Optimizing Data Locality in Analytic Workloads over Distributed Computing Environments

by

Yehia Elshater

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
June, 2017

Copyright © Yehia Elshater, 2017
Abstract

With the explosion of data that are generated every second, there is an emerging need for big data analytics using scalable systems and platforms for exploration, mining and decision making purposes. To gain better business insights, the business users are interested to integrate different kinds of analytics to achieve their goals. These analytics may involve accessing the same data for different purposes. Modern data intensive systems co-locate the computation as close as possible to the data to achieve greater efficiency. This placement of computation close to the data is called data locality. Data locality has a significant impact on the performance of jobs in a large cluster since higher data locality means there is less data transfer over the network.

In this work, we examine data locality in parallel processing frameworks and propose approaches to optimize it. First, we conduct a literature review of the existing systems that maximize data locality while processing big data analytics workflows.

Second, we provide YARN Locality Simulator (YLocSim), a simulator tool that simulates the interactions between YARN components in a real cluster to report the data locality percentages. This tool gives the users better insights about the expected performance of the computing cluster.

Third, we develop YARN Dynamic Replication Manager (YDRM), which is a new component in YARN that interacts with the existing YARN’s Resource Manager to improve the data locality.
Co-Authorship


Acknowledgements

I would like to thank my supervisor Professor Patrick Martin who offered me the opportunity to pursue my graduate studies at Queen’s University and guided me through my research work. I acknowledge his patience and his continuous support during my study.

I would also like to thank my thesis committee: Professor Nicholas Graham and Dr Ying Zou for their fruitful comments and feedback. Thanks to NSERC and IBM CAS Lab for funding my research.

My appreciation extends to my fellow labmate: Shadi Khalifa for his insightful discussions during my research. I would like to thank my friends: Abdullah Ibrahim, Mohamed Sami and Safwat Ibrahim for those memorable times we shared together.

I dedicate this dissertation to my parents, Zakaria Elshater and Nagah Saad and my sister Nora Elshater for their care and love. Also, I would like to thank my parents-in-law: Fathy Abokhadra and Ashgan Kotb for encouraging me during my study.

Last but not least, I owe my PhD to my spouse, Hoda Fathy who showed an immense support and love to me and to my kids: Mariam and Yusuf.
Statement of Originality

I hereby certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

Yehia Elshater

(June, 2017)
# Table of Contents

Abstract i  
Co-Authorship ii  
Acknowledgements iii  
Statement of Originality iv  
List of Tables ix  
List of Figures x  
List of Abbreviations 1  

Chapter 1: Introduction 2  
  1.1 Big Data 2  
  1.2 Distributed Computing 3  
  1.3 Data Locality 4  
  1.4 Thesis Statement 4  
  1.5 Contributions 5  
  1.6 Thesis Organization 5  

Chapter 2: Background and Related Work 7  
  2.1 Background 7
3.3 YARN Data Locality Analysis ........................................ 42
  3.3.1 Data locality and running time trade-off ......................... 42
  3.3.2 Unbalanced allocation of containers ............................ 43
  3.3.3 Redundant I/Os ................................................. 44
3.4 Discussion and Summary .............................................. 45

Chapter 4: Extending YLocSim to Support Scalability .............. 48
  4.1 YLocSim Scalability .............................................. 48
    4.1.1 Vertical Scalability ........................................ 49
    4.1.2 Horizontal Scalability ...................................... 49
  4.2 Extended YLocSim Design .......................................... 50
  4.3 HDFS Block Placement Policy ..................................... 52
  4.4 Evaluation ....................................................... 52
    4.4.1 Evaluation Metrics .......................................... 53
      4.4.1.1 Vertical Scalability Validation ......................... 55
      4.4.1.2 Horizontal Scalability Validation (Without Data Re-Distribution) 57
      4.4.1.3 Horizontal Scalability Validation (With Data Re-Distribution) 59
      4.4.1.4 Discussion ............................................. 62
    4.4.2 Using YLocSim to Predict Impact of Scaling .................. 63
  4.5 Summary .......................................................... 69

Chapter 5: YARN Dynamic Replication Manager (YDRM)............. 70
  5.1 YARN Dynamic Replication Manager ............................... 71
    5.1.1 Scoring Data Blocks ........................................ 77
  5.2 Performance Evaluation ........................................... 78
    5.2.1 Evaluation Methodology ..................................... 78
      5.2.1.1 Workloads ............................................. 78
List of Tables

2.1 Comparison between different dynamic replication management systems . . . 29

3.1 Private cluster setup for validating YLocSim . . . . . . . . . . . . . . . . . . 35

4.1 Private cluster setup for validating YLocSim’s extension . . . . . . . . . . 54

5.1 Private cluster setup for validating YDRM . . . . . . . . . . . . . . . . . . 79
List of Figures

2.1 Hadoop 1.0 job execution flow ................................................. 9
2.2 YARN components interaction, adapted from [62] ......................... 11
2.3 HDFS Architecture .................................................................. 12
2.4 HDFS Rack Awareness Policy ................................................... 13
2.5 Data Locality Hierarchy ............................................................ 24
3.1 YLocSim Components ............................................................... 34
3.2 TPC-H Migration Process ......................................................... 37
3.3 Pig queries as DAGs ................................................................. 38
3.4 Node Locality Percentages ......................................................... 38
3.5 Rack Locality Percentages ......................................................... 39
3.6 Off-Switch Locality Percentages ............................................... 39
3.7 Node Locality Percentages ......................................................... 40
3.8 Rack Locality Percentages ......................................................... 41
3.9 Off-Switch Locality Percentages ............................................... 41
3.10 Locality Percentages of Q7, Q8 and Q21 .................................. 43
3.11 Locality Percentages of Q18 and Q21 ..................................... 44
3.12 Running Times in Seconds ...................................................... 45
3.13 Containers Allocation Distribution Over the Cluster Nodes .......... 46
3.14 Loading Input Data Blocks Into Memory .................................. 47
4.1 Example of Horizontal Scalability ............................................. 50
5.8 Replicated Blocks Transfer Percentage .......................... 86
5.9 Geometric Mean Turnaround Time (GMTT) .................... 87
5.10 Imbalance Percentage ............................................ 89
A.1 TPC-H Schema ................................................... 103
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>SLS</td>
<td>YARN Scheduler Load Simulator</td>
</tr>
<tr>
<td>TPC-H</td>
<td>Transaction Processing Council Ad-hoc/decision support benchmark</td>
</tr>
<tr>
<td>YARN</td>
<td>Yet Another Resource Negotiator</td>
</tr>
<tr>
<td>YDRM</td>
<td>YARN Dynamic Replication Manager</td>
</tr>
<tr>
<td>YLocSim</td>
<td>YARN Data Locality Simulator</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Big data analytics require efficient orchestration of workflows of large scale components which cooperate with each other and pool computational resources to achieve the analytics objectives. Big data analytics originated for business organizations, decision makers and scientific research communities who are interested to transform, store, uncover hidden patterns, report and visualize their big data [71, 43].

1.1 Big Data

'Big data' is still a vague expression and not fully understood by the community. However, there are some basic features that are captured by the term 'big data'. According to Manyika et al., big data refers to data sets whose size is beyond the ability of the traditional software tools to capture, store, manage, and analyze [50]. IBM defines basic characteristics of big data which are: volume, variety, velocity and veracity. [74]. Regarding volume, IBM predicts that the amount of data stored around the world will reach 35 zettabytes by 2020 compared to 800,000 petabytes in 2000. The variety dimension describes the different data formats and data sources (e.g. smart phones, social media, enterprise data) that are used to generate these huge amounts of data. Velocity is the third dimension of big data which defines how
quickly the data is created. Velocity is a challenging for organizations that need real-time decision making. Veracity is an important dimension that refers to the level of reliability associated with certain types of big data [57]. Data cleansing methods cannot remove the inherent unpredictability and uncertainty of some data like weather, economy, or a customer’s actual future buying decisions.

1.2 Distributed Computing

With the emergence of distributed computing technologies, like cloud computing, business entities and research institutions are attracted to moving their data analytic workflows to the cloud due to its flexible and on demand resource usage. Instead of investing expensive up-front costs in private clusters for big data analytics, cloud providers offer unlimited resources according to a 'pay-as-you-go' pricing model (i.e. users only pay for the lease time they already used the resources).

A number of systems and frameworks are being developed to leverage the cloud computing benefits for building and executing big data analytics workflows [17, 38, 33, 66, 53, 18, 19].

MapReduce is a parallel programming framework originated by Google [29] that has became very popular for performing big data analytics. Hadoop is the open source implementation of MapReduce and one of the popular systems that is adapted to execute big data analytics efficiently [3]. Apache YARN is a later version of Hadoop that overcomes its initial performance shortcomings [17]. Apache YARN provides better performance due to its efficient utilization of the resources. Additionally, Apache YARN is designed to support both MapReduce and non-MapReduce jobs on the same cluster.

Usually, MapReduce analytical jobs are formed as Analytical Workflows. Each analytical workflow is a set of MapReduce jobs. An analytic workflow starts with one or more data sources and performs a set of MapReduce jobs on the data. Each analytic workflow job is a MapReduce execution of a set of tasks. These tasks are either transform, compute,
analyze, merge or summarize these data until the needed information are produced during the final stage of the workflow’s execution. Analytic workflows can involve a variety of jobs to achieve the desired results, including data cleaning, exploration, summarization, reporting, visualization, prediction and decision making.

1.3 Data Locality

Data locality refers to the degree to which the data and processing for a computation are co-located on the same physical node. Maximizing data locality is an important goal for many data intensive systems [3, 17] because it can have a significant impact on the performance of the data intensive jobs. The higher data locality, the less data transfer over the network [30, 68]. Most of the previous data locality studies are conducted over Hadoop 1.0.

There are potential performance problems that are related to data locality in YARN, which is the new version of Hadoop. First, changing YARN’s delay scheduler configuration parameters to different values has a clear impact on the performance. Second, users do not have any tool to calculate the potential data locality with different scheduler configuration values. Third, users do not have any insights about the performance before adding physical resources to their clusters. Fourth, fixing the number of replicas in Hadoop Distributed File System (HDFS) does not help to improve the data locality for data-intensive workloads.

1.4 Thesis Statement

The data locality in the modern distributed systems, like YARN, can be better understood and improved by addressing three technical challenges. These challenges are simulating interactions between YARN components in a real cluster to report data locality, estimating the data locality before scaling a real cluster with much physical resources, and applying dynamic replication mechanism to maximize data locality.
1.5 Contributions

Our research makes the following contributions towards the optimization of data analytics:

- A study of the data locality in YARN for different analytic workflows.
- The development of a YARN data locality simulator to support the study of data locality. The tool simulates a YARN cluster and estimate the data locality achieved by a certain workload on a cluster. The simulator supports the following scalability requirements:
  1. Vertical scalability: If the user plans to add more resources (e.g. CPU cores and/or memory) to some or all machines in the cluster, the YARN data locality simulator is able to report the anticipated data locality percentages after this upgrade.
  2. Horizontal scalability: The YARN data locality simulator reports the estimated data locality percentages in case of adding or removing machines to the cluster.
- The development of an online dynamic replication mechanism to improve data locality in YARN that does not require any prior knowledge about the input workload characteristics.

1.6 Thesis Organization

The rest of this dissertation is organized as follows: in Chapter 2, we introduce the background on the new MapReduce framework and we survey related work on data locality management techniques. In Chapter 3, we introduce our data locality study and *YLocSim*, a simulator tool that simulates the interactions between YARN components to report the data locality. In Chapter 4, we extend *YLocSim* to simulate a real cluster after either horizontal
or vertical scaling and report the expected data locality and running time for a given work-
load. In Chapter 5, we propose YDRM, a new component to YARN architecture that applies
online dynamic replication to improve data locality. Chapter 6 summarizes the dissertation
work and proposes a few directions for future work.
Chapter 2

Background and Related Work

2.1 Background

Our focus of our research is MapReduce based processing frameworks since MapReduce as it is the most popular execution framework for big data analytics. We discuss Hadoop, the open source implementation of MapReduce and its related shortcomings. Then, we discuss Apache YARN (Hadoop 2.0) that is designed to overcome Hadoop’s shortcomings. Also, we briefly discuss the Hadoop Distributed File System (HDFS) as it is the main Hadoop file system.

2.1.1 MapReduce Frameworks

Due to its simplicity, MapReduce has recently gained more popularity among data analytics users, business owners and scientists who are interested in processing complex and data intensive jobs in a parallel paradigm. MapReduce is a parallel programming framework which is inspired by Google [29] and it has recently been extended with another tools (like Oozie [38]) to support various disciplines that require data intensive workflows, such as physics, chemistry and the medical and social life sciences [32]. Hadoop is the open source
implementation of MapReduce [3].

A Hadoop job consists of a set of tasks. Each task holds logic written by programmers and it can be either map or reduce task. The Hadoop job tracker distributes and executes the tasks on a cluster of commodity hardware machines by performing two elementary stages: map and reduce stage. In the map stage, the map task reads the input data split, which is stored in a distributed file system like Hadoop Distributed File System (HDFS), and parses them into input key/value pairs. The Hadoop framework applies the user-defined map function logic on each input pair to generate an intermediate set of key/value pairs.

The map tasks can optionally sort and partition the data for different reduce tasks according to a partition function. In the reduce stage, a reduce task retrieves its assigned input split of intermediate key/value pairs from all the map tasks. This process is called the shuffle phase. Afterwards, the reduce task aggregates the intermediate values with the same key and applies the user-defined reduce function logic to the value list with the same key. These results consist of the output of one Hadoop job which are stored in the distributed file system to be possibly used later as an input to another Hadoop job based on the Hadoop workflow specification. Figure 2.1 shows the execution flow of a Hadoop job.

Although Hadoop is popular and simple, it has some limitations. First, a Hadoop cluster is fully dedicated to MapReduce jobs which may cause cluster under utilization. Second, the number of map and reduce slots should be configured in advance during MapReduce cluster setup, which causes cluster under-utilization. For example, if a node is configured to have 5 mapping slots, then no other mappers can run even if there are available resources on this node. Third, Hadoop has scalability problems because each job tracker can only manage up to 4000 nodes.

Apache Hadoop YARN (also known as Hadoop 2.0) was originally designed by Yahoo! in 2008 [16] and later in 2013, YARN was promoted to be a sub-project under Apache Hadoop group [17]. YARN is designed to overcome most of Hadoop’s issues. In YARN, there is
no ‘node slots’ configuration paradigm anymore. Each node has its own resources (CPU, memory) which are allocated to the jobs when requested. Different jobs (either MapReduce or non-MapReduce jobs) can be deployed on a YARN cluster and YARN’s resource manager is responsible for allocating the proper resources based on a job’s requirements using different scheduling techniques.

The main components of YARN and their interactions are shown in figure 2.2. YARN involves three main components namely the Resource Manager, Application Master and Node Manager.

**Resource Manager (RM)** tracks the resource usage across the cluster and also receives heartbeats from the cluster nodes to check their liveness. A Resource Manager is installed per cluster and manages the available resources among the applications and users. The Resource Manager has a global view of the cluster resources and it ensures properties like fairness between the users and data locality among the different applications. Resource manager has two main components: the Applications Manager and the Scheduler.

**Applications Manager (AsM)** is the component of Resource Manager that is responsible for accepting the submitted applications (step 1 in figure 2.2). These applications can be MapReduce or non-MapReduce applications. Applications Manager negotiates with the
Scheduler to obtain the first container for executing the Application Master of a job (step2). The Application Master component is described later. The Applications Manager can restart the Application Master containers in case of failures.

**The Scheduler** is responsible for allocating resources to the running applications based on the resources' availability and other objectives like maximizing fairness and data locality. The Scheduler is not responsible of monitoring the status of the current applications nor does it guarantee restarting the tasks that fail. The Scheduler receives the applications’ computational requirements and it assigns the available resources in terms of a new notion called resource containers. A resource container is introduced by Apache YARN to encapsulate computational resource elements like memory, CPU, network and disk into one entity. Each resource container is bound to one node and each node in the cluster can hold multiple containers. The current version of the YARN Scheduler considers only memory and CPU in a container.

**Application Master (AM)** is responsible for generating the logical resource allocations plan for each individual application and communicates with the Resource Manager to receive the required resources. For example, the Application Master of a MapReduce job computes the resource requirements based on the locations of the input splits to maximize data locality. Then, these resource requirements are translated to a list of resource requests and sent to the Scheduler (step 3). Each resource request includes the preferred resources, the number of containers, the locality preferences (strict or flexible) and the priority of each container within the application. Since the Resource Manager has a global view of the available resources, it replies to the Application Master with a set of containers based on the resources’ availability and the scheduler policy (step 4). The Application Master manages the lifecycle of the job, including increasing and decreasing the number of resource requests and the job execution flow (e.g. running the reducers phase after the mappers phase).

**Node Manager** is a daemon running on every worker machine in the cluster. A Node
Manager is responsible for launching containers for applications once the Resource Manager allocates them to the application (step 5). Each container is wrapped by a Container Launch Context (CLC) which includes a set of environment variables, dependencies stored in remotely accessible storage, security tokens required for authentication and the commands necessary to create the process. A Node Manager also ensures that the allocated containers do not exceed the available resources on its machine before returning them to the Application Master (step 6).

2.1.2 Hadoop Distributed File System (HDFS)

Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware and it is the main distributed file system for Hadoop 1.0 and YARN. HDFS is a fault tolerant system that works under the assumption that the hardware failure is the norm. HDFS detects the faults due to the hardware failure and performs quick recovery efficiently. Since the applications use HDFS for processing large data sets, HDFS supports storing large files in a distributed fashion on the cluster nodes. HDFS provides scaling to thousands of nodes per cluster to accommodate the storage requirements.
Figure 2.3: HDFS Architecture

HDFS architecture follows a master/slave paradigm such that there is a NameNode instance per cluster that acts as a master server that manages a set of DataNodes. A NameNode manages the file system namespace and facilitates accessing the distributed file system by the clients. In HDFS, each file is split into fixed-size blocks which are physically stored on the DataNodes. Figure 2.3 shows the HDFS architecture. A NameNode stores the metadata information of these blocks like the mappings between each file and its blocks and the DataNodes that store each data block. Also, a NameNode manages the file system namespace operations like opening, closing, and renaming files and directories. The DataNodes manage the read and write requests from the clients and perform block creation, deletion and replication based on the instructions received from the NameNode.

Since the NameNode and DataNodes are deployed on commodity machines, HDFS records and stores every transaction into a log file called EditLog for fault tolerance. The NameNode stores the file system namespace and the file-block mappings into a file called Fsimage. When NameNode starts up, it loads EditLog and Fsimage from the disk into the memory and executes all the transactions from EditLog to the in-memory representation of the Fsimage. Afterwards, the NameNode saves the new version of the Fsimage file on the disk.
Regarding data replication, the NameNode replicates each data block on multiple machines to prevent the failure of one machine causing the loss of all copies of the block. One naive replica selection policy is to replicate the data blocks on different racks. However, this policy is not fully optimized when writing data as each write operation requires transferring blocks to multiple racks. Also, the communication between two nodes in different racks must go through switches and usually network bandwidth between two machines on different racks is less than the bandwidth between two machines in the same rack. The current HDFS replica placement policy is called Hadoop Rack Awareness which balances between placing the blocks on the same rack and different racks. For example, if the replication factor is three, then HDFS places one replica in a local rack, the second on a node in different remote rack and the third one on a different node on the same remote rack. Figure 4.3 shows an example of the HDFS replica placement policy.

2.2 Related Work

In this chapter, we survey the following:

1. Previous studies on Hadoop performance.

2. The popular analytical workflow systems.

3. The different analytical workflow optimization techniques and methods. In this part, we focus more on the data locality.

2.2.1 Previous Studies on Hadoop Performance

Jiang et al. study the design factors that hurt MapReduce performance [39]. They conclude that the running time could be affected by the number of tasks to be scheduled, the scheduling algorithm, the I/O mode (either streaming or direct) and the method to convert the raw data into key/value format (known as data parsing).

A study of network and disk performance in data centers by Ananthanarayanan [23] observes that network performance is improving faster than disk performance so that data locality may become irrelevant. However, Abad et al. show that there is a significant difference between disk and network bandwidths and reading data locally from the disk drives is still better than transferring it over network in practice [20].

A recent study compares the performance of different systems that extend Hadoop [40]. The study shows that there are many design factors that may affect the overall system performance like indexing, data compression, data locality, fault tolerance and scalability.

From the industrial perspective, Chen et al. study the MapReduce-based systems from different organizations like Facebook, Cloudera and other retail and telecommunication organizations [27]. The study shows that there is a significant variation in Hadoop performance behavior between different workloads due to the skewness of the data, the size of the jobs and the amount of the generated data. According to these variations, the study discusses the
main challenges towards building a big data benchmark that would be a representative to measure Hadoop performance. This study includes uniformity/skewness of the data, what are the common use of query-like systems (like Hive) and a performance comparison between the MapReduce systems.

2.2.2 Analytical Workflow Systems

In this section, we discuss a number of popular systems that have been adopted for executing analytic workflows like Pegasus, ASKALON and Oozie. Most of these systems assume that the input analytic workflow can be represented as a directed acyclic graph (DAG) such that each node in the graph represents a workflow job and each edge represents data dependency between two workflow jobs. The input data files and their locations should be well defined before the workflow execution. Also we discuss systems that model analytic workflows as SQL data flows or relation algebra expressions to gain the benefits of relational algebra optimization techniques.

2.2.2.1 Pegasus

Pegasus is an analytic workflow system that is used to process and execute scientific workflows [31]. The goal of Pegasus is to find the optimal mapping of workflow tasks to the distributed resources that minimizes the overall workflow execution time. Pegasus uses abstract workflows that are constructed using different tools like Wings [35] and Composition Analysis Tool (CAT) [46, 45]. Pegasus was originally deployed on grid environments and recently it can be deployed on cloud environments [19]. Pegasus collects fundamental information dynamically from the execution environment like the characteristics of the available resources (e.g. the job queue length, the load of each resource, . . . ), the location of the data files and their replication information and the availability of the software libraries that are
required to execute the workflow tasks. The input workflow to Pegasus is an abstract workflow represented as a directed acyclic graph (DAG). The abstract workflow simplifies the workflow representation by hiding the physical resource mapping complexity. Also the abstract workflow provides flexibility in that individual workflow components can be replaced with different implementations.

2.2.2.2 ASKALON

ASKALON is another system for executing workflows using grid and cloud resources [64, 18]. Like Pegasus, ASKALON aims to improve the workflow execution running time but additionally, ASKALON provides a solution for handling unbalanced workflows. ASKALON assumes that there are two possible scenarios. The first one is that there is not any prediction about workflow tasks’ execution time provided to the scheduler. In this case, ASKALON assumes that the execution time for all workflow tasks is the same. The second scenario is that the prediction of workflow tasks execution times are provided to the scheduler. The experimental results show better results for the latter scenario using different scheduling algorithms.

2.2.2.3 Oozie

Oozie is a workflow scheduler that designs, coordinates and manages workflow jobs [38]. Since Oozie can be installed on top of YARN, Oozie coordinates different types of workflow jobs (e.g. MapReduce, Java programs, shell scripts . . . ). The Oozie server accepts a specification of workflow DAGs that are submitted by multiple Oozie clients. Then, Oozie splits these workflow DAGs into sub-tasks and submits each sub-task to a YARN cluster for processing. Oozie isolates the processing of each user and lets them have the privilege to access only their authorized resources. Oozie is highly scalable and has been utilized by Yahoo! for execution of more than 770,000 workflows executing around 2 Million Hadoop jobs on 16
2.3 Analytical Workflow Optimization

In this section, we survey the different methods to optimize analytical workflow execution using public cloud resources. These optimization techniques can be categorized to provisioning methods, avoiding redundant processing and maximizing data locality.

2.3.1 Provisioning

Since public cloud providers give a large number of resources with different types to their users, choosing the best combination of cloud services to perform data analytics under user defined SLAs becomes more challenging. Moreover, managing the trade-off between different user objectives (e.g. minimizing monetary cost along with minimizing running time) is not a straightforward task.

Kllapi et al. propose a technique for optimizing the completion time and the monetary cost given a fixed budget [47]. This work shows that there is a clear trade-off between the completion time and the monetary cost for a given data flow to be executed. The proposed greedy scheduling algorithm explores the completion time and monetary cost space and finds the optimum assignment of cloud resources that satisfies the user’s budget. Also this scheduling algorithm estimates how long every operator in the data flow will run, consequently, the completion time and monetary cost can be estimated for the whole schedule.

Provisioning MapReduce applications in public clouds is a challenging problem. RS (Resource Set) Maximizer is designed to provision MapReduce applications to minimize the different Hadoop clusters for 50 different users.
monetary cost, while achieving better performance based on choosing the best Hadoop configuration parameters [41]. The argument is that the default Hadoop cluster configurations (e.g. the number of mappers and reducers) does not fit all the workloads. The optimization is performed in two stages. The first stage is to build a database that stores signatures of the optimum configurations for different workloads. In the second stage, given a new Hadoop workload, RS Maximizer creates a new signature for it, then this signature is used to lookup in the signatures database to find the optimum Hadoop configurations. However, building a signature database that covers a wide variety of applications and configurations (more than 150 configuration values) may not feasible. Also, RS Maximizer assumes that there is a fixed number of cloud machines in the resource pool.

Mao and Humphrey propose two auto-scaling algorithms to minimize the job turnaround time (the time elapsed from job submission to the job completion) within budget constraints for executing workflows using cloud resources [51]. The scheduling-first algorithm, distributes the application-wide budget to each individual job based on its priority. For instance, the scheduling-first algorithm assigns more budget for high priority jobs in the workflow. Then, it determines the fastest execution plan and accordingly acquires the required cloud resources. The scaling-first algorithm determines the size and the type of the cloud resources first based on the budget constraint and then schedules the workflow jobs on the acquired cloud resources.

Mian et al. present a formulation of the provisioning problem and they design a framework that predicts the cost of executing a data-intensive workload given a set of configurations [52]. This approach begins with exploring the space of all possible configurations based on the predicted monetary cost of each configuration. The search space is described as a
Directed Acyclic Graph (DAG) where each node in this graph represents a unique configuration and each edge represent a possible movement (e.g. add the cheapest VM, add the same VM, upgrade ...) from one configuration to another. Then, a greedy search algorithm is developed to traverse this graph to find the optimal configuration.

Conductor is a system that orchestrates execution of MapReduce jobs on the cloud by choosing the most suitable cloud services to use according to user defined goals (e.g. minimizing the monetary cost or minimizing the execution time) [65]. Conductor provides an abstraction layer to separate the heterogeneous cloud services. For example, most of cloud providers combine the storage and computation services under the same interface, which complicates the deployment plan in case of finding a better storage or computation service. Conductor supports hybrid deployment plans when the users want to make use of their own local hardware infrastructure combined with other services that are provided by public cloud providers. Also Conductor considers the dynamic pricing for the cloud services, which can vary periodically over time by the cloud providers. For example, the prices of Amazon EC2 spot instances vary frequently (e.g. every 10 minutes) [2], consequently, predicting the market prices is quite challenging.

Purlieus provisions virtual MapReduce clusters in a locality-aware manner to provide access of MapReduce virtual machines (VMs) to input data and importantly, intermediate data from local or near by physical machines [54]. It proposes an algorithm to minimize the network distance between storage nodes and compute nodes for the map and reduce phases. For the map phase, the VM that is executing the map task should be as close to the node that stores the input data blocks as possible. For the reduce phase, the VM executing a reduce task should be close to the map-task VMs, such that the generated intermediate data are close from the reduce task. In contrast, the MapReduce framework places the data
independently of the map and reduce tasks and only the map task locality is considered by MapReduce scheduler. Purlieus manages the data placement and VM placement problems. For the data placement problem, it defines the set of physical machines that should store the input datasets. Afterwards, it selects the set of VMs that should be provisioned to execute the MapReduce jobs and changes the locality-awareness approach based on the job specification. For example, it is better to distribute the data blocks across the network to utilize all the physical machines for the map-input heavy jobs (i.e. the map tasks generate small intermediate data) than distribute the data blocks on a subset of the physical machines.

Simitsis et al. show that there are many trade-offs between different quality metrics [59]. For example, adding recovery points to improve recoverability requires paying additional I/O cost which may hurt the performance and affect negatively the execution running time metric. The qualitative metrics (soft goals) like reliability, and other quantitative measures like CPU utilization and execution time can be represented as an interdependency graph to show the different trade-offs between these goals (or metrics). Such an interdependency graph is useful to demonstrate the trade-off to the business users and it can be used to derive the optimization search space.

Extending the latter work, the authors propose optimization techniques that can be applied on the logical and physical levels while executing data flows using different execution engines (e.g. MapReduce, ETL and RDBMS) [60]. For example, on the logical level, the optimizer should choose the most selective join first in the flow. On the physical level, the optimizer uses joins between tables that are stored on the same RDBMS instead of using the ETL engine. Adding a recovery point after expensive operations (e.g. sentiment analysis and surrogate key generation) to provide recoverability is another example. On the physical level, the optimizer examines if the data generated have small size, then it would be better
to migrate it to a single node to do further computation rather than storing the data on a
MapReduce engine which has more overhead.

2.3.2 Redundant Computations

Redundant processing is one of the strategies to manage the straggler nodes, which are the
executing nodes that have poor performance due to hardware failures or misconfiguration.
For example, MapReduce runs a speculative copy (backup) of a slow task on another node
to finish the computation faster [70]. MapReduce starts the speculative tasks based on a
heuristic comparison of each task’s progress and the average processing time. However, this
heuristic estimation works under the assumption that all the nodes are homogeneous and the
tasks progress at a constant rate throughout time, which is not always true. Zaharia et al
propose an alternative approach for speculative execution called Longest Approximate Time
to End (LATE) algorithm [70]. LATE computes the estimated time remaining of a task
instead of calculating the task’s progress, which gives an accurate estimation of a stragglng
task and its impact on the job completion time.

MapReduce Garbage Collector is designed to collect slow tasks in Hadoop [44]. MapRe-
duce Garbage Collector shows that terminating the slow tasks before a certain job’s progress
threshold (e.g. 40%) can save time and cost. Otherwise, it is more efficient with respect to
time and cost to keep these slow tasks running till the job completes.

In big data analytics, it is often that multiple analytic jobs need to perform analytic
processing on the same data. This scenario typically happens if these jobs are executed in-
dependently which results in redundant processing. For instance, different queries that need
to be processed on the same database tables to answer different analytic questions. Pega-
sus follows a workflow reduction technique to modify (or reduce) the structure of the input
abstract workflow [31]. This done by consulting the Replication Location Service about the availability of the intermediate data. If it exists, then Pegasus reduces the abstract workflow by eliminating the redundant tasks that are used to generate these intermediate data.

MapReduce does not support caching the intermediate and output results for future use instead of recomputing these data again. ReStore is a system that materializes the intermediate results that are generated by MapReduce jobs that could be reused by future jobs [33]. However, ReStore affects the job’s completion time as it stores the intermediate results at each stage. Also ReStore requires quite large disk space to store the output results.

MRShare identifies the sharing opportunities between different queries (jobs) that perform the exact work to minimize the I/O operations [53]. The sharing opportunities are determined in terms of sharing scans of the input data and sharing the map output by tagging each map output key with its job id to distinguish its origin job. Then, MRShare reconstructs the jobs into groups such that each group can be executed as one job.

2.3.3 Modeling Analytic Workflows as SQL Data Flows

Modeling analytic workflows as SQL data flows or relation algebra expressions is a good way to borrow relation algebra optimization techniques. For example, moving the most restrictive algebra operator close to the beginning of a workflow could reduce the amount of data generated and consequently the transfer time is reduced. However, expressing analytic workflows as SQL data flows or relational algebra expressions is not an easy task. An analytical workflow in the SQL consists of a graph of jobs. Each job consists of a set of tasks that execute a set of SQL operations (e.g. selection and projection). Analytic workflow system should be aware of each workflow job’s estimated execution time which is not accurate in most cases. Also, estimating a workflow job’s execution time is difficult to
be calculated in dynamic environments [58, 59]. Additionally, in many cases, users define quality of service (QoS) constraints for analytic workflows. Setting execution time deadlines or monetary budgets are examples of these constraints. However, there are many trade-offs among quality of service metrics [59, 72, 71, 47]. For example, to satisfy a recoverability constraint, a workflow system should add recovery points after executing expensive workflow jobs which requires performing additional I/O operations. These additional I/O operations may increase the execution time and violate execution time constraints set by the user. Trade-offs between the monetary cost of cloud execution and the workflow completion time is another problem that arises when using cloud resources for analytic workflow execution. For example, the user has an option to execute a workflow faster at increased cost, or slower at a reduced cost.

2.4 Data Locality Management

Data locality is the property of a job that refers to the degree to which the data and processing for a computation are co-located on the same physical node. It has a significant impact on the performance of jobs in a large cluster since higher data locality means there is less data transfer over the network. For data-intensive jobs with large data sets the network has been shown to be a potential bottleneck [30, 68]. Data locality is a design goal of many systems [31, 56, 30, 62], however, problems may arise while maximizing data locality. For example, some nodes that store the input data may have more contention than the others. Potentially, these nodes could become hotspots in the cluster and consequently the other nodes become under-utilized.

Figure 2.5 categorizes the approaches to data locality management in the literature. The data locality management could target either disk locality or memory locality. The systems that are designed for disk locality try to co-locate the data files and the processing on the same machine. Most of the disk locality management systems assume that each data item
is replicated over a set of machines in the cluster. Some systems apply static replication where the number of replicas per data item does not change along the workload execution. However, static replication may lead to poor data locality. Therefore, dynamic replication methods have been proposed to overcome the poor data locality and they can be executed either offline or online. In our work, we focus on the dynamic disk locality approaches.

2.4.1 Disk Locality

2.4.1.1 Static Replication

Pegasus is a workflow management system that uses a Replica Location Service (RLS) to achieve data locality [31]. RLS is a distributed replica management system that stores information about logical to physical data file name mappings and the available distributed indexes. Pegasus scheduling algorithms query RLS to retrieve data replica locations for the incoming tasks.

Ranganathan and Foster found that scheduling algorithms that target only processor utilization by mapping jobs to idle processors without regard the cost of retrieving the data from a remote site are inefficient [56]. They propose a decoupled scheduling framework for data intensive applications that separates the job scheduling policy and the replication policy. This framework consists of three components: the External Scheduler (ES) that
decides to which node that the jobs must be submitted, a Local Scheduler (LS) that is installed on each node that decides the priority of the jobs arriving at this node and the Dataset Scheduler (DS) that monitors the popularity of the dataset items and decides which dataset items to replicate or delete. In simulation experiments, the ES schedules jobs to either the least utilized site or the site where the data is stored. The DS either does no replication or randomly replicates a file or replicates a file at the least loaded site among its neighbours. The results of this work conclude that scheduling a job to a machine where the data is available results in better response time than an scheduling a job that fetches the data remotely. Also, the proposed technique causes a few sites which hold the data to become overloaded and in this case, dynamic replication should be applied.

Energy-Efficient Adaptive File Replication System (EAFR) proposes a replication scheme that minimizes the energy required to run the cluster [49]. EAFR selects a server with acceptable hardware capabilities (including network bandwidth and capacity) to hold a replica. EAFR is designed to work in a heterogeneous cluster environment, which has servers with heterogeneous network bandwidth, hardware configuration and capability (i.e., the maximum number of service requests that can be fulfilled by a node in parallel).

Prior study has noted that there is a conflict between fairness in scheduling the resources and data locality (placing tasks on nodes that contain their input data) [68]. Considering an exact implementation of fair sharing compromises data locality, because the job to be scheduled according to the fairness policy might not have data on the currently available nodes. Zaharia et al. propose a delay scheduling algorithm that relaxes the fairness constraints such that a job waits for a limited amount of time for a better scheduling opportunity on a node that holds its input data. In chapter 3, we discuss the delay scheduler works in more detail.

Zhenhua et al. formulate the MapReduce data locality problem as a mathematical model, that is used to find the optimal scheduling that maximizes the data locality [36]. This paper shows that scheduling multiple tasks all at once outperforms the delay scheduling approach,
where the scheduling is performed task by task. Delay scheduling assigns the tasks one by one without considering the impact of this assignment on the other tasks. To reach the global minimum of data transfer over the network, a scheduling approach should calculate the cost of each assignment and the its impact of the other tasks.

2.4.1.2 Dynamic Replication

Since the data access patterns on distributed files systems (e.g. HDFS) vary frequently, fixing the number of replicas per data item may lead to poor data locality. Data access patterns can be classified into three categories: popular data that have a large number of concurrent accesses; unpopular data that have a limited number of concurrent accesses; normal data that are the rest of the data.

**Offline Dynamic Replication**  Scarlett is an off-line system that replicates blocks based on the observed probability from the Hadoop job history logs in a previous unit of time (e.g. days or weeks) [22]. Scarlett uses a sample of the historical statistics from running systems and tries to predict the files’ popularity. Scarlett allocates the available disk space budget to the popular files using two main approaches. The first one is called the *priority* approach where Scarlett sorts the popular files according to their sizes and replicates them one by one until it runs out of the disk budget. The intuition behind the *priority* approach is that the files with a large size are accessed more often compared to the small files. However, the *priority* approach distributes the disk budget over a small number of files. Alternatively, the *round-robin* approach increases the replication factor of each file by at most one in each iteration and continue to iterate over the file list until the budget runs out. This approach distributes the budget as many files instead of allocating all the disk space budget to a small number of large files. Regarding the replica placement, Scarlett distributes the blocks of each popular file over as many racks as possible to ensure spreading the load uniformly across all the machines and racks. During the replication process, Scarlett tracks the load of each rack.
and each machine to control the placement of each block. For de-replication, Scarlett deletes the blocks of the unpopular files in a lazy manner by overwriting them when another block needs to be written on the disk.

A Cost-effective Dynamic Replication management Scheme for Cloud Storage Cluster (CDRM) is an offline replication placement scheme for Hadoop that calculates the ideal number of replicas per data block to satisfy the availability requirements [63]. CDRM introduces a model that determines the minimal number of replicas that should be set in the system to satisfy availability requirements. In this model, CDRM explores the trade-off between the replication factor and the availability with different node failure probabilities to find the optimum replication factor. CDRM’s replication policy is based on placing the replica onto the data node with the lowest blocking probability to maintain load balancing. However, CDRM does not increase the number of replicas dynamically during the running time.

Elastic Replication Management System (ERMS) is an offline replication management system for HDFS [28]. ERMS categorizes the data blocks to three categories: hot data which are the data that have a high intensity of access, cold data blocks which are the unpopular data with low number of access requests and normal data which are the rest of the data. ERMS parses the HDFS logs and extracts the events that have been executed over the data blocks. Then, ERMS takes the advantage of Condor, a high performance complex event processing system [34], to classify the data blocks according to the aforementioned categories. ERMS introduces an active/standby storage system. The standby nodes are used to store the extra replicas of hot data. When the active nodes are heavily used, ERMS activates the standby nodes to serve and load balance the requests. ERMS introduces a “Data Judge” component to determine the proper replication factor. When the hot data loses popularity, ERMS shuts down the standby nodes for energy savings. Table 2.1 summarizes the comparison between the aforementioned systems.
Bui et al. propose ARM, an adaptive replication management system by building a supervised machine learning algorithm to predict the popular blocks [26]. ARM works in two main phases. The first phase collects training data (in an offline fashion) from the HDFS heartbeats. ARM uses the training data to build a prediction model that is used for applying the dynamic replication. The second phase uses the prediction model to detect the similarity in the access pattern of the new files under the assumption that two files with similar access behaviors should be treated with the same replication strategy.

**Online Dynamic Replication** Adaptive Data Replication for Efficient Cluster Scheduling (DARE) is an adaptive data replication mechanism for HDFS [20]. DARE assumes that any non-local data access originated by a remote map task is worth replicating with a certain probability value without additional network cost. DARE helps the scheduler to achieve better data locality by replicating data blocks into remote machines using a disk space budget. DARE is scheduler independent and can work with any Hadoop scheduler (e.g. Delay scheduler or FIFO scheduler) to improve the data locality. DARE follows a greedy approach to assign the disk space budget to the replicas. The greedy approach assumes that any block that is requested remotely from a remote map task should be replicated. However, this approach leads to poor locality when unpopular blocks are replicated. Hence, DARE adopts a randomized approach such that a coin is tossed to decide if the remote block should be replicated or not. This approach helps to decrease replicating the unpopular blocks, however, it may miss replicating popular blocks as well. The approach requires a careful adjustment of the probability threshold.

When the disk space budget runs out, DARE iterates over the dynamically replicated blocks and reduces their access count by half each time. Then, another thread runs over the dynamically replicated blocks to delete all the data blocks that have access count less than a pre-defined threshold.

Although these systems are designed to maximize the data locality, it is worth mentioning
<table>
<thead>
<tr>
<th>Replication dynamicity</th>
<th>Scarlett</th>
<th>DARE</th>
<th>CDRM</th>
<th>RRMN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline</td>
<td>Offline</td>
<td>Offline</td>
<td>Offline</td>
<td>Offline</td>
</tr>
<tr>
<td>Replication action</td>
<td>Proactive</td>
<td>Reactive</td>
<td>Proactive</td>
<td>Proactive</td>
</tr>
<tr>
<td>Replication granularity</td>
<td>File</td>
<td>Block</td>
<td>File</td>
<td>Block</td>
</tr>
<tr>
<td>Priority to large files</td>
<td>Greedy approach</td>
<td>Replication factor and availability model</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Rack-aware replica allocation</td>
<td>Random/Probabilistic approach</td>
<td>Placing replicas on the least blocking popularity machines</td>
<td>Active/Standby storage model</td>
<td></td>
</tr>
<tr>
<td>Replica allocation</td>
<td>Uniform placement over all the racks and machines</td>
<td>Random/Probabilistic approach</td>
<td>Placing replicas on the least blocking popularity machines</td>
<td>Active/Standby storage model</td>
</tr>
<tr>
<td>De-Replication</td>
<td>Lazy deletion</td>
<td>Lazy deletion</td>
<td>N/A</td>
<td>Shutting down stand-by nodes</td>
</tr>
<tr>
<td>Type of jobs</td>
<td>MapReduce, Dryad</td>
<td>MapReduce</td>
<td>MapReduce</td>
<td>MapReduce</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison between different dynamic replication management systems

that none of them makes use of the new Hadoop 2.0 (a.k.a YARN) architecture to achieve a better dynamic replication mechanism. These systems are developed to work efficiently with Hadoop 1.0 applications only, which are not applicable for non MapReduce applications.

2.4.2 Memory Locality

In-memory caching takes the advantage of large memories machines in the cluster as a way to speed up the big data analytic jobs.

PACMan is a caching manager framework that coordinates the access requests to the managed distributed caches for waves of MapReduce jobs [24]. This coordination provides a global view to determine which data item should be evicted from the cache, as well as to decide how to place the tasks so that they get memory locality.

Main Memory MapReduce ($M^3$), extends Hadoop to support streaming applications. $M^3$ performs data processing through main memory [21]. To avoid multiple read and write operations from/to the local disks and HDFS, $M^3$ adapts mappers to send the intermediate data through direct communication using Java Remote Method Invocation (RMI) which is a memory based communication protocol.

Shark [66] uses a main memory abstraction called resilient distributed dataset (RDD) that is introduced by Spark [69]. Shark makes use of Spark’s RDD and Hive query processing to improve the performance mainly by caching the queries’ intermediate data in memory. In this way, Shark reduces the read and write operations on disks and HDFS. The size of available main memory is the main restriction of Shark, especially if map/reduce tasks
produce large output data that does fit in the main memory.
Chapter 3

A Study of Data Locality in YARN

Co-locating the computation as close as possible to the data is an important consideration in the current data intensive systems. This is known as the data locality problem. In this chapter, we analyze the data locality provided by YARN, which is the new version of Hadoop. We investigate the data locality offered by the YARN delay scheduler for a variety of workloads and configurations. We specifically examine three issues related to data locality. First, we study the trade-off between the data locality and the job completion time. Second, we observe that there is an imbalance of resource allocation when considering the data locality, which may under-utilize the cluster. Third, we look at the redundant I/O operations when different YARN containers request input data blocks on the same node.

We propose YARN Locality Simulator (\emph{YLocSim}), a tool that simulates the interactions between YARN components in a real cluster and reports the data locality percentages in real time. We validate \emph{YLocSim} over a real cluster setup and use it in our study.

3.1 YARN Delay Scheduler

In this section, we briefly discuss the delay scheduler and its relevant parameters. The delay scheduler is considered the only data locality aware scheduler among the current Hadoop
and YARN schedulers [68, 62]. The delay scheduler recognizes three locality levels:

1) **Node Locality**: Node locality is considered the most efficient locality category where the delay scheduler decides to schedule the incoming task on a node that stores the input data block. In this case, there is no need for remote data retrieval.

2) **Rack Locality**: If node locality is not possible because none of the local nodes have sufficient resources available, the delay scheduler waits $T_1$ seconds hoping that one of the local nodes will be available. If not, the delay scheduler assigns the incoming task to a node on the same rack as one of the local nodes.

3) **Off-Switch Locality**: In the worst case, if the rack locality is not available, the delay scheduler waits a further $T_2$ seconds, and if there is no node on the same rack as a local node available, the delay scheduler assigns the incoming task to an off-switch node which is located on a different rack to avoid task starvation. Off-switch locality is the most expensive decision for the delay scheduler.

### 3.2 YARN Locality Simulator (YLocSim)

Studying the behavior of scheduling algorithms in YARN is both complex and time-consuming. For example, we find that changing delay scheduler parameters (i.e. $T_1$ and $T_2$) on a real cluster is a time consuming process and may increase the downtime of the cluster while restarting YARN services. In addition, the set of experiments one is able to perform is limited by the available physical resources in the cluster. We therefore chose to develop the YARN Location Simulator (*YLocSim*) as a tool to facilitate the study of data locality in YARN clusters. *YLocSim* will also be a benefit to researchers and developers as a tool to evaluate the impact of new features and scheduling algorithms on the data locality provided by YARN.
3.2.1 YARN Scheduler Load Simulator (SLS) Overview

As YLocSim is built by extending the features of Apache YARN Scheduler Load Simulator (SLS) [15], we first describe the goals and the main features of SLS. SLS is a tool that simulates large-scale YARN clusters and application loads in a single machine such that every YARN component is represented as a daemon thread in the memory within a single Java Virtual Machine (JVM). SLS emulates the real YARN Resource Manager and it exercises all the communication heartbeat messages between Application Masters and Node Managers by dispatching heartbeat events within the same JVM.

The network topology and application information like the task start/end time, the type of the task (e.g. map or reduce task in case of MapReduce applications), the containers that are used for executing the task, the user who submitted the application and the queue information are given as inputs to SLS. The application information can be acquired using a tool called Apache Rumen[6], which is a data extraction and analysis tool that mines Hadoop job history logs to extract meaningful data and digest it into a compact format. SLS provides real time statistics for the simulated cluster like the resources usage (e.g. memory and CPU cores), the scheduler handling cost, the number of running containers and the resource shares for each queue.

However, SLS does not measure the data locality. Data locality monitoring helps the users to determine potential data hotspots as well as the number of data blocks that are transferred over the network during execution.

3.2.2 YLocSim Architecture

Figure 3.1 shows the interactions between the main components of YLocSim. First, Hadoop job history files are consumed by Apache Rumen to generate JSON files that contain the workload traces and cluster topology. Second, the JSON files are integrated with any configuration information related to the YARN scheduler algorithm and input to YLocSim.
Finally, *YLocSim* reports in real time the current data locality percentages to the user.

We modified components of Rumen and YARN Scheduler Load Simulator (SLS) in order to support data locality reporting. First, we modified Rumen to extract the data locality preferences for each map task in the input workload. This requires processing of additional logs to bind the input data blocks with the containers requests. These logs are generated by each container and stored on the job history server. Then, we use HDFS `fsck` utility to extract the physical locations of each input data block. \(^1\) Secondly, we adapted the SLS Application Master to request the needed resources based on the data locality preferences to match the behavior of the YARN MapReduce Application Master. *YLocSim* uses the container simulation process and the heartbeat dispatching events from SLS to report the current data locality percentages of the containers that have been allocated so far.

### 3.2.3 YLocSim Validation

#### 3.2.3.1 Setup

We use a private cluster with 16 machines for validating *YLocSim*. Table 3.1 summarizes the configuration of the cluster. Because the private cluster contains only one physical rack, we simulate the network bisection bandwidth to give an estimate of the running time for

\[^1\]https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html#fsck
Number of nodes: 17 nodes (1 master node and 16 slave nodes)
Number of racks: 1 physical rack, simulated as 5 logical racks where rack0 contains the master node and each other rack contains 4 slave nodes
CPU Cores: 4 CPU cores for the master node, 2 CPU cores for each slave node
Memory: 16 GB for the master node, 4 GB for each slave node
Operating System: Ubuntu Server 12.04 LTS - 64 bits
Network: 1 Gigabit Ethernet
Hadoop Version: Hadoop 2.3.0
Data replication factor: 2
Data block size: 64 MB

Table 3.1: Private cluster setup for validating YLocSim

off-switch tasks. Given the fact that the bandwidth between two nodes within the same rack is much higher than the bandwidth between two nodes on different racks [68], we use the following formula to estimate the required time (delay time) that we should delay each off-switch task compared to the rack tasks:

\[ \text{delay time} = \left( \frac{b}{\text{bisection bw}} \right) - \left( \frac{b}{\text{rack bw}} \right) \]  

(3.1)

where \( b \) is the block size, \( \text{bisection bw} \) is the bandwidth between the aggregated switches and \( \text{rack bw} \) is the bandwidth between two nodes on the same rack.

For example, suppose that the bandwidth between two nodes on the same rack is 1024 Megabits per second and the network bisection bandwidth is 512 Megabits per second and the data block size is 64 Megabytes. We need to suspend the off-switch task by \( \left( \frac{64 \times 8}{512} \right) - \left( \frac{64 \times 8}{1024} \right) = 0.5 \) seconds. We assume that the network bisection bandwidth is 512 Megabits per second (50% of the rack bandwidth) in our experiments based on estimates provided by Hadoop [10]. Clearly, this estimation method is used only to simulate the running time, not the network traffic.
3.2.3.2 Benchmarks

Synthetic benchmarks should consider workloads that have variety of characteristics like having short/long and sequential/parallel jobs. We generate 8 GB synthetic text files and we use `grep` [8] (extracts matching strings from text files and counts how many time they occurred), `mrbench` which generates small Hadoop jobs based on random text to test the cluster efficiency and `wordcount` which counts the occurrence of each word in the input text files. These benchmarks are commonly used for Hadoop performance evaluation.

A Hadoop workflow defines the logical sequence and dependencies between related Hadoop jobs. We select TPC-H queries [13] to serve as examples of Hadoop workflows. TPC-H is a decision support benchmark. It includes decision support queries that examine large volumes of data to answer several business questions. We select a subset of the queries that are complex enough to evaluate the data locality, namely $Q_7, Q_8, Q_9, Q_{18}$ and $Q_{21}$.

First, we use TPC-H tools 3 to generate the 10 gigabyte TPC-H database on MySQL [9], and migrate the data from MySQL to Hive [4] using Apache Sqoop [7]. Hive stores the database in Hadoop Distributed File System (HDFS). We then convert the TPC-H SQL queries to Pig queries [5, 11] that are executable over Hive. Figure 3.2 shows the migration process. Each Pig query can be represented as a Directed Acyclic Graph (DAG) such that each node represents a Hadoop job and each edge represents a data dependency between two Hadoop jobs. Figure 3.3 shows DAG examples of three Pig queries ($Q_7, Q_8$ and $Q_{21}$). We list the TPC-H schema, TPC-H SQL queries and the Pig queries in the appendix.

Data locality for the map tasks in the Hadoop jobs is the main validation measure for YLocSim. We consider only map locality as the locations of the map data input blocks are well-defined in HDFS. Calculating the locality of the reduce phase is quite complicated as the data input of each reduce task is dynamically generated from scattered nodes where the map tasks ran. Also, the locations of the input data blocks of reduce tasks are not captured.

---

in the log traces. We calculate the locality percentages for each locality level based on the number of allocated containers divided by the total number of allocated containers. For example, we use the following equation to calculate the node locality percentage:

\[ \text{node\_locality\_percentage} = \frac{\text{node\_containers}}{\text{total\_containers}} \times 100\% \]  

(3.2)

where \text{node\_containers} is the number of containers that the YARN delay scheduler allocated on nodes that stores the input data blocks. \text{total\_containers} is the total number of containers that have been allocated so far at all locality levels. We calculate \text{rack\_locality\_percentage} and \text{offswitch\_locality\_percentage} in a similar manner.

3.2.3.3 \textit{YLocSim} Validation

Figures 3.7, 3.8 and 3.9 show the locality percentage of each locality level achieved in YLocSim and the real cluster for each of the synthetic benchmark. Figures 3.4, 3.5 and 3.6 show the locality percentage of each locality level reported by \textit{YLocSim} and the real cluster using the Hadoop workflows. We run each job or workflow five times on both \textit{YLocSim} and the real cluster and show the average. The average locality percentage is quite a good representation
Figure 3.3: Pig queries as DAGs

of the results as the standard deviation value is small (from 0 to 0.72). We observe an error between $YLocSim$ and real cluster of at most $\pm 3.3\%$ for all cases.

Figure 3.4: Node Locality Percentages
An explanation for the error is that the YARN scheduler assigns a container for the application master on a random node in case that all the nodes are available. Since in each run of \textit{YLocSim}, the application master’s container can be placed on a different node and this container runs until the application finishes, the locality percentages could be slightly
changed for each run. We tried to force YARN scheduler to allocate an application master container on the same node as in the workload log traces, but unfortunately, there is an open bug that prevents YARN from allocating containers on particular nodes[1, 14]. Additionally, we observe that the delay scheduler is efficiently minimizing the bisection network transfer in general. Figures 3.9 and 3.6 show that off-switch locality percentages are less than 4.5% on average.

Figure 3.7: Node Locality Percentages
Figure 3.8: Rack Locality Percentages

Figure 3.9: Off-Switch Locality Percentages
3.3 YARN Data Locality Analysis

In this section, we present our observations about the impact of YARN data locality and highlight potential data locality problems with the YARN delay scheduler by performing three experiments. The goal of the first experiment is to examine the trade-off between the data locality and the running time. We perform the second experiment to identify the imbalance of containers allocation when some nodes hold more input data blocks than the other nodes. The third experiment identifies the redundant I/O operations when different containers request the same the input data blocks on the same node.

3.3.1 Data locality and running time trade-off

We observe there is a clear trade-off between the data locality and the running time when we change the values of the delay scheduler’s $T_1$ and $T_2$ parameters. Figures 3.10 and 3.11 show the impact on data locality when varying $T_1$ and $T_2$ values using YLocSim. For example, figure 3.11 shows that changing $T_1$ from 5 seconds to 10 seconds and $T_2$ from 10 seconds to 15 seconds improves the node locality percentage of Q18 from 65% to 72%. This means setting $T_1 = 10$ seconds and $T_2 = 15$ seconds saves approximately 7% of the data being transferred over the network compared to setting $T_1 = 5$ seconds and $T_2 = 10$ seconds for the same query. However, figure 3.10 shows that the node task percentage does not have evident improvement for Q9 (less than 3%) when changing $T_1$ from 15 to 20 seconds and $T_2$ from 20 to 25 seconds.

Figure 3.12 shows the running time of each query with different $T_1$ and $T_2$ values. Clearly, the running time increases as $T_1$ and $T_2$ values are increased because the delay scheduler delays the non-local tasks to maximize the data locality. The more $T_1$ and $T_2$ values, the more delay to the remote resource requests. Consequently, varying $T_1$ and $T_2$ affects the running time. For example, changing $T1$ from 5 seconds to 20 seconds and $T2$ from 10 seconds to 25 seconds for Q7 causes running time increases from 305 to 494 seconds, but at the same time,
figure 3.10 shows that the node locality percentage of $Q_7$ increases from 65% to 85% and consequently the rack locality percentage decreases from 31% to 15% and off-switch locality percentage drops from 3.3% to 0%. Thus, the higher data locality is achieved at the cost of higher delays waiting for local resources to become available. There is a clear trade-off between maximizing data locality and minimizing the running time in our experiments. Also there is a trade-off between maximizing the data locality and minimizing the stress on the network with respect of transferring the data blocks to the allocated containers.

### 3.3.2 Unbalanced allocation of containers

The second part of the study shows that there is more contention on the nodes that have a larger share of the input data blocks. Potentially, these nodes become hotspots in the cluster and consequently the other nodes become under-utilized. We run the five TPC-H queries over the same Hive database and figure 3.13 shows that $slave_9$, $slave_{10}$, $slave_{4}$ and $slave_{16}$ have the most allocated containers. The reason that these nodes hold more input
data blocks comparing than the other nodes is that in trying to maximize data locality the delay scheduler tends to over-utilize this subset of nodes. This approach can make the other nodes under-utilized and eventually affect the job completion times because the delay scheduler holds incoming tasks until the hotspot nodes are available.

3.3.3 Redundant I/Os

In the third part of the study, we observe that YARN node managers load input data blocks multiple times in order to be processed by different containers. This issue appears when different containers are processing the same input data blocks, which can often be the case in analytic workflows. The YARN node manager loads an input data block from disk to memory for each container that wants to process it. This behaviour increases the number of I/O operations dramatically.

In our experiments, the TPC-H database is loaded into Hive, then each table is split into fixed size blocks (of 64 MB) and each block is replicated over a number of nodes (replication factor = 2 in our case). We give a unique id for each input data block as:

Figure 3.11: Locality Percentages of Q18 and Q21
For example, \textit{lineitem}\_1 means block number 1 of \textit{lineitem} table. Figure 3.14 shows examples of redundant I/Os. We filter out all the blocks that are loaded less than 4 times into memory. For example, blocks \textit{lineitem}\_46 and \textit{lineitem}\_20 are loaded 5 times by 5 different containers from the disk to the memory on node \textit{slave9}.

### 3.4 Discussion and Summary

In this chapter, we study the impact on data locality of the YARN delay scheduler. We identify three potential problems. The first problem is that there is a trade-off between the running time and the data locality. The experiments show that slightly changing the node delay time ($T_1$) and rack delay time ($T_2$) affects the response time and the data locality percentages. Also, there is no single pair of parameter values that fits all the workload types for a particular cluster. We propose that \textit{YLocSim} can help users and administrators to choose the best delay time values given the history logs of workloads before deploying these settings into a real cluster. Additionally, \textit{YLocSim} can be used to validate new or modified
schedulers that target maximizing data locality without the hassle of expensive deployment on a real cluster environment.

The second problem, we identify is the unbalanced allocation of containers. This problem can be expected to happen if the workloads target a subset of the data to accomplish analytic tasks and so cause cluster hotspots, particularly when the number of replicas of data blocks is fixed. Running multiple queries on the same database tables is an example of this scenario. Dynamic replication is an approach that could minimize the hotspots as well as maximizing the data locality. Developing a scheduler that applies dynamic replication requires a prediction technique to predict the popular data blocks that would be requested from different containers based on the current status. Then, these popular data blocks can be replicated to the most under-utilized nodes in the cluster. Similarly, if input data blocks lose their popularity, the scheduler could remove the unnecessary replicas from the worker nodes.
The third problem we identify is redundant I/O operations. Introducing a caching mechanism could reduce the number of I/O operations. For instance, we could extend the capabilities of the YARN scheduler such that it sends caching commands to the node managers for certain input data blocks if these blocks are going to be reused by different containers. Then, the YARN scheduler can give a scheduling priority to the nodes that have cached blocks in the memory. There are some systems, like Spark [69], that caches the input data into the memory. However, Spark assumes that the users have specified what input files should be cached in the memory beforehand, but this is not always the case. In the following chapters, we discuss potential solutions to the aforementioned problems.
Chapter 4

Extending YLocSim to Support Scalability

Our study of data locality in YARN shows that it is challenging for users to determine a combination of values for the scheduler configuration parameters that achieves a good balance of data locality and running time. The problem becomes even more difficult when we include scalability as another factor in the problem. The resources of a cluster can be scaled in two ways: scaling-up, that is, adding more processing and memory resources to the current nodes in the cluster, and scaling-out, that is adding more nodes to the cluster. In studying performance, it is often impractical to physically scale a cluster. In this chapter, we propose an extension to YLocSim that accommodates scaling and supports the study of the potential data locality and performance changes after scaling YARN’s cluster.

4.1 YLocSim Scalability

The scalability requirements for our extension of YLocSim are discussed in detail in the following sections.
4.1.1 Vertical Scalability

The YARN data locality simulator should capture the impact on the data locality if a user scales-up a cluster by adding more memory and/or CPU cores to some or all nodes in the cluster. For specific use cases, users prefer scaling-up their resources, also known as vertical scalability, because it simplifies debugging when runtime problems occur. In other words, having a smaller number of more powerful machines may be a better solution than a larger number of less powerful machines for specific types of applications. Vertical scalability increases the number of containers that a node holds. On the other hand, it causes downtime while scaling up in physical environments. This downtime includes stopping all processing services (e.g. YARN’s Resource Management) and distributed file system services (e.g. HDFS services).

4.1.2 Horizontal Scalability

Horizontal scaling (or scaling out) refers to increasing resources by adding more machines to the cluster. The Application Masters’ resource requests can be spread across more machines which leads to reducing the excess load on the hotspot machines. There are two main approaches to dealing with the existing input data blocks while applying horizontal scalability:

1. Adding more nodes to the cluster while preserving the locations of the original input data files. \textit{YLocSim} should allow the users to horizontally scale the cluster by adding additional machines without re-distributing the original input data file blocks over the new machines (i.e. the input data file blocks are still stored on the original machines).

2. Adding more nodes to the cluster and redistributing the input data blocks on all the cluster machines. \textit{YLocSim} should provide an option to the user to re-distribute the input data blocks over the total of old and new machines based on a pre-defined data
block placement policy before the workload simulation.

Figure 4.1 shows the two horizontal scalability scenarios. Assume that there are four machines before applying the horizontal scaling. Obviously, the input data files are already stored on these four machines. The first scenario is to add two more machines to the cluster without re-distributing the data. In this case, the new machines are considered processing machines (not processing and data machines). The second scenario is to re-distribute all the data blocks on the six machines in the cluster.

![Figure 4.1: Example of Horizontal Scalability](image)

### 4.2 Extended YLocSim Design

To support horizontal scalability with data redistribution over all the nodes in the cluster, we propose a new design solution to simulate distributing the data blocks over the cluster based on a pre-defined block placement policy. Figure 4.2 illustrates the additional components and interactions between them that are needed to support this horizontal scalability requirement. The new components are:
Figure 4.2: YLocSim Horizontal Scalability with Data Re-distribution

1) **Input Files Meta-data Extractor** Given the original Hadoop jobs traces files (i.e. log files), *Input Files Meta-data Extractor* extracts the meta-data of the input HDFS files that are used by Hadoop jobs (steps 1 and 2). The meta-data for an HDFS file are the HDFS file path and its number of blocks.

2) **HDFS Block Placement Simulator** Given the new cluster topology, a pre-defined HDFS block placement policy and the meta-data for the input HDFS files, the HDFS Block Placement Simulator places the HDFS blocks on the new machines in the new topology (steps 3a and 3b). We design HDFS Block Placement Simulator to exercise the typical actions performed by the HDFS Name Node while placing HDFS blocks on a cluster of data nodes. The HDFS Block Placement Simulator’s output are the new physical locations of the data blocks on the new cluster topology (step 4).

3) **New Traces Generator** The *New Traces Generator* component uses the new HDFS
blocks' physical locations (step 5a) and the original Hadoop jobs traces (step 5b) to generate new traces based on the new cluster topology and the new physical locations of the input data blocks. The new traces are ingested by YLocSim to report the expected data locality and running time of the given workload in the new environment (step 6).

4.3 HDFS Block Placement Policy

One objective of the HDFS Block Placement Policy is to achieve a good balance between the network bandwidth and reliability. The NameNode replicates each data block on multiple machines to prevent the failure of one machine from losing all copies of a block. A naive replication policy is to replicate the data blocks on different racks. However, this policy is not fully optimized when writing data as each write operation requires transferring blocks to multiple racks. Also, the communication between two nodes in different racks must go through switches and usually network bandwidth between two machines on different racks is less than the bandwidth between two machines in the same rack.

The current HDFS replica placement policy is called Hadoop Rack Awareness which balances between placing the blocks on the same rack and different racks. For example, if the replication factor is three, then HDFS places one replica in a local rack, the second on a node in a different remote rack and the third one on a different node on the same remote rack. Figure 4.3 shows an example of the HDFS replica placement policy.

4.4 Evaluation

We use a private cluster to validate YLocSim’s extension. Table 4.1 summarizes the configuration of the cluster.

\footnote{http://hadoop.apache.org/}
We use the TPC-H benchmark described earlier to validate \textit{YLocSim}'s scalability extension. We execute the Pig queries in parallel on the same Hive database to validate \textit{YLocSim} scalability under an intensive workload.

### 4.4.1 Evaluation Metrics

\textit{YLocSim}'s extended support for scalability is validated using the following metrics:

1. The difference between the data locality percentages reported by \textit{YLocSim} verses the data locality percentages reported from executing the same benchmark on a real YARN cluster. This can be quantified as follows:

\[
error_{node} = \frac{|c.\text{node}\_locality\% - y.\text{node}\_locality\%|}{c.\text{node}\_locality\%}
\]

where \(c.\text{node}\_locality\%) is the node locality percentage that is reported from the real cluster and \(y.\text{node}\_locality\%) is the node locality percentage that is reported from \textit{YLocSim} (node locality percentage is already defined in equation 5.4).
Table 4.1: Private cluster setup for validating YLocSim’s extension

2. The difference between the running time reported by YLocSim verses the running time reported from a real YARN cluster based on the same benchmark. The running time is the time between submitting the first job of the benchmark and finishing the execution of the last job. Similar to Node Locality Percentage error, the running time error can be quantified as:

\[
error_{running\_time} = \frac{|c.r\_running\_time - y.r\_running\_time|}{c.r\_running\_time}
\]  

(4.2)

where \(c.r\_running\_time\) is the running time that is reported from the real cluster and \(y.r\_running\_time\) is the running time that is reported from YLocSim.

Obviously, the smaller the data locality percentage and running time errors, the more accurate are YLocSim’s predictions for data locality and running time. For the following experiments, we run each workload 5 times and report the average node locality percentages and running times. The average locality percentage and the average running time are good representations because the standard deviations are small. For instance, the maximum standard deviation for of the node locality percentage and the running time for all cases are
approximately 3.8% and 41.6 seconds respectively.

4.4.1.1 Vertical Scalability Validation

Figures 4.4, 4.5 and 4.6 show that the more resources (i.e. CPU cores and memory) are added to the clusters' machines, the better YLocSim's quality with respect of reporting node locality percentage and running time.

![Diagram showing node locality percentage validation of vertical scalability](image)

Figure 4.4: Node Locality Percentage Validation of Vertical Scalability
The smallest node locality and runtime errors occurred on the original topology (i.e. before scaling up the cluster) which are 4.1% and 4.8% respectively. The original cluster topology is 8 machines and each machine holds three containers at a time. Each container is
1 CPU core and 1024 MB memory. The error of the node locality percentage is dropped from 12% with 5 containers per node to 7.4% with 10 containers per node. An explanation for the decay of the error while scaling up the cluster is that the more resources that a local node has, the more node local containers it can run. Consequently, \textit{YLocSim} assigns more node local containers to the data nodes. The main result shows that the error in the simulation is less than 14% in all cases, which means that \textit{YLocSim} does a good job of modeling data locality when clusters are scaled-up. Our rationale is that the other sources of errors are discussed in detail later in the following section.

4.4.1.2 Horizontal Scalability Validation (Without Data Re-Distribution)

Figures 4.7, 4.8 and 4.9 show the differences between the node local percentages and running times that are reported from \textit{YLocSim} and the real cluster after adding new more machines. In these set of experiments, we do not redistribute the data on all the machines after adding new nodes. In other words, the TPC-H database is stored across the original 8 machines. These experiments show that \textit{YLocSim} reports more node locality percentages and less running time than the real cluster. For example, after adding 12 new machines to the original cluster, \textit{YLocSim} reports the node locality percentage as 47.7% and the running time as 1041.7 seconds. Whereas, the actual node locality percentage is 42.2% and the actual running time is 1247 seconds.
Figure 4.7: Node Locality Percentage Validation of Horizontal Scalability (Without Data Re-Distribution)

Figure 4.8: Running Time Validation of Horizontal Scalability (Without Data Re-Distribution)
Also, the figures show that the error increases when adding new machines to the cluster. For instance, the node locality percentage error increases from 4% to 16% and the running time error increases from 4% to 18% when adding 16 machines to the original cluster. In the following section, we discuss why the errors while scaling-out a cluster are larger than with scaling-up.

However, the node locality percentage and the running time that are reported by *YLocSim* follow the same trend as in the real cluster. This means that *YLocSim* can be useful when planning to scale out a cluster.

### 4.4.1.3 Horizontal Scalability Validation (With Data Re-Distribution)

In this section, we validate *YLocSim* horizontal scalability with re-distributing the original input data files over the whole cluster (i.e. the old and the new machines). Figures 4.10, 4.11 and 4.12 show the differences between the node locality percentages and the running time that are reported from *YLocSim* and the real cluster. From these set of experiments, we conclude that the node locality percentage and running time errors are increasing while

![Figure 4.9: Error of *YLocSim* Horizontal Scalability (Without Data Re-Distribution)](image-url)
adding more machines to the cluster. For example, after adding 8 machines to the original cluster, the \textit{YLocSim}'s error is 19%. Whereas, after adding 16 machines to the original cluster, the \textit{YLocSim}'s error is 26%.

Figure 4.10: Node Locality Validation of Horizontal Scalability (With Data Re-Distribution)
Figure 4.11: Running Time Validation of Horizontal Scalability (With Data Re-Distribution)

Figure 4.12: Error of YLocSim Horizontal Scalability (With Data Re-Distribution)
Additionally, the errors are relatively greater than the errors in case of horizontal scalability without data re-distribution. For example, when adding 16 machines without re-distributing the data, the node locality percentage error is approximately 16%. However, with data re-distribution scenario, the node locality percentage error increases to 26%. The interpretation of these types of errors is discussed in the following section.

4.4.1.4 Discussion

In this section, we discuss the possible reasons for the differences in the values produced by \textit{YLocSim} while simulating a real YARN cluster. The first reason is that, for each run of \textit{YLocSim}, the Resource Manager places the Application Master’s container on a different node and this container runs until the application finishes. This can cause a slight change in node locality percentages and the running time for each run. In addition, due to this random assignment, the application master’s container could be placed on a node that does not hold a lot of data blocks, which leads to higher data locality (while the opposite is also true). We tried to force YARN scheduler to allocate an Application Master container on the same node as in the workload log traces, but unfortunately, there is an open bug that prevents YARN from allocating containers on particular nodes\cite{1, 14}. Additionally, this problem arises when adding more nodes because the probability of placing the Application Master container on a different node increases. This explains why the error when simulating vertical scalability is less than with horizontal scalability.

The validation experiments also show that there is an increase in the error for the case of horizontal scalability with data re-distribution compared with horizontal scalability without data re-distribution. Our rationale is that this increase in error is due to the block placement simulator. The block placement simulator follows the same policy of assigning the HDFS blocks as the NameNode. However, the block placement simulator does not place the data blocks over the cluster nodes on the same machines like the real cluster (even if both are
applying the same block placement policy). Using the default block placement policy, that is provided by HDFS, the NameNode includes randomness while choosing the proper rack or machine to place a data block. In other words, if a user uploads the input dataset twice, the NameNode would place the data blocks differently both time.

In addition, the data locality of the reduce tasks is quite undefined as the data input of each reduce task is dynamically generated from scattered nodes where the map tasks ran. Consequently, the Application Masters place the reduce containers randomly on any available node(s) without regard the number of blocks that are stored on these nodes, which affects the data locality in the map phase.

4.4.2 Using YLocSim to Predict Impact of Scaling

In the previous section, we validated YLocSim against a real cluster. In this section, we show how YLocSim can be used to predict the impact of scaling resources on data locality and the running time for a given workload.

We saw in Chapter 3 that the current YARN implementation experiences a load imbalance of the map containers among the cluster resources as some nodes have more input data blocks than others. We introduce, Imbalance Percentage, a metric that measures the overall cluster under-utilization [55]. Imbalance Percentage is defined as:

$$\lambda = \left( \frac{L_{\text{max}}}{\bar{L}} - 1 \right) \times 100\%$$ (4.3)

where $L_{\text{max}}$ is the maximum number of map containers that a node has already executed, $\bar{L}$ is the average number of containers that are executed across all the cluster nodes. $\lambda$ represents the imbalance percentage for the whole nodes in the cluster. The lower $\lambda$, the better the cluster utilization. Figure 4.13 shows some examples of different map containers allocation scenarios using a cluster of nine nodes. If the nine nodes in the cluster execute the same number of containers, $\lambda$ equals to zero. On the other hand, if nodes 1, 5 and 6
execute more containers than the other nodes, then $\lambda$ is greater than zero (e.g. 89%).

Figures 4.14 and 4.15 show that adding more resources per node in the cluster leads to better node locality percentage and running time. For example, increasing the number of containers per node from 3 to 6 improves the node locality percentage by approximately 10.1% and the running time by 203 seconds. However, at some cut-off, the results show that increasing the number of containers does not lead to significant improvement in either the data locality or running time. For instance, doubling the resources from 12 containers per machine to 24 containers per machines leads to approximately 3.4% improvement in data locality and approximately 5.6 seconds in running time.
Figure 4.14: Node locality percentage for the Vertical Scalability

Figure 4.15: Running time in seconds for the Vertical Scalability

Figure 4.16 shows the overall cluster under-utilization for different vertical scalability setups. We see that increasing the number of containers per machine leads to a linear
increase in the imbalance percentage. For example, increasing the number of containers per machine from 3 containers to 24 containers leads to a 37.6% increase in the imbalance percentage. This is because increasing the computational power of the hotspot nodes makes YARN’s Resource Manager assign more local containers to improve the data locality. On the other hand, the other nodes in the cluster become under-utilized compared to the hotspot machines that store more of the needed data blocks.

![Imbalance Percentage for Vertical Scalability](image)

Figure 4.16: Imbalance Percentage for Different Vertical Scalability Setups

Figures 4.17 and 4.18 show the data locality percentage and the running time for the horizontal scalability. The first observation is that, in general, increasing the number of machines leads to lower node locality percentages. For example, the node locality percentage for a cluster setup with 100 machines and with re-distributing the data is approximately 41.2%, whereas a cluster with 16 machines achieves 60.2%. This is because the more machines in the cluster, the lower the probability that the Resource Manager will allocate the containers on the local nodes, especially, when the delay scheduler timing thresholds (i.e. $T_1$ and $T_2$) expire. For the same reason, the Resource Manager tries to utilize the other non-local machines, which leads to a better running time while increasing the number of machines.
The second observation is that increasing the number of machines in the cluster without re-distributing the data leads to more remote data block transfers (i.e. lower node locality percentage) compared to horizontal scalability with re-distributing data. We assume that the remote data blocks are the blocks that are transferred over the same rack or over the switch (off-switch). For example, the expected node data locality percentage of a cluster with 100 machines without re-distributing the data is 18.7% and the running time is 964.2 seconds. On the other hand, the node locality percentage for the same cluster setup, but with re-distributing the data over all machines in the cluster is 41.2% and the running time is 874.15 seconds. The reason is that the original 8 machines in the cluster are considered hotspots as they hold all the input data blocks. Consequently, every container that is assigned to any new machine added to the cluster requires transferring data blocks over the network.
The third observation is that the data re-distribution decision in horizontal scaling impacts the cluster utilization dramatically. For instance, the imbalance percentage of a cluster with 100 machines without re-distributing the data is approximately 125.8%. While the imbalance percentage of the same cluster setup, but with re-distributing the data is 97.2%. This is because the Resource Manager tries to over-utilize the machines that store the input data blocks, however, the rest of the machines are under-utilized until the local machines finish executing the containers or the remote locality thresholds expire (i.e. $T_1$ and $T_2$).
4.5 Summary

In this chapter, we discuss YLocSim’s extension that allows the administrators to have performance insights before scaling the real cluster. The extension of YLocSim supports vertical and horizontal scalability and it is an open-source and can be run on a single node to simulate a cluster with hundreds of machines. The source code is available on GitHub. The validation experiments show that YLocSim’s simulation error is less than 14% when clusters are scaled-up and 26% when clusters are scaled-out.

The experiments show that adding more resources per node in the cluster leads to better node locality percentage and running time. However, at a certain threshold, increasing the hardware resources per machine does not lead to significant improvement in either the data locality or running time. Also, the experiments show that increasing the number of machines in the cluster without re-distributing the data leads to more remote data block transfers (i.e. lower node locality percentage) compared to horizontal scalability with re-distributing data.

---

3https://github.com/yelshater/hadoop-2.3.0
Chapter 5

YARN Dynamic Replication Manager (YDRM)

Maximizing data locality is an important design goal for many data intensive systems like YARN. Jobs that can access the data locally complete faster than those that must access remote data. As we have seen in the previous studies, YARN has shortcomings with its approach to data locality. First, the policy of a fixed number of replicas in HDFS does not help to improve the data locality as the data access frequencies from different applications may vary. Second, when a YARN container requests a remote data block for processing, YARN does not keep a local copy of this data block for future containers that may require the same data block.

In this chapter, we propose YARN Dynamic Replication Manager (YDRM), a new component in YARN that applies online dynamic replication to improve the data locality. YDRM takes advantage of YARN’s Resource Manager’s global view of resources to proactively detect the popular data blocks and then place new replicas on the most under-utilized nodes in the cluster in order to minimize the hotspots. Experiments show that YDRM improves the data locality for our test data-intensive workloads and the resource utilization of the cluster.
5.1 YARN Dynamic Replication Manager

The main objective of the YARN’s delay scheduler is to maximize the utilization of the nodes that store the input data blocks. This can cause increased contention on the nodes having more input data blocks. Another problem is that YARN does not keep a local copy of data blocks for future containers that may require the same data block, and consequently, this causes more data blocks to be transferred over the network.

Dynamic replication is an approach to minimizing the hotspots while maximizing the data locality. We considered two approaches to detect the hotspots in a cluster. The first approach is to monitor the current performance parameters (e.g. the CPU utilization and available memory) on each node in the cluster to identify the nodes that are heavily utilized. We can then reallocate tasks from the heavily-utilized nodes to the under-utilized ones [67]. This approach is efficient while achieving load balancing of cluster’s resources like CPU and memory. However, it fails to improve the data locality as it does not replicate the data files that are likely requested from different tasks. Consequently, the data files must be transferred over the network each time the tasks are migrated from one node to another.

The second approach we considered is to detect the popular data blocks that are requested by the different containers from the executing applications. We then proactively replicate the popular data blocks to the most under-utilized nodes in the cluster without interfering with the running applications. Similarly, unnecessary replicas can be removed as data blocks lose their popularity. This approach is more effective for data locality than the first approach as YARN’s scheduler assigns containers (in terms of CPU and memory) based on the location of the data. To achieve realtime dynamic replication, we introduce a new component into the YARN’s architecture called YARN Dynamic Replication Manager (YDRM). Figure 5.1 shows the new YARN’s architecture after deploying YDRM.

YDRM works along with the Resource Manager such that it receives the resource requests from the different Application Masters, which can be responsible for both MapReduce and
Figure 5.1: YARN Dynamic Replication Manager

non-MapReduce jobs. Each Application Master calculates the required resources based on the data locations and sends these requests to the Resource Manager (figure 5.1, step 1).

YDRM updates the blocks’ statistics with their access counts and scores each data block based on the number of containers that request access to it. Then, YDRM marks those blocks with a score greater than a pre-defined score threshold as popular blocks and issues replication requests to the HDFS NameNode (figure 5.1, step 2). We discuss the scoring mechanism in detail in the following section. YDRM selects the most under-utilized node to place the new replica. Later, we discuss how YDRM orders the nodes in the cluster. Usually in the data intensive applications, the number of the resource requests is greater than the number of the running containers. YDRM leverages the waiting time of the resource requests and issues synchronous copies of data blocks. Users can set REPLICA_TIMEOUT in case that the replication of a given block does not completed successfully (i.e. the target node is down
or the network is too slow). If the \textit{REPLICA\_TIMEOUT} expires for a given block, then \textit{YDRM} ignores replicating this block to avoid task starvation. If the NameNode succeeds in replicating the popular blocks, then it sends an acknowledgement to \textit{YDRM} (figure 5.1, steps 3 and 4). After receiving the acknowledgement, \textit{YDRM} updates the resource requests with the new replicas' locations (figure 5.1, step 5). Finally, the Resource Manager assigns the containers based on the locations of the data blocks. For each container, the Resource Manager binds its block ID and sends them back to the Application Masters (figure 5.1, step 6). These bindings are used by the Application Masters to identify the containers that it has been allocated. The new dynamically-allocated containers are used first by the Application Masters as they are allocated on the most under-utilized nodes.

In our design, we decouple the logic of \textit{YDRM} and the logic of the Resource Manager to achieve the \textit{separation of concerns} design aspect [37], such that the Resource Manager remains concerned with resource management in terms of container allocation based on the set of resource requests. Also, this makes \textit{YDRM} a pluggable component such that administrators can enable or disable it without changing the logic of the original YARN components. In addition, the calculation of a block's popularity and replication/de-replication decisions are delegated to the \textit{YDRM}. The same approach can be used in other systems like Mesos \footnote{http://mesos.apache.org} that allocates resources in a similar way to YARN.

Algorithm 1 describes the steps in packaging the resource requests. Each Application Master prepares a list of Resource Requests (called \textit{ask} in line 1) and sends it to the Resource Manager. We modify the messaging protocol to include the input data blocks' information in each local resource request. We construct a logical block ID that uniquely identifies any input data block stored in HDFS. This logical block ID consists of the file location on HDFS appended to its block's starting byte. Then, the Application Master iterates over the block's physical locations to construct the node local, rack and off-switch resource requests. The
Application Master includes the block id in the case of node local resource requests (line 9).

**Algorithm 1** Package Maps’ Resource Requests Algorithm

1: \( \text{ask} = \text{List} < \text{ResourceRequest} > \)
2: \textbf{for} mapAttempt \textbf{in} allMapTasks \textbf{do}
3: \hspace{1em} taskMetaInfo = mapAttempt.getMetaInfo()
4: \hspace{1em} fileUrl = taskMetaInfo.getSplitFileUrl()
5: \hspace{1em} startingByte = taskMetaInfo.getStartingByte()
6: \hspace{1em} blockId = constructBlockId(fileUrl, startingByte)
7: \hspace{1em} blockLocations = getBlockLocations(blockId)
8: \textbf{for} localNode \textbf{in} blockLocations \textbf{do}
9: \hspace{2em} localReq = ResourceRequest(localNode, blockId)
10: \hspace{2em} rack = localNode.rack
11: \hspace{2em} rackReq = ResourceRequest(rack)
12: \hspace{2em} offSwitchReq = ResourceRequest(*)
13: \hspace{2em} ask.add(localReq)
14: \hspace{2em} ask.add(rackReq)
15: \hspace{2em} ask.add(offSwitchReq)
16: \textbf{end for}
17: \textbf{end for}
18: \textbf{return} ask

In algorithm 2, \( \text{YDRM} \) receives the Resource Requests list from the Application Masters and extracts the block IDs (line 3). \( \text{YDRM} \) updates the blocks’ access counts through maintaining a max-heap data structure in memory for faster retrieval of the popular blocks (line 4). Also, \( \text{YDRM} \) incrementally updates the mean and the variance of the blocks’ access counts, which are used in the scoring. \( \text{YDRM} \) updates the blocks’ access counts within a sliding time window \( W \) (e.g. using the most recent requests within \( W \) hours) and this window can be set by the administrators.

After updating the aforementioned statistics, \( \text{YDRM} \) calculates the blocks’ popularity by assigning a score to each data block based on their access counts (line 7). The higher a block’s score, the higher the probability that the block will be requested remotely multiple times by different containers. We discuss the scoring in detail in the following section. If the score of a block is greater than a pre-defined threshold and its number of replicas is less
Algorithm 2 YDRM Dynamic Replication Algorithm

1: ask = getResourceRequests()
2: for request in ask do
3:   blockId = request.getBlockId()
4:   updateBlockStats(blockId)
5: end for
6: for block in allRequestedBlocks do
7:   bscore = calculateScore(block)
8:   if bscore > THRESHOLD and replicas(block) < BUDGET then
9:     targetNode = selectMinNode(block)
10:    NameNode.replicate(targetNode,block)
11:    decreaseBlockScore(block)
12:    newRequest = ResourceRequest(targetNode,blockId)
13:    ask.add(newRequest)
14: end if
15: end for

than a user-defined storage budget per block, then this block is scheduled to be dynamically
replicated on the under-utilized nodes (line 8).

We assume the most under-utilized node is the node with the lowest average number of
requests to its blocks, which is calculated using the following equation:

\[
\text{target\_node} = \min_{\forall n \in N} \frac{|n.\text{requests}|}{|n.\text{blocks}| + 1}
\]  \hfill (5.1)

where \text{target\_node} is the node that receives the replica of the candidate data block. \(N\)
is the set of the nodes in the cluster. \(|n.\text{requests}|\) is the number of resource requests that
node \(n\) receives from different application masters. \(|n.\text{blocks}|\) is the number of blocks that
node \(n\) stores. We add 1 to \(|n.\text{blocks}|\) in the denominator to avoid dividing by zero in case of
a node that does not store any input data block requested by the current workload. Then,
YDRM decreases the block’s popularity by subtracting the mean access count from its access
count (line 11). Afterwards, YDRM updates the resource requests list by appending the new
physical locations that hold the new replicas (line 13). This updated resource requests list
gives more choices to the Resource Manager to allocate containers on local nodes.

For each container, the Resource Manager binds its corresponding block ID. This binding
Algorithm 3 Application Master allocation algorithm

1: for c in response.containers do
2:   # Allocate containers on the nodes that store new replicas.
3:   for mapTask in scheduledMapTasks do
4:     if mapTask.blockId == c.blockId then
5:       launchNewContainer (mapTask, c)
6:       scheduledMapTasks.remove(mapTask)
7:       response.containers.remove(c)
8:     end if
9:   end for
10: end for
11: # Allocate rest of the containers based on the default policy
12: # First, allocate the node local containers.
13: for c in response.containers do
14:   for mapTask in scheduledMapTasks do
15:     if mapTask.nodeId == c.nodeId then
16:       launchNodeContainer (mapTask, c)
17:       scheduledMapTasks.remove(mapTask)
18:       response.containers.remove(c)
19:     end if
20:   end for
21: end for
22: # Do the same for allocating the rack local containers
23: ... 
24: # Do the same for allocating the off-switch containers
25: ... 

process gives the Application Masters a hint to consume the allocated containers that are based on the new replicas as top priority (since these containers are allocated to the most under-utilized nodes).

Algorithm 3 describes the container allocation process. The Application Master iterates over the scheduled map tasks and checks if there are any of these tasks need to process blocks that are dynamically replicated (lines 1 to 10). After allocating the new containers, each Application Master allocates the rest of the containers using the default allocation policy (i.e. first, allocating node containers, then rack local containers and finally the off-switch containers). These steps are described in algorithm 3 lines 13 to 25.

The process to remove a replica is the reverse of the one to increase the number of replicas.
Given a pre-defined time-to-live value, YDRM starts to cleanup the unpopular blocks from the most utilized nodes to alleviate the contention. Then, YDRM communicates with the Resource Manager to update the new blocks’ physical locations.

5.1.1 Scoring Data Blocks

Since the access requests to the different input data blocks vary from one analytics workload to another, we need to choose a standardized score to define the access threshold given the differences among workloads. The score should be normalized and dependent on the relevant number of accesses. z-score is a standard score that is used to normalize the values of a random variable (e.g. access counts in our case) [48]. For each input data block $b_i$, YDRM calculates its $z_i$ score using the following equation:

$$z_i = \frac{x_i - \mu}{\sigma} \quad (5.2)$$

Where $x_i$ is the number of access requests to the input data block $b_i$, $\mu$ is the mean value of the number of access requests across all the input data blocks for a given workload and $\sigma$ is the standard deviation of the number of access requests. After assigning a score for each data input block, YDRM sorts the data blocks according to this score and issues a replication request for each data block with a score greater than the pre-defined $z$-score threshold.

Calculating the $z$-score threshold requires the data access requests to be normally distributed, which is not a common case in production workloads as discussed in Chapter 1. We use the intuition of the central limit theorem to transform a skewed data access distribution to a normal distribution by sampling the average number of data access requests using a sufficient number of samples. The central limit theorem (CLT) states that the sampling distribution of the sample mean is approximately normally distributed regardless of the underlying distribution [73, 25]. We use the following equation to estimate the minimum sample size that ensures that the CLT applies given a pre-defined threshold:
\[ n \geq \left( \frac{Z_{\text{threshold}} \cdot \sigma}{E} \right)^2 \]  \hspace{1cm} (5.3)

where \( n \) is the required sample size and \( Z_{\text{threshold}} \) is the normalized access count threshold. \( YDRM \) considers that any block \( b_i \) with score \( z_i \geq Z_{\text{threshold}} \) is a popular block that is worth replicating. \( E \) is the margin error that defines the range of values for which we are confident that the true average access count of the population is within this range.

Algorithm 2 (line 8) shows how \( YDRM \) flags the popular blocks. Setting a high \( Z_{\text{threshold}} \) value results in creating too few replicas of the popular blocks and consequently, failing to mitigate the contention on the hotspot nodes. On the other hand, setting a low \( Z_{\text{threshold}} \) value results in replicating a large number of blocks which wastes storage and network resources. In the evaluation (section 5.2), we examine these possible trade-offs by setting \( Z_{\text{threshold}} \) and \( BUDGET \) (which is the maximum number of replicas per block) to different values.

## 5.2 Performance Evaluation

In this section, we present the evaluation methodology and the performance analysis of \( YDRM \) through different experiments on a real cluster.

### 5.2.1 Evaluation Methodology

We evaluate the performance of deploying \( YDRM \) on a private cluster with 1 master node and 24 worker nodes. Table 5.1 summarizes the configuration of the cluster.

### 5.2.1.1 Workloads

To evaluate the performance of \( YDRM \), we ensure that the workloads are data intensive and complex enough to evaluate the data locality. In addition, they include sequential and
Table 5.1: Private cluster setup for validating \textit{YDRM}

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>25 nodes (1 master node and 24 worker nodes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of racks</td>
<td>4 racks each rack contains 6 slave nodes</td>
</tr>
<tr>
<td>CPU Cores</td>
<td>8 CPU cores for the master node</td>
</tr>
<tr>
<td></td>
<td>4 CPU cores for each slave node</td>
</tr>
<tr>
<td>Memory</td>
<td>16 GB for the master node</td>
</tr>
<tr>
<td></td>
<td>8 GB for each slave node</td>
</tr>
<tr>
<td>Container capability</td>
<td>1024 MB memory</td>
</tr>
<tr>
<td></td>
<td>1 virtual CPU</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu Server 12.04 LTS - 64 bits</td>
</tr>
<tr>
<td>Storage</td>
<td>160 GB for the master node</td>
</tr>
<tr>
<td></td>
<td>80 GB for each worker node</td>
</tr>
<tr>
<td>Network</td>
<td>1 Gigabit Ethernet</td>
</tr>
<tr>
<td>Hadoop Version</td>
<td>Hadoop 2.3.0</td>
</tr>
<tr>
<td>Data replication factor</td>
<td>2</td>
</tr>
<tr>
<td>Data block size</td>
<td>64 MB</td>
</tr>
</tbody>
</table>

parallel jobs that frequently access the input data files from HDFS. The workloads we use in this study are the following:

**Hadoop workflows:** we use the same Hadoop workflows derived from TPC-H as in the previous chapters.

**Facebook Traces:** The Statistical Workload Injector for MapReduce (SWIM) (also known as Facebook cluster traces) includes a set of historical traces that are sampled from a 600 machine cluster at Facebook [12]. These jobs are synthesized by a set of replay scripts that are developed to generate the original workload in production. The data access pattern of these traces is skewed to the popular files (figure 5.2). SWIM includes two datasets, namely the Facebook 2009 and Facebook 2010 traces. We execute 600 jobs from the Facebook 2010 traces.

Figure 5.3 shows the data access distribution of the sample jobs that are used in our experiment. It is clear that the distribution of the Facebook traces is more skewed than the TPC-H distribution. 70% of the jobs request access to less than 12.5% of the input data files. However, in TPC-H, 70% of the jobs request access to less than 35% of the input data
5.2.1.2 Metrics

We evaluate YDRM based on the following metrics:

**Node Locality Percentage**: Data locality for the map tasks, as defined in the earlier chapters, is the main evaluation metric used to measure the quality of data locality in YARN. We consider only map locality as the locations of map data input blocks are easily known in HDFS. We calculate the node locality percentage as the number of locally allocated containers divided by the total number of allocated containers using the following equation:

\[
\text{node\_locality\_percentage} = \frac{\text{node\_containers}}{\text{total\_containers}} \times 100\%
\]  

(5.4)

where \text{node\_containers} is the number of containers that the YARN’s scheduler allocated on nodes that store the input data blocks. \text{total\_containers} is the total number of containers.
that are allocated at all locality levels. Obviously, the larger node locality percentage value, the better the data locality is.

**Percentage of Replicated Blocks**: We use this metric to measure how many data blocks have been dynamically replicated by YDRM and transferred remotely over the network. We assume that any data block that is transferred over the network is a remote block. It is important to report this metric to study the sensitivity of YDRM against the different parameters like the score threshold and the maximum budget per block. This metric is calculated as follows:

\[
\text{replicated blocks percent} = \frac{R_{ydrm}}{B}
\]

(5.5)

where \(R_{ydrm}\) is the total number of blocks that are dynamically replicated by YDRM and transferred over the network. \(B\) is the total number of blocks assuming the HDFS replication factor is equal to a predefined maximum budget per block (i.e. replication factor = BUDGET). In other words, \(B\) is the number of unique data blocks times the replication
factor. In the Sensitivity Analysis section, we discuss how the replicated blocks percentage changes against the different values of score threshold and maximum budget.

**Completion Time Metric:** Since the benchmarks we are using in the evaluation involve both short and long jobs, the average turnaround time may not be a good choice as it is dominated by long jobs. The geometric mean of the turnaround time (GMTT) is a non-biased metric to the long jobs and it is used to evaluate the performance of different cloud and grid systems [20, 61]. GMTT is defined as follows:

$$GMTT = \left( \prod_{k \in K} TT_k \right)^{1/|K|}$$  \hspace{1cm} (5.6)

where $TT_k$ is the turnaround time of job $k$. The turnaround time of a job is the difference between the job’s completion time and the job’s arrival time. $K$ is the set of all the jobs in the current workload. The lower the GMTT, the better the performance of the system.

**Imbalance Percentage:** The current YARN implementation experiences load imbalance of the map containers among the cluster resources as some nodes have more input data
blocks more than others. We use *Imbalance Percentage* (or $\lambda$), which is a good metric to measure the effect of YDRM’s replica placement policy on the allocation of the map containers within the cluster nodes [55]. *Imbalance Percentage* is already defined in section 4.4.2 as:

$$\lambda = \left( \frac{L_{\text{max}}}{\bar{L}} - 1 \right) \times 100\% \quad (5.7)$$

where $L_{\text{max}}$ is the maximum number of map containers that a node has already executed, $\bar{L}$ is the average number of containers that are executed using all the cluster nodes. $\lambda$ represents the imbalance percentage for the whole nodes in the cluster. The lower $\lambda$, the better the cluster utilization.

### 5.2.2 Sensitivity Analysis

In this section, we analyse the sensitivity of YDRM against two main parameters which are (listed in algorithm 2): $Z_{\text{threshold}}$, which is the score threshold value that defines the popular blocks (i.e. a popular block is any block with a score greater than $Z_{\text{threshold}}$) and $BUDGET$, which is the maximum number of replicas per block. We use TPC-H benchmark in the all sensitivity analysis experiments. We use $YLocSim$, to validate the correctness of YDRM’s dynamic replication algorithm, however, the experiments are on a real cluster.

Figure 5.5 shows that increasing the score threshold with a fixed block budget leads to a smaller number of node local containers. The reason is that setting the score threshold too small leads YDRM to flag more data blocks as popular. Consequently, a larger number of data blocks are dynamically replicated which increases the probability of allocating node local containers. Also, the results show that increasing the budget per block leads to a higher data locality for the TPC-H benchmark we test.
Conversely, figure 5.6 shows the percentage of the data blocks that are transferred over the network for each score threshold and budget setting. The higher the score threshold, the smaller the number of blocks that are dynamically replicated and the smaller the amount of data transferred remotely. Additionally, the results show that increasing the block budget to a certain cut-off does not lead to a major data locality improvement, whereas it leads to a major increase in network traffic. For example, increasing the block budget from 8 to 9 (using score threshold = 1.5) improves the node locality percentage from 70% to 72%, however, it increases the percentage of the replicated data blocks that transferred over the network from 32% to 40% for TPC-H benchmark. From the results, we find that setting $Z_{\text{threshold}} = 1.5$ and $BUDGET = 7$ gives reasonable results for the rest of the experiments.
5.2.3 Data Locality

Figure 5.7 shows the node locality percentages for TPC-H and Facebook traces before and after deploying YDRM. YDRM achieves better data locality than the default delay scheduler. YDRM improves the data locality by 15% and 21% for TPC-H and Facebook traces respectively. The results show that YDRM achieves better data locality for Facebook traces than TPC-H. Our interpretation is that YDRM achieves better data locality when the data access distribution of the workload becomes more skewed. For example, Facebook’s data access distribution (figure 5.3) is more skewed than TPC-H’s data access distribution (figure 5.4). The gain by replicating the most popular data blocks increases when the jobs access a smaller percentage of the input data blocks, which happens when the data access distribution becomes more skewed. Figure 5.8 shows that the dynamic replication causes 11% more replicated block transfers in TPC-H than Facebook. The overhead of dynamic replication in the TPC-H workload outweighs its benefit compared to the Facebook workload.
Figure 5.7: Node Locality Percentage

Figure 5.8: Replicated Blocks Transfer Percentage
5.2.4 Completion time

In this section, we analyse the completion time based on the GMTT metric (defined in equation 5.6). Figure 5.9 shows that $YDRM$ speeds up the geometric mean turnaround time of TPC-H queries by 9.8% and Facebook traces by 12.14% compared to the default delay scheduler implementation. Similar to the data locality observation, the more skewed the data access distribution, the larger the improvement in completion time.

![Figure 5.9: Geometric Mean Turnaround Time (GMTT)](image)

5.2.5 Popular Block Placement Policies

One goal of designing $YDRM$ is to alleviate the contention on the hotspots in the cluster which store more data blocks than the others. In this section, we discuss how the dynamic replication helps YARN’s Resource Manager to allocate the map containers more uniformly. Imbalance percentage (defined in equation 5.7) evaluates the quality of the allocating map containers over the cluster nodes. We compare the imbalance percentage of two different
block placement policies namely: minimum and random placement policies. Using the random policy, \textit{YDRM} places the popular blocks on random nodes without considering the hotspot nodes in the cluster. On the other hand, using the minimum placement policy, \textit{YDRM} places the popular blocks on the most under-utilized nodes (i.e., the nodes with minimum load factor) based on the sorting criteria defined in equation 5.1. The larger the imbalance percentage, the higher the probability of causing hotspots due to the unbalanced container allocation for the workloads we test.

Figure 5.10 shows the imbalance percentage of map containers allocation before and after deploying \textit{YDRM}. Clearly, enabling \textit{YDRM} leads to a more balanced allocation of map containers than the default YARN implementation. For the TPC-H benchmark, \textit{YDRM} improves the balance in the allocation by 4% and 13% using the random and the minimum placement policies, respectively. For the Facebook traces, \textit{YDRM} improves the balance of the allocation by 7% and 15% using the random and the minimum placement policies, respectively. In addition, the results show that placing the popular blocks on the most under-utilized nodes is better than the random policy.
5.3 Summary

In this chapter, we provide a method to improve the data locality in the new Hadoop architecture (a.k.a YARN). To achieve this goal, we propose YDRM, a new component for the YARN architecture that implements realtime dynamic replication. We leverage the benefit of YARN’s global resource management to detect the popular data blocks that are requested from different Application Masters. We design YDRM to work with any of YARN’s schedulers and generic enough to support MapReduce and non-MapReduce applications with minimal changes in the Application Master logic.

We validate the first version of YDRM on MapReduce workloads like TPC-H and Facebook traces. The results show that improving data locality results in more balanced utilization of resources and improvement in response times. Our performance evaluation shows that there are many trade-offs between the data locality and the network traffic against...
different YDRM’s parameters.

YDRM performs better when the data access distribution is skewed. YDRM improves the data locality by 15% and 21% for TPC-H and Facebook traces respectively compared to the default delay scheduler. Also, the evaluation results show that YDRM improves the overall cluster utilization by placing the new replicas to the most under utilized nodes.
Chapter 6

Conclusions and Future Work

Our main focus in this work is to provide solutions to study and improve the data locality in the new Hadoop architecture (a.k.a YARN). Maximizing data locality is an important goal for many data intensive systems, like YARN, as it has a significant impact on the performance. Users face a challenge to determine a combination of the scheduler configuration parameter values that achieves a good balance of data locality and running time. Also, users do not have any insights about the performance before adding physical resources to their clusters. In addition, the static replication mechanism of YARN and HDFS causes a clear imbalance of resource allocation when considering data locality.

Our data locality study shows that there is a trade-off between data locality and running time for different scheduler configurations for a given workload. We provide YLocSim as a simulation tool that is able to report this trade-off without deploying the scheduler’s configuration changes on a real cluster.

We extend YLocSim to accommodate scaling and to report the potential data locality and performance changes after scaling a YARN cluster. The experiments show that YLocSim’s simulation error is less than 14% when clusters are scaled-up and 26% when clusters are scaled-out.

We propose YDRM, a new component in YARN that applies dynamic replication to
improve the data locality and cluster utilization. Results show that \textit{YDRM} improves the data locality by 21\% and improves the balance in the containers’ allocation by 13\% for workloads with heavy-tailed data access distributions.

### 6.1 Contributions

We make five contributions in this thesis to study and improve the performance of executing complex workloads over the distributed environment.

The \textbf{first contribution} is a survey study [42], which is presented in Chapter 2. The survey covers the different optimization methods for analytical workloads by different systems. In this study, we focus more on how different systems maximize data locality while executing analytical workloads.

The \textbf{second contribution} is providing a study of data locality in YARN, presented in Chapter 3. The goal of this Chapter is to study the data locality achieved by YARN for different workloads and configurations. Also, the study highlight areas of potential performance problems and suggest solutions and it helps with better understanding of data locality variations on YARN.

The \textbf{third contribution} is introducing \textit{YLocSim} as a simulator tool that helps the researchers and users to evaluate the impact of new features and scheduling algorithms on the data locality provided by YARN.

The \textbf{fourth contribution} is extending \textit{YLocSim} to support scalability, which is presented in Chapter 4. This extension allows the users and administrator to have performance insights (i.e. data locality and running time) before scaling the real cluster. \textit{YLocSim} supports vertical and horizontal scalability. \textit{YLocSim} is an open-source and can be run on a single node to simulate a cluster with hundreds of machines. The source code is available on GitHub \footnote{https://github.com/yelshater/hadoop-2.3.0}.
The fifth contribution is introducing \textit{YDRM}, a new component to YARN’s architecture, presented in Chapter 5. \textit{YDRM} detects the popular data blocks that are requested by the different containers from the executing applications. \textit{YDRM} then proactively replicates the popular data blocks to the most under-utilized nodes in the cluster without interfering with the running applications. \textit{YDRM} is open-source that can be deployed directly to a YARN cluster. We publish the source code on GitHub \(^1\).

6.2 Future Work

The current implementation of \textit{YLocSim} only simulates the MapReduce applications. However, \textit{YLocSim} can be extended to simulate Non-MapReduce applications by implementing new Application Master components. For example, to simulate Spark jobs, a developer should only implement the behaviour of Spark Application Master while requesting resources from YARN’s Resource Manager. Then, \textit{YLocSim} could report the data locality for the Spark Applications.

Another possible extension to \textit{YLocSim} is to design an optimizer to select the best configuration parameters that will maximize the data locality. For example, such an optimizer would choose the best \(T_1\) and \(T_2\) values for the delay scheduler.
Bibliography


[14] YARN ignores host-specific resource requests bug

[15] YARN SLS


[38] Mohammad Islam, Angelo K Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. Oozie: towards a scalable workflow management system for hadoop. In *Proceedings of the 1st ACM*


Appendix A

TPC-H

TPC-H Schema

Figure A.1: TPC-H Schema
TPC-H Queries

Q7 - SQL

select
supp_nation, cust_nation, l_year, sum(volume) as revenue
from
(
select
n1.n_name as supp_nation, n2.n_name as cust_nation, extract(year from l_shipdate) as l_year, l_extendedprice * (1 - l_discount) as volume from supplier,lineitem,orders, customer, nation n1, nation n2
where
s_suppkey = l_suppkey
and o_orderkey = l_orderkey
and c_custkey = o_custkey
and s_nationkey = n1.n_nationkey
and c_nationkey = n2.n_nationkey
and (n1.n_name = ':1' and n2.n_name = ':2')
or (n1.n_name = ':2' and n2.n_name = ':1')
) as shipping
group by supp_nation,cust_nation,l_year
order by supp_nation,cust_nation,l_year;
Q7 - Pig

-- $input is the HDFS directory path that holds TPC-H data tables
-- $output is the HDFS output directory path to store the output of the query.
supplier = load '$input/supplier' USING PigStorage('|') as
    (s_suppkey:long, s_name:chararray, s_address:chararray,
    s_nationkey:int, s_phone:chararray, s_acctbal:double, s_comment:chararray);
lineitem = load '$input/lineitem' USING PigStorage('|') as
    (l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber:long,
    l_quantity:double, l_extendedprice:double, l_discount:double, l_tax:double,
    l_returnflag:chararray, l_linestatus:chararray,
    l_shipdate:chararray, l_commitdate:chararray,
    l_receiptdate:chararray, l_shippingstruct:chararray,
    l_shipmode:chararray, l_comment:chararray);
orders = load '$input/orders' USING PigStorage('|') as
    (o_orderkey:long, o_custkey:long, o_orderstatus:chararray, o_totalprice:double,
    o_orderdate:chararray, o_orderpriority:chararray, o_clerk:chararray,
    o_shippriority:long, o_comment:chararray);
customer = load '$input/customer' USING PigStorage('|') as
    (c_custkey:long, c_name:chararray, c_address:chararray, c_nationkey:int,
    c_phone:chararray, c_acctbal:double, c_mktsegment:chararray, c_comment:chararray);
nation = load '$input/nation' USING PigStorage('|') as
    (n_nationkey:int, n_name:chararray, n_regionkey:int, n_comment:chararray);
filtered_lineitem = filter lineitem by l_shipdate >= '1995-01-01'
    and
    l_shipdate <= '1996-12-31';
lineitem_supplier = join supplier by s_suppkey, filtered_lineitem by l_suppkey;
lineitem_supplier_order = join orders by o_orderkey, lineitem_supplier by l_orderkey;
lineitem_supplier_order_customer = join customer by c_custkey,
lineitem_supplier_order by o_custkey;
lineitem_supplier_o_c_nation1 = join nation by
n_nationkey, lineitem_supplier_order_customer by s_nationkey;
lineitem_supplier_o_c_nation1_nation2 = join nation by
n_nationkey,lineitem_supplier_o_c_nation1 by c_nationkey;
filtered_lineitem_nation1_nation2 = filter lineitem_supplier_o_c_nation1_nation2 by
($1=='FRANCE' and lineitem_supplier_o_c_nation1::nation::n_name=='GERMANY')
or ($1=='GERMANY' and lineitem_supplier_o_c_nation1::nation::n_name=='FRANCE');
shipping = foreach filtered_lineitem_nation1_nation2 GENERATE
lineitem_supplier_o_c_nation1::nation::n_name as supp_nation,
nation::n_name as cust_nation,
SUBSTRING(l_shipdate, 0, 4) as l_year,
l_extendedprice * (1 - l_discount) as volume;
grouped_shipping = group shipping by (supp_nation, cust_nation, l_year);
aggregated_shipping = foreach grouped_shipping GENERATE FLATTEN(group),
SUM($1.volume) as revenue;
ordered_shipping = order aggregated_shipping by group::supp_nation,
group::cust_nation, group::l_year;
store ordered_shipping into '$output/Q7_out' USING PigStorage('|');
Q8 - SQL

select o_year,
sum(case when nation = ':1' then volume else 0 end) / sum(volume) as mkt_share
from
(
select extract(year from o_orderdate) as o_year,
l_extendedprice * (1 - l_discount) as volume,n2.n_name as nation
from part,supplier,lineitem,orders,
customer, nation n1, nation n2, region
where p_partkey = l_partkey
and s_suppkey = l_suppkey
and l_orderkey = o_orderkey
and o_custkey = c_custkey
and c_nationkey = n1.n_nationkey
and n1.n_regionkey = r_regionkey
and r_name = ':2'
and s_nationkey = n2.n_nationkey
and o_orderdate between date '1995-01-01' and date '1996-12-31' and p_type = ':3' ) as all_nations
group by o_year
order by o_year;
Q8 - Pig

customer = load '$input/customer' USING PigStorage('|')
as (c_custkey:long, c_name:chararray, c_address:chararray,
c_nationkey:int, c_phone:chararray, c_acctbal:double,
c_mktsegment:chararray, c_comment:chararray);
orders = load '$input/orders' USING PigStorage('|')
as (o_orderkey:long, o_custkey:long, o_orderstatus:chararray, o_totalprice:double,
o_orderdate:chararray, o_orderpriority:chararray, o_clerk:chararray,
o_shippriority:long, o_comment:chararray);
lineitem = load '$input/lineitem' USING PigStorage('|')
as (l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber:long,
l_quantity:double, l_extendedprice:double, l_discount:double,
l_tax:double, l_returnflag:chararray, l_linestatus:chararray,
l_shipdate:chararray, l_commitdate:chararray, l_receiptdate:chararray,
l SHIPPINGSTRUCT:chararray, l_shipmode:chararray, l_comment:chararray);
supplier = load '$input/supplier' USING PigStorage('|')
as (s_suppkey:long, s_name:chararray, s_address:chararray,
s_nationkey:int, s_phone:chararray, s_acctbal:double, s_comment:chararray);
nation = load '$input/nation' USING PigStorage('|')
as (n_nationkey:int, n_name:chararray, n_regionkey:int, n_comment:chararray);
region = load '$input/region' USING PigStorage('|')
as (r_regionkey:int, r_name:chararray, r_comment:chararray);
part = load '$input/part' USING PigStorage('|')
as (p_partkey:long, p_name:chararray, p_mfgr:chararray, p_brand:chararray,
p_type:chararray, p_size:long, p_container:chararray,
p_retailprice:double, p_comment:chararray);
fregion = filter region by r_name == 'AMERICA';
forders = filter orders by o_orderdate <= '1996-12-31'
and
o_orderdate >= '1995-01-01';

fpart = filter part by p_type == 'ECONOMY ANODIZED STEEL';
n1 = join nation by n_regionkey, fregion by r_regionkey;
seln1 = foreach n1 generate n_nationkey;

c1 = join customer by c_nationkey, seln1 by n_nationkey;
selc1 = foreach c1 generate c_custkey;

o1 = join forders by o_custkey, selc1 by c_custkey;

selo1 = foreach o1 generate o_orderkey, o_orderdate;

l1 = join lineitem by l_orderkey, selo1 by o_orderkey;
sell1 = foreach l1 generate o_orderdate, l_partkey,
     l_discount, l_extendedprice, l_suppkey;

p1 = join fpart by p_partkey, sell1 by l_partkey;

selp1 = foreach p1 generate o_orderdate, l_discount,
       l_extendedprice, l_suppkey;

s1 = join supplier by s_suppkey, selp1 by l_suppkey;

sels1 = foreach s1 generate o_orderdate, l_discount,
       l_extendedprice, l_suppkey;

n2 = join nation by n_nationkey, sels1 by s_nationkey;
seln2 = foreach n2 generate SUBSTRING(o_orderdate,0,4) as o_year,
       l_extendedprice * (1 - l_discount) as volume, n_name;

grResult = GROUP seln2 by o_year;

sumResult = foreach grResult{
    seln3 = filter seln2 by n_name MATCHES 'BRAZIL';
}
generate group, SUM(seln3.volume)/SUM(seln2.volume) as mkt_share;
}

sortResult = order sumResult by group;

store sortResult into '$output/Q8_out' USING PigStorage('|');
Q9 - SQL

select
nation,o_year,sum(amount) as sum_profit
from
(
select
n_name as nation,
extract(year from o_orderdate) as o_year,
l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
from
part,supplier,lineitem,partsupp,orders,nation
where
s_suppkey = l_suppkey
and ps_suppkey = l_suppkey
and ps_partkey = l_partkey
and p_partkey = l_partkey
and o_orderkey = l_orderkey
and s_nationkey = n_nationkey
and p_name like '%:1%'
) as profit
group by nation,o_year
order by nation,o_year desc;

Q9 - Pig

customer = load '$input/customer' USING PigStorage(' | ')
as (c_custkey:long,c_name:chararray, c_address:chararray,
c_nationkey:int, c_phone:chararray, c_acctbal:double,
c_mktsegment:chararray, c_comment:chararray);

orders = load '$input/orders' USING PigStorage('|')
  as (o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
      o_totalprice:double, o_orderdate:chararray, o_orderpriority:chararray,
      o_clerk:chararray, o_shippriority:long, o_comment:chararray);

lineitem = load '$input/lineitem' USING PigStorage('|')
  as (l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber:long,
      l_quantity:double, l_extendedprice:double, l_discount:double,
      l_tax:double, l_returnflag:chararray, l_linestatus:chararray,
      l_shipdate:chararray, l_commitdate:chararray,
      l_receiptdate:chararray, l_shippingstruct:chararray,
      l_shipmode:chararray, l_comment:chararray);

supplier = load '$input/supplier' USING PigStorage('|')
  as (s_suppkey:long, s_name:chararray, s_address:chararray,
      s_nationkey:int, s_phone:chararray, s_acctbal:double,
      s_comment:chararray);

nation = load '$input/nation' USING PigStorage('|')
  as (n_nationkey:int, n_name:chararray, n_regionkey:int, n_comment:chararray);

region = load '$input/region' USING PigStorage('|')
  as (r_regionkey:int, r_name:chararray, r_comment:chararray);

partsupp = load '$input/partsupp' USING PigStorage('|')
  as (ps_partkey:long, ps_suppkey:long, ps_availqty:long,
      ps_supplycost:double, ps_comment:chararray);

part = load '$input/part' USING PigStorage('|')
  as (p_partkey:long, p_name:chararray, p_mfgr:chararray,
p_brand:chararray, p_type:chararray, p_size:long,
p_container:chararray,p_retailprice:double, p_comment:chararray);
fpart = filter part by REGEX_EXTRACT(p_name,'(green)', 1) != '';  
s1 = join nation by n_nationkey, supplier by s_nationkey;  
sels1 = foreach s1 generate s_suppkey, n_name;  
l1 = join lineitem by l_suppkey, sels1 by s_suppkey;  
sell1 = foreach l1 generate s_suppkey, l_extendedprice,
l_discount, l_quantity, l_partkey, l_orderkey, n_name;  
l2 = join partsupp by (ps_suppkey, ps_partkey), sell1  
by (l_suppkey, l_partkey);  
sell2 = foreach l2 generate l_extendedprice, l_discount,
l_quantity, l_partkey, l_orderkey, n_name, ps_supplycost;  
l3 = join fpart by p_partkey, sell2 by l_partkey;  
sell3 = foreach l3 generate l_extendedprice, l_discount,
l_quantity, l_orderkey, n_name, ps_supplycost;  
o1 = join orders by o_orderkey, sell3 by l_orderkey;  
selo1 = foreach o1 generate n_name as nation_name,  
SUBSTRING(o_orderdate, 0, 4) as o_year,
(l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity) as amount;
grResult = GROUP selo1 by (nation_name, o_year);  
sumResult = foreach grResult generate flatten(group),
SUM(selo1.amount) as sum_profit;  
sortResult = order sumResult by nation_name, o_year desc;  
store sortResult into '$output/Q9_out' USING PigStorage('|');
Q18 - SQL

```sql
select
c_name,c_custkey,o_orderkey,o_orderdate,o_totalprice,sum(l_quantity)
from
customer,orders,lineitem
where
o_orderkey in (select
l_orderkey
from
lineitem
group by
l_orderkey having
sum(l_quantity) > :1)
and c_custkey = o_custkey
and o_orderkey = l_orderkey
group by
c_name,c_custkey,o_orderkey,o_orderdate,o_totalprice
order by o_totalprice desc,o_orderdate;
```

Q18 - Pig

```pig
lineitem = load '$input/lineitem' USING PigStorage('|')
as (l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber:long,
l_quantity:double, l_extendedprice:double, l_discount:double,
l_tax:double, l_returnflag:chararray, l_linestatus:chararray,
```
l_shipdate:chararray, l_commitdate:chararray, l_receiptdate:chararray,
l_shippingstruct:chararray, l_shipmode:chararray, l_comment:chararray);
customer = load '$input/customer' USING PigStorage('|')
as (c_custkey:long, c_name:chararray, c_address:chararray,
c_nationkey:int, c_phone:chararray, c_acctbal:double,
c_mktsegment:chararray, c_comment:chararray);
orders = load '$input/orders' USING PigStorage('|')
as (o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
o_totalprice:double, o_orderdate:chararray, o_orderpriority:chararray,
o_clerk:chararray, o_shippriority:long, o_comment:chararray);
lineitem_group = group lineitem by l_orderkey;
orderkeys_sum = foreach lineitem_group generate group,
SUM(lineitem.l_quantity) as l_quantity_sum;
orderkeys_filter = filter orderkeys_sum by l_quantity_sum>300;
lineitem_orders = join orderkeys_filter
by group, orders by o_orderkey;
lineitem_orders_customer = join lineitem_orders
by o_custkey, customer by c_custkey;
lineitem_orders_customer_group = group lineitem_orders_customer
by (c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice);
result = foreach lineitem_orders_customer_group generate group.c_name
as c_name, group.c_custkey as c_custkey,
group.o_orderkey as o_orderkey, group.o_orderdate as o_orderdate,
group.o_totalprice as o_totalprice,
SUM(lineitem_orders_customer.l_quantity_sum) as l_quantity_sum;
out = order result by o_totalprice desc, o_orderdate;
store out into '$output/Q18_out' USING PigStorage('|');
**Q21 - SQL**

```sql
chapter
select
s_name,
count(*) as numwait
from
supplier, lineitem l1, orders, nation
where
s_suppkey = l1.l_suppkey
and o_orderkey = l1.l_orderkey
and o_orderstatus = 'F'
and l1.l_receiptdate > l1.l_commitdate
and exists (select * from lineitem l2
where
l2.l_orderkey = l1.l_orderkey
and l2.l_suppkey <> l1.l_suppkey
)
and not exists (select * from lineitem l3
where
l3.l_orderkey = l1.l_orderkey
and l3.l_suppkey <> l1.l_suppkey
and l3.l_receiptdate > l3.l_commitdate
)
and s_nationkey = n_nationkey
```
and n_name = ':1'

group by s_name

order by numwait desc, s_name;

Q21 - Pig

orders = load '$input/orders' USING PigStorage('|')
as (o_orderkey:long, o_custkey:long, o_orderstatus:chararray,
    o_totalprice:double, o_orderdate:chararray, o_orderpriority:chararray,
    o_clerk:chararray, o_shippriority:long, o_comment:chararray);

lineitem = load '$input/lineitem' USING PigStorage('|')
as (l_orderkey:long, l_partkey:long, l_suppkey:long, l_linenumber:long,
    l_quantity:double, l_extendedprice:double, l_discount:double,
    l_tax:double, l_returnflag:chararray, l_linestatus:chararray,
    l_shipdate:chararray, l_commitdate:chararray, l_receiptdate:chararray,
    l SHIPPINGSTRUCT:chararray, l_shipmode:chararray, l_comment:chararray);

supplier = load '$input/supplier' USING PigStorage('|')
as (s_suppkey:long, s_name:chararray, s_address:chararray,
    s_nationkey:int, s_phone:chararray, s_acctbal:double,
    s_comment:chararray);

nation = load '$input/nation' USING PigStorage('|')
as (n_nationkey:int, n_name:chararray,
    n_regionkey:int, n_comment:chararray);

gl = group lineitem by l_orderkey;

L2 = filter gl by COUNT(org.apache.pig.builtin.Distinct(lineitem.l_suppkey))>1;
fL2 = foreach L2{
    t1 = filter lineitem by l_receiptdate > l_commitdate;
    generate group, t1;
}

store fL2 into '$output/Q21_temp' using PigStorage('|');

fL3 = filter fL2 by COUNT(org.apache.pig.builtin.Distinct($1.l_suppkey)) == 1;
L3 = foreach fL3 generate flatten($1);

fn = filter nation by n_name == 'USA';

fn_s = join supplier by s_nationkey, fn by n_nationkey USING 'replicated';

fn_s_L3 = join L3 by l_suppkey, fn_s by s_suppkey USING 'replicated';

fo = filter orders by o_orderstatus == 'F';

fn_s_L3_fo = join fn_s_L3 by l_orderkey, fo by o_orderkey;

gres = group fn_s_L3_fo by l_orderkey, fo by o_orderkey;

sres = foreach gres generate group as s_name,
    COUNT($1) as numwait;

ores = order sres by numwait desc, s_name;

lres = limit ores 100;

store lres into '$output/Q21_out' using PigStorage('|');

fs -rmr $output/Q21_temp