apply performance optimiza-
tions. More specifically, we focus on performance optimizations for batch-processing,
data-intensive computation platforms. To achieve a large impact and wide applicability
of our research, we have chosen to target system runtimes optimizations, rather than fo-
cusing on a particular application domain. Therefore, we are not interested in cases of
application-specific performance tuning and profiling. Instead, we aim at designing and in-
1.2. RESEARCH METHODOLOGY

tegrating optimization techniques, that would result in more efficient query planning and execution.

From a high-level view, there are two main factors that significantly impact the execution time of a data-intensive application, assuming a static execution environment and available resources: (a) the amount of data that needs to be processed and (b) the amount of computations that need to be performed on this data. Naturally, processing more data requires more time and also, performing more computations slows down analysis. Thus, in order to increase the speed of analysis, we define two main corresponding goals:

1. to reduce the **size of datasets** to be processed, and
2. to reduce the **amount of total computation** to be performed

With respect to these objectives, [25] is a study performed in order to reveal optimization opportunities, [26] presents an optimization towards both objectives, [23] presents work done with respect to reducing dataset sizes and [24] contains work towards the objective of reducing computation.

Finally, we define the following secondary goals, as design directives for all the optimizations presented in this thesis:

- **Application transparency.** It is our goal to make all proposed optimization techniques applicable to existing applications. Ideally, the user should be able to benefit from the optimization, without having to re-write or re-compile their application. In order to achieve this goal, we try to design our optimizations without changing the systems’ application programming interface and programming model.

- **System independence.** We strive to make our optimizations as generic as possible, so that they are applicable to a variety of big data processing systems. The optimizations presented in this thesis span three different and widely used computation platforms. To meet this design objective, we carefully study each system’s architecture, in order to identify the appropriate layer, where our technique can be integrated, while being as decoupled as possible from the specific system.

- **Minimum user involvement.** We strongly believe that in order to relieve users from error-prone and low-level tuning, optimizations should minimize user involvement. With respect to this goal, we are trying to automate the optimization techniques and avoid manual configuration, whenever possible.

### 1.2 Research Methodology

In this section, we provide a high-level view of the methods used in this research work. We give a summary of general principles that we followed and design decisions that we made in order to achieve our goals. We also discuss several challenges that we had to face and how we chose to overcome them. Since several different systems were used to implement each of the optimization techniques presented here, we adjusted our methods according to
the characteristics of each particular system. The reader can find detailed methodologies for each of the optimization techniques, in each of corresponding the papers, which appear in the second part of this thesis.

1.2.1 General Approach

This work does not follow analytical, mathematical optimization methods, but is rather based on an empirical approach. Our approach was based on first identifying performance bottlenecks and limitations in data-intensive computation platforms and then designing and implementing techniques to overcome these limitations. First, we conducted a literature study of recent research papers on the limitations and optimizations for big data platforms. The results of this study allowed us to become familiar with the state of the art in the research field and revealed open issues, that gave us an insight on possible problems to tackle. Next, in order to identify a potential bottleneck, we studied the programming model, architecture and implementation of several big data platforms. After having collected candidate opportunities for performance improvement, we verified the existence of the bottleneck, by experiment. Once we had confirmed the presence of a shortcoming, we designed a technique to improve performance, by either trying to reduce the size of the processed datasets or reduce the amount of the performed computation, as stated in our research goals. Next, we implemented our optimization technique, while trying to meet our secondary goals of transparency, system independence and minimum user involvement. Finally, we evaluated our implementation by comparing the modified system which uses our optimization technique, to the original unmodified system. For the evaluation, we used a set of representative applications and real-world datasets, where possible. We used total execution time as the performance measure and also computed overheads separately, when necessary.

1.2.2 Challenges

We faced a number of challenges throughout this work. First, a lot of the systems described in literature were not freely available to use and experiment with. In such cases, we first contacted the corresponding authors and tried to get access. Unfortunately, in all cases, we received negative responses. Therefore, we used freely available, open-source systems to implement our techniques. Second, when working on the m2r2 system, we were not able to find an available real-world workload to use for our experimental evaluation. Instead, we found a plethora of published studies, which described the characteristics of typical workloads that are used in big companies and organizations. In order to evaluate our implementation, we created a synthetic workload, based on the characteristics of these studies. The details are available in [24]. Finally, it was often challenging to properly configure and tune the systems used. For our experiments, we used configurations based on “best practices” suggested by the systems’ creators and we often got engaged with the corresponding developer communities, in order to assure the correctness of our approaches.
1.3. THESIS CONTRIBUTIONS

1.2.3 Experimental Evaluation

For our experiments, we always chose the latest—at the time of the research work—stable version of the system considered. We also chose publicly available datasets, whenever possible and share our setup configuration in each of our papers, in order to facilitate reproducibility. We conducted all of our experiments using virtual machines in a cloud environment. The use of virtual machines allowed us to create a clean, isolated environment, where only the necessary tools were installed. In order to choose representative applications for our evaluation, we either chose applications which were described in the original research papers of each system or applications that fit the specific domain that our optimization technique was targeting.

1.3 Thesis Contributions

The contributions of this thesis are as follows.

1. A survey of MapReduce and Hadoop related systems, that summarizes limitations of the Hadoop system that might lead to performance or other issues. A set of proposed optimizations and a description of systems that implement the proposed solutions. The survey concludes with open issues, topics for further research and optimization opportunities in the MapReduce context.

2. PonIC, a system that combines Pig, a high-level language for MapReduce, with a more suitable runtime, namely Stratosphere. This work demonstrates how high-level dataflow system can be integrated with a flexible execution engine and transparently gain performance.

3. HOP-S, a system that exploits partial job execution and a novel sampling technique to accurately approximate results of MapReduce jobs.

4. m2r2, a framework which avoids redundant computations for a batch of queries submitted in the same cluster. It uses results materialization and plan matching to reuse intermediate results, already stored in the system’s repository.

5. A categorization and formalization of optimizations for large-scale graph-processing. The work presents an analysis of cases when asymmetry appears in graph processing and explains how this characteristic can be exploited to speed up graph applications.

1.4 Thesis Organization

The rest of this thesis organized as follows. Chapter 2 gives the background and describes the systems used in this research work. Chapter 3 provides an overview of optimization types for data-intensive computation platforms and describes the optimizations proposed in this thesis. Thesis contributions are discussed in Chapter 4. Chapter 5 contains conclusions, a summary of the results of this thesis, as well as a discussion on known limitations and future work. Part II contains the research papers produced during this work.
Chapter 2

Background

The main characteristic of a data-intensive application, as opposed to a computation-intensive application, is that the amount of data is large in comparison to the amount of computation that needs to be done on them. Since the amount of data is so large, data-intensive applications are distributed and require parallel processing, in order to achieve good performance. In such a highly-parallel environment, where data transfers are expensive, minimizing node communication is of essence. One of the prevalent design ideas towards this direction suggests moving the computation close to the data. Thus, both the storage and the processing systems run on the same nodes.

Big data frameworks are typically organized in a layered architecture, like the one shown in Figure 2.1. There are three main components in a big data system:

- **The programming model.** The programming model contains all the elements that allow users to write big data analysis applications. Most frequently, it is a dataflow model, providing operators and data types. Operators describe processing steps, which are linked together to create a data flow. Users can usually specify functions that are encapsulated in operators and define transformations on datasets. The majority of big data systems offer even higher levels of abstractions, such as declarative or SQL-like programming interfaces, domain-specific languages for specific classes of applications, such as machine learning and specialized libraries.

- **The runtime system.** The runtime automatically handles all or some of the following. Data partitioning and distribution, compilation of the user code into parallel tasks, physical plan construction, scheduling and monitoring parallel tasks, resource management and fault-tolerance.

- **The persistent storage.** The storage layer usually consists of a distributed file system, but most big data systems have interfaces for communication with other types of storage as well, such as databases and key-value stores.

In the rest of this chapter, we describe the systems used in the papers included in this thesis; MapReduce/Hadoop [13, 1], a popular data-parallel programming model and its open-source implementation; Pig [18], a platform for high-level data analysis on top of
CHAPTER 2. BACKGROUND

2.1 MapReduce/Hadoop

2.1.1 The MapReduce Programming Model

MapReduce is a data-parallel programming model, designed to efficiently execute programs on large clusters of commodity machines. The model assumes that a distributed file system is deployed on the same cluster. Hadoop, the most popular open-source implementation of MapReduce, uses the Hadoop Distributed File System, HDFS [39]. MapReduce is inspired by functional programming and provides two operators, map and reduce, which form a static pipeline. In the simplest case, a MapReduce user only needs to specify the two functions to be executed by these two operators.

The stages of a MapReduce job are shown in Figure 2.2. In a MapReduce application, input is read from the distributed file system and is parsed as key-value pairs. Next, the pairs are processed by parallel map tasks, which apply the user-defined function, producing intermediate key-value pairs. These intermediate results are sorted and grouped by key. Each group is then individually processed by parallel reduce tasks. The partitioning stage of the framework guarantees that all pairs sharing the same key will be processed by the same reduce task. A user-defined function is applied to each group, producing the output. Finally, each parallel reducer creates a file in the distributed file system containing its results. One of the most important advantages of the schema described above is that most
2.1. MAPREDUCE/HADOOP

Figure 2.2. The stages of a MapReduce job

of the parallelization complexity is handled by the framework. The user only has to write the first-order function that will be wrapped by the Map and Reduce functions. However, this advantage comes with loss of flexibility. Each job must consist of exactly one Map function followed by one Reduce function, and no step can be omitted or executed in a different order. Moreover, if an algorithm requires multiple Map and Reduce steps, these can only be implemented as separate jobs, and data can only be passed from one job to the next through the file system. This limitation can frequently add a significant overhead to the execution time.

2.1.2 The Hadoop ecosystem

Hadoop [1] is an open-source Apache project that provides an implementation of the MapReduce programming model. It includes a runtime and execution engine for scheduling, running and monitoring MapReduce jobs. Moreover, it provides a distributed file system [39] that serves as the default input and output source of MapReduce jobs. HDFS is a fault-tolerant and scalable file system, optimized for large files and batch processing, rather than interactive use. It implements a write-once-read-many access model and automatically handles replication and load balancing. Apart from MapReduce and HDFS, there exists a variety of Hadoop-related and Hadoop-compatible frameworks and tools, which form the fast-growing Hadoop ecosystem. These tools offer services, such as distributed coordination, monitoring, serialization, machine-learning libraries, high-level languages, databases.
2.1.3 YARN

YARN, Yet Another Resource Negotiator, is often referred to as the Next Generation Hadoop. The goal of this component is to allow the system to serve as a general data-processing framework. It supports programming models other than MapReduce, while also improving scalability and resource utilization. YARN makes no changes to the programming model or to HDFS. It consists of a re-designed runtime system, aiming to eliminate the bottlenecks of the master-slave architecture. Figure 2.3 shows YARN’s novel architecture. The responsibilities of the JobTracker are split into two different processes, the ResourceManager and the ApplicationMaster. The ResourceManager handles resources dynamically, using the notion of containers, instead of static Map/Reduce slots. Containers are configured based on information about available memory, CPU and disk capacity. It also has a pluggable scheduler, which can use different strategies to assign tasks to available nodes. The ApplicationMaster is a framework-specific process, meaning that it allows other programming models to be executed on top of YARN, such as MPI, Spark [42] and Stratosphere [7]. It negotiates resources with the ResourceManager and supervises the scheduled tasks.
2.2 Pig

Pig [36, 18] is a high-level system that consists of a declarative scripting language, Pig Latin, and a set of compilers that create MapReduce data-flows from the high-level scripts. Pig’s abstraction aims to hide the complexity of the MapReduce programming model and allows users to write SQL-like scripts. Pig Latin scripts consist of statements, which define transformations on data collections. Most of the common operators are already provided by Pig, but users can also define their own custom functions.

Figure 2.4 shows the execution flow of a Pig query. The input of the system is a script, consisting of Pig Latin statements. The script is parsed and generates a logical plan of operators. The logical plan contains information what kind of operations will be executed. It connects the operators in a directed acyclic data flow graph. The logical plan is then translated into a physical plan. The physical plan is a DAG of physical operators and contains information about how the operations are going to be executed, physically. Finally, the physical plan is translated into a set of MapReduce jobs and subsequently submitted to the Hadoop execution engine.

2.3 MapReduce Online

MapReduce Online [12] is a modified version of Hadoop MapReduce. It supports Online Aggregation and stream processing, while also improving utilization and reducing response time. Traditional MapReduce implementations materialize the intermediate results of mappers and do not allow pipelining between the map and the reduce phases. This approach has the advantage of simple recovery in the case of failures, however, reducers cannot start executing tasks before all mappers have finished. This limitation lowers resource utilization and leads to inefficient execution for many applications. The main motivation of MapReduce Online is to overcome these problems, by allowing pipelining between operators, while preserving fault-tolerance guarantees.
CHAPTER 2. BACKGROUND

Figure 2.5 shows the data pipelining operation in MapReduce Online. On the left, the original MapReduce execution is shown. After the mapper tasks have finished execution, they write their intermediate results in local disk. The reducer tasks are idle until then. Once all mappers have finished, the reduce tasks get notified and retrieve input data from the local disks of map tasks, by issuing an HTTP request.

2.4 Stratosphere

Stratosphere is a general-purpose data-processing framework. It provides a programming model for writing parallel data analysis applications, which extends the MapReduce model with additional operators, allowing more than one inputs. Its execution engine is flexible and accepts jobs in the form of arbitrary DAGs of operators. Stratosphere also supports iterative computations and is compatible with YARN.

2.4.1 The Nephele Execution Engine

Nephele is an execution engine designed to execute DAG-based data flow programs. It takes care of task scheduling and setting up communication channels between nodes. Moreover, it supports dynamic allocation of resources during execution and fault-tolerance mechanisms. The architecture of the Nephele execution engine is shown in Figure 2.6. Like Hadoop, it is designed based on a master-slave model, where the master (Job Manager) is responsible for coordinating the task execution on the slave nodes (Task Managers). The Instance Manager monitors the resources in the cluster and communicates with the scheduler in order to decide on which physical nodes to spawn the parallel worker tasks.

The programming model provided with Stratosphere is PACT. However, it is possible to submit jobs directly to the Nephele engine, in the form of Directed Acyclic Graphs (DAGs). Each vertex of the graph represents a task of the job. There are three types of vertices: Input, Output and Task. The edges of the graph correspond to the communication channels between tasks.
2.4. STRATOSPHERE

2.4.2 Stratosphere’s Programming Model

The PACT programming model is a generalization of the MapReduce programming model. It extends the idea of the map and reduce second-order functions, introducing the Input Contracts. An Input Contract is a secondary function that accepts a first-order user-defined function and one or more data sets as inputs. Input Contracts do not have to form any specific type of pipeline and can be used in any order that respects their input specifications. In the context of the PACT programming model, Map and Reduce are Input Contracts. Apart from these two, three more Contracts are defined:

- The Cross Input Contract accepts multiple inputs of key value pairs and produces subsets of all possible combinations among them, building a Cartesian product over the input. Each combination becomes then an independent subset.

- The Match Contract operates on two inputs and matches each pair of the first input with one pair of the second input that has the same key value. This contract naturally maps to an inner join, in terms of database semantics.

- Finally, the CoGroup function creates independent subsets by combining all pairs that share the same key.
Iterations in Stratosphere

Iterations are implemented in Stratosphere as composite operators, which encapsulate the step function and the termination criterion. The optimizer creates a physical execution plan based on the first iteration and caches the constant path’s result in memory.

Stratosphere’s iterations are very well-suited for fixed point computations. The iteration operator receives a dataset as input, which contains the initial values of some user-defined parameters. This dataset is called the *solution set*. The second input to the iteration operator is the *dependency set*. This dataset contains information on how the parameters of the solution set depend on each other. The user also needs to define an *update function*, which encapsulates the computation logic of the iterative program. The update function can be composed by one or more Stratosphere operators. In a typical iterative Stratosphere program, the solution set is joined with the dependency set and the update function is applied to the resulting dataset. The newly computed parameters are then used to update the solution set elements. The program halts when a convergence criterion has been met or a maximum number of iterations has been reached.

Stratosphere iterations are *synchronous*. All the worker tasks work in well-defined steps and get synchronized by a barrier at the end of each iteration. The are two different iteration types by default: *bulk* and *incremental*. From a high-level point of view, a bulk iteration is one that, in each iteration, performs computations on all the elements of the solution set. An incremental iteration, uses an auxiliary dataset, the *workset*, to keep track of updated elements and uses only these elements to perform computations in subsequent iterations. More details on the implementation of iterative applications in Stratosphere can be found in [17].
Chapter 3

Performance Optimization for Data-Intensive Computation Platforms

In this chapter, we present an overview of general optimization techniques for data processing. Most of these techniques have been invented and researched in the context of relational database systems. We present a categorization of optimizations based on their main objective and we describe popular existing techniques that fit into each category. Next, we discuss the challenges in applying existing database optimizations to general-purpose big data systems and refer to profound research work in this area. An overview of the state of the art in optimization in data-intensive computation platforms can be found in Chapter 6. Finally, we present a summary of the optimization techniques presented in this thesis.

3.1 Optimization Types and Methods

We categorize optimizations based on their main objective, namely Performance, Ease-of-use and Cost reduction. Performance optimizations aim at reducing job execution time and making data processing faster. The goal is to provide faster answers to user queries. Ease-of-use optimizations aim at making data processing easier, from a user perspective. These techniques intent to automate the complex parts of data analysis and facilitate application development and deployment. Cost reduction optimizations refer to methods whose purpose is minimizing operating costs of data analysis systems.

- **Performance Optimizations.** Techniques in this category can be further divided into two main subcategories: (a) single-program optimizations and (b) batch optimizations. Single-program optimizations focus on optimizing a single application at a time. In other words, these techniques consider the characteristics of an application when executed separately. They do not take into account interactions with other applications running at the same time and do not consider possible optimization opportunities in this context. Batch optimizations are techniques that intend to optimize a batch or workload of applications as a whole. These techniques view the system as an environment where applications are running at the same time or during a specified
period of time.

**Single-program optimizations.** A lot of techniques in this category are based on extracting information about the input and intermediate datasets. These optimizations aim at inventing more efficient ways of executing a query. Typically, a user writes a data processing application in some language and the data processing system is responsible for defining an execution strategy for running this application, using the available resources. The process of translating the user’s application into a form that the system can understand and execute is called query planning. Widely used optimizations in this category concern choosing an efficient query execution plan. These techniques usually involve methods for accurate dataset size estimation and estimation of key cardinalities. Static code analysis is another popular method used. An example of a concrete query plan optimization technique is operator reordering.

Another class of optimization techniques in this category is concerned with how to efficiently access data in the processing system’s storage. The goal is to minimize communication and random data accesses. In order to reduce data I/O, several smart data placement and partitioning strategies have been developed. For example, one can predict which data blocks are likely to be accessed at the same time and try to co-locate them. Related issues include efficient data representation, compression techniques, erasure coding and amortizing data skew.

An optimization technique that is very useful when the amount of data to be processed is very large is result approximation. The goal of this class of techniques is to return a result to the user query, as soon as possible, before the analysis has finished. Partial job execution can yield very fast results but it has to be paired with a robust results estimation method. Sampling histogram building and wavelets are popular choice in this area. Regarding sampling, techniques can utilize online sampling or use pre-cached samples of the input or pre-processed datasets. These techniques are especially efficient and accurate when the queries are known beforehand and the system has adequate information about the data distribution.

**Batch optimizations.** The main goal of batch optimization techniques is to increase the performance of an execution environment, such as a cluster or datacenter, as a whole. Therefore, these techniques often refer to metrics like system throughput and query latency. A big class of batch optimization techniques aim at efficiently scheduling and managing applications running in the same environment. In distributed setups, load balancing is another very important issue that significantly affects performance and it is also very closely related to scheduling. Another popular technique makes use of work sharing for simultaneous queries. This optimization considers a batch of jobs submitted for execution and re-orders them to enable sharing of execution pipelines. A related technique is sometimes referred to as non-concurrent work-sharing and aims at avoiding redundant computations by materializing and reusing results of previously executed jobs.

- **Ease-of-use** Ease-of-use optimizations aim at making it easier for users to conduct data analysis. They mainly include tools that analysts can use to boost their produc-
3.2. SUMMARY OF PROPOSED OPTIMIZATIONS

tivity and avoid bad development practices that may lead to buggy programs. These tools for easy application development automate parts of the analysis process and abstract low-level details from the user, for example parallelization and data distribution. Another set of techniques facilitate the deployment and maintenance of analysis applications on complex environments. They might perform automatic system configuration and load balancing or handle fault-tolerance. Related tools facilitate testing and debugging of data analysis programs, tasks that are especially challenging in distributed data processing.

• **Cost reduction** The last category of optimizations primarily intend to minimize the cost of data processing. Such optimizations include tools for reducing storage requirements, for example by using data deduplication techniques or erasure coding. A big class of cost reduction optimizations also concerns resource and cost-aware scheduling. The system has information about the cost of different resources and can evaluate different execution strategies and deployments, in order to efficiently utilize resources and reduce operational costs. Resource utilization optimizations usually also benefit the performance and throughput of a system.

3.2 Summary of Proposed Optimizations

In Chapter 6, we present an overview of the state of the art in Hadoop and MapReduce optimizations. We identify the limitations of the model and its implementation and categorize different systems that extend MapReduce and Hadoop. We summarize the state of the research field and identify open issues and chances for further optimizations. In Chapters 7, 8 and 9 we propose three performance optimization techniques and we describe their implementation and evaluation in popular data-intensive computation platforms.

First, we present an integration of the Pig high-level processing system with Stratosphere. We give a translation process that allows Pig Latin programs to be transparently executed on top of Stratosphere’s flexible execution engine. We identify the features of Pig that negatively impact execution time and we show that by harnessing an alternative execution engine, like Stratosphere, Pig can gain performance, without any loss of expressiveness.

Then, we present an efficient sampling technique that can be used in order to get accurate results, before job completion. Our technique uses block sampling and in-memory shuffling of records to increase accuracy and allow the system to return meaningful results, early. Our evaluation results show that our technique can provide high accuracy for certain classes of applications, after only processing a very small percentage of the input datasets.

Next, we describe a framework for results materialization and reuse in high-level data-flow big data systems. Our framework is independent of the high-level language and the execution engine. We describe an algorithm for matching execution plans and for rewriting submitted queries, in order to benefit from already stored results. Our results show that, when there exists sharing opportunity, query execution time can be significantly reduced, while the imposed overhead is small.
Finally, in Chapter 10, we describe an optimization framework for iterative, fixed-point iterations. We show how dataset dependencies can be used to speed up iterative processing, using a mathematical model. We show how each of the optimizations proposed can be implemented in a general, data-flow processing systems and we provide present template execution plans, in Stratosphere. Our evaluation shows that algorithms and datasets that expose asymmetrical characteristics can have order-of-magnitude gains, when using the optimized execution plans.
Chapter 4

Thesis Contribution

In this chapter, we describe the thesis contributions. First, we list the publications produced during this work. Then, we provide details on the contributions of each publication separately and highlight the individual contributions of the thesis author.

4.1 List of Publications

1. V. Kalavri, V. Vlassov, *MapReduce: Limitations, Optimizations and Open Issues*, 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2013


5. V. Kalavri, S. Ewen, K. Tzoumas, V. Vlassov, M. Volker, S. Haridi, *Asymmetry in Large-Scale Graph Analysis, Explained, To Appear* in 2nd International Workshop on Graph Data Management Experiences and Systems, (GRADES), 2014

4.2 State-of-the-Art Survey on MapReduce

Our study on MapReduce limitations and related systems is published as a research paper [25] and appears as Chapter 6 in this thesis.
CHAPTER 4. THESIS CONTRIBUTION

Paper Contribution

Among the numerous MapReduce implementations that have been proposed, Hadoop is unquestionably the most popular and widely adopted. Hadoop provides an implementation of the MapReduce programming model and also includes a distributed file system and a resource management layer. Despite its popularity, Hadoop has also been criticized for some of its characteristics, such as the static map-shuffle-reduce pipeline and the intensive disk I/O. The lack of support for iterations and indices have also been identified as limiting factors for efficient complex analysis. Configuration and tuning of the system are also two tasks that may be cumbersome and time-consuming for the user. In order to overcome Hadoop’s limitations and unsuitability for some classes of applications, several variations of the system have been developed. These variations include performance optimizations, extensions of the programming model and configuration automation.

In this survey, we first present a comprehensive background on the MapReduce programming model and the Hadoop implementation. We discuss advantages of the model and we describe components, such as the distributed file system and the runtime architecture. We also describe the latest version of the Hadoop implementation, YARN. In this architecture, the resource management is decoupled from the programming model, in order to make the system more scalable and increase resource utilization. Next, we discuss several limitations of the model and the implementation in particular. We classify the identified limitations in three categories: (a) performance issues, (b) programming model issues and (c) configuration and automation issues. We continue by presenting an overview of the state-of-the-art in Hadoop and MapReduce optimizations. We categorize each optimization based on its type and on whether it mainly targets performance, programming model or configuration issues. We describe the systems that implement the optimizations and extensions and provide a comparison of their features. Next, we make an extensive discussion on the current state of the research field. We list resolved problems and we identify trends and open issues, Finally, we describe our own vision regarding possible future directions.

Thesis Author Contribution

The author is the main contributor in this work. She defined the categorization of limitations and selected the systems that are discussed in the paper. She is also responsible for the proposed optimizations categorization and the discussion on open issues and future directions.

4.3 Integration of Pig and Stratosphere

The integration of the Pig system with the general-purpose data processing system, Stratosphere is published as a research paper [26] and appears as Chapter 7 in this thesis.
4.3. INTEGRATION OF PIG AND STRATOSPHERE

Paper Contribution

The paper presents an integration of the Pig System with Stratosphere. Pig is a high-level system for data processing on top of Hadoop. It provides an abstraction that hides Hadoop’s one-input and two-stage dataflow limitations from the user. Having Hadoop as its backend, Pig directly benefits from Hadoop’s scalability and fault-tolerance. At the same time, though, it suffers from the limitations and inflexibility of Hadoop, which often translate to inefficient execution. The Pig system offers common relational operators out-of-the-box, which are translated into the static pipeline of Hadoop, in the backend. This translation often produces an inefficient execution plan, since data have to be materialized and replicated after every MapReduce step.

In order to provide a more flexible execution engine for Pig and improve its performance, we first analyze the internal structure of the system. Next, we designed a suitable integration strategy with the Stratosphere platform, having transparency to user applications as our main goal. In order to evaluate the benefits of the integration, we develop a prototype implementation. Our prototype proves that it is possible to plug a different execution engine into the Pig system. More importantly, we identify the parts of the Pig that can be reused and the point where the integration is feasible. We provide a Pig to PACT translation process for the most common Pig Latin operators and we show that Stratosphere has desirable properties that can significantly simplify the plan generation. Regarding evaluation of our proposed solution, we present a set of basic scripts, using combinations of the translated Pig operators and their native MapReduce and Stratosphere equivalents. We provide a comparison of PonIC with Pig, as well as the corresponding native programs.

Our integration is entirely transparent to Pig’s end-users, meaning that existing Pig scripts can be executed on PonIC, without any modification. We keep the syntax and the semantics of the operators completely unchanged. We show that it is possible to integrate Pig with alternative execution engines and we present one way that this can be achieved. We also identify the characteristics of Pig that negatively impact execution time, mainly due to Hadoop inflexibilities and intensive disk I/O. We show that Pig can be integrated with Stratosphere and that the integration outperforms the original Pig implementation, for some classes of applications. Finally, we describe a complete translation process of Pig logical operators into Stratosphere operators.

Thesis Author Contribution

The author is the main contributor in this work. The idea about this project, the integration design and the implementation were developed by the thesis author. She is also the main contributor to the Pig to PACT translation process described in the paper. The author is also responsible for selecting the experimental queries and setup and for conducting the evaluation experiments.
4.4 Approximate Results in MapReduce

In [23], we propose a block sampling technique for data-intensive computation platforms, which can be used for fast and accurate partial job execution. The paper appears as Chapter 8 in this thesis.

Paper Contribution

The paper presents a statistically profound early estimation technique as a layer over MapReduce Online. We propose a simple, yet efficient random sampling technique implementation, which significantly improves the accuracy of Online Aggregation. In order to overcome the challenge of the unstructured nature of data, our implementation performs the sampling before data is sent to the mapper tasks, as long as it is organized in blocks. Moreover, in order to avoid slow random disk accesses, we propose in-memory shuffling of data blocks, thus efficiently achieving random sampling and reducing data bias. We evaluate our implementation in terms of performance and accuracy, using real-world datasets of various sizes and distributions. We assess the accuracy of early estimations and study its dependency over our introduced block-level sampling technique parameters. We show that our system delivers highly accurate results, while matching MapReduce Online in performance.

The paper introduces a novel, efficient, in-memory block sampling technique for MapReduce applications. It also describes a method for integrating the block sampling technique with a distributed data-intensive computation platform. We provide an implementation of the block sampling technique for the Hadoop Online Prototype (HOP), but the technique is generic and could be integrated in other systems easily. We give an experimental evaluation of the proposed technique, focused on performance and accuracy, using real-world datasets and applications.

Thesis Author Contribution

The author is a major contributor in this work, including writing most of the paper. She played a major role in designing the sampling technique and choosing the system of implementation. She led the implementation effort by supervising and closely advising the developer of the prototype.

4.5 Results Materialization and Reuse

Our work on results materialization and reuse for high-level dataflow systems for big data, is published as a conference paper [24]. The complete paper appears as Chapter 9 of this thesis.

Paper Contribution

In this paper, we propose an optimization that helps avoid redundant computations in a workload of queries. The goal is to identify similar or identical queries in different jobs.
and try to reuse previously computed results. Recent research studies have shown a great amount of redundancies in typical analysis workloads. It is common to have parts of queries or even whole jobs submitted for execution to re-appear unchanged in future job submissions.

In this paper, we present m2r2 (materialize-match-rewrite-reuse), a language-independent and extensible framework for storing, managing and using previous job and sub-job results. We aim to achieve generality and support for different languages and backend execution engines. Thus, we have chosen to base our design at the logical plan level. In the paper, we describe general techniques for identifying candidate sub-plans for materialization. We discuss mechanisms for efficiently storing and retrieving previously stored plans and sub-plans. We demonstrate how to exploit reuse opportunities and we also discuss garbage collection and management of the results repository.

More specifically, we discuss and identify architecture and design similarities in high-level dataflow processing systems for big data. This generalization and high-level view gives us the opportunity to find the proper level for providing a reuse mechanism. We propose a generic materialization and reuse framework, m2r2, which is independent of the high-level language and the execution engine. We describe a prototype implementation of our materialization and reuse framework for the Pig system. Finally, we provide an evaluation of our prototype implementation of m2r2, using the TPC-H benchmark for Pig.

**Thesis Author Contribution**

The author is a major contributor in this work. The study and summarization of the system design and common characteristics of popular high-level dataflow systems for big data analytics is her individual work. Also, she is the main contributor in the m2r2 design goals, architecture and choice of systems. She supervised and revised the implementation of the system and closely worked with her co-authors to perform the experimental setup and evaluation of the prototype.

**4.6 Asymmetry in Large-Scale Graph Analysis**

Our work on large-scale graph analysis optimization was initiated while the author was at a research internship at the Database Systems and Information Management Group, Technical University of Berlin. The work has been accepted as a workshop paper and appears as Chapter 10 in this thesis.

**Paper Contribution**

We propose a general optimization framework for fixed point iterative algorithms, using a common mathematical model. We study the characteristics of fixed point algorithms in a formal way and we describe what these characteristics mean and how they can be safely exploited, in order to derive optimized algorithms. More importantly, we give the necessary conditions under which, it is safe to apply each of the proposed optimizations, by
expanding problem-specific properties. Our contributions could enable a cost-based optimizer to relieve the programmer of part of the burden for exploiting asymmetry. We use general-purpose dataflow operators and traditional fixed point theory to create template optimized execution plans, which can detect converged parts and avoid redundant computations, while providing functionality equivalent to that of Pregel and GraphLab. We evaluate our proposed optimizations using two characteristic iterative algorithms, Connected Components and PageRank. We present extensive experiments using real-world datasets of varying sizes. We show that optimized algorithms can yield order of magnitude gains compared to the naive execution.

We provide a categorization of optimizations for fixed point iterative algorithms, using a common mathematic model. We also develop a formal proof of equivalence between the categorized algorithms and a formal description of the necessary conditions under which the relevant optimizations can be safely applied. We present a mapping of the optimized algorithms to existing graph processing abstractions. We develop and describe an implementation of template optimized execution plans, using general data-flow operators. Additionally, we give an experimental evaluation of the proposed optimizations, using a common runtime.

**Thesis Author Contribution**

The thesis author is a major contributor in this work. She is the main contributor to the mathematical model, formalization and proofs provided in the article. She also collaborated closely with the co-authors to implement the proposed template execution plans. She implemented the applications used for evaluation and she was in charge of the experiments and collection of measurements.
Chapter 5

Conclusions and Future Work

In this thesis, we present techniques and tools for performance optimizations for data-intensive computation frameworks. We give an introduction to big data analysis systems and we provide the specific background knowledge on the systems we use in our research. We then summarize our contribution to this area, by presenting five articles: a survey on MapReduce limitations and optimizations; an integration of a high-level analysis system with a more flexible and efficient backend; a sampling technique for retrieving accurate results from partial job execution; a materialization and reuse framework for high-level big data systems; a formal optimization framework for large-scale fixed point iterative analysis.

Big data is a very active research area with a plethora of open issues and challenges to be addressed. Developing ways to process the vast amounts of data available drives business innovation, health discoveries, science progress and allows us to find novel ways to solve problems, which we considered very hard or even impossible in the past. Existing programming models for big data analytics, such as MapReduce, have been a great contribution and are widely used.

5.1 Summary of Results

High-level languages for big data analysis built on top of Hadoop, like Pig, are very useful tools, especially for analysts who are not experts in programming and have no background in distributed systems. These languages resemble SQL and usually use an intuitive, declarative way for defining data analysis tasks. However, these systems are often quite slow and suffer from the inflexibility of the backend engine. With our work, we show that such systems can significantly benefit from alternative execution engines and gain performance. Moreover, we show that this can be achieved in a transparent way, so that the end-users can execute their applications on the enhanced system, without having to modify them.

In many cases, accurate answers can be returned after only processing a small subset of the available data and great value can be extracted by only partial results. Several analysis applications can tolerate approximate answers to queries and highly benefit from lower latency. Such a functionality can also be used for rapid prototyping or pattern discovering in huge datasets. Query result approximation is not a novel idea. There has been extensive
research on the topic from the database community in the past. The data explosion that we are experiencing today, leaves us no choice but to reconsider approximation techniques, in the context of data-intensive MapReduce-style systems, in order to reduce query response times. However, adoption of existing techniques is not straight-forward and proves to be challenging. In the MapReduce world, data is not organized in tables or properly structured, but is often schema-less and stored in raw files. Moreover, analysis applications are usually much more complex than simple aggregations and can use arbitrary user-defined functions. With our work, we show that early estimations of the most frequent values in such datasets can be very accurate, thus the complete process of the input data is not always necessary.

Computation redundancy is another problem, very relevant to big data systems. Several studies have shown large similarities in data analysis queries and suggest that any type of caching techniques would greatly benefit data analysis frameworks. Following the idea of materialized views in relational databases, we have examined the possibility of porting this technique in big data environments. We observe that the majority of systems follow very similar designs in their upper layers, namely the language and logical plan layer, while their physical layers differ significantly. Based on this observation, we present a design for integrating a materialization and reuse framework after the logical layer of high-level dataflow processing systems. Our results show that when there exists sharing opportunity, query execution time can be immensely reduced by reusing previous results. We also show that the induced overhead of materialization is quite small, while non-I/O overheads are negligible. We note that both benefit and overhead are very sensitive to framework parameters, such as sub-job selection strategy and garbage collection policy, as well as specific workload characteristics.

In the last paper presented in this thesis, we focus on the optimization of iterative fixed point applications. Iterations are inevitably in the heart of many data-intensive applications, such as graph-parallel algorithms and machine learning applications. We present a taxonomy of optimizations for iterative fixpoint algorithms and describe ways to exploit the asymmetrical convergence behavior to implement optimized iterative execution plans. We offer proof of equivalence between different approaches and we provide a mapping to existing iterative and graph processing programming models. Our results demonstrate order of magnitude gains in execution time, when the optimized plans are used.

5.2 Evaluation of Results

For the evaluation of PonIC, we used the Piggish data generator \(^1\) to create evaluation datasets. We developed five scripts for evaluation, four of which were used to evaluate single operators and one was used to evaluate a combination of operators. The queries were developed in Pig Latin (executed both on Pig and PonIC), native MapReduce and Stratosphere. We measured the performance overhead for the Pig system over the corresponding native Hadoop MapReduce implementations and for PonIC over PACT. The results for Pig confirm already published results, showing that Pig is around 1.2 to 2 times slower than a native MapReduce application. PonIC’s overhead was found to be significantly lower and

\(^1\)http://cwiki.apache.org/PIG/pigmix.html
5.2. EVALUATION OF RESULTS

smaller than 1.6 in all cases. To have a better idea on how the overhead changes depending on the dataset size, we also ran one of the queries for different sizes of the input dataset. The results suggest a larger overhead for smaller datasets. Next, we measured the execution time ratio of Pig and native Hadoop MapReduce over PonIC. The results show that PonIC matches or is significantly faster than Pig. When compared to native MapReduce, PonIC is faster for three out of the five queries.

For the evaluation of HOP-S, we retrieved and pre-processed a number of varying size datasets and performed both performance and accuracy evaluation experiments. We first evaluated the overhead of the snapshot materialization frequency, which represents how often estimations are materialized into the distributed file system. Both bias reduction and sampling were disabled in this experiment. The results show that there is a moderate execution overhead (up to 35%) in the case of frequent snapshots. For the second experiment, we enabled bias reduction and compared the performance of HOP-S to Hadoop and MapReduce Online. The measured performance difference between the standard Hadoop framework and the other two frameworks was found to be insignificant, after taking into account the overhead of frequent early result snapshots. In the last experiment, we measured the overhead of the sampling process to the overall job execution time. We performed several tests varying the block-level sampling rate. We observed a noticeable overhead in comparison to job execution time over the MapReduce Online framework, which grows with the number of block-level samples. Regarding our technique’s precision of early results, we defined accuracy as the absolute error between the early returned result of a partially executed job and the result returned after processing the entire dataset. We ran the MapReduce application that measures the average yearly temperature on Hadoop, HOP and HOP-S. We observed that the default setting of 4 block-level samples per map task is too conservative. Results obtained in case of 10 to 16 blocks per map are very accurate even after processing just 10% of the input. At later stages of the job execution, the estimations become relatively accurate independently of the number of processed blocks. When running the same experiments on a sorted input, we found only slight variations in the maximum values of the error range and after 20% of processed input, estimations have very low errors. We also observed that MapReduce Online did not provide any estimation, in some cases, while it needed to process as much as 40%. Overall, according to our results, Hadoop Online did not provide statistically meaningful results. On the other hand, HOP-S, after processing only 10% of the input, gives a maximum error of 30%, with 25th/75th percentiles being less than 20%. Furthermore, we observed a steadily increasing accuracy with the amount of processed input.

For the evaluation of m2r2, we generated data sets using the DBGEN tools of the TPC-H Benchmark. We also created a synthetic workload that contains reuse opportunity, by changing substitution parameters. First, we investigated the overhead of the materialization. Then, we ran experiments to quantify the benefits gained by reusing materialized results for query execution. Finally, we examined the performance of our implementation and the throughput of the results repository. Our results show that when materialization is enabled, the overhead is negligible, except for one query, whose size of the intermediate results was very large. When comparing query execution times of the system when using the reuse framework to exploit results of sub-jobs, with query execution times of the sys-
tem without materialization, we observed a speedup of 50% - 80% for most queries. The non-I/O overheads measured were found to be negligible when compared to I/O overheads.

For evaluating the performance optimizations on iterative fixed point applications, we first validated our motivation by examining two popular iterative algorithms and showing the non-uniform convergence behavior that they expose. Next, we executed these algorithms using the proposed execution plans and measured the execution time per iteration, for different real-world datasets. We implemented the Connected Components and the PageRank applications in Stratosphere. Our results show that in all cases, the convergence of the algorithms is non-uniform and clearly some parts of the graphs converge faster than others. We also observed that the intensity of the phenomenon differs among datasets and depends on the dependency graph properties and the algorithm. Regarding execution time, we observed that two of the proposed execution plans perform consistently better than the baseline. On the other hand, the Dependency plan, was found to be consistently less efficient than the base plan, during the initial iterations. However, for most of the examined datasets, the Dependency plan outperforms the baseline, when the candidate elements for recomputation drop under a threshold. We concluded that the performance of this plan varies with the dataset and algorithm.

5.3 Known Limitations

One of the identified limitations of our research is that our survey on MapReduce systems contains no experimental comparison of the examined systems. This is partially due to the fact that, unfortunately, the majority of the systems are not open-source or even available to use. Another challenge was that each of the systems included in the survey targets a different limitation of the MapReduce/Hadoop implementation and, often, a different application domain. Therefore, it would be unfeasible to make a global, yet fair comparison of the systems.

Regarding PonIC, we are aware of the limitation of only considering a subset of Pig operators. However, it is important to stress that the completeness of our proposal is guaranteed. The PACT programming model is a generalization of the MapReduce programming model, containing operators equivalent to the ones of MapReduce. Since every Pig Latin program and Logical Plan can be translated into a MapReduce Plan, it can therefore also be translated into a PACT Plan.

An obvious restriction of our proposed sampling technique is that HOP-S does not offer any statistical error bounds or guarantees. However, such estimators are not possible to obtain, without making assumptions about the dataset distributions and schema. In our work, we rather chose to implement a general solution that operates in the level of blocks and is not aware of the input schema or distribution. Also, our technique does not take into account the locality of the chosen samples. We believe that working towards this direction in the future, would reduce the sampling overheads significantly.

An obvious limitation of m2r2 is the use of a synthetic workload for evaluation. The reason for this is that, to the best of our knowledge, no representative workload exists for Pig, containing all the necessary information for evaluation our technique, such as the...
5.4. FUTURE WORK

There exist a lot of opportunities to extend the research described here and improve the proposed optimizations. We are particularly interested in designing a cost model for fixed point iterative algorithms and implement it in the Stratosphere system. In order to realize this task, we need to complete two steps. First, we are planning to modify the execution engine of Stratosphere to support plan switching between iterations. Then, we are going to implement an optimizer for fixed point iterations as a system component. The optimizer will be collecting statistics during the execution of iterations. It will then use the measurements to decide which iterative plan to be chosen, in subsequent iterations.

Furthermore, we would like to evaluate several other large-scale graph-processing systems, such as Giraph and GraphLab. We are planning to implement several applications and make a wide comparison among these systems and our optimized Stratosphere plans. Next, we would like to explore other characteristics and alternative programming models for large-scale graph processing, such as asynchronous execution.

Moreover, we are willing to work towards extending m2r2. First, we are planning to completely decouple it from the Pig framework and build a general, component-based and extensible materialization and reuse framework. We then intend to integrate it with other popular high-level systems, such as Hive. Then, we would like to examine how benefits are related to different workload characteristics, data distributions and cluster sizes. We are also particularly interested in exploring possibilities of sharing and reusing results among different frameworks. In order to evaluate the materialization framework under more realistic conditions, we believe that it is essential to obtain execution traces from industrial partners and big organizations. We also intend to minimize the imposed overhead, by overlapping the materialization process with regular query execution, thus moving it out of the critical path of the execution. Finally, we wish to examine the possibility of extending m2r2 in order to support incremental computations and exploit concurrent-sharing opportunities.
Bibliography


[44] Paul Zikopoulos, Chris Eaton, Dirk deRoos, Tom Deutsch, and George Lapis. Understanding big data. analytics for enterprise class hadoop and streaming data.
Part II

Research Papers
Chapter 6

MapReduce: Limitations, Optimizations and Open Issues

Vasiliki Kalavri and Vladimir Vlassov

In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications* (TrustCom), 2013
CHAPTER 6. MAPREDUCE: LIMITATIONS, OPTIMIZATIONS AND OPEN ISSUES

Abstract

MapReduce has recently gained great popularity as a programming model for processing and analyzing massive data sets and is extensively used by academia and industry. Several implementations of the MapReduce model have emerged, the Apache Hadoop framework being the most widely adopted. Hadoop offers various utilities, such as a distributed file system, job scheduling and resource management capabilities and a Java API for writing applications. Hadoop’s success has intrigued research interest and has led to various modifications and extensions to the framework. Implemented optimizations include performance improvements, programming model extensions, tuning automation and usability enhancements. In this paper, we discuss the current state of the Hadoop framework and its identified limitations. We present, compare and classify Hadoop/MapReduce variations, identify trends, open issues and possible future directions.

6.1 Introduction

Recent advances in technology have allowed organizations to collect extremely large amounts of data, anticipating high value in analyzing them. “Big Data” management and processing has been one of the biggest challenges of our era. Current approaches consist of processing systems deployed on large amounts of commodity machines and exploit massive parallelism to efficiently analyze enormous datasets. The most successful system is the Google’s MapReduce framework [15], which hides the complexity of data distribution, communication and task scheduling and offers a simple programming model for writing analytical applications, while also providing strong fault-tolerance guarantees.

Several implementations of the MapReduce programming model have been proposed, with open-source Apache Hadoop framework [2] being the most widely adopted. Apart from the MapReduce programming model, Hadoop offers various other capabilities, including a distributed file system, HDFS [33] and a scheduling and resource management layer. Despite its popularity, the MapReduce model and its Hadoop implementation have also been criticized [4] and have been compared to modern parallel database management systems (DBMSs), in terms of performance and complexity [31]. There have been extensive studies on MapReduce characteristics, identifying a set of shortcomings of the model and current implementations [29, 25, 21]. Features such as the static map-shuffle-reduce pipeline, the frequent data materialization (writing data to disk), the lack of support for iterations and state transfer between jobs, the lack of indexes and schema and sensitivity to configuration parameters have been confirmed to contribute negatively in its performance, for certain classes of applications.

Numerous variations of Hadoop MapReduce have been developed during the last few years, proposing performance improvements, programming model extensions, automation of use and tuning. Each one of the extensions deals with one or more shortcomings of the vanilla Hadoop MapReduce implementation. The amount of these variations has grown significantly, making it hard for users to choose the appropriate tool. Existing surveys summarize some of these systems, however, there exists no complete study categorizing them and clarifying the trade-offs for a potential user. Having all these alternatives available,
users either need to spend a lot of time researching which system would best fit their needs or resort to common Hadoop installations, even if such a choice would be suboptimal for their problem.

In this survey, we examine existing MapReduce implementations based on Hadoop. The scope of our study is strictly limited to systems extending or enhancing Hadoop and does not include more generalized data-flow systems, such as Dryad [23], Spark [35] and Stratosphere [9]. The contributions of this paper are:

- An overview of the state-of-the-art in Hadoop/MapReduce optimizations;
- A comparison and classification of existing systems;
- A summary of the current state of the research field, identifying trends and open issues;
- A vision on possible future directions.

The rest of this paper is organized as follows. In Section 9.2, we provide background on Hadoop/MapReduce and present the current state of the project. In Section 6.3, we discuss the limitations of the model and implementation, as demonstrated in recent literature. Section 7.5.2 makes a categorization and comparison of existing MapReduce variations. In Section 10.5 we focus on trends and open issues and propose future directions. We conclude in Section 9.8.

### 6.2 Background

This section gives an introduction to the MapReduce model and its open-source implementation, Hadoop.

#### 6.2.1 The MapReduce Programming Model

The MapReduce programming model is designed to efficiently execute programs on large clusters, by exploiting data parallelism. A distributed file system is deployed on the same machines where the applications run, so that execution can benefit from data locality, by trying to move computation where the data reside. The model is inspired by functional programming and consists of two second-order functions, Map and Reduce, which form a static pipeline, where the Map stage is followed by the Reduce stage.

Data are read from the distributed file system, in the form of user-defined key-value pairs. These pairs are then grouped into subsets and serve as input for parallel instances of the Map function. A user-defined function must be specified and is applied to all subsets independently. The Map function outputs a new set of key-value pairs, which is then sorted by key and partitioned according to a partitioning function. The sorted data feed the next stage of the pipeline, the Reduce function. The partitioning stage of the framework guarantees that all pairs sharing the same key will be processed in the same Reduce task. In a similar way, a user-defined function is applied to the pairs, producing one output file per Reduce task, in the distributed file system.
CHAPTER 6. MAPREDUCE: LIMITATIONS, OPTIMIZATIONS AND OPEN ISSUES

One of the important advantages of the above schema is that the parallelization complexity is handled by the framework. The user only has to write the first-order functions that will be wrapped by the Map and Reduce functions. However, this advantage often comes with loss of flexibility. Each job must consist of exactly one Map function followed by an optional Reduce function, and steps cannot be executed in a different order. Moreover, if an algorithm requires multiple Map and Reduce steps, these can only be implemented as separate jobs, and data can only be transferred from one job to the next, through the file system.

In the initial implementations of Hadoop, MapReduce is designed as a master-slave architecture. The JobTracker is the master managing the cluster resources, scheduling jobs, monitoring progress and dealing with fault-tolerance. On each of the slave nodes, there exists a TaskTracker process, responsible for launching parallel tasks and reporting their status to the JobTracker. The slave nodes are statically divided into computing slots, available to execute either Map or Reduce tasks. The Hadoop community realized the limitations of this static model and recently redesigned the architecture to improve cluster utilization and scalability. The new design, YARN is presented in section 6.2.3.

6.2.2 HDFS

HDFS [33] is the distributed file system used by the Hadoop project. Hadoop MapReduce jobs read their input data from HDFS and also write their output to it. HDFS has been very popular because of its scalability, reliability and capability of storing very large files.

There are two types of nodes in HDFS: the DataNodes and the NameNode. Typically, a Hadoop deployment has a single NameNode, which is the master and a set of DataNodes, which serve as slaves. The main responsibility of a DataNode is to store blocks of data and to serve them on request over the network. By default, data blocks are replicated in HDFS, for fault-tolerance and higher chance of data locality, when running MapReduce applications. The NameNode is unique in an HDFS cluster and is responsible for storing and managing metadata. It stores metadata in memory, thus limiting the number of files that can be stored by the system, according to the node’s available memory.

6.2.3 YARN

YARN, Yet Another Resource Negotiator, is included in the latest Hadoop release and its goal is to allow the system to serve as a general data-processing framework. It supports programming models other than MapReduce, while also improving scalability and resource utilization. YARN makes no changes to the programming model or to HDFS. It consists of a re-designed runtime system, aiming to eliminate the bottlenecks of the master-slave architecture. The responsibilities of the JobTracker are split into two different processes, the ResourceManager and the ApplicationMaster. The ResourceManager handles resources dynamically, using the notion of containers, instead of static Map/Reduce slots. Containers are configured based on information about available memory, CPU and disk capacity. It also has a pluggable scheduler, which can use different strategies to assign tasks to available nodes. The ApplicationMaster is a framework-specific process, meaning that it allows
other programming models to be executed on top of YARN, such as MPI or Spark [6]. It negotiates resources with the ResourceManager and supervises the scheduled tasks.

6.3 Hadoop/MapReduce Limitations

Even though YARN manages to overcome the well-known limitations of the Hadoop scheduling framework and improves the scalability and resource utilization, there still exist several opportunities for optimizations in Hadoop/MapReduce. Having studied the recent literature, we group the optimization opportunities in three main categories: performance issues, programming model extensions and usability enhancements. In this section, we discuss the limitations which lead to these optimization opportunities.

6.3.1 Performance Issues

Even though Hadoop/MapReduce has been praised for its scalability, fault-tolerance and capability of processing vast amounts of data, query execution time can often be several hours [26]. This is orders of magnitude higher than what modern DBMSs offer and prevents interactive analysis. Performance highly depends on the nature of the application, but is also influenced by inherent system characteristics and design choices. A quite large percentage of the execution time is spent in task initialization, scheduling, coordination and monitoring. Moreover, Hadoop/MapReduce does not support data pipelining or overlap of the Map and the Reduce phases. Data materialization for fault-tolerance and intensive disk I/O during the shuffling phase have also been found to significantly contribute to the overall execution time. It has been suggested that Hadoop performance would benefit from well-known optimization techniques, already used by database systems and query optimizers. Even though Hadoop lacks a built-in optimizer, many of the suggested techniques have been implemented in Hadoop extensions, as discussed in the next section. Optimizations include index creation [16], data co-location [19], reuse of previously computed results [18], exploiting sharing opportunities [30], mechanisms dealing with computational skew [27] and techniques allowing early approximate query results.

6.3.2 Programming Model Issues

Developing efficient MapReduce applications requires advanced programming skills and deep understanding of the system architecture. Common data analysis tasks usually include processing of multiple datasets and relational operations, such as joins, which are not trivial to implement in MapReduce. Therefore, the MapReduce programming model has been often characterized as too "low-level" for analysts used to SQL-like or declarative languages. Another limitation of the programming model comes from its "batch" nature. Data need to be uploaded to the file system and even when the same dataset needs to be analyzed multiple times, it has to be read every time. Also, the computation steps are fixed and applications need to respect the map-shuffle-sort-reduce sequence. Complex analysis queries are realized by chaining multiple MapReduce jobs, having the results of one serving as the input for the next. These characteristics make the model inappropriate for certain
classes of algorithms. Various applications, including machine learning algorithms and graph processing, often require iterations or incremental computations. Since MapReduce operators are stateless, MapReduce implementations of iterative algorithms require manual management of state and chaining of iterations. Abstractions and high-level languages, have been built to facilitate MapReduce application development [20, 34]. Also, a set of domain-specific systems have emerged, extending the MapReduce programming model. We present these systems in section 6.4.2.

6.3.3 Configuration and Automation Issues

The third category of optimizations are related to automatic tuning and ease of use. There are numerous configuration parameters to set when deploying a Hadoop MapReduce cluster. Performance is often quite sensitive to them and users usually rely on empirical "rules of thumb". Options include the number of parallel tasks, the size of the file blocks and the replication factor. Proper tuning of these parameters requires knowledge of both available hardware and workload characteristics, while misconfiguration might lead to inefficient execution and underutilization of resources [26, 22]. Hadoop variations dealing with automatic tuning are discussed in Section 6.4.3.

6.4 Hadoop/MapReduce Variations

Some of the optimizations discussed in this Section can have multiple effects, therefore, some of the presented systems could fall into more than one category. At this point, we need to stress that our categorization and comparison is based on the primary motivation of each system examined.

6.4.1 Performance Optimizations

Operator Pipelining and Online Aggregation One of the first successful Hadoop extensions is MapReduce Online [14]. It improves performance by supporting online aggregation and stream processing, while also improving resource utilization. The motivation of MapReduce Online is to enable pipelining between operators, while preserving fault-tolerance guarantees. Pipelining is implemented both between tasks and between jobs. In the initial design, each reducer opens one TCP connection to each mapper. When a mapper computes a record, it determines to which partition it belongs and sends it via the appropriate socket. Opening a large number of TCP connections proved to be problematic, so the design was refined to use a "mixed" push/pull approach. Each reducer is allowed to open a bounded number of TCP connections, while pulling data from the rest of the mappers in the traditional Hadoop way. One problem that arises due to pipelining, is the nullification of the effect of combiners. To solve this problem, MapReduce Online buffers intermediate data up to a specified threshold, applies the combiner function on them and spills them to disk. As a side-effect of this design, early results of the jobs can be computed making approximate answers to queries available to users. This technique is called online aggregation and returns useful early results much faster than final results. Simply by applying the
reduce function to the data that the reducer has seen so far, the system can provide an early snapshot. In combination to the job progress metrics, a user can approximate the accuracy of the provided snapshot.

**Approximate Results** A more sophisticated approach to approximate results in MapReduce is proposed by Laptev et al. [28]. The EARL library is a Hadoop extension which allows incremental computations of early results using sampling and the bootstrapping technique. An initial sample of the data is obtained and the error is estimated using bootstrapping. If the error is too high, the sample is expanded and the error recomputed. This process is repeated until the error is under a user-defined threshold. In order to implement EARL, Hadoop was extended to support dynamic input size expansion. First, pipelining between mappers and reducers was implemented, similarly to MapReduce Online, so that reducers can start processing data as soon as they become idle. Then, mappers are kept active and reused instead of being restarted in every iteration. This modification saves a significant amount of setup time. Finally, a communication channel was built between mappers and reducers, so that the termination condition can be easily tested. EARL is an addition to the MapReduce API and existing applications require modifications in order to exploit it.

**Indexing and Sorting** Quite a few of the proposed optimizations for Hadoop/MapReduce come from well-known techniques of the database community. Long query runtimes are often caused due to lack of proper schemas and data indexing. Hadoop++ [16] and HAIL [17] are two remarkable attempts dealing with this matter. Hadoop++ is a transparent addition to Hadoop implemented using User Defined Functions (UDFs). It provides an indexing technique, the *Trojan Index*, which extends input splits with indexes at load time. Additionally to the Trojan Index, the paper also proposes a novel Join technique, the *Trojan Join*, which uses data co-partitioning in order to perform the join operation using only map tasks. HAIL proposes inexpensive index creation on Hadoop data attributes, in order to reduce execution times in exploratory use-cases of MapReduce. It modifies the upload pipeline of HDFS and creates a different clustered index per block replica. HAIL uses the efficient binary PAX representation [8] to store blocks and keeps each physical block replica in a different sort order. Sorting and indexing happen in-memory at upload time. If index information is available, HAIL also uses a modified version of the task scheduling algorithm of Hadoop, in order to schedule tasks to nodes with appropriate indexes and sort orders. The block binary representation and in-memory creation of indexes improves upload times for HDFS, while query execution times also greatly improve when index information is available. HAIL preserves Hadoop’s fault-tolerance properties. However, failover times are sometimes higher, due to HAIL assigning more blocks per map task, therefore limiting parallelization during recovery. In a system with the default degree of replication, three different sort orders and indexes are available, greatly increasing the probability of finding a suitable index for the corresponding filtering attribute of the query. HAIL benefits queries with low selectivity, exploratory analysis of data and applications for which there exists adequate information for index creation.
CHAPTER 6. MAPREDUCE: LIMITATIONS, OPTIMIZATIONS AND OPEN ISSUES

Work Sharing  MRShare [30] is a Hadoop extension that aims to exploit sharing opportunities among different jobs. It transforms a batch of queries into a new batch, by forming an optimization problem and providing the optimal grouping of queries to maximize sharing opportunities. MRShare works on the following levels: sharing scans when the input to mapping pipelines is the same and sharing map outputs when the reducers will have to push each tuple to the correct reduce function. Hadoop was modified to support tagging of tuples and merge the tags into the keys of tuples, so that their origin jobs can be identified. Moreover, reducers were enabled to write to more than one output files.

Data Reuse  ReStore [18] is an extension to Pig [20], a high-level system built on top of Hadoop/MapReduce. It stores and reuses intermediate results of scripts, originating from complete jobs or sub-jobs. The input of ReStore is Pig’s physical plan, i.e. a workflow of MapReduce jobs. ReStore maintains a repository where it stores job outputs together with the physical execution plan, the filename of the output in HDFS and runtime statistics about the MapReduce job that produced the output. The system consists of a plan matcher and rewriter which searches in the repository for possible matches and rewrites the job workflow to exploit stored data. It also has a sub-job enumerator and a sub-job selector, which are responsible for choosing which sub-job outputs to store, after a job workflow has been executed. Sub-job results are chosen to be stored in the repository based on the input to output ratio and the complexity of their operators. Repository garbage collection is not implemented, however guidelines for building one are proposed.

Skew Mitigation  SkewTune [27] is a transparent Hadoop extension providing mechanisms to detect stragglers and mitigate skew by repartitioning their remaining unprocessed input data. In order to decide when a task should be treated as a straggler, while avoiding unnecessary overhead and false-positives, SkewTune is using Late Skew Detection. Depending on the size of the remaining data, SkewTune may decide to scan the data locally or in parallel. In Hadoop, skew mitigation is implemented by SkewTune as a separate MapReduce job for each parallel data scan and for each mitigation. When repartitioning a map task, a map-only job is executed and the job tracker broadcasts all information about the mitigated map to all the reducers in the system. When repartitioning a reduce task, due to the MapReduce static pipeline inflexibility, an identity map phase needs to be run before the actual additional reduce task.

Data Colocation  The last system we present in this category is CoHadoop [19] and it allows applications to control where data are stored. In order to exploit its capabilities, applications need to state which files are related and might need to be processed together. CoHadoop uses this information to collocate files and improve job runtimes. While HDFS uses a random placement policy for load-balancing reasons, CoHadoop allows applications to set a new file property, in order for all copies of related files to be stored together. This property, the locator, is an integer and there is a N:1 relationship between files and locators, so that files with the same locator are stored on the same set of datanodes. The mapping is managed by saving information in a locator table, in the Namenode’s memory. If the
selected set of datanodes runs out of space, CoHadoop simply stores the files in another set of datanodes. CoHadoop may lead to skew in data distribution and also loss of more data in the presence of failures. Collocation and special partitioning are performed by adding a preprocessing step to a MapReduce job, which itself is a MapReduce job.

6.4.2 Programming model extensions

High-Level Languages

Developing applications using high-level languages on top of Hadoop has proven to be much more efficient regarding development time than using native MapReduce. Maintenance costs and bugs are also greatly reduced, as much less code is required. Pig [20] is one such high-level system that consists of a declarative scripting language, Pig Latin, and an execution engine that allows the parallel execution of data-flows on top of Hadoop. Pig offers an abstraction that hides the complexity of the MapReduce programming model and allow users to write SQL-like scripts, providing all common data operations (filtering, join, ordering, etc.).

One of the most widely-used high-level systems for Hadoop is Hive [34]. Initially developed by Facebook, Hive is not just an abstraction, but a data warehousing solution. It provides a way to store, summarize and query large amounts of data. Hive’s high-level language, HiveQL, allows users to express queries in a declarative, SQL-like manner. Very similar to Pig, HiveQL scripts are compiled to MapReduce jobs and executed on the Hadoop execution engine.

Another popular query language is Jaql [10]. Jaql is less general than the systems we have introduced in this Section, as it is designed for querying semi-structured data in JSON format only. The system is extensible and supports parallelism using Hadoop. Although Jaql has been specifically designed for data in JSON format, it borrows a lot of characteristics from SQL, XQuery, LISP, and PigLatin.

Cascading [1] is a Java application framework that facilitates the development of data processing applications on Hadoop. It offers a Java API for defining and testing complex dataflows. It abstracts the concepts of map and reduce and introduces the concept of flows, where a flow consists of a data source, reusable pipes that perform operations on the data and data sinks. Cascading quickly gained popularity among the industry and Twitter even developed and open-sourced a Scala API for it, Scalding [7].

Domain-specific Systems

Support for Iterations Iterative algorithms are very common in data-intensive problems, especially in the domains of machine learning and graph processing. HaLoop [12], is a modified version of Hadoop, with built-in support for development and efficient execution of iterative applications. HaLoop offers a mechanism to cache and index invariant data between iterations, significantly reducing communication costs. It extends Hadoop’s API, allowing the user to define loops and termination conditions easily. The authors also propose a novel scheduling algorithm, which is loop-aware and exploits inter-iteration locality.
It exploits cached data in order to co-locate tasks which access the same data in different iterations.

**Support for Incremental Computations** A special class of iterative applications is that of incremental computations. These include jobs which need to be run repeatedly with slightly different, most often augmented input. Performing such computations in MapReduce would obviously lead to redundant computations and inefficiencies. In order to overcome this problem, one has to specially design their MapReduce application to store and use state across multiple runs. Since MapReduce was not designed to reuse intermediate results, writing such programs is complex and error-prone. Incoop’s [11] goal is to provide a transparent way to reuse results of prior computations, without demanding any extra effort from the programmer. Incoop extends Hadoop to support incremental computations, by making three important modifications: (a) Inc-HDFS. A modified HDFS which splits data depending on file contents instead of size. It provides mechanisms to identify similarities between datasets and opportunities for data reuse, while preserving compatibility with HDFS. (b) Contraction Phase. An additional computation phase added before the Reduce phase, used to control task granularity. This phase leverages the idea of Combiners to "break" the reduce task into a tree-hierarchy of smaller tasks. The process is run recursively until the last level, where the reduce function is applied. In order to result into a data partitioning suitable for reuse, content-based partitioning is again performed on every level of Combiners. (c) Memoization-aware Scheduler. An improved scheduler which takes into account data locality of previously computed results, while also using a work-stealing algorithm. The memoization-aware scheduler schedules tasks on the nodes that contain data which can be reused. However, this approach might create load imbalance if some data is very popular. To avoid this situation, the scheduler implements a simple work-stealing algorithm. When a node runs out of work, the scheduler will locate the node with the largest task queue and delegate a task to the idle node.

### 6.4.3 Automatic tuning

**Self-Tuning** Configuring and tuning Hadoop MapReduce is usually not a trivial task for developers and administrators, often resulting to poor performance, resource under-utilization and consequently increased operational costs. Starfish [22] is a self-tuning system, built as an extension to Hadoop, which dynamically configures system properties based on workload characteristics and user input. Starfish performs tuning on three levels. In the **job-level**, it uses a *Just-in-Time Optimizer* to choose efficient execution techniques, a *Profiler* to learn performance models and build job profiles and a *Sampler* to collect statistics about input, intermediate and output data and help the Profiler build approximate models. In the **workflow-level**, it uses a *Workflow-aware Scheduler*, which exploits data locality on the workflow-level, instead of making locally optimal decisions. A *What-if Engine* answers questions based on simulations of job executions. In the **workload-level**, Starfish consults the *Workload Optimizer* to find opportunities for data-flow sharing, materialize of intermediate results for reuse or reorganize jobs inside a batch and the *Elastisizer* to automate node and network configuration.
Disk I/O Minimization  Sailfish [32] is another Hadoop modification also providing auto-tuning opportunities, such as dynamically setting the number of reducers and handling skew of intermediate data. Additionally, it improves performance by reducing disk I/O due to intermediate data transfers. The proposed solution uses KFS [3] instead of HDFS, which is a distributed file system allowing concurrent modifications to multiple blocks of a single file. The authors propose I-files, an abstraction which aggregates intermediate data, so that they can be written to disk in batches. An index is built and stored with every file chunk and an offline daemon is responsible for sorting records within a chunk.

Data-aware Optimizations  Manimal [24] is an automatic optimization framework for MapReduce, transparent to the programmer. The idea is to apply well-known query optimization techniques to MapReduce jobs. Manimal detects optimization opportunities by performing static analysis of compiled code and only applies optimizations which are safe. The system’s analyzer examines the user code and sends the resulting optimization descriptors to the optimizer. The optimizer uses this information and pre-computed indexes to choose an optimized execution plan, the execution descriptor. The execution fabric then executes the new plan in the standard map-shuffle-reduce fashion. Optionally, an index generation program creates an additional MapReduce job to generate an indexed version of the input data. Example optimizations performed by Manimal include Selection and Projection. In the first case, when the map function is a filter, Manimal uses a B+Tree to only scan the relevant portion of the input. In the second case, it eliminates unnecessary fields from the input records.

Table 6.1 shows a brief comparison of the systems discussed in this survey. We have excluded high-level languages from this comparison, since they share common goals and major characteristics among them.

6.5 Discussion

MapReduce is a quite recent paradigm and its open-source implementation, Hadoop, still has plenty of optimization opportunities to exploit. However, implementing even traditional optimization techniques can be very challenging in architectures of shared-nothing clusters of commodity machines. Scalability, efficiency and fault-tolerance are major requirements for any MapReduce framework and trade-offs between optimizations and these features need to be carefully studied.

One can identify several trends when studying the systems discussed in this survey. In contrast to traditional applications, MapReduce programs are data-intensive instead of computation-intensive and, in order to achieve good performance, it is vital to minimize disk I/O and communication. Therefore, many systems seek ways to enable in-memory processing and avoid reading from disk when possible. For the same reason, traditional database techniques, such as materialization of intermediate results, caching and indexing are also favored.

Another recurring theme in MapReduce systems is relaxation of fault-tolerance guarantees. The initial MapReduce design from Google assumed deployments in clusters of
<table>
<thead>
<tr>
<th>Optimization Type</th>
<th>Major Contributions</th>
<th>Open-Source / Available to use</th>
<th>Transparent to Existing Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapReduce Online</td>
<td>Pipelining, Online aggregation</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>EARL</td>
<td>Fast approximate query results</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Hadoop++</td>
<td>Performance gains for relational operations</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>HAIL</td>
<td>Performance gains for relational operations</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>MRShare</td>
<td>Concurrent work sharing</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>ReStore</td>
<td>Reuse of previously computed results</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>SkewTune</td>
<td>Automatic skew mitigation</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>CoHadoop</td>
<td>Communication minimization by data co-locations</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>HaLoop</td>
<td>Iteration support</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Incoop</td>
<td>Incremental processing support</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Starfish</td>
<td>Dynamic self-tuning</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Sailfish</td>
<td>Disk I/O minimization and automatic tuning</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Manimal</td>
<td>Automatic data-aware optimizations</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 6.1. Comparative Table of Hadoop Variations
hundreds or even thousands of commodity machines. In such setups, failures are very common and strict fault-tolerance and recovery mechanisms are necessary. However, after the release of Hadoop, MapReduce has also been used by organizations much smaller than Google. Common deployments may consist of only tenths of machines [5], significantly decreasing failure rates. Such deployments can benefit from higher performance, by relaxing the fault-tolerance guarantees of the system. For example, one can avoid materialization of task results and allow pipelining of data. In this scenario, when a failure occurs, the whole job would have to be re-executed, instead of only the tasks running on the failed node.

Many important steps forward have been made since the launch of MapReduce and Hadoop, but several open issues still exist in the area. Even though it is clear that relaxing fault-tolerance offers performance gains, we believe that this issue needs to be further studied in the context of MapReduce. The trade-offs between fault-tolerance and performance need to be quantified. When these trade-offs have become clear, Hadoop could offer capabilities of tunable fault-tolerance to the users or provide automatic fault-tolerance adjustment mechanisms, depending on cluster and application characteristics.

Another open issue is clearly the lack of a standard benchmark or a set of typical workloads for comparing the different Hadoop implementations. Each system is evaluated using different datasets, deployments and set of applications. There have been some efforts in this direction [32, 13], but no complete solution has been introduced and no clear answer exists to what a "typical" MapReduce workload would be.

As far as programming extensions are concerned, we believe that the main problem with all the specialized systems proposed is transparency to the developer. In our view, such programming extensions need to be smoothly integrated into to the framework, so that existing applications can benefit from the optimizations, automatically, without having to change or re-compile the source code.

Finally, even if successful declarative-style abstractions exist, Hadoop MapReduce is still far from offering interactive analysis capabilities. Developing common analysis tasks and declarative queries has indeed been significantly facilitated. However, these high-level systems still compile their queries into MapReduce jobs, which are executed on top of Hadoop. According to our judgment, these systems could greatly benefit from more sophisticated query optimization techniques. Mechanisms such as data reuse and approximate answers should also be more extensively studied and exploited in high-level systems.

Unfortunately, the majority of the proposed systems are not open-source or even available to use. This prevents researchers from studying or extending them and stalls progress. Also, proposed systems usually only compare to vanilla Hadoop, not yielding very interesting results. Consequently, very few of the optimizations proposed have been incorporated to official Hadoop releases.

6.6 Conclusions

In conclusion, Big Data systems and specifically MapReduce, are an active research area, still at its infancy. Currently, the interest for MapReduce is at its peak and there exist a
lot of problems and challenges to be addressed. There lies a bright future ahead for Big Data, as businesses and organizations realize more and more the value of the information they can store and analyze. Developing ways to process the vast amounts of data available drives business innovation, health discoveries, science progress and allows us to find novel ways to solve problems, which we considered very hard or even impossible in the past.

Acknowledgment

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF) under the contract RIT10-0043.
Bibliography


BIBLIOGRAPHY


Chapter 7

PonIC: Using Stratosphere to Speed Up Pig Analytics

Vasiliki Kalavrim Vladimir Vlassov and Per Brand

In *Euro-Par 2013 Parallel Processing*, 279-290
CHAPTER 7. PONIC: USING STRATOSPHERE TO SPEED UP PIG ANALYTICS

Abstract

Pig, a high-level dataflow system built on top of Hadoop MapReduce, has greatly facilitated the implementation of data-intensive applications. Pig successfully manages to conceal Hadoop’s one input and two-stage inflexible pipeline limitations, by translating scripts into MapReduce jobs. However, these limitations are still present in the backend, often resulting in inefficient execution.

Stratosphere, a data-parallel computing framework consisting of PACT, an extension to the MapReduce programming model and the Nephele execution engine, overcomes several limitations of Hadoop MapReduce. In this paper, we argue that Pig can highly benefit from using Stratosphere as the backend system and gain performance, without any loss of expressiveness.

We have ported Pig on top of Stratosphere and we present a process for translating Pig Latin scripts into PACT programs. Our evaluation shows that Pig Latin scripts can execute on our prototype up to 8 times faster for a certain class of applications.

7.1 Introduction

Large-scale data management and analysis is currently one of the biggest challenges in the area of distributed systems. Industry, as well as academia, is in urgent need of data analytics systems, capable of scaling up to petabytes of data. Such systems need to efficiently analyze text, web data, log files and scientific data. Most of the recent approaches use massive parallelism and are deployed on large clusters of hundreds or even thousands of commodity hardware.

MapReduce [4], proposed by Google, is the most popular framework for large-data processing; its open-source implementation, Hadoop \(^1\), is nowadays widely used. However, it has several limitations, including the limitation on the number of input datasets (only one input set) and the limitation on a structure of a program that must follow a static fixed pipeline pattern of the form split-map-shuffle-sort-reduce. This pipeline is suitable for simple applications, such as log-file analysis, but severely complicates the implementation of relational queries or graph algorithms. These limitations have led researchers to develop more general-purpose systems, inspired by MapReduce [8, 14, 15, 1, 2]. One of them is Stratosphere [2], which consists of a programming model, PACT (Parallelization Contracts), and the Nephele execution engine. The system is essentially a generalization of MapReduce and aims to overcome the limitations mentioned above.

Both models, MapReduce and PACT, require significant programming ability and in-depth understanding of the systems’ architectures. Applications usually lead to complex branching dataflows which are low-level and inflexible. In order to save development time and make application code easier to maintain, several high-level languages have been proposed for these systems. Currently, high-level platforms on top of Hadoop include JAQL [3], Hive [13] and Pig [6]. Pig Latin, which is the language of the Pig platform [11], offers

\(^1\)http://hadoop.apache.org/
the simplicity and declarativeness of SQL, while maintaining the functionality of MapReduce. Pig compiles Pig Latin into MapReduce jobs which are executed in Hadoop. Pig hides Hadoop’s one-input and two-stage dataflow limitations from the programmer and provides built-in functions for common operations, such as filtering, join and projection. It also directly benefits from Hadoop’s scalability and fault-tolerance. However, even if not obvious to the users, the limitations and inflexibility of Hadoop are still present in the Pig system. The translation of relational operators for the static pipeline of Hadoop produces an inefficient execution plan since data have to be materialized and replicated after every MapReduce step.

The goal of Pig was to make MapReduce accessible to non-experts and relieve the programmer from the burden of repeatedly coding standard operations, like joins. Another goal was to make Pig independent of any particular backend execution engine. However, Pig was developed on top of Hadoop, ended up solving specific Hadoop problems and became highly coupled with its execution engine. The Stratosphere data-parallel computing framework offers a superset of MapReduce functionality, while overcoming some of the major weaknesses of the MapReduce programming model. It allows data pipelining between execution stages, enabling the construction of flexible execution strategies and removing the demand for materialization and replication in every stage. Moreover, the PACT programming model of Stratosphere supports multiple inputs.

In this paper, we present PonIC (Pig on Input Contracts), an integration of the of Pig System with Stratosphere. We have analyzed the internal structure of Pig and have designed a suitable integration strategy. In order to evaluate the benefits of the integration, we have developed a prototype implementation. The current prototype supports a subset of the most common Pig operations and it can be easily extended to support the complete set of Pig Latin statements. Thus, we show that it is possible to plug a different execution engine into the Pig system and we identify the parts of Pig that can be reused. With our Pig to PACT translation algorithm and our prototype, we show that Stratosphere has desirable properties that significantly simplify the plan generation. We have developed a set of basic scripts and their native MapReduce and PACT equivalents and we provide a comparison of PonIC with Pig, as well as the corresponding native programs. We observe that Stratosphere’s relational operators are much more efficient than their MapReduce equivalents. As a result, PonIC has a great advantage over Pig on Hadoop and often executes faster than native Hadoop MapReduce. The main contributions of this paper are as follows.

- Our integration is entirely transparent to Pig’s end-users and existing Pig Latin applications can be executed on PonIC without any modification. The syntax and the semantics are completely unchanged.

- We show that Pig can be harnessed to alternative execution engines and present a way of integration.

- We identify the features of Pig that negatively impact execution time.

- We show that Pig can be integrated with Stratosphere and gain performance.
We propose a complete translation process of Pig Logical Plans into Stratosphere Physical Plans and we present and evaluate PonIC.

The rest of this paper is structured as follows. In Section 9.2, we provide the necessary background on the Pig and Stratosphere systems. Section 7.3 discusses the restrictions that MapReduce poses on Pig’s current implementation and presents our Pig-to-Stratosphere translation process. In Section 9.5, we discuss our prototype implementation in detail. Section 9.6 contains the evaluation of PonIC against Pig on Hadoop, native Hadoop MapReduce and native PACT Stratosphere. In Section 10.4, we comment on related work, while we provide our conclusions, open issues and vision for the future in Section 9.8.

7.2 Background

In this section, we provide the essential background. We briefly discuss the MapReduce programming model, the Pig system and the Stratosphere system.

7.2.1 The MapReduce Programming Model

MapReduce is a data-parallel programming model. Its architecture is inspired by functional programming and consists of two second-order functions, Map and Reduce, which form a static pipeline. Data are read from an underlying distributed file system and are transformed into key-value pairs, which are grouped into subsets and processed by user-defined functions in parallel. Data distribution, parallelization and communication are handled by the framework, while the user only has to write the first-order functions wrapped by the Map and Reduce functions. However, this abstraction comes with loss of flexibility. Each job must consist of exactly one Map function followed by one Reduce function and no step can be omitted or executed in a different order. Moreover, if an algorithm requires multiple Map and Reduce steps, these can only be implemented as separate jobs, and data can only be passed from one job to the next through the file system. This limitation can frequently add a significant overhead to the execution time. MapReduce was initially proposed by Google and its open-source implementation, Hadoop and HDFS [12] are nowadays widely used.

7.2.2 Pig

Pig consists of a declarative scripting language, Pig Latin, and an execution engine that allows the parallel execution of data-flows on top of Hadoop. The Pig System takes a Pig Latin program as input and produces a series of MapReduce jobs to be executed on the Hadoop engine. Compilation happens in several steps. First, the parser transforms a Pig Latin script into a Logical Plan. Each Logical operator is compiled down to one or more Physical Operators. The Physical Plan is then passed to the compiler that transforms it into a DAG of MapReduce operators. MapReduce operators are topologically sorted and connected between them using a store-load combination, producing the MapReduce Plan as output. The generated jobs are finally submitted to Hadoop and monitored by Pig.
7.3 PLAN COMPILATION

7.2.3 Stratosphere

Stratosphere is a parallel data-processing framework, which consists of a programming model, PACT (Parallelization Contracts), and an execution engine, Nephele, capable of executing dataflow graphs in parallel. Nephele is an execution engine designed to execute DAG-based data flow programs. It manages task scheduling and setting up communication channels between nodes. Moreover, it supports dynamic allocation of resources during execution and fault-tolerance mechanisms. The PACT programming model is a generalization of the MapReduce programming model. It extends the idea of the Map and Reduce second-order functions, introducing the Input Contracts. An Input Contract is a secondary function that accepts a first-order user-defined function and one or more data sets as inputs. Input Contracts do not have to form any specific type of pipeline and can be used in any order that respects their input specifications. In the context of the PACT programming model, Map and Reduce are Input Contracts. The following three more Contracts are defined in PACT:

- The Cross Input Contract accepts multiple inputs of key value pairs and produces subsets of all possible combinations among them, building a Cartesian product over the input.
- The Match Contract operates on two inputs and matches each pair of the first input with one pair of the second input that has the same key value.
- The CoGroup Contract creates independent subsets by combining all pairs that share the same key.

7.3 Plan Compilation

As explained in the previous section, a Pig Latin script is parsed and transformed into a graph of logical operators, each corresponding to one command. This graph, the Logical Plan, is then translated into a Physical Plan, a graph of physical operators, which defines how the logical operations will be executed. Multiple strategies can be used to map logical operators to physical ones and it’s the system’s compiler job to choose a strategy, depending on the underlying execution engine’s capabilities, dataset characteristics, hints provided by the developer, etc. The translation process in Pig is briefly explained next.

7.3.1 Plan Compilation in Pig

Pig’s compiler translates logical to physical operators, with the additional restriction that each physical operator needs to be expressed in terms of MapReduce steps or parts thereof. The compiler keeps track of the current phase during translation and knows if it is a map or a reduce step. For each operator, it checks if it can be merged into the current phase. If communication is required, the current phase is finalized and a new phase is started in order to compile the operator. If the current phase is a map, a reduce phase will be initiated; otherwise, a new MapReduce job needs to be created and store-load combination
is required to chain the jobs. We explain the translation process using an example from a slightly modified query of the PigMix benchmark \(^2\) shown below:

**Example Query 1**

A = load 'page_views' as (user, timestamp, revenue);
B = foreach A generate user, revenue;
alpha = load 'users' as (name, phone, address, city);
beta = foreach alpha generate name;
C = join beta by name, B by user;
D = group C by $0;
E = foreach D generate group, SUM(C.revenue);
store E into 'out';

The simple Example Query 1 loads two datasets, performs a join on a common attribute to find the set of users who have visited some webpages, groups the resulting dataset and generates the estimated revenue for each user. Figure 7.1(a) shows the simplified Logical Plan for the above script, whereas Figure 7.1(b) shows the generated Physical Plan. Note that the join operator is replaced by four new operators and the group operator is translated into three physical operators similarly. The Physical Plan is then translated into a MapReduce Plan, as shown in Figure 7.1(c). First, a map phase is created and as the Physical Plan is traversed, operators are added to it. When the global rearrange operator is reached, shuffling is required, therefore the map phase is finalized and a reduce phase is initiated. When a new MapReduce job is created, a store-load pair is added in between to set the output of the first as the input of the second.

---

\(^2\)http://cwiki.apache.org/PIG/pigmix.html
7.3. PLAN COMPILATION

Our example shows that even for a small script, generated plans can be long and cumbersome. If the generated Logical Plan does not fit well the MapReduce static pipeline, performance might degrade. Adding store-load combinations and materialization of results in between jobs is also a source of inefficiency.

In contrast to MapReduce, using Stratosphere as the backend for Pig significantly simplifies the translation process. Input Contracts can be greatly exploited to generate shorter and more efficient plans, without any extra effort from the programmer. We present our translation algorithm next.

7.3.2 Pig to PACT Plan Translation

Pig Latin offers a large set of commands that are used for input and output, relational operations, advanced operations and the declaration of user-defined functions. We chose the most common and useful ones and we describe here how they are be translated into PACT operators. A more detailed description of the translation process we followed can be found in [9].

**Input/Output** Pig provides the LOAD and the STORE commands for data input and output. These two logical operators can be mapped directly to the `GenericDataSource` and the `GenericDataSink` Input Contracts of Stratosphere. In our implementation, we only support input and output from and to files, so we have based our implementation on the more appropriate Contracts, `FileDataSource` and `FileDataSink`. The generic Contracts can be easily extended to support other kinds of input and output sources.

**Relational Operators** PACTs support most of the common relational operations. The FILTER and FOREACH statements correspond to a Map Contract. The GROUP logical operator naturally maps to the `Reduce` Input Contract, while INNER and OUTER JOIN operations can be implemented using the `Match` and `CoGroup` Input Contracts. Pig’s ORDER BY operator can sort the input records in ascending or descending order, specifying one or more record fields as the sorting key. Pig realizes the ORDER BY operation by creating two MapReduce jobs. With PACTs, the same functionality can be offered in a much simpler way using the `GenericDataSink` Contract.

**Advanced Operators** From the set of the advanced Pig operators, we choose CROSS and UNION. The CROSS operator can be directly mapped to the `Cross` Input Contract, while the Map Input Contract can be used to realize UNION. The Map Contract (Stratosphere 0.2) offers a method, which provides the functionality we need to implement UNION.

Our translation consists of two stages. At the first stage the Logical Plan is translated into a plan of PACT operators. This PACT Plan is the equivalent of Pig’s Physical Plan. The second stage translates the PACT Plan into actual Input Contracts and submits the PACT Plan to the Nephele execution engine.

The Plan generation for the Example Query 1 is shown in Figure 7.2(a). There is an one-to-one mapping of logical operators to PACT operators and consequently Input Contracts, which makes the graph and the translation process much simpler. The resulting graph can be further optimized, by merging filter and foreach operators into the preceding Contracts, as shown in Figure 7.2(b).
7.3.3 Discussion

Even though we have considered only a subset of Pig operators, it is important to stress that the completeness of our proposal is guaranteed. The PACT programming model is a generalization of the MapReduce programming model. Since every Pig Latin program and Logical Plan can be translated into a MapReduce Plan, it can therefore also be translated into a PACT Plan.

Using Stratosphere and Input Contracts as the backend results into a more straightforward translation process. The one-to-one Pig-to-PACT mapping requires less communication, due to less shuffling. Data is pipelined between Input Contracts, eliminating the need for frequent materialization. Also, the execution plan benefits from optimizations of the Logical Plan by Pig’s Logical Plan optimizer and of the PACT Plan by Stratosphere’s optimizer.\(^3\)

7.4 Implementation

PonIC has been implemented as an extension to the Pig system and reuses Pig functionality where possible. Pig classes or wrappers are used in order to make them compatible with the new features. The source code is publicly available.\(^4\)

We have identified the parts of the Pig software stack that are not tightly coupled to the Hadoop execution engine, namely the parser and the Logical Plan layer. The underlying layers have been replaced with our compilation layer that transforms the Logical Plan into a Stratosphere execution plan.

Pig’s Logical Plan is traversed in a depth-first fashion. The traversal starts from the plan’s roots and a visit() method is responsible for recognizing the operator type and cre-

---
\(^3\)http://stratosphere.eu/wiki/doku.php/wiki:pactcompiler  
\(^4\)http://github.com/PonIC/PonIC
7.5. EVALUATION

Table 7.1. Pig to PACT operators mapping (for the chosen subset of Pig operators)

<table>
<thead>
<tr>
<th>Pig Operator</th>
<th>Input Contract</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>FileDataSource</td>
</tr>
<tr>
<td>STORE</td>
<td>FileDataSink</td>
</tr>
<tr>
<td>GROUP</td>
<td>Reduce</td>
</tr>
<tr>
<td>INNER JOIN</td>
<td>Match</td>
</tr>
<tr>
<td>OUTER JOIN / COGROUP</td>
<td>CoGroup</td>
</tr>
<tr>
<td>UNION</td>
<td>Map</td>
</tr>
<tr>
<td>FILTER / FOREACH</td>
<td>Map</td>
</tr>
<tr>
<td>ORDER</td>
<td>FileDataSink</td>
</tr>
</tbody>
</table>

Evaluating the appropriate PACT operator, according to the mappings of Table 7.1. It is also responsible for setting the correct parameters, such as data types, operator alias, result types, as well as connecting the newly created operator to its predecessors. This way, a graph of PACT operators is gradually constructed. When the PACT Plan has been created, it is submitted to Nephele for execution. Table 7.1 summarizes the Pig to PACT translation mappings for the subset of the Pig operators considered in this study.

The most significant extensions made to the Pig codebase are:

- An additional execution mode to allow starting Pig in Stratosphere execution mode with the command `pig -x strato`.
- An extension of Pig’s `HExecutionEngine` class as an engine for Stratosphere.
- A re-implementation of the relational and expression operators to support the new APIs.
- A `LogToPactTranslationVisitor` class, based on Pig’s `LogToPhyTranslationVisitor` class, as the first-level compiler.
- A package of PACT operators, based on Pig’s physical operators.
- A `PactCompiler` class, as the second-level compiler.
- Stratosphere-specific load and store functions.
- A `contractsLayer` and a `stubsLayer` packages, which contain wrapper classes of Stratosphere’s Input Contracts and Stub classes.

7.5 Evaluation

We conducted our experiments on an OpenStack cluster, using 10 ubuntu Virtual Machines (VMs), each having 4 VCPUs, 90GB of disk space and 8GB of RAM. We deployed Hadoop version 1.0.0, Pig version 0.10.0 and Stratosphere version 0.2. Hadoop’s NameNode and
JobTracker, as well as Stratosphere’s JobManager run on a dedicated VM, while the remaining 9 VMs serve as slave nodes. Default parameters were used for HDFS block size and replication factor.

We used the PigMix data generator to create a page_views dataset of 10 million rows (approximately 15GB) and the corresponding users table. We developed five scripts for evaluation, namely a Load/Store operation, a Filter script which filters out 50% of the input, a Group operation, a Join of the page_views and the users dataset and a Mixed query, corresponding to the Example Query 1, containing a combination of Load, Group, Join and Store operators. Each test was executed 5 times and the results presented here have a standard deviation of less than 1% in all cases. The test applications were developed in Pig Latin (executed both on Pig and PonIC), native MapReduce and PACT.

7.5.1 Implementation Overhead

Whenever using high-level languages, there is an overhead users have to pay in exchange for the abstraction offered. This overhead is one of the factors defining the value of the abstraction. Figure 7.3 shows the performance overhead for the Pig system over the corresponding native Hadoop MapReduce implementations and for PonIC over PACT. For Pig, this overhead includes setup, compiling, data conversion and plan optimization time. The results for Pig confirm already published results [6]; Pig is around 1.2 to 2 times slower than a native MapReduce application. Figure 9.5 also shows that PonIC’s overhead is significantly lower and smaller than 1.6 in all cases. We believe that the smaller overhead is mainly due to the more efficient translation process. Since PonIC only supports a subset of Pig’s features, the overhead could increase in a complete implementation. However, as we described in Section 7.3.3, in the worst case, an operator could be translated into PACT, using only the Map and Reduce Contracts. Such a naive translation would result into an overhead comparable to Pig’s overhead.

In order to have a better idea on how the overhead changes depending on the dataset size, we ran the Group query for three different sizes of the page_views dataset. The results in Figure 7.3(b) show that the overhead caused by setup and compilation time has a heavier influence on smaller datasets.
7.6. RELATED WORK

Figure 7.4. Evaluation Results: Execution Time Comparison

7.5.2 Comparison with Pig and Hadoop MapReduce

Figure 7.4(a) shows the execution time ratio of Pig and native Hadoop MapReduce over PonIC. Y axis is in logarithmic scale. PonIC matches Pig’ execution time for the Load-/Store and the Filter queries, while it is significantly faster in the rest of the cases. When compared to native MapReduce, PonIC is also faster, except from the Load/Store and Filter operations, for which setup and data conversion times are dominant. In the case of Mixed query, PonIC is 8 times faster than Pig. The MapReduce Plan that Pig creates for this query contains two MapReduce jobs in order to implement the join and the group operations, involving a materialization step in between them. On the other hand, PonIC can execute faster, exploiting Stratosphere’s data pipelining between Input Contracts. The main reason why PonIC is generally faster than Pig is demonstrated in Figure 7.4(b), which is a comparison between the execution time of native MapReduce and PACT implementations. It shows that, in all the cases except Load/Store, Stratosphere is faster than native MapReduce.

7.6 Related Work

Among the supported high-level languages for MapReduce, Hive is probably the most popular and has been used in work similar to ours. Hive has been integrated with the ASTERIX system [1]. ASTERIX provides a data-agnostic algebra layer, which allows Hive to run on top of the Hyracks runtime. Hive execution plans are translated to ASTERIX algebra plans and better performance is achieved without any changes in the HiveQL queries. To our knowledge, no published evaluation measurements exist to support this claim.

The Shark system [5] allows HiveQL queries to execute on top of Spark [15], in an analogous way to ours with Pig and Stratosphere. However, Shark’s goal is to provide a unified system where both SQL queries and iterative analytics applications can co-exist and execute efficiently. Our work and the Shark project share some discoveries regarding the limitations of the MapReduce-based execution engines, which result in inefficient execution, namely the expensive data materialization and inflexibility of static pipelines over general DAGs.

There has been recent work in integrating JAQL with the Stratosphere system [10], which led to the creation of Meteor [7]. Meteor is a high-level language inspired by JAQL and lies on top of a relational algebra layer, Sopremo. Meteor programs are translated into Sopremo operators, which are then compiled into Input Contracts, in a way similar to our work. However, Meteor, like JAQL, only supports the JSON data model and no
performance measurements are yet available, as far as we know. With our work, we benefit both Pig and Stratosphere users. Pig developers can gain improved performance without changing their applications, while Stratosphere users can now exploit the expressiveness of the Pig Latin language to develop applications faster and execute them on the Nephele execution engine, with only minimal compilation overhead.

### 7.7 Conclusions and Future Work

Existing programming models for Big Data analytics, such as MapReduce and PACT, have been a great contribution and are widely used. However, in order to fully exploit the possibilities provided by the increasing amounts of data in business and scientific applications, data analysis should become accessible to non-experts, who are used to work with higher-level languages. Therefore, improving the performance of systems like Pig is of great importance.

In this paper, we examined the feasibility of integrating Pig with Stratosphere. We show that Pig can highly benefit from using Stratosphere as the backend system and gain performance, without any loss of expressiveness. We concluded that, even though Pig is tightly coupled to the Hadoop execution engine, integration is possible by replacing the stack below the Logical Plan layer. The translation algorithm and prototype integration of Pig with Stratosphere allows execution of Pig Latin scripts in the Stratosphere execution engine, without modifying the scripts, while offering improved performance.

Several issues remain unexplored and are interesting for further investigation. We certainly believe that creating a system that fully supports Pig Latin and generates Stratosphere jobs is not the limit of this research. Several optimizations can now be added to Pig because of the underlying Nephele execution engine. For example, Pig Latin could be extended to include keywords corresponding to Output Contracts or PACT’s compiler hints. Since Stratosphere now offers its own high-level language, Meteor, it would also be very interesting to compare its expressiveness, usability and performance against Pig.

### Acknowledgements

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF) under the contract RIT10-0043. The authors would also like to thank the Stratosphere team for their help throughout this work.
Bibliography


Chapter 8

Block Sampling: Efficient Accurate Online Aggregation in MapReduce

Vasiliki Kalavri, Vaidas Brudza and Vladimir Vlassov

In 5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2013
CHAPTER 8. BLOCK SAMPLING: EFFICIENT ACCURATE ONLINE AGGREGATION IN MAPREDUCE

Abstract

Large-scale data processing frameworks, such as Hadoop MapReduce, are widely used to analyze enormous amounts of data. However, processing is often time-consuming, preventing interactive analysis. One way to decrease response time is partial job execution, where an approximate, early result becomes available to the user, prior to job completion. The Hadoop Online Prototype (HOP) uses online aggregation to provide early results, by partially executing jobs on subsets of the input, using a simplistic progress metric. Due to its sequential nature, values are not objectively represented in the input subset, often resulting in poor approximations or “data bias”.

In this paper, we propose a block sampling technique for large-scale data processing, which can be used for fast and accurate partial job execution. Our implementation of the technique on top of HOP uniformly samples HDFS blocks and uses in-memory shuffling to reduce data bias. Our prototype significantly improves the accuracy of HOP’s early results, while only introducing minimal overhead. We evaluate our technique using real-world datasets and applications and demonstrate that our system outperforms HOP in terms of accuracy. In particular, when estimating the average temperature of the studied dataset, our system provides high accuracy (less than 20% absolute error) after processing only 10% of the input, while HOP needs to process 70% of the input to yield comparable results.

8.1 Introduction

Real-time and near real-time large-scale data management and analysis have emerged as one of the main research challenges in computing. Modern large-scale analytics applications include processing web data, transaction and content-delivery logs, scientific and business data. Popular systems nowadays use massive parallelism and are usually deployed on clusters of inexpensive commodity hardware. However, even in such a highly parallel setting, analyzing big data sets takes a considerable amount of time. Many data analysis applications can tremendously benefit from using early approximate results, which can be available before job completion. Such applications can tolerate some inaccuracy in the results, while gaining significantly reduced response time. Example applications include search engines, estimation of Twitter trending topics, weather forecasts and recommendation systems.

Early accurate approximation techniques have been extensively studied in the context of relational databases. Popular works suggest using Wavelet transformations [2, 6, 9, 18], histogram-based approximations to return results with known-error bounds [17, 12] or sampling techniques [15, 20].

Hadoop [1], an open source implementation of Google’s MapReduce [8], is the most popular and widely used big data processing framework. Due to its batch processing nature, though, it does not allow partial job execution. Even though existing research in approximate techniques for relational queries can serve as a starting point, applying such techniques to MapReduce-style processing frameworks is very challenging [3, 16, 13]. Apart from the aforementioned batch-style processing nature, the biggest challenge is posed by the unstructured data format that such systems need to analyze. Having sufficient knowl-
8.1. INTRODUCTION

![Figure 8.1. Popularity Variation Keyword Searches over a 3-month Period](image)

edge on the input data structure, types and cardinalities of a query greatly facilitates the estimation of result accuracy. Unfortunately, this is not the case in Hadoop, where data is available as raw files stored in a distributed file system. Data is transformed into key-value pairs only after a job has been submitted for execution. Moreover, different applications can freely choose to interpret input data in different ways. MapReduce allows applications to specify arbitrary user-defined functions to be executed during the map and the reduce phase, thus making result estimation even harder.

These challenges have not discouraged researchers to try to integrate approximate result techniques in MapReduce. However, existing solutions are either far from matching the success of their database predecessors or are limited to specific operators and query templates [3, 16, 13]. A notable research work, MapReduce Online [7] integrates Online Aggregation into the Hadoop processing framework. MapReduce Online uses pipelining between operators, thus enabling partial job execution. However, input is read in a sequential manner and estimation accuracy is based on a simplistic job progress metric. Consequently, results are often highly inaccurate and are likely to exhibit data bias. Consider an application that computes the popularity of a search keyword over some specified period. It is often the case that some keywords, representing topics or events, become very popular over a short period of time or their popularity varies depending on the time of the day or day in the week. Two examples of keyword search distributions are shown in Figure 8.1, generated by Google Trends. If we estimate popularity using MapReduce Online, the result will be highly dependent on the selected period of time. For example, a partial job estimating the average popularity of the keyword “G1 league” of Figure 8.1(a), would sequentially process data from left to right, missing the values represented in the later spike, thus returning an inaccurate answer.

In this paper, we present a statistically profound early estimation technique as a layer over MapReduce Online. We propose a simple, yet efficient random sampling technique implementation, which significantly improves the accuracy of Online Aggregation. In order to overcome the challenge of the unstructured nature of data, our implementation performs the sampling before data is sent to the mapper tasks, as soon as it is organized in blocks. Moreover, in order to avoid slow random disk accesses, we propose in-memory shuffling of data blocks, thus efficiently achieving random sampling and reducing data bias. We evaluate our implementation in terms of performance and accuracy, using real-world datasets of various sizes and distributions. We assess the accuracy of early estimations and study its dependency over our introduced block-level sampling technique parameters. We show that our system delivers highly accurate results, while matching MapReduce Online in perfor-
The main contributions of this paper are as follows.

- A novel, efficient, in-memory block sampling technique for MapReduce applications.
- A method for integrating the block sampling technique with a distributed large-scale processing framework.
- An implementation of the block sampling technique for the Hadoop Online Prototype (HOP).
- An experimental evaluation of the proposed technique, focused on performance and accuracy, using real-world datasets and applications.

The rest of this paper is organized as follows. Section 9.2 gives the necessary background for the paper. Section 8.3 presents our block sampling technique and gives details on the design, architecture and implementation of our solution. In Section 9.6, we provide evaluation results and comparison with Hadoop and the HOP. Section 9.7 discusses related work. We discuss conclusions and future work in Section 9.8.

8.2 Background

In this section, we briefly review the MapReduce programming model, the MapReduce Online framework and the Online Aggregation technique.

8.2.1 The MapReduce Programming Model

MapReduce [8] is a programming model for large-scale parallel data processing. In the MapReduce programming model, one simply has to specify an input path in the distributed file system, two user-defined functions, map and reduce, and an output path. Data is read from the file system, organized in blocks and shipped to parallel map tasks, where they are parsed into key-value pairs. Each parallel map task processes one block and applies the used-defined map function on each key-value pair, producing new pairs as output. The output pairs are then grouped by key and are sent to parallel reduce tasks, which apply the reduce function on each group. The result of each reduce task produces one file in the distributed file system. MapReduce rapidly became popular, as it ensures efficient and reliable execution of tasks across large numbers of commodity machines and successfully manages to hide the complex details of parallelization, data distribution, fault tolerance and load balancing from the user.

8.2.2 MapReduce Online

MapReduce Online [7] is a modified version of Hadoop MapReduce, a popular open-source implementation of the MapReduce programming model. It supports Online Aggregation
8.3. THE BLOCK SAMPLING TECHNIQUE

and stream processing, while also improving utilization and reducing response time. Traditional MapReduce implementations materialize the intermediate results of mappers and do not allow pipelining between the map and the reduce phases. This approach has the advantage of simple recovery in the case of failures, however, reducers cannot start executing tasks before all mappers have finished. This limitation lowers resource utilization and leads to inefficient execution for many applications. The main motivation of MapReduce Online is to overcome these problems, by allowing pipelining between operators, while preserving fault-tolerance guarantees.

8.2.3 Online Aggregation

Online Aggregation [11] is a technique enabling interactive access to a running aggregation query. In general, aggregate queries are executed in a batch-mode, i.e. when a query is submitted, no feedback is given during the query processing time. Consequently, the accumulated results are returned only after the aggregation process is completed. The technique enables partial query processing, without requiring prior knowledge of the query specifications, such as types of operators and data structures. As a result, users are able to observe the progress of running queries and control their execution (e.g. stop query processing in case early results are acceptable). Due to the lack of knowledge on query and data characteristics, Online Aggregation relies on random sampling to provide early results. The system is then able to provide running confidence intervals along with an estimated query result. A number of estimators for several types of running confidence interval computations has been proposed in [10]. Though the Online Aggregation technique never made a big impact in the commercial database products, it becomes newly relevant due to the rising interest in fast large-scale data processing.

8.3 The Block Sampling Technique

In this Section, we describe the block sampling technique in detail. We discuss the design goals of our implementation and the integration of the proposed technique with the MapReduce Online framework.

8.3.1 Design Objectives

Providing early accurate query results in big data frameworks is a very challenging problem. Our vision is to provide a simple, yet efficient and robust solution, independent of the specific processing engine. Our design is based on the following principles:

- **Statistical Robustness**: The system should be able to provide a valid estimate of the final job results at any given time during the job execution. We consider an estimate valid, if it has been computed over a uniform random sample of the entire job input dataset. Statistical properties should be guaranteed independently of the input data format or its underlying distribution.
CHAPTER 8. BLOCK SAMPLING: EFFICIENT ACCURATE ONLINE AGGREGATION IN MAPREDUCE

- **Framework Independence:** The solution should be generic, with a simple design, easy to integrate with any big data processing framework using a block-based storage or file system, such as HDFS. Thus, we have chosen to realize the implementation as a thin independent layer, on top of the MapReduce Online stack.

- **Application Transparency:** Existing applications should be able to benefit from the proposed technique, without any modification. Furthermore, users should be able to select whether the new functionality will be active or inactive during execution.

- **Efficiency:** The introduced modifications should impose minimal overhead when compared to the original system. This includes both the MapReduce processing and data collection phases.

### 8.3.2 System Architecture

Before describing our system architecture, we first briefly review the MapReduce Online framework execution workflow. Like in standard Hadoop, in MapReduce Online, data is stored in HDFS [5] and is read in blocks, which are sent to map tasks wrapped into Input Splits. Before applying the map function, mappers parse the Input Splits into key-value records. The structure of the input data is described by the InputFormat method. This method selects a valid data reader, which reads an input file sequentially, parses it and returns the input records to the map tasks. The output of each map task is stored in an in-memory buffer and is periodically sent to the reduce tasks. The reduce tasks collect and merge partial results sent by mappers and the reduce function is applied to each record in the merged results file. Based on the initial job configuration, the output of a reduce task is materialized as part of the final job results or as a snapshot.

Standard map tasks access one data block and process it sequentially, one record at a time. In order to obtain a uniform random data sample, we chose to slightly modify the read operation and have each call to the record reader return a random record from the spill file. The implementation challenge here comes from the fact that common distributed file systems, like HDFS, are highly optimized for sequential reads, while random accesses are very expensive. Furthermore, the record reader should have sufficient prior information about the spill file, i.e., how many records each data block stores, which is not available at this stage of the workflow. In order to avoid expensive random disk accesses, we chose to process file splits in memory as shown in Figure 8.2. Each map task retrieves a number of record blocks from several data splits (1). The accessed record blocks are stored in memory (2), shuffled in-place in order to reduce possible data bias (3) and then sent to the map function (4). This way, block-level sampling reduces the I/O related overhead, which is inevitable when accessing multiple files in the distributed file system. We are convinced that the shuffling phase is also necessary, as it reduces the bias of collected data and guarantee the statistical properties of partial results (taken at any time during the execution). Finally, the input data structure remains consistent, so there are no MapReduce applications based constraints.
8.3. THE BLOCK SAMPLING TECHNIQUE

8.3.3 Implementation

We have implemented a prototype of the proposed block sampling technique on top of HOP, that we call HOP-S. The source code is publicly available \(^1\). In this section, we discuss implementation details. Likewise the described design and implementation process can be split into two phases: bias reduction and sampling.

Bias Reduction

Data randomization is a necessary step in order to reduce the overall bias of the data and ensure that the randomness property is guaranteed at any time during the execution. We have implemented the data randomization phase as part of the initial reading process, after running several tests and verifying that it is an efficient solution. The functionality can be enabled by setting up a newly introduced MapReduce daemon parameter. The list of supplementary Hadoop framework parameters is given in Table 8.1.

The bias reduction process is done in the following steps:

1. **Access phase.** During the initial step, the map tasks retrieve the file URI from the HDFS NameNode and access the specified file. By default, a file is read as a stream of bytes and parsed into lines of text (based on specified separator characters), text is then parsed into records and finally sent to the map user-defined function. We have introduced a new data collection method, which processes the blocks of the entire input split and stores them in the local task’s memory. Further processing is delayed until the next described shuffling phase is completed.

2. **Shuffling phase.** The input data is stored in a custom in-memory data structure. Its design focuses on optimizing data randomization: a number of bytes (corresponding to a single line of text) retrieved from HDFS is stored as a tuple, with an additional, randomly generated, integer value in range of \([0...2^{31} - 1]\). The randomly generated integer value facilitates the shuffling: stored data is sorted according to the assigned integer value, thus returning randomly shuffled input data. Our tests showed that 512

\(^1\)https://github.com/vasia/HOP-S
CHAPTER 8. BLOCK SAMPLING: EFFICIENT ACCURATE ONLINE AGGREGATION IN MAPREDUCE

Table 8.1. Newly Introduced Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>io.file.shuffle</td>
<td>Input records shuffling method for data bias reduction.</td>
</tr>
<tr>
<td>io.split.maxsubsplit</td>
<td>Set the number of split files from which block-level sample will be created. Default value = 4.</td>
</tr>
<tr>
<td>io.split.insort</td>
<td>If enabled, split files are shuffled before sampling method is applied.</td>
</tr>
</tbody>
</table>

MB of memory allocated to TaskTracker child processes is sufficient for the default 64 MB block size of HDFS.

3. **Processing phase.** Even though storing the shuffled data to disk would enable us to reuse the initial Hadoop data reading function, this step would introduce unnecessary I/O overhead. Therefore, we have developed an additional read method, which is used when the data randomization is enabled. It skips the data access and preprocessing steps and instead serves the line of text from local memory.

**Block-level sampling**

The block-level sampling requires several modifications of the map phase, mostly related with the storage system access. The processes of file sampling and processing are presented in detail next.

By default, each mapper retrieves and processes a single data block. Our aim is to force a map task to fetch and process data from multiple HDFS data blocks, thus emulating data sampling. The initial file information is acquired by issuing the request to HDFS. We introduce the RandomFileInputFormat which stores each file’s information in local memory. Initial data splits are then divided into equally-sized blocks, to be processed by the separate map tasks. This approach requires a minimum number of random accesses (one-per-block), reducing the cost of HDFS I/O operations. We experimentally investigate the overhead of I/O operations in Section 9.6.

The number of blocks each split is divided into is determined in the MapReduce job configuration. In case the data split cannot be divided into equal-sized parts, the sampling process will determine the optimal sizes for each block ensuring that mappers receive a comparable workload, in terms of input size. Each map task will be responsible for processing a number of blocks from separate splits, equal to the number of parts each split is divided to. For example, if the user sets the `io.split.maxsubsplit` parameter to 6, map tasks will process 6 blocks of data from 6 separate input splits. Furthermore, the list of input splits, obtained in the data access phase, can be shuffled by setting the `io.split.insort` parameter, before the sampling process starts.
8.4. EVALUATION

We have additionally developed a RandomFileSplit format, necessary for storing the complete block-level samples information, including URIs of the input splits, data offsets and block lengths. This information is used to query the HDFS NameNode during the data access phase, thus enabling map tasks to sequentially open several input data streams.

Finally, we have implemented two additional InputFormat methods, in order to enable processing of the block-level samples, which are produced during the sampling phase. There are two main differences between the default and our introduced formats. First, our methods are able to access multiple input splits and read the blocks of data from each of them. This process is cyclic: as one data stream is consumed (certain number of bytes is read), a next one is initiated, until the whole sample is processed. Furthermore, block-level samples can start or end in the middle of a text line of the initial dataset, since the sampling phase relies solely on the stored file’s meta-data. This issue is addressed with a simple rule: a task that receives a data block with the beginning of text line will process the entire line, otherwise, it will skip that fragment of the line.

8.4 Evaluation

We have evaluated the block sampling technique and our implementation on top of the Hadoop Online Prototype, HOP-S. This section presents the results of our evaluation.

8.4.1 Evaluation Environment and Datasets

Our setup consists of an OpenStack cluster, deployed on top of 11 Dell PowerEdge servers, each with 2 x Intel Xeon X5660 CPUs (24 cores in total), 40 GB of memory and 2 TB of storage. We ran all experiments using 8 large-instance virtual machines, each having 4 virtual CPUs, 8 GB of memory and 90 GB of disk space. Nodes run Linux Ubuntu 12.04.2 LTS OS and have 1.7.0_14 version JavaTM SE Runtime Environment installed. We configured Hadoop, HOP and HOP-S to use up to 17 map tasks and 5 reduce tasks per job, HDFS block size of 64MB and set the data replication factor set to 2.

For our experiments, we retrieved and pre-processed a number of varying size datasets. For the performance evaluation, we acquired weather data for several decades, available from the National Climatic Data Center ftp server. Data is present for each year separately, (available years 1901 to 2013) and contains log files from a number of different weather stations. The logs were compressed and stored separately, per station. As a pre-processing step, we merged each weather station log files into a single yearly weather dataset. The size of aggregated log varies from 2 to 10 GB, depending on the year of measurement. In total, we acquired 21 aggregate log files consisting of 100 GB of data.

For the accuracy evaluation experiments, we prepared several datasets with different data distributions of size between 11 and 25 GB. The first dataset is an extract of the previously mentioned 100GB weather dataset and consists of 10 arbitrarily selected files of 25 GB size in total. The yearly log files of the each weather station are merged sequentially, one after the other. For our second experiment, we reuse this dataset, after sorting it ac-

\[\text{ftp://ftp3.ncdc.noaa.gov/pub/data/noaa/}\]
CHAPTER 8. BLOCK SAMPLING: EFFICIENT ACCURATE ONLINE AGGREGATION IN MAPREDUCE

8.4.2 Results

First, we present experiments to measure performance and investigate possible sources of overhead. Next, we evaluate the estimation accuracy of the proposed technique. We try to identify sensitivity factors and how they influence the results’ accuracy.

Performance Evaluation

We first evaluate the influence of the snapshot materialization frequency, namely how often estimations will be materialized into HDFS, on the overall system performance. This is an overhead present in both HOP-S and HOP and it is usually nullified by the gains provided by pipelining [7]. Note that both bias reduction and sampling are disabled in this experiment. For this test, we set the maximum memory size available for map and reduce tasks to 512 MB per child. Our results are displayed in Figure 8.3(a). Each value is obtained by averaging the results of 3 to 5 executions over an extended period of time, to reduce the influence of cluster performance fluctuations. Overall, the results show that there is a moderate execution overhead (up to 35%) in the case of frequent snapshots. This overhead mainly occurs due to the reduce tasks being unable to process the output of map tasks during the period of snapshot materialization to the HDFS file system. Based on our observations, we recommend to set the snapshot materialization parameter to every 25% of processed input, as it has less overhead (about 15%) in comparison to executions with higher snapshot frequency settings. However, as we show in the next section, even earlier estimations of various MapReduce jobs can have high accuracy.

3http://www.gutenberg.org/
For the second experiment, we enable bias reduction and compare the performance of HOP-S to Hadoop and MapReduce Online. For this task, we prepared seven different size datasets, varying from 5.5GB to 100GB, which were used in the aggregate average temperature job execution. In order to minimize the influence of clusters performance variation, we ran multiple executions over each dataset and averaged the results, which are shown in Figure 8.3(b). We demonstrate the evaluation results of three systems: Hadoop (ver. 0.19), MapReduce Online and HOP-S with bias reduction enabled. The measured performance difference between the standard Hadoop framework and the other two frameworks is insignificant, after taking into account the overhead of frequent early result snapshots. Note that the snapshot frequency is set to 10% for all tests. We further investigate how the bias reduction process performance depends on system parameters. While the input data size is relatively small, bias reduction has little to no overhead. However, in the case of large inputs, the overhead noticeably increases, up to 20% over execution time with no bias reduction. Such results can be explained by the choice of system parameters, namely the relatively low number of map tasks, resulting in additional processing time being aggregated over a large number of sequentially processed data blocks each JVM executes a large number of map tasks sequentially). We strongly believe, though, that the shuffling phase is the main source of the inspected overhead, when input data records are arranged by the assigned random prefix. However, there is another source of potential overhead, not reflected in the present figure. Before sorting, all records are stored in the local memory of the processing nodes. The amount of required memory depends on the size of the HDFS data blocks. As a rule of thumb, we recommend increasing the available memory of each node to 4 times the blockSize.

In the last experiment, we measure the overhead of the sampling process to the overall job execution time. We use the 100 GB size input dataset, on which we ran an average temperature computation MapReduce application, using HOP-S and also the MapReduce Online framework. We performed several tests varying the block-level sampling rate. Results are shown in Figure 8.3. We observe that there is a noticeable overhead in comparison to job execution time over the MapReduce Online framework, which is growing with the number of block-level samples. We identify the following overhead sources: the sample generation process, the block-level samples movement across the nodes and the increased number of random accesses to HDFS, as there is a linear dependency between the number of additional random accesses introduced and the chosen sampling rate.

Accuracy Evaluation

In this section, we discuss experiments which evaluate the precision of early results, as produced by MapReduce Online, with and without using our proposed technique. We define accuracy as the absolute error between the early returned result of a partially executed job and the result returned after processing the entire dataset. We use the two previously described weather datasets with different distributions of values.

For the first experiment, we use a 25GB dataset of ten yearly aggregate logs of weather stations. We ran the MapReduce application that measures the average yearly temperature on Hadoop, HOP and HOP-S. First, we evaluate the variation of estimation accuracy over
the number of block-level samples processed by each map task. The estimations were materialized and stored to HDFS every 10% of the processed input data. For this test we enabled both data sorting and sampling. Figure 8.4(a) shows the aggregated estimation results of 2 randomly selected yearly log files. The graph illustrates the absolute error range variation over the number of block-level samples given for each map task. Each box defines the 25th/75th percentiles, while the drawn line shows the minimum and maximum values of the measured absolute error. Each column consists of 8 values obtained after 10 up to 80% of input data is processed. We observe that the default setting of 4 block-level samples per map task is too conservative. When a low number of block-samples is processed by each map task, there is a higher probability that one or few of yearly datasets will be present to a lesser extent, thus the estimation accuracy might suffer. On the other hand, results obtained in case of 10 to 16 blocks per map are very accurate even after processing just 10% of the input. At later stages of the job execution, the estimations become relatively accurate independently of the number of processed blocks. Figure 8.4(b) shows the results for the same experiment ran on the sorted weather data. The main difference is the values distribution of the data. If this dataset is read sequentially (as in MapReduce Online), winter months will be processed first. In comparison to the unsorted weather data, there are only slight variations in the maximum values of the error range. However, the accuracy of estimations converges promptly. After 20% of processed input, estimations have very low errors. Based on these results, we set 10 samples per block for the rest of the experiments.

Figure 8.5(a) shows the results for the yearly average temperature calculation applica-
8.5. RELATED WORK

In an experiment, over varying-size processed inputs. The gray boxes correspond to execution of the application on the Hadoop Online system, while the colored boxes correspond to executions of the application on HOP-S. For Hadoop Online, some of the values have 100% error rate. This means that the system did not provide any estimation for a particular year. Also, all ten estimations were available only after 40% of the input data was processed. Furthermore, we notice that the system mostly processes blocks of one or few input files at a time, therefore resulting in a maximum error value between 40 to 70 percent of the processed input data which does not change. Overall, we conclude that Hadoop Online does not provide statistically meaningful results. Some of the early estimations might be reasonably accurate even at early stages of processing, while others required the whole input data to be processed in order to return a fairly accurate estimation. On the other hand, HOP-S gives promising results. Even after processing only 10% of the input, the maximum value of the absolute error is around 30%, with 25th/75th percentiles being less than 20%. Furthermore, accuracy steadily increases with the amount of processed input. We also notice that even at 10% of processed input, average temperature estimations are available for all 10 years, whereas the Hadoop Online framework required 40% of input to do the same. Similar results were observed when running the same experiment over logs of the weather data, sorted by date. The results are illustrated in Figure 8.5(b).

In a final experiment, we evaluated the estimation accuracy of a top-100 words application MapReduce job, for which we used the e-books dataset described in Section 8.4.1. One important difference from the previous datasets is that it has a Zipfian distribution: a small number of words occur very frequently, while many others occur rarely. We measured the number of missed words (out of the top-100 final words) over the part of processed input data and observed that even after 10% of the processed input, both MapReduce Online and our designed system give fairly precise results, with up to 7 misses. As the segment of the processed input data grows, the number of missed words is reduced. We observed close to no difference in estimations of the HOP and our system. Due to the Zipfian distribution, the block-level sampling technique does not provide tangible benefit over sequential processing, however, it still offers a great advantage over traditional batch-processing, since complete processing is not necessary.

8.5 Related Work

8.5.1 Large-Scale Parallel Data Processing Systems

Adapting the approximation techniques, previously used in databases, to large-scale, distributed processing systems is not straight-forward. Major challenges include the unstructured nature of data, the shared-nothing distributed environments such systems are deployed on and the need to support more complex analysis operations than simple aggregations.

Apart from MapReduce Online, which we briefly cover in Section 9.2, there have been several other important contributions towards the direction of providing fast, approximate yet accurate results in large-scale MapReduce-like systems.
CHAPTER 8. BLOCK SAMPLING: EFFICIENT ACCURATE ONLINE AGGREGATION IN MAPREDUCE

The EARL library [13] is an extension to the Hadoop framework and focuses on accurate estimation of final results, while providing reliable error estimations. It uses the bootstrap technique, which is applied to a single pre-computed sample. Consequently, numerous subsamples are extracted and used to compute the estimate. The accuracy of estimation can be improved with an expansion of the initial sample size and increase in number of used subsamples. EARL allows estimations for arbitrary work-flows and requires only minimal changes to the MapReduce framework. However, efficiency is very dependent on the job and the provided dataset properties and distribution. The bootstrap technique often proves to be very expensive and would result in even slower execution than running the complete job without sampling.

The BlinkDB [3] approximate query engine creates and uses pre-computed samples of various sizes to provide fast answers. It relies on two types of samples: large uniform random and smaller multi-dimensional stratified. Queries are evaluated on a number of selected samples and an initial estimate is produced. BlinkDB is based on the predictable query column sets (QCS) model, so it assumes that a constant set of data columns, used for group or filter predicates, exist in the stored datasets. As a result, the system can estimate results of standard aggregate queries easily. More complex queries, including arbitrary joins, are currently not supported. In case the accuracy or time constraints of a query are not met, larger or smaller samples can be selected. The accuracy of various queries estimation depends on the composition and sizes of stored samples. However, samples creation (primarily stratified) is an expensive task and can take considerable amount of time. Consequently, the processing of newly arrived data to be included in the preceding samples can be delayed. On the other hand, our designed system can return continuously improving accuracy estimates without the additional pre-processing or the requirement to previously store samples. Further, it does not require pre-assumptions made in the predictable QCS model.

Another important work builds Online Aggregation [16] for MapReduce jobs, using the Hyracks execution engine [4]. The authors argue that Online Aggregation is newly relevant for the Cloud Computing cost model, as it can save computations and therefore money. They aim to adjust the classic work of databases in a MapReduce environment and mainly focus on facing the challenges that rise because of the shared-nothing cluster environment. In such a setup, where failures are also quite frequent, it is hard to guarantee the statistical properties of the partially processed input. They propose an operational model and a Bayesian framework for providing estimations and confidence bounds for the early returned results. However, in order to guarantee such properties, the system only supports applications conforming to a specialized interface and limited to the set of common aggregate functions.

Finally, Facebook’s Peregrine [14] is a distributed low-latency approximate query engine, built on top of Hive [19] and HDFS. It supports a subset of operators and provides approximate implementations for some aggregate functions. A user has to explicitly use the approximate functions in their query and can get terminate it before the execution is complete. After termination, information is provided on the number of scanned records and possible failures that occurred during execution. In order to provide fast results, Peregrine uses one-pass approximate algorithms and an in-memory serving tree framework for
computing aggregations.

8.6 Conclusions and Future Work

The amount of data organizations and businesses store and process everyday is increasing with tremendous rates. In order to analyze data efficiently and at a low cost, the academic and industry communities have relied on data-parallelism and have developed distributed, shared-nothing processing architectures and frameworks, like MapReduce. However, even with these highly distributed solutions, query latency is still very high. Data analysts often have to wait for several minutes or even hours to acquire a result. In many cases, however, quite accurate answers can be returned after only processing a small subset of the available data and great value can be extracted by only partial results. Several analysis applications can tolerate approximate answers to queries and highly benefit from lower latency. Such a functionality can also be used for rapid prototyping or pattern discovering in huge datasets.

Query result approximation is not a novel idea. There has been extensive research on the topic from the database community in the past. The data explosion that we are experiencing today, leaves us no choice but to reconsider approximation techniques, in the context of large-scale MapReduce-style systems, in order to reduce query response times. However, adoption of existing techniques is not straight-forward and proves to be very challenging. In the MapReduce world, data is not organized in tables or properly structured, but is often schema-less and stored in raw files. Moreover, analysis applications are usually much more complex than simple aggregations and can use arbitrary user-defined functions.

In this paper, we present the block-level sampling technique, which can provide a random sample of the provided dataset, without requiring pre-processing or additional storage space. We integrated block-level sampling with the MapReduce Online framework and we show that, together with an additional bias reduction technique, it can provide accurate estimations of results, without requiring a-priori knowledge of the query. In order to achieve that, data is sampled before the map stage and is shuffled in-memory, in order to introduce randomization. As a result, map tasks still can access data sequentially, avoiding the overhead of random disk accesses.

The evaluation of HOP-S shows superb results over the standard MapReduce Online framework, in terms of the early aggregate jobs estimations accuracy. Consequently, the execution time of most aggregate applications can be reduced noticeably, while still maintaining the high accuracy of the estimations. We demonstrate, that our system can estimate the average temperature of 100GB weather dataset with as low as 2% error, up to 6 times faster than a complete job execution time. However, we also show that the benefit varies and is highly dependent on the data distribution. For example, in the case of Zipfian distribution, MapReduce Online can return quite accurate results, even with sampling disabled. Nevertheless, we display that early estimations of the most frequent values in such datasets can be very accurate, thus the complete process of the input data is not always necessary. An interesting alternative metric that is subject of our future work, is the time needed to achieve a certain accuracy level.

Several open and interesting issues remain to be explored in the context of our work.
First, we would like to explore the feasibility of integrating statistical estimators into our system, in order to provide error bounds or similar useful feedback to users. We are also interested in trying to automate the sampling process and optimize the strategy, based on system configuration parameters, such as block size and available memory. The automatic process would fine tune the sampling process to reach the best possible performance for the provided system configuration. Another direction would be to investigate alternative sampling techniques, or even wavelet-based early approximation techniques and explore the possibility for integration with large-scale processing frameworks.

Acknowledgment

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF) under the contract RIT10-0043.
Bibliography


Chapter 9

m2r2: A Framework for Results Materialization and Reuse in High-Level Dataflow Systems for Big Data

Vasiliki Kalavri, Hui Shang and Vladimir Vlassov

In 2nd International Conference on Big Data Science and Engineering (BDSE), 2013
CHAPTER 9. M2R2: A FRAMEWORK FOR RESULTS MATERIALIZATION AND REUSE IN HIGH-LEVEL DATAFLOW SYSTEMS FOR BIG DATA

Abstract

High-level parallel dataflow systems, such as Pig and Hive, have lately gained great popularity in the area of big data processing. These systems often consist of a declarative query language and a set of compilers, which transform queries into execution plans and submit them to a distributed engine for execution. Apart from the useful abstraction and support for common analysis operations, high-level processing systems also offer great opportunities for automatic optimizations. Existing studies on execution traces from big data centers and industrial clusters show that there is significant computation redundancy in analysis programs, i.e., there exist similar or even identical queries on the same datasets in different jobs. Furthermore, workload characterization of MapReduce traces from large organizations suggest that there is a big need for caching job results, that will enable their reuse and improve execution time.

In this paper, we propose m2r2, an extensible and language-independent framework for results materialization and reuse in high-level dataflow systems for big data analytics. Our prototype implementation is built on top of the Pig dataflow system and handles automatic results caching, common sub-query matching and rewriting, as well as garbage collection. We have evaluated m2r2 using the TPC-H benchmark for Pig and report reduced query execution time by 65% on average.

9.1 Introduction

Big data collection, processing and analysis is becoming one of the major concerns for large and small organizations, companies and academic institutions. Operations, business decisions, product recommendations and numerous other everyday tasks are increasingly relying on processing and analyzing large datasets of diverse formats and heterogeneous sources. The need for making the power of big data analysis available to non-experts and analysts with no programming experience, quickly led to the development and adaptation of high-level, dataflow systems for data analysis.

Pig [9], Hive [18] and Jaql [5] are among the most widely used high-level dataflow frameworks for big data analytics. They offer an easy programming experience, using declarative languages and support for common data analysis operations, such as filtering, projection, join, grouping, etc. Their advantages include ease of use, fast prototyping, readable and easily-maintainable programs. Studies show that a big percentage of analytics is performed using such high-level layers [7]. Apart from their obvious benefits for users, high-level dataflow systems also offer great opportunities for automatic optimizations.

In this paper, we propose an optimization regarding identifying and avoiding computational redundancies, i.e., similar or identical queries in different jobs. It is based on recent research studies revealing that redundancies exist in a big extend among typical analysis workloads [14, 16, 11], showing a 30-60% of similarity in queries submitted for execution. In other words, parts of queries or even whole jobs submitted for execution re-appear unchanged in future job submissions. A common case is some kind of initial filtering or transformation on a dataset before the main analysis task. For example, filtering out badly formatted records, spam e-mails or transforming a data representation into another.
9.1. INTRODUCTION

The common way to implement this optimization is by caching query and sub-query results, to avoid re-computing identical tasks in future jobs. The idea is based on the materialized view technique, popular in relational databases [10, 20, 3]. In order to exploit materialized results, the system needs to identify the redundancy in future job submissions and provide a mechanism to rewrite queries, so that stored results can be reused. In the context of high-level dataflow systems, there has been some remarkable previous work, mainly for DryadLINQ and Pig [11, 14, 16, 8]. The proposed materialization and reuse frameworks handle the problem by providing an execution plan matcher and query rewriter. These frameworks are naturally highly coupled to the underlying processing systems and execution engines. ReStore [8], for example, matches plans at the physical level, assuming that Pig scripts are translated into MapReduce jobs, while DryadInc [16] assumes a Dryad DAG as the execution model.

However, there has been recent interest in integrating popular high-level dataflow languages with alternative execution engines or developing new ones. In particular, Shark [19] offers an implementation of Hive on top of Spark [22] and PonIC [15] translates Pig scripts into Stratosphere [4] jobs. Furthermore, several different parallel-dataflow systems can be used by the same organization, inside the same datacenter or cluster to process common datasets. Since all these technologies are quite recent and are still evolving, it might be the case that different development teams, inside the same company, use different frameworks and languages to implement similar analysis tasks and process common datasets.

We are, therefore, in need of a language and execution engine-independent and extensible framework for storing, managing and re-using query results. We observe that, despite the differences in the backends of existing high-level dataflow processing systems, they do share common design characteristics on higher layers. These systems often offer a high-level declarative-type language, whose statements correspond to data operators, defining how data will be organized and processed. Thus, a query is normally translated into a DAG of operators, called the logical plan. This plan is then optimized and translated into a physical execution plan, which defines how the operators will be implemented and executed on the underlying parallel execution engine. For example, in the case of Pig and Hive, the final plan is a DAG of MapReduce jobs, while in the case of Shark, it is a Spark dataflow and in the case of PonIC it is a Stratosphere job.

In this paper, we present m2r2 (materialize-match-rewrite-reuse), a language-independent and extensible framework for storing, managing and using previous job and sub-job results. In order to achieve generality and support different languages and backend execution engines, we have chosen to base our design at the logical plan level. We provide a detailed description of general techniques for identifying candidate sub-plans for materialization. We propose mechanisms for efficient storage and retrieval of plans and sub-plans, in order to exploit reuse opportunities and we also discuss garbage collection and management of the results repository. We report our on-going work on a prototype implementation using the Pig framework and MySQL Cluster as the repository for storing and managing plans and sub-plans. We present very promising preliminary results, using the TPC-H benchmark for evaluation. The main contributions of this paper are as follows.

- We discuss and identify architecture and design similarities in high-level dataflow...
processing systems for big data, in order to find the proper level for providing a reuse mechanism based on materialization and caching of previous query results.

• We propose a language and execution engine-independent framework, m2r2, for results materialization and reuse.

• We describe a prototype implementation of our materialization and reuse framework for the Pig system.

• We provide an evaluation of our prototype implementation of m2r2, using the TPC-H benchmark for Pig.

The rest of this paper is organized as follows. Section 9.2 provides a brief overview of the materialized view techniques in relational databases. Section 9.3 discusses the similarities in the design and implementation of popular high-level dataflow systems for big data. Section 9.4 presents the design of m2r2 and Section 9.5 discusses implementation details regarding our prototype on Pig. In Section 9.6, we provide evaluation results. Section 9.7 discusses related work, while we conclude and share our future research directions in Section 9.8.

9.2 Background

9.2.1 Materialized Views in Relational Databases

A materialized view in the context of relational databases is a derived relation, stored in the database. Creating and managing materialized views is driven by several applications, such as query optimization, maintaining physical data independence, data integration and others [13]. In this work, we are only interested in materialized views techniques used for query optimization. The idea is based on the fact that queries can be computed from materialized views, instead of base relations, by reusing results of common sub-queries. We briefly discuss the three main problems related to this technique: view design, view maintenance and view exploitation.

View Design

View design determines which views will be materialized, considering the trade-off between the limited space for storing the views and the search cost for finding a related view. Thus, it is not practical to create a view for every sub-query, but instead, use a technique to select which sub-queries to materialize. View design is carried out in two steps, view enumeration and view selection. View enumeration aims to reduce the number of candidate views to be considered by the selection phase, by filtering out the non-related views. View selection is based on a cost-benefit model. A view is considered beneficial if it is expensive to compute and if it can be reused by other queries. The cost is computed based on the overhead to select, create and store the views and the overhead to keep the views updated. Since relational databases use cost-based query optimizers, View selection can be easily
9.3. Design and Similarities of High-Level Dataflow Systems for Big Data

integrated with the query optimizer. Therefore, views are selected by the query optimizer based on their benefit and cost.

View Maintenance

View maintenance refers to the problem of updating the materialized views when the base relations change. Specifically, when operations such as Insert, Update or Delete are performed on the base relations, the materialized views get “dirty” and they should either be updated or garbage collected. [12] discusses the view maintenance problems and techniques in detail. We are not interested in view maintenance, since most parallel processing platforms assume append-only input data. However, we discuss garbage collection in Section 9.4.5.

View Exploitation

View exploitation describes how to efficiently use materialized views for query optimization. It includes two phases, view matching and query rewriting. View matching defines how to find the related views that can be used for answering queries. Query rewriting generates a new query, using the selected views. The rewritten query can either be an equivalent expression to the original and provide an accurate answer or, only provide a maximal answer. In this paper, when referring to query rewriting, we always mean equivalent rewriting.

9.3 Design and Similarities of High-Level Dataflow Systems for Big Data

In this section, we summarize the system design and common characteristics of popular high-level dataflow systems for big data analytics. Our study is mainly based on Pig [9], Hive [18], Jaql [5] and DryadLINQ [21]. We discuss the main system components and focus on the architecture and compilation similarities, which motivate our design decisions.

9.3.1 Language Layer

The majority of high-level dataflow processing systems offer a declarative, SQL-like scripting language for writing applications. Programs consist of series of statements, each of which, defines a transformation on one or more collections of datasets and produces new collections. Data is usually read from a distributed file system and final results are also stored there. All systems allow user-defined functions, which can be used in conjunction with the language-provided statements. A wide variety of datatypes is also supported, allowing schema specification and nested data structures.

9.3.2 Data Operators

The language statements correspond to data operators, objects that encapsulate the logic of the transformations to be performed on datasets. Data operators have one or multiple inputs.
and usually one output. We refer to the special operator which accepts input from persistent storage, such as a file system, as the \textit{Load} operator and to the special operator that writes its output to persistent storage as the \textit{Store} operator. We refer to common analysis operators for manipulating collections of data such as Filter, Group By, Join, Order, etc as \textit{Relational} operators. Relational operators are distinguished from \textit{Expression} operators, such as Sum, Count, Avg, (elsewhere also seen as functions), which can be composed to form expressions and are usually nested inside relational operators. Some high-level languages also support \textit{control-flow} operators, which we do not consider at present.

\subsection*{9.3.3 The Logical Plan}

After a script is submitted for execution, it is sent to the parser, which is responsible for creating the Abstract Syntactic Tree (AST). If there are no syntax violations, the AST is transformed into a Directed Acyclic Graph (DAG), called the \textit{logical plan}. The nodes of this plan correspond to data operators, which are connected by directed edges, denoting data flow. The logical plan usually has one or more sources, corresponding to Load operators and at least one sink, which corresponds to the Store operator.

\subsection*{9.3.4 Compilation to an Execution Graph}

In order to produce the final execution plan, the logical plan goes through two main phases, optimization and translation. During the optimization phase, the plan is simplified and data-flow optimizations are applied, such as filter pushdown, column pruning, etc. The goal is mainly to reduce the amount of data that will be transferred from one operator to the other or create parallelization opportunities. The output of the optimization phase is a rewritten, optimized logical plan. The optimized logical plan is next transformed into a lower-level representation, usually referred to as the \textit{physical plan}. The physical plan is a more detailed and usually larger DAG of fine-grained operators, the \textit{physical} operators.

While the logical operators only contain information about the semantics of an operation, the physical operators encapsulate information regarding its actual physical execution. For example, a logical operator Join represents the result of matching two or more datasets based on a key and collecting the set of records that are matched. On the other hand, a physical operator Join contains information about the specific execution strategy that will be used for executing the join (hash-based approach, a merge-sort, the replicated strategy, etc.). Finally, the physical plan is translated into an engine-specific dataflow DAG of tasks or jobs, such as Map-Reduce jobs or Dryad jobs, each encapsulating a part of the initial logical plan.

\subsection*{9.3.5 Discussion}

We observe that all of the systems share very similar designs on the upper layers, from the language layer until the optimization of the logical plan. The step of translating the logical plan into a physical plan is where the system logics start to divert. This is mainly due to the differences that exist in the backend frameworks, since physical operator implementations essentially depend on the capabilities of the underlying execution engine. We,
therefore, believe that it is not a wise design choice to make any assumptions about the backend execution engine. Instead, since our main goal is to make a general framework, we build the materialization and reuse framework at the logical level, which is more stable and less likely to change. The logical plan level is abstract enough to provide us with some information about operator costs, even if we might lose some reuse opportunities, due to reduced granularity. It is true that the more we move down in the compilation process layers, the more fine-grained operators we can exploit, with higher chance for reuse and more information regarding operator costs. However, in that case, we would have to build a specialized system for each language and execution engine. It is our goal to be able to exploit reuse and sharing opportunities among different frameworks, as well as provide support for adding new operators and new languages in the future. We are certain that the logical plan layer is the appropriate point for integration, in order to build a language-independent and execution-engine independent, extensible and configurable framework for results materialization and reuse.

9.4 m2r2 Design

We summarize here our main design goals.

- **Independence of the high-level language.** Our design is based on the assumption that there exists an abstract logical operator layer, similar to the one described in Section 9.3.2. If necessary, this layer can be customized based on the different characteristics and requirements of each language.

- **Independence of the execution engine.** This design goal is fulfilled by choosing the logical layer for plan matching. Query rewriting happens before the compilation into physical operators and is, therefore, independent of their implementation.

- **Extensibility.** It is our intention to create a fully extensible framework, so that new operators, languages and execution engines can be easily supported.

- **Configurability.** Our preliminary experiments show that system parameters are very sensitive to workload characteristics. Therefore, we have decided to allow the users to tune important system parameters, such as the set of operators after which sub-plans are materialized, the degree of replication of stored results and the garbage collection policy.

- **Effectiveness and Efficiency.** The reuse mechanism should provide some gain in query execution and this gain should exceed the overhead associated with the reuse mechanism.

The five components of our framework, Plan Matcher and Rewriter, Logical Optimizer, Results Cache, Plan Repository and Garbage Collector, are discussed next.
9.4.1 Plan Matcher and Rewriter

The Plan Matcher and Rewriter takes as input an optimized logical plan and produces a rewritten logical plan, as output, compatible with the logical plan that the compiler of the underlying framework is expecting. It stores a representation of the (sub) plan and execution statistics about the job in the repository. The Match phase performs lookups in the repository for finding potential matches. If the fingerprint of the input plan or its sub-plans exists in the repository and the output exists in the results cache, then we have a match. In this case, the Rewrite phase is activated and generates new logical plan by adding or removing operators from/to the original logical plan. The tasks performed by this component can be summarized in the four following steps: (1) Choose a set of sub-plans from the input plan to materialize; (2) Calculate the fingerprints of the plans and its sub-plans; (3) Store the fingerprints and execution statistics of the selected plan in the repository, if no match is found or rewrite the query if there is a match. A flow diagram of the Plan and Rewrite Algorithm is shown in Figure 9.1.

Sub-Plan Selection Criteria

One of the main challenges we need to solve is selecting which sub-plans to materialize, since each sub-plan has a different cost and possibilities to reoccur in a future job submission. In order to select the most beneficial sub-plans, we examine the following three criteria: (1) Whether the data size of the output is reduced; (2) whether the computation is expensive; (3) reuse possibilities from future queries. Commonly used operators, such as Filter, often used in the very beginning of the logical plan are good candidates for reducing the data size. Projection is another good candidate, as it often performs column pruning. On the other hand, Group and Join are two commonly used expensive operators, which, for some execution engines, such as MapReduce, incur significant I/O overhead. Thus,
9.5. IMPLEMENTATION

materializing these operators is also a good idea. In general, users should better combine the selection strategy with a specific workload.

9.4.2 Plan Optimizer

The logical optimizer optimizes the rewritten logical plan. First, Store operators are inserted in order to generate the sub-plans. Second, the matched sub-plan is replaced by a Load operator, which might not be able to recognize the type of the input data. We must therefore translate the new data scheme. Most of the high-level dataflow frameworks already provide a logical optimizer layer. We highly encourage users to use the provided optimizer or extend it, in case some additional functionality is necessary.

9.4.3 Results Cache

The Results Cache stores intermediate results and results of whole jobs. In order to simplify implementation, we suggest using the already provided data storage by each framework, e.g. a distributed file system. This way, rewriting plans is greatly simplified and reuse becomes transparent to the underlying system. However, since most systems use replication for fault-tolerance, using the same configuration for storing intermediate results might increase the overhead. We, therefore, encourage users to disable replication for the results cache.

9.4.4 Plan Repository

The Plan Repository stores the hash of the selected plan, the output path of the plan and statistics about the stored results, such as the reuse frequency and the last access timestamp. It can be implemented as a key-value store or an in-memory relational database.

9.4.5 Garbage Collector

The Garbage Collector operates based on information collected by the Repository manager and the Results Cache. It locates obsolete records in the Repository and deletes the related data in both the Repository and the Cache. The garbage collection policy is configurable and can be implemented as least-recently-used, based on reuse frequency or a combination of statistics. Garbage collection can be implemented as a periodical background process or can be explicitly invoked by the user. Alternatively, several threshold values can be set, so that garbage collection is triggered when one of them is violated, for example if the available disk space is too low.

9.5 Implementation

We implemented a prototype reuse framework on top of Pig/Hadoop. The system architecture is shown in Figure 9.2.
CHAPTER 9. M2R2: A FRAMEWORK FOR RESULTS MATERIALIZATION AND REUSE IN HIGH-LEVEL DATAFLOW SYSTEMS FOR BIG DATA

Figure 9.2. Implementation of the Reuse Framework on Pig

9.5.1 Match and Rewrite Phase

We have chosen to represent logical plans simply as Strings. The choice is driven by two facts. First, Pig already provides a method for transforming an operator into a String, containing all necessary information for identifying the operator. Therefore, composing operators becomes as simple as concatenating Strings. Second, having the plans in a String representation greatly simplifies the computation of their fingerprints, which we need to store in the repository.

In Pig, a query can be uniquely identified by its input and its logical plan. For simplicity, we assume that input files are identified by file names and file paths and that these do not change. Therefore, two identical files with different names are treated as different files by our implementation. To calculate the hash of a sub-query, we first calculate the fingerprint of its logical plan, then we calculate the fingerprint of its input and then use their concatenation to uniquely identify the query. We compute the fingerprint of a logical plan with the following steps: (1) A depth-first traversal to retrieve a list of ordered operators in the plan, as demonstrated in Figure 9.3. (2) Concatenation of the string representations of the operators; (3) Calculation of the hash of the final string. We extend Pig’s getSignature() method in the LogicalPlan class to also acquire the fingerprints of the plan’s inner plans during the traversal.

9.5.2 Results Cache and Repository

We have used HDFS [17] as the Results Cache and MySQL Cluster [1] as the Repository for our implementation. MySQL Cluster is an in-memory database which provides high read/write throughput. It is a highly available, scalable and fault-tolerant store. We use ClusterJ to connect Pig with MySQL Cluster, because it is simple to use and can well satisfy our needs. We created one table in MySQL Cluster to represent the Repository, with four columns: hashcode - hash of the plan; filename - output path of the plan; frequency - the
9.6. EVALUATION

Figure 9.3. Bottom-Up Logical Plan Traversal

reuse frequency; last access - last time to access the record. Correspondingly, we created an annotated interface in Pig, having the same four properties. When a sub-plan is selected for materialization, we insert a new record at the MySQL Cluster table and store the result in HDFS. By using HDFS as the Results Cache, we can simply rewrite Pig Latin scripts by removing the corresponding sub-plan operators and adding a Load operator in their place, with the specified HDFS path. Similarly, in order to store a sub-plan, we only have to add a Store operator, after the chosen materialization point in the logical plan.

9.5.3 Garbage Collection

We have implemented garbage collection as a separate component, which can be invoked at will by the user. The Garbage Collection process includes three main steps: filter out the obsolete records, delete the corresponding outputs from the Results Cache and then delete the records from MySQL database. We have based the garbage collection policy on reuse frequency and the last access time. Since the frequently reused records tend to have a more recent access time than the less reused ones, our garbage collector is only based on the last access time threshold, which is a user-defined configuration parameter. For example, if we set the threshold to 5 days, the records that have not been accessed within 5 days from the time we ran the garbage collector will be deleted. In this case, the materialized results that have never been reused will also be eventually deleted, since their last access value does not exist. The reuse frequency threshold can also be specified by user, but we did not use it for our experiments.

9.6 Evaluation

In this section, we describe the environment we used for evaluating our prototype, present and discuss our results.
9.6.1 Environment Setup

We have set up a Hadoop cluster and a MySQL cluster on top of OpenStack. We have used 20 Ubuntu 11.10 virtual machines, each one running Java(TM) SE Runtime Environment version 1.7.0. We have installed Hadoop version 1.0.4 and configured one Namenode and 15 Datanodes. The Namenode has 16 GB of RAM, 8 cores and 160GB disk, while the Datanodes have 4GB of RAM, 2 cores and 40 GB disk. For convenience, we have installed Pig 0.11 on the Hadoop Namenode. We have also used MySQL cluster version 7.2.12, with one manage node, one SQL node and 2 data nodes. Each node has 4GB of RAM, 2 cores and 40GB of disk space.

9.6.2 Data and Queries

We have used Jie Li’s work [2] for running the TPC-H benchmark in Pig. We have generated the data sets using the DBGEN tools of the TPC-H Benchmark and have generated 107 GB of data in total. In order to create reuse opportunities, we have created some new queries by changing substitution parameters. Finally, we selected 20 queries in total for evaluation, 14 from the original TPC-H Benchmark queries and 6 newly created queries, with reuse opportunity.

9.6.3 Sub-Plan Selection Strategy

We have configured m2r2 to materialize results after the Join and the CoGroup operators. For logical plan with multiple inputs, if one input branch changes, then the final Join of all the inputs must be recomputed. Thus, if the plan has multiple inputs, we only materialize the intermediate results on each input branch before the final Join operator. The TPC-H queries usually contain multiple inputs, such as Query 20. Starting from the Store operator, Match and Rewrite would not be performed until reaching the branches. An Example Pig logical plan showing a selection of a sub-plan to be materialized is shown in Figure 9.4.
9.6. EVALUATION

![Figure 9.5](image.jpg)

**Figure 9.5.** Comparison of Query Execution Times on Pig without Materialization and Query Execution Times with Materialization. When the materialization mechanism is enabled, materialized results are produced but not exploited.

### 9.6.4 Results

In this section, we present the results of our evaluation. We have first executed the set of the 20 TPC-H queries (14 original and 6 with modified substitution parameters) in Pig with and without enabling our materialization framework. We have run each test at least 5 times, ensuring that standard deviation of the measurements is under 8%. In the first set of experiments, we are interested in investigating the overhead that our materialization framework incurs compared to regular Pig scripts execution. Although we only discuss here the overhead regarding total execution time, we are also planning to look into storage overhead in future experiments. In the second set of experiments, we focus on quantifying the benefits gained by reusing materialized results for query execution. Finally, we also examine the performance of our implementation and the throughput of MySQL Cluster.

#### Materialization Overhead

Figure 9.5 shows a comparison of the execution time for the 14 original queries on Pig without materialization with the query execution time on Pig with materialization enabled. We clarify that, in this experiment, no materialized results were exploited, so that the measurements depict the pure overhead of the materialization mechanism. The numbers on x axis are in correspondence with the query numbers in TPC-H Benchmark. When having materialization enabled, the overhead consists of the Match and Rewrite algorithm, the optimization of the rewritten logical plan, storing execution statistics to MySQL Cluster and storing results in HDFS. We observe that for queries 4, 6, 12, 13 and 17 there is no overhead or negligible overhead. This is due to the fact that either no intermediate results were materialized or their size was too small. For queries 2, 3, 5, 7, 8, 10, 18 and 20 we notice very small overhead, from 5% to 25% over the total execution time. Finally, we observe a quite large overhead for query 18, around 120%. In this case, the size of the intermediate results chosen for materialization was very large. We expect this case not to be so frequent and note that this is a one-time overhead that will benefit all future matching queries.
CHAPTER 9. M2R2: A FRAMEWORK FOR RESULTS MATERIALIZATION AND REUSE IN HIGH-LEVEL DATAFLOW SYSTEMS FOR BIG DATA

Figure 9.6. Comparison of Query Execution Times With and Without Using Sub-Job Materialized Results

Benefit of Reusing Plans and Sub-Plans

After having materialized the plans and selected sub-plans of the first 14 queries, we have run the 6 additional queries containing common sub-queries. The results are shown in Figure 9.6. The figure shows a comparison of query execution times of the system when using the reuse framework to exploit results of sub-jobs, with query execution times of the system without materialization. On the x axis, it is shown in parentheses the number of the original query from which each new script was produced. We observe a speedup of 50% - 80% for most queries, except for query 25, which has the lowest speed up of 30%. This is because we did not modify this particular query so that it can reuse its most I/O dominant sub-plan.

For the second part of this experiment, we run again all 20 queries, after having stored all relevant data in MySQL and HDFS, in order to measure the benefit of reusing results of whole jobs. As expected, the achieved speedup, shown in Figure 9.7, is tremendous and the execution time is around 30 - 45 sec for all queries. The speedup depends on the base execution time, the size of the final results and the execution time variance. In this case, for all queries, the execution time essentially consists of the time to make a lookup in the repository and return the location of the stored results in HDFS. The total execution time of all 20 queries with materialization disabled is around 753 min and is reduced to around 13 minutes, when the results of whole jobs can be reused. This scenario is probably not realistic, but we believe it is highly indicative of the potential benefits of the materialization and reuse framework.

Non-I/O Overheads

The non-I/O overheads are introduced by the execution of the Match and Rewrite algorithm and the access to MySQL Cluster. We have found that the non-I/O overheads are negligible when compared to I/O overheads. Creating a session to access MySQL cluster takes the longest time. We have collected around 200 values of session creation time, 62.6% of them being around 780 ms. The execution time of the Match and Rewrite algorithm is also at the
9.7. RELATED WORK

The measured time includes computing the fingerprints, lookup of matched plan and MySQL Cluster read/write operations.

9.7 Related Work

Restore [8] is a non-concurrent sharing framework built on top of Pig, which uses materialized results for query optimization. The main idea is very similar to our work but it is designed exclusively for Pig on Hadoop and operates on the physical level. Restore consists of four components: a sub-job enumerator, a matcher and rewriter, a repository and a sub-job selector. The sub-job enumerator selects sub-jobs for materialization and the matcher and rewriter performs match and rewrite of plans. In our implementation, both functionalities are performed by the Plan Matcher and Rewriter. The repository is used to store execution statistics and is analogous to MySQL Cluster in m2r2. An important implementation difference is that ReStore stores and matches the plan object, while we use fingerprints to identify and match plans. Moreover, garbage collection is not implemented. We have not included an comparative evaluation of our implementation with ReStore, because, to the best of our knowledge, the system was not publicly available at the time this paper was written.

Incoop [6] is a non-concurrent sharing framework that can reuse results for incremental computation. It detects input data changes by using content-based chunks instead of fixed-size chunks to locate files. During the map phase, results of the unchanged chunks are fetched from the file system and processed by mapper. During the incremental reduce phase, Incoop not only stores the final results, but also stores sub-computation results by introducing an additional phase, called the contraction phase. During contraction, the input of a reduce job is split into chunks which are processed by combiners. Thus, the result is computed by combining the results of unchanged chunks and the output of the combiners. Unlike Incoop, we focus at identifying the computations sharing the same input and not inputs sharing the same computations.

Microsoft has conducted remarkable research on query optimization by materializing results, DryadInc [16], the Nectar [11] and Comet [14], being the most representative systems related to our work. All are built upon the Dryad/DryadLINQ and like us, assume an append-only file system. DryadInc is similar to Incoop and focuses on non-concurrent sharing incremental computation. It uses two heuristics to do incremental computation:
identical computation and mergeable computation. Nectar is a non-concurrent sharing framework, which aims to improve both computation efficiency and storage utilization. Regarding computation, it uses a program rewriter and a cache server to avoid redundant computations, similar to DryadInc. For data management, it uses a mechanism to store all executed programs and materialize all computation results. When the garbage collection process detects the existence of computation results that have not been accessed for a long time, it replaces them with the programs that produce them. Comet enables both concurrent sharing and non-concurrent sharing. It performs three kinds of optimizations: query normalization, logical optimization and physical optimization. In query normalization phase, a single query is split into sub-queries, which can be reused by other sub-queries. Moreover, each sub-query is tagged with a time stamp, so that the system can identify whether a set of sub-queries from different queries that can be executed together. During logical optimization, common sub-queries are detected and are executed for only once. Reordering is also enabled, in order to expose more common expressions. Comet also considers co-location when replicating materialized results, for reducing network traffic. During physical optimization, shared scan and shared shuffling are performed. While all the above systems assume Dryan/LINQ as the programming and execution environment, in our work, we have designed a language-independent and extensible framework for storing, managing and using previous job and sub-job results. Our initial prototype of the m2r2 framework is built on top of the Pig dataflow system and can be easily extended to support other platforms.

9.8 Conclusions and Future Work

The problem of computation redundancy is very relevant in big data systems. Several studies have shown large similarities in data analysis queries and suggest that any type of caching techniques would greatly benefit data analysis frameworks.

Following the idea of materialized views in relational databases, we have examined the possibility of porting this technique in big data environments. In this work, we present m2r2, a results materialization and reuse framework for high-level dataflow systems. We have examined and compared several popular high-level dataflow languages and execution engines and we have summarized their common properties and design characteristics. We observe that the majority of systems follow very similar designs in their upper layers, namely the language and logical plan layer, while their physical layers differ significantly. We, therefore, present a design for integrating a materialization and reuse framework after the logical layer of high-level dataflow processing systems.

We have implemented an initial prototype framework on top of Pig/Hadoop and have used the TPC-H Benchmark to evaluate our work. The results show that when there exists sharing opportunity, query execution time can be immensely reduced by reusing previous results. We also show that the induced overhead of materialization is quite small, around 25% of the total execution time without materialization, while non-I/O overheads are negligible. We note that both benefit and overhead are very sensitive to framework parameters, such as sub-job selection strategy and garbage collection policy, as well as specific workload characteristics.
9.8. CONCLUSIONS AND FUTURE WORK

There are a number of open issues and interesting paths to explore in the context of this research. First, we plan to extend m2r2, by decoupling it from the Pig framework and building a general, component-based and extensible framework, as described in Section 9.4. We then intend to integrate it with other popular high-level systems. We are also particularly interested in exploring possibilities of sharing and reusing results among different frameworks. Furthermore, we believe it is essential to obtain execution traces from industrial partners and big organizations, in order to evaluate the gains and overheads of the materialization framework under more realistic conditions. We would like to examine how benefits are related to different workload characteristics, data distributions and cluster sizes. We intend to minimize the imposed overhead, by overlapping the materialization process with regular query execution, thus moving it out of the critical path of the execution. Finally, we also wish to examine the possibility of extending m2r2 in order to support incremental computations and exploit concurrent-sharing opportunities.

Acknowledgment

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF) under the contract RIT10-0043.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


Chapter 10

Asymmetry in Large-Scale Graph Analysis, Explained

Vasiliki Kalavri, Stephan Ewen, Kostas Tzoumas, Vladimir Vlassov, Volker Markl and Seif Haridi

To Appear in 2nd International Workshop on Graph Data Management Experiences and Systems
Abstract

Iterative computations are in the core of large-scale graph processing. In these applications, a set of parameters is continuously refined, until a fixed point is reached. Such fixed point iterations often exhibit non-uniform computational behavior, where changes propagate with different speeds throughout the parameter set, making them active or inactive during iterations. This asymmetrical behavior can lead to many redundant computations, if not exploited. Many specialized graph processing systems and APIs exist that run iterative algorithms efficiently exploiting this asymmetry. However, their functionality is sometimes vaguely defined and due to their different programming models and terminology used, it is often challenging to derive equivalence between them.

We describe an optimization framework for iterative graph processing, which utilizes dataset dependencies. We explain several optimization techniques that exploit asymmetrical behavior of graph algorithms. We formally specify the conditions under which, an algorithm can use a certain technique. We also design template execution plans, using a canonical set of dataflow operators and we evaluate them using real-world datasets and applications. Our experiments show that optimized plans can significantly reduce execution time, often by an order of magnitude. Based on our experiments, we identify a trade-off that can be easily captured and could serve as the basis for automatic optimization of large-scale graph-processing applications.

10.1 Introduction

Iterations are inevitably in the heart of many graph-parallel algorithms. Commonly, in these algorithms, the task is to iteratively refine the values of the graph vertices, until a termination condition is satisfied. In each iteration, new values of the vertices are computed, using an update function. The algorithm terminates when some convergence criterion is met.

Many iterative graph algorithms expose non-uniform behavior, where changes propagate at different speeds across iterations. Examples include any algorithm that contains some kind of an iterative refinement process. Ideally, one would like to detect this phenomenon and stop the computation early for the inactive parts. This would allow the system to avoid redundant computation and communication. Applying this simple optimization requires detecting inactive vertices and identifying the parts for which computation can halt. However, one must examine how, halting computation for some vertices, could potentially affect the correctness of the computation for the rest of them. In other words, even if inactive parts can be accurately identified, it might not always be possible to halt computation for these parts and still obtain correct results.

To clarify these issues, we use the single source shortest paths (SSSP) algorithm. Consider the graph of Figure 10.1, where S is the source, the weights on the edges represent distances, i is the iteration counter and the values in the boxes show the distance computed for each vertex after each iteration. In this example, the algorithm is refining the distances of vertices from the source vertex S. In each iteration, a vertex receives new candidate distances from its incoming-neighbors, selects the minimum of these candidates and its
current value, adopts this minimum as the new value and then propagates it to its out-going neighbors (in a vertex-centric programming model). For this problem, it is trivial to detect the vertices that have reached convergence; the ones whose value does not change between two consecutive iterations (shown in gray in Figure 10.1). It is also easy to see that if we halt computation for these vertices (i.e. the vertices do not send or receive any values), the final result will still be correct. For example, in iteration 3, the value of A does not need to be recomputed. Moreover, A does not need to propagate its distance to B or C again. Any future distances they will compute can only be equal or lower than their current values.

Even though the way to apply this optimization to SSSP may seem obvious, it cannot be easily generalized for all similar algorithms. Consider that the given graph is part of a web graph and the task is to iteratively compute PageRank on this graph. In its simplest form, during one iteration of PageRank, a vertex computes its new rank by summing up the weighted ranks of all of its incoming-neighbors. If we try to apply the previously described optimization in this case, and assuming that vertex A converges in iteration 3, then during this iteration, vertex C will only receive the weighted ranks of vertices B and D, sum them up and therefore compute an incorrect value (missing A's contribution).

A large number of highly-specialized systems for large-scale iterative and graph processing have emerged [8, 9, 11], while there also exist general-purpose analysis systems with support for iterations [13, 12, 6]. Specialized systems are usually designed to exploit dataset dependencies, in order to efficiently execute applications and avoid redundant computations. General-purpose systems often do not match the specialized systems in performance, as they typically do not embody sophisticated optimizations for graph processing. Each system requires computations to be expressed using different programming abstractions and it is not always trivial to derive a mapping from one model to another. Pregel
[9], for example, uses the vertex model and defines that if a vertex does not receive any messages during an iteration, it becomes deactivated and does not execute or produce messages in the subsequent superstep. GraphLab [8] realizes similar behavior with its adaptive execution mechanism. However, it is left to the developer to decide when it is safe to deactivate vertices or halt parts of the computation. This requires the user to understand both models and to carefully verify the correctness of the algorithm.

In this work, we present an overview of general optimizations for graph processing, in the presence of asymmetrical behavior in computations. We study the characteristics of several iterative techniques and we describe what these characteristics mean and how they can be safely exploited, in order to derive optimized algorithms. More importantly, we give the necessary conditions under which, it is safe to apply each of the described optimizations, by exploiting problem-specific properties. We use general-purpose dataflow operators to create template optimized execution plans, which can detect converged parts and avoid redundant computations, while providing functionality equivalent to this of Pregel and GraphLab. We evaluate the optimizations using two characteristic iterative algorithms, Connected Components and PageRank. We present extensive experiments using real-world datasets of varying sizes. We show that optimized algorithms can yield order of magnitude gains compared to the naive execution. Our contributions can serve as the foundation for building a cost-based optimizer that would relieve the programmer from the burden of manually exploiting asymmetry.

The contributions of this paper are the following:

1. A categorization of optimizations for fixed point iterative graph processing, using a common mathematic model.

2. A formal description of the necessary conditions under which the relevant optimizations can be safely applied.

3. A mapping of the optimization techniques to existing graph processing abstractions.

4. An implementation of template optimized execution plans, using general data-flow operators.

5. An experimental evaluation of the optimizations, using a common runtime.

The rest of this paper is organized as follows. In Section ?? we introduce the notation used throughout this document. In Section 10.2 we present four different iteration techniques. We describe the conditions, under which, each technique can be used and we make a connection between the described optimizations and existing graph-processing systems. We also describe the implementation of each technique as a template execution plan, while we present our experimental results in Section 10.3. We discuss related work in Section 10.4 and conclude in Section 10.5.
10.2. GRAPH ITERATION TECHNIQUES

We use common graph notation to explain the different iteration techniques. Let $G(V, E)$ be an invariant directed graph, where $V$ is the set of vertices and $E$ is the set of directed edges. We also define the following auxiliary problem constructs:

- the **solution set**, containing the values of all vertices in $V$.
- the **dependency set**, containing the in-neighbors of a vertex $v_j$ and
- the **out-dependency set**, containing the out-neighbors of $v_j$.

Figure 10.2. Graph processing techniques as data flow plans.
CHAPTER 10. ASYMMETRY IN LARGE-SCALE GRAPH ANALYSIS, EXPLAINED

### Table 10.1. Iteration Techniques Equivalence

<table>
<thead>
<tr>
<th>Iteration Technique</th>
<th>Equivalent to Bulk?</th>
<th>Vertex Activation</th>
<th>Vertex Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bulk</td>
<td>n/a</td>
<td>always</td>
<td>using values of all in-neighbors</td>
</tr>
<tr>
<td>Dependency</td>
<td>always</td>
<td>if any in-neighbor is updated</td>
<td>using values of all in-neighbors</td>
</tr>
<tr>
<td>Incremental</td>
<td>f is idempotent and weakly monotonic</td>
<td>if any in-neighbor is updated</td>
<td>using values of changed in-neighbors</td>
</tr>
<tr>
<td>Delta</td>
<td>f is linear over composition operator</td>
<td>if any in-neighbor is updated</td>
<td>using values of changed in-neighbors</td>
</tr>
</tbody>
</table>

In the SSSP example of Figure 10.1, the initial solution set would be \( S_1 = \{0, na, na, na, na\} \) and the final solution set would be \( S_4 = \{0, 3, 4, 6, 7\} \). The dependency sets for each vertex would be \( D_S = \emptyset, D_A = \{S\}, D_B = \{S, A\}, D_C = \{A, B, D\} \) and \( D_D = \{B\} \) and the out-dependency sets \( U_S = \{A, B\}, U_A = \{B, C\}, U_B = \{C, D\}, U_C = \emptyset \) and \( U_D = \{C\} \), respectively.

If \( F \) is the update function defined over the domain of the values of the vertices of the iteration graph, \( F \) is decomposable and has a fixed point, then we can compute the fixed point by executing the following procedure: \( S_{i+1} := F(S_i, D) \) until \( S_{i+1} = S_i \).

Next, we describe four graph iteration techniques. We introduce the general bulk technique and then describe three possible optimizations. For each optimization, we give the necessary conditions, under which the optimization is safe. We provide a proof of equivalence with the general-purpose technique in the Appendix.

#### 10.2.1 The Bulk Technique

During a bulk iteration, all the elements of the solution set \( S \) are recomputed, by applying the update function \( f \) to the result of the previous iteration. In the end of each iteration, \( S \) is updated with the newly computed values. The algorithm terminates when none of the values of the solution set changes, i.e. the newly computed \( S \) in iteration \( i \) is identical to the solution set of the previous iteration.

An implementation of the bulk processing model, using common data-flow operators is shown in Figure 10.2(a). The solution set \( S \) contains the vertices of the input graph and the dependency set \( D \) contains directed edges (in case of an undirected graph, each edge appears twice, covering both dependencies). In each iteration, the set of vertices is joined with the set of edges to produce the dependencies of each vertex. For each match, a record is emitted, having the target vertex as key. Then, records with the same key are grouped together and the update function is applied. The newly computed values are emitted and then joined with the previous values in order to update the solution set and check the convergence criterion.
10.2. The Dependency Technique

In the bulk case, a value of a vertex, when recomputed, may produce the same result as the one of the previous iteration. This may happen because (a) either none of the values in the dependency set of the vertex has changed since the previous iteration or (b) applying the update function to the changed values happens to return an identical result. Ideally, we would like to only recompute the values of the vertices that are guaranteed to change value during an iteration. Instead, we can exploit the dependency set to safely select the vertices that are likely to change value in the next iteration.

We introduce two auxiliary sets, the workset \( W \) and the candidate set \( Z \). In each iteration, \( W \) stores the vertices which have changed value since the last iteration and \( Z \) stores the candidate vertices for recomputation. In other words, \( Z \) contains the vertices whose at least one in-neighbor has changed value during the last iteration. \( Z \) is essentially an overestimation of the ideal set of vertices that are guaranteed to require recomputation.

The intuition behind this algorithm is that if a vertex of the iteration graph changes value, then, all its out-neighbors are likely to be affected by this change. On the other hand, if none of the dependencies of a vertex changes value, it is safe to exclude this vertex from the next computation, since recomputing the update function on the same arguments, would return an identical result.

An implementation of the dependency processing model is shown in 10.2(b). The workset \( W \) is isomorphic to the solution set \( S \) and contains only the records corresponding to the vertices that changed value during the previous iteration. \( W \) is joined with the dependency set to generate candidates, emitting a record with the target vertex id as the key for each match. The candidate vertices for recomputation are grouped to remove duplicates and joined with the dependency set on target vertex id, producing a subset of the dependency set. The resulting set serves as input to a subplan equivalent to the previously described bulk model, which only recomputes the new values of the vertices, whose at least one input changed during the previous iteration. The last join operator, only emits records containing vertices that changed values back to the workset.

10.2.3 The Incremental Technique

In some cases, the dependency technique can be further optimized. Specifically, if the function \( f_j \) is of the form \( f_j = t_1 \sqcup t_2 \sqcup \cdots \sqcup t_n \), where \( t_1, t_2, \cdots t_n \) represent independent contributions and \( f_j \) is distributive over the combination operator \( \sqcup \), then we can optimize by only computing \( f_j \) on the changed values of the dependency set in each iteration and then, combine the result with the previous value. For example, if \( t \) is the identity function and the combination operator is minimum, then \( f_j = \min(t(D_j)) = t(\min(D_j)) \). In the graph of Figure 10.1, the value of node B depends on the values of nodes S and A, thus, \( D_B = \{S, A\} \). Then, \( f_B = \min(t(\text{value}(S)), t(\text{value}(A))) = t(\min(\text{value}(S), \text{value}(A))) = \min(\text{value}(S), \text{value}(A)) \). In the Appendix, we prove that, if \( f \) is also idempotent and weakly monotonic, then the combination can be reduced to applying \( f_j \) to the previous value and the partial result. Returning to the SSSP example, minimum is also idempotent (\( \min(a, a) = a \)) and also weakly monotonic, since, for
\[ a \leq a' \text{ and } b \leq b', \min(a, b) \leq \min(a', b'). \]

The incremental technique uses the introduced above workset \( W \), which, in every iteration, contains the elements of \( S \) that changed value since the last iteration and the candidates set \( Z \), which contains the candidate elements for recomputation. Its implementation using common data-flow operators is shown in Figure 10.2(c). This execution plan takes \( W \) as input, which stores only the vertices with changed values. First, the workset is joined with the dependency set to generate the candidate set \( Z \). The result is grouped by key and the update function is applied to each group. After the new values have been computed, only the parameters whose value has changed are emitted into the workset.

### 10.2.4 The Delta Technique

Ideally, for each change \( \delta x \) in the input, we would like to have an efficient function \( \delta F \), such that:
\[
F(x \oplus \delta x) = F(x) \oplus \delta F(x, \delta x)
\]
where \( \oplus \) is a binary composition operator. In this ideal scenario, we could propagate only the differences of values, or *deltas*, from each iteration to the next one. That would potentially decrease the communication costs and make the execution more efficient. However, there are two major factors one has to consider. First, it might not always be the case that computing \( \delta F(x, \delta x) \) is more efficient than simply computing \( F(x \oplus \delta x) \). Moreover, even if we are able to find an efficient \( \delta F \), combining its result with \( F(x) \) could still prove to be a costly operation. In the special case where the update function \( f \) is linear over the composition operator \( \oplus \), then
\[
F(x \oplus \delta x) = F(x) \oplus F(\delta x),
\]
in which case we can use the same function \( f \) in the place of \( \delta f \).

For example, if \( f = \text{sum}(D) \), this optimization is applicable. Let us assume that \( D^i = \{a, b\} \) and \( D^{i+1} = \{a', b\} \), where \( a' = a + \delta a \). Then,
\[
f^{i+1} = \text{sum}(a', b) \Rightarrow f^{i+1} = \text{sum}(a + \delta a, b) \Rightarrow f^{i+1} = \text{sum}(a, b, \delta a) = \text{sum}(f^i + \delta a).
\]

A data-flow execution plan implementing the delta technique is shown in Figure 10.2(d). In this plan, only the differences of values are propagated to the workset.

Table 10.1 summarizes the equivalence among the different techniques and the conditions for safely applying each optimization.

### 10.2.5 Iteration Techniques in Iterative and Graph Processing Systems

In the Pregel [9] model, an iteration corresponds to one superstep. The vertex-centric Pregel model naturally translates to the incremental iteration technique. The vertices receive messages from neighboring vertices and compute their new value using those messages only. The candidates set \( Z \) can be seen as maintaining the subset of the active vertices for the next superstep. The delta iteration technique can be easily expressed using the vertex-centric-model, if vertices produce deltas as messages for their neighbors. To emulate a bulk iteration in the Pregel model, vertices simply need to transfer their state to all their neighbors, in every iteration. Vertices would remain active and not vote to halt, even if they do not have an updated state. Finally, emulating the dependency iteration in Pregel is not that trivial, since vertices in Pregel can only send messages to their out-neighbors. However, in the dependency technique, if a vertex is candidate for recomputation, it needs to *activate* all its in-coming vertices, therefore, it needs to send them a message. A way
10.2. GRAPH ITERATION TECHNIQUES

around this would be to add a pre-processing step, where all vertices send their ids to their out-neighbors. Then, when a vertex receives messages from the in-neighbors, it can use the ids to create auxiliary out-going edges to them. The computation could then proceed by using a three-step superstep as one dependency iteration: during the first step, vertices with changed values produce messages for their out-neighbors. During the second step, vertices that receive at least one message are candidates for recomputation and produce messages for all their in-neighbors, while the rest of the vertices become inactive. In the third step, the candidates for recomputation receive messages from all their in-neighbors and update their value.

GraphLab’s [8] programming abstraction consists of the data graph, an update function and the sync operation. The data graph structure is static, similar to what we assume for the dependency set. GraphLab introduces the concept of the scope of a vertex, which is explicitly declared and refers to the set of values of a vertex and its neighbors. This scope corresponds to the dependency set in the bulk and dependency techniques and to the intersection of the dependency set of a vertex and the workset, in the incremental and delta iterations. Therefore, all four algorithms can be implemented by GraphLab, by computing the appropriate scopes.

PowerGraph [7] is a graph processing system for computation on natural graphs. It introduces the Gather-Apply-Scatter (GAS) abstraction, which splits a program into these three phases. During the gather phase, a vertex collects information from its neighborhood, which then uses during the apply phase to update its value. During the scatter phase, the newly computed values are used to update the state of adjacent vertices. The GAS abstraction can be used to implement both the Bulk and the Incremental iteration plans, while the Delta plan is equivalent to PowerGraph’s delta caching mechanism. The model does not intuitively support the dependency technique. However, it can be implemented in a similar way to the three-step superstep described for Pregel.

GraphX [12] is a graph processing library built on top of Spark [13], in order to efficiently support graph construction and transformation, as well as graph parallel computations. The programming model of GraphX is equivalent to that of Pregel and PowerGraph.

Stratosphere [3, 2] supports flexible plans in the form of a Directed Acyclic Graph (DAG) of operators. Iterations are implemented in Stratosphere as composite operators, which encapsulate the step function and the termination criterion. The implementation of the Bulk and the Incremental algorithms are described in [6]. Nevertheless, all of the algorithms described above can be easily implemented in Stratosphere.

Table 10.2 summarizes the support of each technique in popular graph processing and iterative systems. Most of the existing systems implement the bulk technique by default and special implementations of operators to support the delta optimization. These models assume that the update function has the required characteristics, or that it can be easily re-written, in order to fulfill the required conditions and, therefore, usually do not expose the implementation of an equivalent to the more general dependency technique. Indeed, it is usually trivial to derive an incremental or delta version of a typical aggregate update function. Apart from the cases when an incremental/delta version of the update function cannot be easily derived, the dependency technique can prove to be beneficial in cases when the properties of the update function are not known to the user, for example, if the function
CHAPTER 10. ASYMMETRY IN LARGE-SCALE GRAPH ANALYSIS, EXPLAINED

<table>
<thead>
<tr>
<th>System</th>
<th>Bulk</th>
<th>Dependency</th>
<th>Incremental</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pregel</td>
<td>**</td>
<td>***</td>
<td>*</td>
<td>**</td>
</tr>
<tr>
<td>GraphLab</td>
<td>*</td>
<td>*</td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td>GraphX</td>
<td>**</td>
<td>***</td>
<td>*</td>
<td>**</td>
</tr>
<tr>
<td>Powergraph</td>
<td>*</td>
<td>***</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Stratosphere</td>
<td>*</td>
<td>**</td>
<td>*</td>
<td>**</td>
</tr>
</tbody>
</table>

Table 10.2. Iteration Techniques in Graph and Iterative systems. *: provided by default, **: can be easily implemented, ***: possible, but non-intuitive

used belongs to an external library and the user has no access to its source code.

10.3 Experiments

In this Section, we evaluate optimization strategies for graph processing, by examining two popular iterative algorithms. We implement the applications in Stratosphere [3, 2], an open-source general-purpose framework for large-scale data analysis, which has operators for common processing tasks, such as mapping, group and join. In [6], it is shown that Stratosphere iterations are comparable in performance with Spark and Giraph, even without using the optimizations discussed in this paper.

Our setup consists of an OpenStack cluster, using 9 virtual machines, each having 8 virtual CPUs, 16 GB of memory and 170 GB of disk space. Nodes run Linux Ubuntu 12.04.2 LTS OS.

We evaluate the performance of two iterative algorithms, Connected Components and PageRank. The update function of the Connected Components algorithm (minimum) satisfies the conditions of the incremental technique. Therefore, we implement this application using the Bulk, Incremental and Dependency plans. We execute the Connected Components plans using datasets from [1]. The update function of PageRank (summation of partial ranks) satisfies the conditions of the delta technique. Therefore, we implement this application using the Bulk, Delta and Dependency plans. Initial deltas can be derived from the difference between the uniform initial rank and the in-degree proportional rank. We execute the PageRank plans using datasets from SNAP [1] and the Laboratory for Web Algorithmics respectively [5, 4].

10.3.1 Results

Figure 10.3 shows the number of elements that actually change value in each iteration, for both the Connected Components and PageRank experiments. If this behavior is not accounted for, a lot of redundant computations will be performed. We also observe that the intensity of the phenomenon differs among datasets and depends on the dependency graph properties and the algorithm.

In Figures 10.4 and 10.5, we present execution time measurements for the Connected Components and PageRank algorithms, respectively. In each figure, we have plotted the execution time per iteration for the Bulk, Dependency and Incremental or Delta plan im-
10.3. EXPERIMENTS

As expected, the time for all Bulk iterations is quite stable throughout the execution, for all the cases examined. Regarding the Dependency plan, we observe that in the first few iterations, it is consistently less efficient than the Bulk plan. This is due to the fact that the Dependency plan first needs to identify the candidate elements for recomputation and retrieve their dependencies. This pre-processing step imposes an overhead compared to the Bulk execution. When the amount of elements in the workset is close to the total amount of elements in the solution set, the overhead of the pre-processing is larger than the time saved by updating less elements. In the case of the Connected Components, the Dependency plan outperforms the Bulk plan in later iterations, for both the Livejournal and Friendster datasets. For the Livejournal dataset (Figure 10.4(a)), the execution time of the Dependency plan drops significantly after iteration 7. As seen in Figure 10.3, it is this iteration that the elements in the workset also greatly decrease. Regarding the Friendster dataset (Figure 10.4(b)), the execution time of the Dependency plan outperforms the Bulk plan after iteration 8, but its cost remains more or less stable until convergence, due to the much slower decrease rate of the workset elements, as seen in Figure 10.3. In the case of PageRank, we observe similar behavior for the Livejournal and Wikipedia datasets in...
Figures 10.5(a) and 10.5(b). The Dependency plan cost keeps decreasing as the workset is shrinking across iterations. Regarding the Twitter dataset, Figure 10.5(c) shows that the overhead of the pre-processing step remains dominant in the Dependency execution, during all iterations. This is in accordance to Figure 10.3, where we observe a much smaller decline in the workset size of the Twitter dataset, compared with the other datasets.

Unsurprisingly, the Incremental and Delta plans match the Bulk plan performance during the first iterations, while they significantly outperform it as the workset shrinks. In all cases, the optimized plans outperform the Bulk plan by an order of magnitude or more, as iterations proceed. Finally, we observe that the Delta plan always performs better than both the Bulk and the Dependency plans and its cost continuously decreases.

Our experiments show that the Incremental and Delta plans save a lot of redundant computations and should always be preferred over Bulk plans. However, these plans can be used only when the update function of the algorithm satisfies the conditions described in Section 10.2 for the incremental and the delta techniques respectively. What is more interesting to examine is when and how the more general Dependency plan can be used to speed up total execution time. Our results show that there is a trade-off that depends on the size of the workset. We intend to build a cost model that will be able to capture the overhead of the Dependency plan over the Bulk plan. We plan to use this cost model and the results of our analysis to build a cost-based optimizer to choose the most efficient iteration plan, at runtime.
10.4 Related Work

To the best of our knowledge, no directly related work exists that categorizes and formalizes optimizations for large-scale graph processing. GraphX [12] is the work closest to ours. Like us, the authors realize that graph computations can be expressed using common relational operators, including joins and aggregations. Regarding optimizations, the system supports incremental computation, by maintaining a replicated vertex view, in memory. Our proposed execution plans are more general and do not rely on maintaining views, in order to implement incrementalization.

Delta and Incremental optimizations have been used in several other systems as well. REX [10] is a system for recursive, delta-based data-centric computation, which uses user-defined annotations to support incremental updates. It allows explicit creation of custom delta operations and lets nodes maintain unchanged state and only compute and propagate deltas. The system optimizer discovers incrementalization possibilities during plan optimization, while the user can also manually add delta functions to wrap operators, candidates for incrementalization. Naiad [11] is a stream processing framework which supports computation on arbitrarily nested fixed points. Naiad assumes a partial order, keeps a timestamped full-version history and responds to additions and subtractions of records by maintaining state.

10.5 Conclusions and Future Work

In this work, we present a taxonomy of optimizations for iterative fixpoint algorithms. We describe ways to exploit the asymmetrical convergence behavior to implement optimized iterative execution plans. We offer proof of equivalence between different approaches and we provide a mapping to existing iterative and graph processing programming models. We implement template execution plans, using common dataflow operators and we present experimental evaluation, using a common runtime. Our results demonstrate order of magnitude gains in execution time, when the optimized plans are used.

In the future, we plan to design a cost model to accurately predict the cost of subsequent iterations. Then, using information about the changed elements in the current iteration, a cost-based optimizer could be used to choose the most efficient execution plan, at runtime. We also intend to extend our proposed iteration plans to more general iterative algorithms. We plan to implement a set of iterative applications and compare performance with other iterative and graph processing systems.
Appendix A

1. Notation

An instance of a fixed point iteration problem consists of the following constructs:

- A **solution set** $S = \{x_1, x_2, ..., x_n\}$, which contains a finite number of variables $x_j$, for which we want to reach convergence. The problem is considered solved when all variables converge to their final values, so that none of the values in the set is affected by a subsequent iteration.

- An **update function** $F: \text{domain}(x_j) \rightarrow \text{domain}(x_j)$, decomposable to partial functions $f_1, f_2, ..., f_n$, for computing each $x_j \in S$, where $\text{domain}(x_j)$ is the domain of values in which $x_j$ are defined and $j \in [1, n]$.

- A **dependency set** $D = \{D_1, D_2, ..., D_n\}$, where $D_j \subseteq S$, describes the dependency relation of element $x_j$ on other elements of $S$ and $j \in [1, n]$. $D_j$ therefore contains the elements of $S$ on which the computation of $x_j$’s value depends. In other words, $D_j$ contains the elements of $S$ which appear on the right hand side of the function $f_j$.

From the dependency set $D$, we can easily derive the auxiliary **out-dependency** set, $U = \{U_1, U_2, ..., U_n\}$, where $U_j \subseteq S$. $U$ contains the elements of $S$ whose values depend on $x_j$ and is formally defined as $U_j = \{x_k | x_k \in S \text{ and } x_j \in D_k\}, j, k \in [1, n]$. In the rest of this paper we assume familiarity with standard set notation. Indices $k, j$ are used to refer to elements of $S$ and its subsets, while index $i$ is used to denote an instance of a construct during the $i$-th iteration.

2. Pseudocode

**Bulk Algorithm**

We give the pseudocode for a bulk iteration in Algorithm 1.

**Algorithm 1** Bulk Iteration Algorithm

begin

$S^0 := \{x^0_1, x^0_2, ..., x^0_n\}$

$i := 0$

repeat

\[ x^{i+1}_j \in S^i \]

\[ x^{i+1}_j := f_j(D_j) \] \hspace{1cm} \{(A.1.1)\}

end

$S^{i+1} := \{x^{i+1}_1, x^{i+1}_2, ..., x^{i+1}_n\}$ \hspace{1cm} \{(A.1.2)\}

\[ i := i + 1 \]

until $S^i = S^{i-1}$ \hspace{1cm} \{(A.1.3)\}

end.
Dependency Algorithm

We introduce two auxiliary sets, the workset $W$ and the candidate set $Z$. In each iteration, $W$ stores the parameters of $S$ which have changed value since the last iteration and $Z$ stores the candidate elements for recomputation. In other words, $Z$ contains the parameters whose at least one dependency has changed value during the last iteration. Therefore, $Z$ covers case (a) described above and is essentially an overestimation of the ideal set of parameters that are guaranteed to require recomputation.

The Dependency Algorithm works as follows: The workset $W$ is initialized to be equal to the solution set $S$ and we compute the candidates $Z$, by adding into the set the out-neighbors of each element in $W$. In each iteration, we compute the new value of each variable $x_j$ in $Z$. If the newly computed value of $x_j$ is different from the one of the previous iteration, we add it into the workset.

The pseudocode for the dependency iteration is given in 2.

Algorithm 2 Dependency Iteration Algorithm

\begin{verbatim}
begin
  $S^0 := \{x_0^1, x_2^0, \ldots, x_n^0\}$
  $W^0 := S^0$
  $i := 0$
  repeat
    foreach $x_j \in W^i$
      $Z^i := Z^i \cup U_j$ \{generate candidates\}
    endforeach
    foreach $x_j \in Z^i$
      $x_j^{i+1} := f_j(D_j)$ \{(A.2.1)\}
      if $x_j^{i+1} \neq x_j^i$ then
        $W^{i+1} = W^{i+1} \cup \{x_j\}$ \{(A.2.2)\}
      end
    end
    $S^{i+1} := \{x_1^{i+1}, x_2^{i+1}, \ldots, x_n^{i+1}\}$
    $i := i + 1$
  until $W^i = \emptyset$
end

3. Proofs

Proposition 1. The Dependency Algorithm is equivalent to the Bulk Algorithm.

Proof. We prove this statement using the method of contradiction. Let us assume that the two algorithms are not equivalent. Then, there exists an initial input set $S_0$ and a function $f$ for which the two algorithms converge to different solution sets. Let us assume that the algorithms give identical partial solution sets until iteration $i$, but the results diverge in iteration $i + 1$. If $S_b^{i+1}$ is the partial solution set produced by the execution of the Bulk Algorithm and $S_w^{i+1}$ is the partial solution set produced by the execution of the Dependency
Algorithm after iteration $i+1$, there should exist at least one element that is different in the two sets.

Since $W$ is a subset of $S$, that would mean that the Dependency Algorithm failed in identifying all the vertices that required recomputation during iteration $i$, i.e. there exist $x^i_{j,b} \in S^i_b$, $x^{i+1}_{j,b} \in S^{i+1}_b$, $x^i_{j,z} \in S^i_w$, $x^{i+1}_{j,w} \in S^{i+1}_w$, such that
\[ x^i_{j,b} \neq x^{i+1}_{j,b} \quad (10.1) \]
and
\[ x^i_{j,w} = x^{i+1}_{j,w} \quad (10.2) \]
From the relations 10.1 and (A.1.1) we can derive the following relation:
\[ f_j(D_{i-1}^j) \neq f_j(D_i^j) \]
From the relations 10.2 and (A.2.1) we can derive the following relation:
\[ f_j(D_{i-1}^j) = f_j(D_i^j) \]
and we have therefore arrived at a contradiction.

**Proposition 2.** If $f_j$ is of the form $f_j = t_1 \sqcup t_2 \sqcup \cdots \sqcup t_n$, where $t_1, t_2, \cdots t_n$ represent independent contributions to the value of $f_j$, i.e. $f_j$ is distributive over the combination operator $\sqcup$ and $f_j$ is also idempotent and weakly monotonic, then the Incremental Algorithm is equivalent to the Bulk Algorithm.

**Proof.** Let $x^i_j$ be the value of an element of $S$ in iteration $i$. Since $f_j$ is distributive over $\sqcup$ and idempotent then $x^i_j = f_j(t_1 \sqcup t_2 \sqcup \cdots \sqcup t_n) = f_j(T \sqcup t_n)$, where $T = t_1 \sqcup t_2 \sqcup \cdots \sqcup t_n-1$. Let us assume that during iteration $i$, only $t_n$ changed value and therefore
\[ x^{i+1}_j = f_j(T \sqcup t'_n) \quad (10.3) \]
Since $f_j$ is idempotent,
\[ f_j(t_n, t_n) = t_n \]
and
\[ f_j(t'_n, t'_n) = t'_n \]
Let us also assume that $f_j$ is weakly increasing (the case of decreasing is analogous). Then, we have the following two cases:

- Case 1: $t'_n < t_n : f_j(T \sqcup t'_n) \leq f_j(T \sqcup t_n) \Rightarrow f_j(T \sqcup t'_n) \leq f_j(T \sqcup t_n \sqcup t'_n) \quad (10.4)$
- Case 2: $t'_n > t_n : f_j(T \sqcup t'_n) \geq f_j(T \sqcup t_n) \Rightarrow f_j(T \sqcup t'_n) \geq f_j(T \sqcup t_n \sqcup t'_n) \quad (10.5)$

From equations 10.4 and 10.5 we conclude that
\[ f_j(T \sqcup t'_n) = f_j(T \sqcup t_n \sqcup t'_n) \]
Consequently, equation 10.3 becomes: $x^{i+1}_j = f_j(T \sqcup t_n \sqcup t'_n) \Rightarrow x^{i+1}_j = f_j(x^i_j \sqcup t'_n). \quad \square$
Bibliography


