Pingo: a Framework for the Management of Storage of Intermediate Outputs of Computational Workflows

by

Jadiel de Armas

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved April 2017 by the
Graduate Supervisory Committee:

Rida Bazzi, Chair
Dijiang Huang
Violet Syrotiuk

ARIZONA STATE UNIVERSITY

May 2017

ABSTRACT

Scientific workflows allow scientists to easily model and express the entire data processing steps, typically as a directed acyclic graph (DAG). These scientific workflows are made of a collection of tasks that usually take a long time to compute and that produce a considerable amount of intermediate datasets. Because of the nature of scientific exploration, a scientific workflow can be modified and re-run multiple times, or new scientific workflows are created that might make use of past intermediate datasets. Storing intermediate datasets has the potential to save time in computations. Since storage is limited, one main problem that needs a solution is determining which intermediate datasets need to be saved at creation time in order to minimize the computational time of the workflows to be run in the future. In this research thesis I propose the design and implementation of Pingo, a system that is capable of managing the computations of scientific workflows as well as the storage, provenance and deletion of intermediate datasets. Pingo uses the history of workflows submitted to the system to predict the most likely datasets to be needed in the future, and subjects the decision of dataset deletion to the optimization of the computational time of future workflows.

DEDICATION

To my wife, who has taken the torch from my mom and kept it burning.

To my uncle, my aunt and my cousins. They made sure nothing bad happened from mom to wife

ACKNOWLEDGEMENTS

First, I want to thank God because he gives me the daily strength and curiosity to keep learning new things. I also want to thank my advisor, Dr. Rida Bazzi, for his guidance in the research and the time that he has taken to help me put together a good document. I also want to thank Dr. Dijian Huang and Dr. Violet Syrotiuk for accepting to be part of the Committee and for accommodating me into their busy schedules. Finally, I want to thank my wife for her support and understanding during the different stages of the research.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

1 INTRODUCTION 1

1.1 Contributions 3

LIST OF SYMBOLS 1

PREFACE 1

CHAPTER

2 RELATED WORK 5

2.1 Scientific workflows 5

2.2 Scientific Workflow Management Systems 6

2.3 Parallel Processing Frameworks 7

3 THEORETICAL CONSIDERATIONS OF THE DATA REUSE PROBLEM 9

3.1 Skipping unnecessary computations 9

3.1.1 Actions 10

3.1.2 Workflows 11

3.2 Bounded storage space 16

3.2.1 The History of Workflows 16

3.2.2 The Data Reuse Problem 17

3.2.3 Two families of algorithms 18

4 IMPLEMENTATION OF THE SYSTEM 21

4.1 Bird's eye overview 21

4.2 The Workflow Definition Language 22

4.2.1 Actions 24

CHAPTER Page

4.2.2 Determining Action's output path 25
4.3 The Action Manager 28
4.3.1 Action States 29
4.3.2 The Action Scraper 29
4.3.3 The Action Submitter 31
4.4 The Workflow Manager 32
4.4.1 Datasets 32
4.5 The Callback System 36
4.5.1 The Success Callback 36
4.5.2 The Action Failed Callback 36
4.5.3 The Action Killed Callback 37
4.6 The Dataset Manager 37
4.6.1 The Dataset Scraper 37
4.6.2 The Dataset Deletor 38
4.7 The Decision Manager 38
5 EVALUATION METHODOLOGY OF THE DECISION ALGORITHMS AND RESULTS 40
5.1 Evaluation Methodology 41
5.1.1 Workflows generator 41
5.1.2 Ideal Execution Time calculation 43
5.2 Evaluation Experiments 48
5.2.1 How computation time is affected by the amount of storage in the system 49

CHAPTER Page

5.2.2 How computation time is affected by the types of workflows in the history of workflows 51
5.3 Conclusions of our evaluations 52
6 FUTURE RESEARCH 54
REFERENCES 56
APPENDIX
A SYSTEMS' USER GUIDE 59
B IMPLEMENTATION OF WORKFLOW GENERATOR ALGORITHM . 60
C PARAMETERS OF EXPERIMENTS 63
BIOGRAPHICAL 65

LIST OF TABLES

Table Page

4.1 Action State Descriptions. 30

4.2 Dataset State Descriptions. 33

LIST OF FIGURES

Figure		Page
3.1	Hypothetical workflow and its corresponding hash structure	14
4.1	Workflow definition	23
4.2	Command Line Action definition	26
4.3	Example with two different orderings of input parameters	28
5.1	Example of a tree workflow that is produced with parameters:	43
5.2	Example of a more complex DAG	44
5.3	Example of four DAGs with variance of distributions increased	44
5.4	Effective life diagram of datasets of hypothetical history of workflows.	47
5.5	Computation time as storage increases	51
5.6	Computation time percentage as percentage of actions from previous workflows increases	52
C.1	Parameters of Workflows' Generator in Experiment 1	63
C.2	Parameters of Workflows' Generator in Experiment 2	64

Chapter 1

INTRODUCTION

The scientific process increasingly benefits from the use of computation to achieve advances faster. Many times these computations can be naturally broken into steps where each step may filter, transform or perform computations on the data it receives as input from a previous step. Modeling and expressing these workflow computations has been widely adopted by the scientific community as a directed acyclic graph (DAG) of computations (see Liu *et al.* (2015)). These workflows (or scientific workflows) have many tasks that take a long time to compute and that produce a considerable amount of intermediate datasets. Because of the nature of scientific exploration, a scientific workflow is usually modified and executed multiple times, or new scientific workflows are created that might make use of past intermediate datasets.

Storing intermediate datasets has the potential to save time in computations. Instead of re-computing a dataset, one could potentially use previously stored values. Since storage is limited, one main problem that needs a solution is determining which intermediate datasets need to be saved in order to minimize the computational time of workflows that will be submitted to the system in the future. Some systems provide a solution to the problem (Yuan *et al.* (2012)). Also there have been some research efforts in algorithms that determine the data sets that need to be saved in order to optimize the computation time of the workflows to be computed (see Zohrevandi and Bazzi (2013)). But that effort has been mainly applied to systems that can make optimal decisions with full knowledge of the future workflow submissions. In realistic scenarios this might not be possible, since due to the nature of scientific exploration, researchers usually need to use the results of current computations in order to design

future workflows.

Another current issue in the field is the divide that exists between academic and industry-level workflow systems. Academic systems are mostly responsible for the use directed acyclic graphs as the model to express workflows of computations. They have also been adopting the latest research in the area of data reuse optimization algorithms. On the other hand, industry-level workflow systems such as Apache Oozie and Apache Airflow have mainly focused on managing the execution of workflows of highly scalable computations that run in the successful Hadoop ecosystem. Both Oozie and Airflow have great monitoring capabilities and are very extensible. Unfortunately they don't provide any data reuse functionality. There is a need for a system that can manage highly scalable computations and that provides support for intermediate computations and data reuse.

In order to tackle those two issues, I have created Pingo, a system that is capable of managing the computations of scientific workflows for the Hadoop ecosystem. Pingo also provides a solution to the problem of optimization of data reuse without knowledge of future submissions of workflows, since it is capable, under some general and reasonable assumptions, of predicting what intermediate datasets will be needed by future workflow submissions to the system. The focus of the research is not in the predictive algorithms themselves, but on designing a plug-and-play framework that is able to support a variety of algorithms. Pingo also takes care of the management of storage and provenance information of the intermediate datasets. This work describes the design and implementation of the system's features. The need for most of those features is established from research as well as from my experience using scientific workflow systems in the past.

1.1 Contributions

Pingo has a commonality with previous systems (Altintas *et al.*, 2004; Yuan *et al.*, 2012; Deelman *et al.*, 2015) in that it stores intermediate datasets produced by workflows with the purpose of skipping the computations of future workflows. But differing from previous systems, Pingo does it in a "reactive way". That is, most previous systems decide which intermediate datasets to keep in storage with foreknowledge of the definition of the future workflows that will be submitted to the system. In this work, I analyze the history of workflows already submitted to the system in order to determine which datasets might be important to keep for the future. That difference in design makes it suitable for fast-paced research environments where it is impossible for the researcher to foresee what are the next steps to take.

Another important contribution of this work is that, to the best of my knowledge, this is the first system bringing management of intermediate datasets and its computational optimization advantages to the Hadoop ecosystem. A consequential contribution of designing it with Hadoop in mind is that I have also designed it to be a scalable multi-user system. There are other smaller contributions scattered throughout the thesis. For example, in order to support data reuse, the system needs to make use of a data provenance system that keeps track of the origin of each dataset stored in the file system. There are many ways of implementing such system, and I propose the use of a Merkle tree like structure as an efficient alternative to determining the provenance of datasets.

The rest of the thesis is organized as follows: Chapter 2 presents research work related to the data reuse problem and describes existing systems for managing scientific workflows. Chapter 3 formally introduces the problem and explores the theoretical motivations behind the design decisions and tradeoffs of the system. Chapter 4 pro-

vides a detailed description of the design and implementation of the system. Chapter 5 proposes a methodology to evaluate the performance of the decision algorithms introduced in 3. It also reports the results of that evaluation methodology on Pingo. And Chapter 6 talks about future directions that can improve the functionality of the system.

Chapter 2

RELATED WORK

2.1 Scientific workflows

A workflow is the automation of a process, during which data is processed by a sequence of computations in a preordered way (Liu *et al.*, 2015). From a conceptual point of view, workflows can be divided into two types: business workflows and scientific workflows (Hollingsworth and Hampshire, 1995; Taylor *et al.*, 2014). Scientific workflows are typically used for modeling and running scientific experiments. Taylor *et al.* (2014) defines scientific workflows as the assembly of complex sets of scientific data processing activities with data dependencies between them. More simple workflows can be represented as sequences (pipelines) of activities, but the most general representation is DAG, where nodes correspond to data processing actions and edges represent the data dependencies. Scientific workflows must be fully reproducible (Barker and Van Hemert, 2007). That is, the same results must be obtained after repeated executions of the computations of the workflow if the same data was used as input. Such requirement introduces the opportunity to store intermediate datasets to optimize the execution of computations of workflows.

In a workflow, an activity describes a piece of work that forms a logical step within the workflow representation. The associated data in an activity consists of the input data and configuration parameters. The execution of an activity is a job or task. In the literature, the terms **activity**, **job** and **task** are sometimes used interchangeably and special attention needs to be given to the context where the term is used to determine if they refer to the definition or to the execution of a task.

2.2 Scientific Workflow Management Systems

A Workflow Management System is a system that defines, creates and manages the execution of workflows. Many seminal works on the topic of workflow management systems began to appear in the mid 2000's (Yu and Buyya, 2005; Fox and Gannon, 2006; Gil *et al.*, 2007, e.g.), and many workflow systems were developed, such as Kepler (Altintas *et al.*, 2004), Taverna (Oinn *et al.*, 2006) and the e-Science project (Deelman *et al.*, 2009). These legacy systems provided the foundations for the field of scientific workflows, but were designed with the intent of executing computations in local standalone machines.

A more advanced example is Pegasus (Singh *et al.*, 2008), a workflow management system for scientific applications. It enables workflows to be executed both locally and on a cluster of computers in a simultaneous manner. It has a rich set of APIs that allow the construction and representation of workflows as DAGs. It also has more advanced job scheduling and monitoring facilities than previous systems.

One important capability that has been added as a functionality to some of these systems (see Yuan *et al.* (2012)) is the ability to store the computations of intermediate datasets with the purpose of optimizing the computation time of future workflows that are to be computed by the system and that make use of those intermediate datasets. For a scientific workflow system, there are two costs associated with running workflows. On the one hand there is execution cost associated for running the workflow and on the other hand there is the storage cost for storing intermediate datasets. Finding the optimal balance is at the heart of this problem. This optimization problem is ofter formulated under the realistic assumption that there is an upper bound available for storage and trying to make the best use of the available storage. The problem is also formulated as a cost optimization problem where there are no

a priori bounds on storage, and costs are associated with units of computation and storage. In this thesis I adopt the bounded-storage formulation.

Some systems research has included in their efforts research on the topic of optimizations for different versions of the data reuse problem. Ramakrishnan *et al.* (2007) addressed the problem of minimizing the amount of space a workflow requires by removing datasets at run-time when they are no longer required. As mentioned previously, the research of Yuan *et al.* (2012) proposes a strategy to find a trade-off between computation cost and storage cost. Also, some theoretical research has been done on the data reuse problem itself. For example, Adams *et al.* (2009) proposes a model to represent the trade-off of computation cost and storage cost, but does not give any strategy to solve the optimization problem that can be derived from their model. Yuan *et al.* (2011) presented two algorithms as the minimum cost benchmark of the data reuse problem, one for the case of linear-structure workflows, which takes $O(n^4)$ time, and a general algorithm for parallel structure. Cheng *et al.* (2015) improve on the work of Yuan *et al.* (2011) by presenting a new algorithm for the "linear-structure" type of workflows that runs in $O(n^3)$ time. Zohrevandi and Bazzi (2013) formally introduce the data reuse problem under the assumption that the workflows to optimize are known before-hand. They model the problem using a non-linear integer programming formulation and show that it is NP-Hard. They also propose two algorithms: a branch and bound optimal algorithm, and a heuristic algorithm that is on average within 1% of the optimal answer.

2.3 Parallel Processing Frameworks

The scale of computations have been growing with time, and the ability of the systems cited above to process large amounts of data and to execute the placement of task execution on a distributed environment is limited or can only be done on

High Performance Computing (HPC) systems of expensive hardware. Orthogonal to the development of workflow management systems, distributed parallel processing frameworks have been designed and developed to meet the growing demands of computation.

One of such frameworks is MapReduce (Dean and Ghemawat, 2008). It was originally developed by Google as a proprietary product to process large amounts of unstructured or semi-structured data clusters of machines with commodity hardware. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. There are multiple implementations of the framework, the most commonly used being part of the Hadoop ecosystem (White, 2012).

Islam *et al.* (2012) introduced Apache Oozie, which is Apache Hadoop's workflow and scheduling system. Its workflow definition API rivals that of Pegasus, while it takes advantage of the superior scalability of the Hadoop ecosystem. Unfortunately, it does not provide any capability to optimize the computation of workflows by saving the output of intermediate datasets in the hope of skipping the computation of future actions submitted to the system.

Chapter 3

THEORETICAL CONSIDERATIONS OF THE DATA REUSE PROBLEM

This chapter discusses the theoretical motivations behind data reuse systems. It introduces the data reuse problem after providing necessary background. It also presents two families of algorithms that provide solutions to the problem. The discussion is organized from a functional standpoint, discussing the theoretical motivations and tradeoffs that drive the design as they relate to the desirable functions of a system that attempts to solve the data reuse problem.

3.1 Skipping unnecessary computations

If a user submits the description of an action A to be computed, but the system has previously computed an action B that produced the exact same output that action A will produce when computed, then action A does not need to be computed. The output of action B can be returned immediately as the output of action A.

In order to achieve such functionality, there must be an equivalence relation between actions A and B. This equivalence relation is not only restricted to the description of the actions provided by the user, but also to the place of the action in the workflow, since the output of an action is determined by both the definition of the computations represented by the action as well as the input to that action from previously computed actions. To define more precisely the equivalence relation between two actions, precise definitions of **Actions** and **Workflows** are needed. The discussion follows the definitions of the model introduced by Zohrevandi and Bazzi (2013) in their work, with some slight variations in terminology, and some additions, such as the concept of History of Workflows.

3.1.1 Actions

An action can be thought of as a pure function $f : A \rightarrow B$. Suppose that the system is asked to compute $f(a)$. If the system has previously computed $b = g(c)$, and it can be determined that $g = f$ and that $c = a$, then the system can skip the computation of $f(a)$ (if it is that output b has not been deleted). Determining if a and c are equal to each other is trivial, but it can be time consuming if the inputs are big. Determining if f and g are equivalent is in general undecidable (Turing, 1937).

But comparing actions to pure mathematical functions is not completely accurate. Gifford and Lucassen (1986) say that a pure function is a function that is "referentially transparent: it does not cause side-effects, its value is not affected by side-effects, and it returns the same value each time it is evaluated." In other words, the function result cannot depend on any hidden or random information or state that may change while the function executes, or across two different executions of the function. In some cases, as next chapter shows, there will be nothing stopping actions from reading values from the outside environment, or from using randomization. Users can also submit binary files (instead of source code) for execution, or computations with third party library dependencies over which the system has no control.

Because of these difficulties, the system needs a practical approach to determine the equivalency between two actions. Instead of looking into the source code (or executable code) of the actions to determine if they are equal or equivalent, it is more convenient to look into the configuration file that corresponds to the actions. Those configurations are simply a collection of parameters such as: path to input folders, path to the executable (or source code) of the action, extra input parameters, etc.

A measure of equivalency between two actions should satisfy the following two simple properties. The description (or configuration parameters) of an action sub-

mitted to the system becomes then very relevant, since it is the starting point to determine equivalence between actions.

1. If two action descriptions represent actions that produce the same output **when executed in an ideal controlled environment**, they should be considered equivalent.

2. If two action descriptions represent actions that produce different output, they are inequivalent.

The language in the second property is absolute, since the property must be satisfied at all times for the measure of equivalency to be considered usable. As we have seen, a challenge of the second property is that computations performed by actions might not be **pure functions**. Because of that, if the end user thinks that the execution of an Action A might produce a different output to that of an existing output of an equivalent previously submitted action B, then the user must let the system know that no optimizations should be applied on action A and that its computation must be executed.

3.1.2 Workflows

In essence, a **workflow** is a DAG where nodes correspond to **actions** or to **original datasets** (datasets not derived from the computation of an action). A directed edge from a node a to a node b means that the output of action A is used as input by action B. Actions are identified with the lower letter a (a_1, a_2, ..., a_n), original datasets with the lower letter o, and derived datasets with the lower letter d. Functional notation is used to represent actions outputs from inputs. For example, if action a_1 takes as inputs original dataset o_1, derived dataset d_2 and the output of action a_3 on original dataset o_2, action a_1's output d_1 is represented as: $d_1 = a_1(o_1, d_2, a_3(o_2))$.

A topological sort of the workflow dictates a possible order of execution of the computations of the actions: actions with no parents, or actions whose parents' output are already known, can start executing whenever computational resources are available. The fact that the workflow is a DAG guarantees that all actions in the workflow will be computed at some point in time, given that there are no malformed computations or failures. Chapter 4 has a complete discussion of failure handling.

Equivalence of actions in the workflow setting

Consider the previous example of the definition of action a_1. What strategy can be devised to efficiently determine if its corresponding output dataset d_1 is already in storage? As a starting point, for every dataset d_i kept in storage, the system needs to keep some accounting, including the definition of the workflow that produced d_i. Then it is possible to search over those definitions to find one that matches the current workflow submitted to the system.

In order to analyze the running time complexity of such problem, let's look at a naive search algorithm (Algorithm 1). The algorithm takes as input action a and hypothetical procedure E, whose job is to compare two action descriptions for equivalency, returning **true** if they are equivalent, and **false** otherwise.

The algorithm goes over each dataset d_i in storage, and uses a simple Breadth First Search (BFS) strategy to compare the DAGs induced by the ancestors of both actions a and a_i (where a_i is the action whose output is d_i). If it finds an dataset a' whose DAG is equivalent to the DAG that produced a, it returns the output dataset of action a', otherwise, it returns $None$. Notice that for simplicity, the algorithm assumes that parent actions of an action are given to the corresponding queue in a correct order, so that when it pops them from the queue to compare them, they correspond to each other. Chapter 4 discusses how that assumption is valid for some

Algorithm 1 Naive dataset search algorithm:

```
1: procedure DatasetSearch(a, E)
2:     for dataset d_i in storage do
3:         a_i ← action(d_i)                          ▷ a_i is action that outputted d_i
4:         Q ← queue(), P ← queue()
5:         Q.add(d_i), P.add(a)
6:         areEqual ← True
7:         while Q not empty and P not empty do
8:             q ← pop(Q), p ← pop(P)
9:             if E(q, p) then
10:                Q.addAll(parents(q))               ▷ parents of q in the workflow
11:                P.addAll(parents(p))               ▷ parents of p in the workflow
12:            else
13:                areEqual ← False
14:                Break
15:        if areEqual and Q.size = 0 and P.size = 0 then return d_i
       return None
```

action types, but not for others.

To analyze the running time of Algorithm 1, let m be the number of datasets in storage, and let M and N be the number of nodes and of edges of the submitted workflow. The running time of the algorithm would then be $O(m(M + N))$.

The hashing alternative

Another strategy that can be used to determine if an action's output already exists in the storage system is **hashing**. Given an action node a_i in a workflow, it is possible to recursively compose the description of action a_i together with the descriptions of its parent actions to produce a hash value that "uniquely" identifies dataset d_i produced by action a_i as output; then compare that hash value with the hash values corresponding to the actions of the datasets stored in the system in order to find an equivalent dataset. This idea is similar to the concept of a Merkle tree (Merkle, 1990). Merkle trees are binary (or more generally, k-ary) trees where each non-leaf node is the hash of the concatenation of its children, and each leaf node is the hash of one file from the collection. In general for a workflow (see Figure 3.1) its hash structure is not

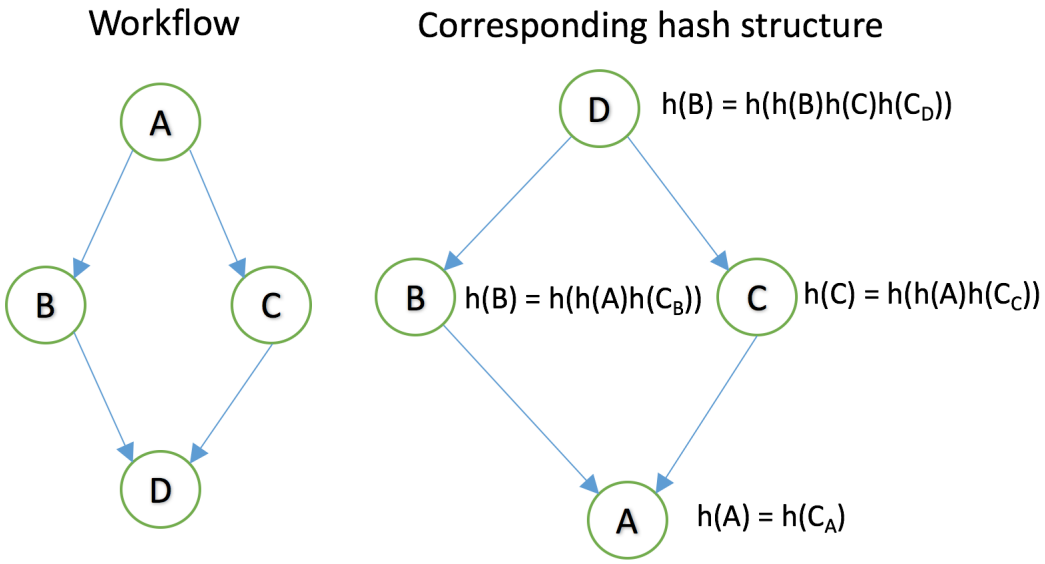

Figure 3.1: Hypothetical workflow and its corresponding hash structure

a tree but a DAG. Also, for each node in the hash structure, its corresponding hash value is not just he concatenation of its parents' hash value but also the configuration parameters of the action.

An algorithm that implements the ideas described above is highly dependent on the semantics of the description of the actions. Because of that, we leave its discussion to Chapter 4. But there are enough elements that allow us to arrive at certain conclusions regarding the viability of the approach. Firstly, comparing the hash value of action a_i to the hash values of actions whose datasets are in storage is something that can be done in constant time with the proper indexing structure. Secondly, since it is impossible to know beforehand the set of keys that are going to be hashed, the possibility of perfect hashing is discarded, and at least a lower bound analysis of what would be the probability of a clash between the hash signatures of non-equivalent actions is in place.

Assume that n is the number of bits of the hash signatures produced by the

hashing algorithm, and that N is the number of datasets currently in storage. First, I provide a lower bound of the probability of a single collision on the leafs. There are 2^n possible hash values. Assuming that all of them have the same probability of occurring (see **random oracle model** by Bellare and Rogaway (1993)), then the probability that all the N hash values are different, $1 - p(N)$ is:

$$\begin{aligned} 1 - p(N) = 1 \times \frac{2^n - 1}{2^n} \times \frac{2^n - 2}{2^n} \times ... \times \frac{2^n - N + 1}{2^n} \\ 1 - p(N) = \frac{2^n \times 2^n - 1 \times ... \times 2^n - N + 1}{2^{n^N}} \\ 1 - p(N) = \frac{2^n!}{2^{n^N}(2^n - N)!} \end{aligned} \tag{3.1}$$

Then $p(N)$ is the probability that at least two of the N hash values are the same. For $n = 80$ (SHA-1 function) and $N = 1,000,000$, the probability of a collision is $4.135580766728708e - 13$.

Now, take for example, two simple binary trees with two leaves. Let tree 1's leaves be A and B, and tree 2's leaves X and Y. The probability that the tree hashes collide at the root is the probability that $(h(A), h(B)) \neq (h(X), h(Y)) \wedge h(h(A), h(B)) = h(h(X), h(Y))$ plus the probability that $(h(A), h(B)) = (h(X), h(Y))$. The probability of the second part is equal to $p(N)^2$, and the probability of the first is $p(N) * (1 - p(N)^2)$. The total probability becomes $p(N) + p(N)^2 - p(N)^2 * p(N)$, which is slightly greater than $p(N)$, the probability of a single collision. As the depth of the tree increases, the probability of a collision on the root also increases slightly. Coron *et al.* (2005) show under which conditions collision resistance of the compression function f is sufficient to obtain collision resistance of the hash function of a Merkle tree. Their work shows that a compression function like $SHA2$ is sufficient.

At any rate I introduce a verification step to the search process. The first step of the search process finds all the datasets whose corresponding workflow's hash value is equal to the one of the workflow under consideration. The second step compares the workflows themselves to determine if they are equivalent.

3.2 Bounded storage space

Placing a constraint on the amount of space available to store intermediate datasets makes the problem more interesting. There are physical limitations and technical limitations that won't allow us to have unlimited space. Therefore, at most moments in time, the system needs to decide which datasets to keep in storage and which datasets to delete. The system also has enforce the decisions.

3.2.1 The History of Workflows

The concept of **History of Workflows** becomes useful when the constraint of bounded storage capacity is added to the system. Since the final objective is to optimize the time of computing future workflows, the decision system can be thought of as an optimization problem that tries to "guess" which datasets will be more relevant in the future. How to accurately predict which datasets are needed in the future is an open question that has multiple valid answers. Most of the potential valid strategies make use of the history of workflows submitted to the system up to that point. Different strategies might need different information from the history of workflows. Because of that, there needs to be an API layer to query the history of workflows to obtain a diverse set of statistics over specified **ranges of time** and at different **resolutions**.

3.2.2 The Data Reuse Problem

A core functionality of any workflow management system that wants to tackle the data reuse problem would be its decision system. The decision system determines which datasets need to remain and which datasets need to be deleted from storage. It can either run each time a new workflow is submitted to the system, or it can run only when space occupied by stored datasets reaches a certain threshold of the total capacity. The second option makes more sense, since under it the system remains in compliance with the bounded storage constraint, but uses less computational resources. The decision system defines an interface that can be implemented by different decision algorithms to determine which datasets currently in storage need to be deleted. Let D be the set of datasets currently stored in the file system. The goal of the algorithms is to output a set $B \subset D$ of datasets such that $\sum_{d \in B} s(d)$ is greater than or equal to F. Set B needs to be selected in such a way that there is no other subset B' of D of storage size equal or greater than B such that if datasets of B' instead of B are removed from storage, the system would spend less time computing future workflow submissions.

The interface takes the following elements as input:

1. The History of Workflows submitted to the system, $H = (W_1, W_2, ..., W_n)$, where each W_i is a DAG. It is left to the algorithm to decide if it will use the entire history or only a subset of it.

2. The set D of datasets currently stored in the file system.

3. $s : D \to \mathtt{N}$, where $s(d_i)$ is the storage space that dataset d_i occupies in the file system.

4. $c_H : D \to \mathtt{N}$, where $c_H(d_i)$ is the number of times that dataset d_i appears in

History of Workflows H.

5. $t : A \rightarrow \mathtt{N}$, where $t(a_i)$ is the computational time that it takes action a_i to compute its output d_i. If action a_i has been computed multiple times, the average of those times is reported.

6. F, the amount of space to free on the file system.

3.2.3 Two families of algorithms

Similar versions of the problem presented above have been studied in other areas of Computer Science. For example similar problems appear in the operating systems literature in the areas of memory caches (Smith, 1982) and scheduling (for a survey of algorithms, see the work of Ramamritham and Stankovic (1994)). But I have found the nature of the problems in the field of web caching and prefetching to be the most similar to the data reuse problem. Web catching and prefetching techniques play a key role in improving web performance by keeping web objects that are likely to be visited in the near future closer to the client. They exploit both the temporal and spatial locality for predicting revisiting objects. In their context, spatial locality is modeled as a graph of objects. That characteristic makes their solutions more suitable for application to the data reuse problem in workflows, since one can think of the DAG that defines the workflows as a way of representing "spatial locality". For excellent surveys of web caching and prefetching algorithms, see the works of Wang (1999) and Ali *et al.* (2011).

This section introduces two families of algorithms that solve the problem introduced in Section 3.2.2 with varying degrees of success. They use the most basic ideas from the caching literature referred above. Chapter 5 introduces a methodology to evaluate the algorithms' performance from a practical standpoint.

Least Valuable Algorithm Family

This is a very simple algorithm that represents a big family of possible algorithms that can be implemented. The idea is to retain in storage the datasets with the most valuable datasets, according to some evaluation metric. Implementations of this base algorithm define their own evaluation metrics. For example, Algorithm 2 defines value as the number of times the dataset is used throughout the history of workflows times the time it takes to compute such dataset. That is a very simple definition of value, but provides the most straightforward implementation. Algorithm 2 is a good baseline to compare against.

Algorithm 2 Least-Valuable-Datasets Algorithm

```
1: procedure LeastValuable(H, D, c_H, s, t, F)
2:     datasetValues ← List()
3:     for d_i in H do
4:         datasetValues.append((d_i, c_H(d_i) * t(d_i)))
5:     sortedDatasetValues ← sortByValue(datasetValues)
6:     spaceFreed ← 0
7:     toDelete ← List()
8:     for d, value in sortedDatasetValues do
9:         if spaceFreed ≥ F then
10:            break
11:        toDelete.append(dataset)
12:        spaceFred ← spaceFreed + s(d)
13:    return toDelete
```

Simple Adaptive Algorithm Family

Algorithms that belong to the *LeastValuableAlgorithm* family are a good starting point, but they have the weakness that they use the entire history of workflows to compute the value of datasets. That strategy might be out of touch with reality, especially in fast-paced research settings where more recent datasets are more likely to be reused than less recent ones. One important question to ask is: How far back into the history of workflows do we need to look to determine the value of datasets?

Algorithm 3 Adaptive Least-Valuable-Datasets Algorithm

```
1: procedure AdaptiveLeastValuable(H, D, c_H, s, t, F)
2:     recencyList ← List()
3:     n ← length(H)
4:     for i in from 1 to n do
5:         for dataset d in W_i do
6:             W_j ← previous workflow where d was used.
7:             if W_j is not Null then
8:                 recencyList.append(i − j)
9:     μ ← mean(recencyList)
10:    σ ← std(recencyList)
11:    SH ← H.subList(max(0, int(n − μ − 2σ)), n + 1)
12:    return LeastValuable(SH, D, c_SH, s, t, F)
```

I propose a simple adaptive algorithm, described in Algorithm 3. Its strategy is that for each workflow, for each dataset produced in that workflow, the algorithm samples how many workflows into the past one has to look back to find the previous time that such dataset was needed (not necessarily computed). After building that sample of intervals, the algorithm finds its mean (μ) and standard deviation (σ). By passing to the *LeastValuable* procedure a sub-history of workflows that goes back $\mu - 2\sigma$ steps into the past, it guarantees that the *LeastValuable* procedure receives a history of workflows that although not complete, its statistics reflect more accurately the current usage trends of the system.

Chapter 4

IMPLEMENTATION OF THE SYSTEM

4.1 Bird's eye overview

The Pingo system is a group of independent processes that together manage scientific workflow computations and their outputs. This chapter first gives a bird's eye overview of Pingo, and then explains each of the components with more detail.

From a user point of view, the interaction process proceeds as follows: The end-user defines a workflow of computation in JSON format and submits it to an endpoint of the system. But before submitting the workflow of actions to the system, the user needs to have placed the files that contain the "executables" that carry those computations in a folder in a distributed file system that is accessible to Pingo. There is a **Submission Manager** whose function is to parse the JSON description of the workflow and to analyze the directed acyclic graph of computations defined in it in order to determine which actions need to be computed. Those actions are assigned one of two states: $WAITING$ or $READY$.

There exists a process called the **Action Submitter** that is constantly pulling actions that are ready to be submitted for computation. The **Action Submitter** converts the JSON definition of the action into an XML definition that is understood by Hadoop, and submits the computation to Hadoop for execution (Hadoop here means any of the supported subsystems of the Hadoop ecosystem, such as MapReduce, Spark, Pig scripts, etc). There can be multiple **Action Submitter** processes working at the same time. To synchronize efforts, the processes use a database as synchronization mechanism to guarantee that no action is submitted to Hadoop twice.

Another independent piece of the system is the **Callback Manager**. When an action is submitted to Hadoop, a callback endpoint is provided so that Hadoop can notify back Pingo with information about the computation: if it finished, failed or was killed, etc. As a redundancy measure, the **Callback Manager** constantly polls Hadoop for information regarding each of the actions submitted for computation. Once an action finishes computing, the **Callback Manager** updates the state of the action and of the dataset produced by the action in the database. It also updates the state of any action that depends on the output of the computation that just finished.

The **Decision Manager** is another independent process that frequently queries the database and the file system to determine which of the datasets produced by actions should be kept in the file system, and which ones should be deleted. The Decision Manager is the predictive brain of the system, and its decisions will affect the computation time of future workflow submissions to the system.

The last independent process of the system is the **Dataset Manager**. The **Dataset Manager** enforces the decisions of the **Decision Manager**. It queries the database for actions whose state has been changed to TO_DELETE. If the dataset is not currently locked by any action, it deletes it from the distributed file system.

4.2 The Workflow Definition Language

The JSON format is a good choice for the definition of workflows because its expressiveness is sufficient for the needs of Pingo, and it is also very human readable. Bray (2014) provides a memo that defines the set of formatting rules of the JSON format.

As shown in Figure 4.1, a workflow is made of a **name**, an **start action id**, an **end action id** and a **list of actions**.

```
{
    "name": "Example Workflow",
    "startActionId": 1,
    "endActionId": 2,
    "actions": [
        {
            "id": 1,
            "name": "action1-name",
            "type": "command-line",
            .
            .
            .
        },
        {
            "id": 2,
            "name": "action1-name",
            "parentActions": [
                {
                    "id": 1,
                }
            ],
            "type": "command-line",
            .
            .
            .
        }
    ]
}
```

Figure 4.1: Workflow definition

The workflow definition in Figure 4.1 consists of two actions whose ids are 1 and 2. In this workflow, action 2 must be executed after action 1 finishes. This is expressed by making action 1 a parent of action 2.

These are the constraints imposed by the system on the structure of a workflow:

1. A workflow most have at least one action.

2. No two actions can have one same id in a workflow definition.

3. If an action *id* is referenced somewhere in the workflow definition (they can

be referenced in *startActionId*, *endActionId*, and within the array of *parentActions*), that action must be defined in the array of actions of the workflow.

4. The *parentActions* attribute of an action will define relationships among the actions that can be represented as a directed graph. Specifically, this directed graph must be a DAG.

If one of the constraints is not satisfied, the application will throw an error at workflow submission time.

4.2.1 Actions

Actions must have *id*, *name* and *type* attributes. They have two optional boolean attributes: *forceComputation* and *isManaged*. If *forceComputation* is set to *True*, it means that the action will compute its output regardless of if its dataset already exists in storage or not. If it is set to *False*, it means that the system has the freedom to determine if the action will be computed or not. The default is *False*.

If the attribute *isManaged* is set to *True*, it means that the path where the output of this action will be stored is determined and managed by the system. If *isManaged* is set to *False*, it means that the path where the output of this action will be stored is not determined or managed by the system, and that path must be provided by the user. The system needs to have Read/Write permissions to any path the user provides, otherwise, the execution of the action will fail when attempting to persist the output. The default value for *isManaged* is *True*. The system does not apply computation optimizations on non-managed actions.

Action names do not need to be unique. An action name is just a mnemonic resource to help with the recall of what the action does. Also, depending on the action type, there might be other required attributes too. Currently, Pingo has implemented

functionality for two types of actions: **Command-line action** and **MapReduce v1.0 action**. Only **Command-Line actions** are fully supported. The roadmap includes adding support for **Spark actions** and **Sqoop actions**.

Command Line Action

A Command Line action is a Java program to be executed by some machine in the Hadoop cluster. Before the user submits a workflow with the action, they need to have created an action folder in the Hadoop cluster's file system that contains the Java jar file with the action's executables, as well as any jars with other libraries needed by the Java program. The JSON description of the action (Figure 4.2), includes the main class of the action, as well as any command line parameters that are passed as arguments to the main class. For each kind of action, there is a contract that defines the order on which parameters in the action's JSON description are passed as arguments to the executables. For Command-Line actions the contract is that the Main class reads first the configuration parameters given in the field *additionalInput* in the order given, and then the rest of the arguments are the paths to the output of its parent actions, ordered by the id of the parent action.

4.2.2 Determining Action's output path

There must a one-to-one relationship between an action and its output path, therefore the output path can also serve as the identifier of an action. As described in Section 3.1.2, it is better to search for equivalent actions using hashes of actions. A hash will derive from an action's description and from its lineage (the hashes of its parents). Algorithm 4 describes the procedure to produce a unique string. It takes as input parameters ps and a, where ps is a list with the generated unique string of parents, and a is the action description. Depending on the type of Action a,

```
{
    "actionFolder": "/user/hadoop/examples/apps/workflows",
    "actionId": 1,
    "additionalInput": [
    {
     "key": "sizeInMB",
     "value": "306.709032093"
    },
    {
     "key": "timeInSeconds",
     "value": "199.156048492"
    },
    {
      "key": "nameNode",
      "value": "hdfs://ec2-23-32.compute-1.amazonaws.com:8020"
    },
    {
      "key": "uniqueRandomInput",
      "value": "SNI31N35RS"
     }
     ],
     "forceComputation": false,
     "mainClassName": "io.biblia.workflows.job.Main",
     "name": "action1",
     "parentActions": [
          0,
          2,
          3
     ],
   "type": "COMMAND_LINE"
I}
```


Figure 4.2: Command Line Action definition

Algorithm 4 uses a different subroutine to analyze the action's description. Algorithm 5 shows the subroutine for the Command Line Action type.

Algorithm 4 Action Hashing from Description and Lineage

```
1: procedure ActionHashing(ps, a)
2:     concatenationString ← ""
3:     for parentString in ps do
4:         concatenationString.append(parentString)
5:         concatenationString.appendSeparator()
6:     if a.type is COMMAND_LINE then
7:         concatenationString.append(CommandLineActionHashing(a))
8:     else
9:         throw NotSupportedOperation exception
10:    return hash(concatenationString)
```

Algorithm 5 Command Line Action Hashing

```
1: procedure CommandLineActionHashing(a)
2:     concatenationString ← ""
3:     concatenationString.append(a.name)
4:     concatenationString.appendSeparator()
5:     for key, value in additionalInputs do
6:         concatenationString.append(key)
7:         concatenationString.appendSeparator()
8:         concatenationString.append(value)
9:         concatenationString.appendSeparator()
10:    return concatenationString
```

Considerations to keep in mind to design subroutines for new types of actions

A hash subroutine for an action type behaves like a hash function of that action's description. Only the description fields that make that action unique need to be used. Some of the fields of the JSON description are sequences. Consider, for example, the **additionalInput** field in Figure 4.3. In the case of a COMMAND_LINE action, the order of additional input parameters matters, since they are designed to be passed as arguments to the Main class of the action. Since order matters, the two exam-

ples of Figure 4.3 represent two actions that are not equivalent. In the case of a MAP_REDUCE action, order in a sequence field does not matter, and both examples represent actions that are equivalent between themselves. For sequence fields where order does not matter, the hash subroutine needs to order the elements of the list alphabetically by key and then value, so that the hash value that corresponds to the action description is the same, no matter the order of the elements in the list.

```
{
    .
    .
    .
    "additionalInput": [
    {
     "key": "sizeInMB",
     "value": "306.709032093"
    },
    {
     "key": "timeInSeconds",
     "value": "199.156048492"
    }]
    .
    .
    .
}
```

```
{
    .
    .
    .
    "additionalInput": [
    {
     "key": "timeInSeconds",
     "value": "199.156048492"
    },
    {
     "key": "sizeInMB",
     "value": "306.709032093"
    }]
    .
    .
    .
}
```

Figure 4.3: Example with two different orderings of input parameters

4.3 The Action Manager

The **Action Manager**'s purpose is to submit individual actions to the Hadoop cluster for computation. Its current implementation in the Pingo system uses **Apache Oozie** as an intermediary. The Action Manager is safe for use in a distributed manner, since it has synchronization mechanisms that allow multiple action managers to work together.

This is a description of its functionality:

1. It maintains a synchronized queue Q with the actions to be submitted to the

Hadoop cluster. The queue is capacity bounded and supports operations that wait for it to become non-empty when retrieving an element and that wait for space to become available in the queue when storing an element. All operations are thread safe.

2. The queue is filled by an **Action Scraper** entity that queries the database for actions that are ready to be submitted.

3. The **Action Manager** pops new actions from queue Q and hands them to a pool of **Action Submitter** threads that will submit the actions to Hadoop and will also update the state of those actions in the database.

4.3.1 Action States

In order to support a cluster of servers working as action managers and to avoid the need to add a dependency to a distributed coordination server such as **Apache Zookeper**, the system implements synchronization using the database as a shared resource. It defines a synchronization oriented semantic for each of the different states of an action.

Actions can be in one of the following states: *WAITING*, *READY*, *PROCESSING*, *SUBMITTED*, *RUNNING*, *FINISHED*, *FAILED*, and *KILLED*. (See Table 4.1 for a complete reference).

4.3.2 The Action Scraper

Every certain amount of time, the **Action Scraper** queries the database to find available actions and adds them to queue Q. Available actions are actions that are in the *READY* state, or actions that have been in the *PROCESSING* state for a long time. The reason why it queries for actions that have been in the *PROCESSING*

Action State	Description
WAITING	The action has been submitted as part of a workflow and is waiting for parent actions to finish before it can be submitted to Hadoop.
READY	The action is ready to be submitted to Hadoop because it either does not depend on any other action, or because all the actions on which it depends have finished their computations.
PROCESSING	The **Action Scraper** found a READY action in the database and has placed it in the actions' queue of the actions to be submitted.
SUBMITTED	The **Action Manager** removed the action from the queue and submitted it to Hadoop.
RUNNING	Hadoop is running the computations that correspond to the action.
FINISHED	Hadoop has finished executing the action successfully.
FAILED	A run time error has occurred and the action did not finish executing.
KILLED	The user killed the action after it started executing.

Table 4.1: Action State Descriptions.

state for a long time is to account for the rare case where another **Action Manager** began processing those actions, but because of some failure the process died before it finished to process them.

Before adding the action to Q, the **Action Scraper** attempts to update the state of the action to *PROCESSING*. If the update fails because the action entity has changed after it was queried by the scraper, then the scraper drops the action and does not add it to the **Action Manager**'s queue. Otherwise, if the update is successful, it adds the action to the queue. To illustrate how this synchronization technique is valid, consider the following example with action scrapers A and B and

their corresponding action managers. Both scrapers A and B query the database for ready actions and both find action $a1$ to be in the *READY* state. Without loss of generality, assume that A is the first scraper to update the state of action $a1$ to *PROCESSING*. When B also attempts to update the state of action $a1$, it will realize that action $a1$ has already been updated by someone else, and it will immediately drop it.

The synchronization technique described and exemplified in the above paragraph is used by other components of the system. In general, that synchronization pattern can be applied in situations where multiple processes can potentially move an object o from state $S1$ to state $S3$ (in the previous example $S1$ would be equivalent to the *READY* state, and $S3$ to the *SUBMITTED* state) but only one of the processes should be allowed to do it. In order to solve the problem, there is an intermediate state $S2$ (*PROCESSING* in this case). All the processes compete to be the first one to change the state of o to $S2$. All the loosing processes drop the processing of object o, and the winning process carries on.

4.3.3 The Action Submitter

The **Action Manager** is constantly taking new elements from the queue and passing them to the **Action Submitter** threads that submit the actions to Hadoop. The decision to add actions that have been in the *PROCESSING* state for a long time to the queue makes the design of the Action Submitter more careful. The submitter first attempts to update the state of the action to *SUBMITTED*. If it succeeds, then it actually submits the action to Hadoop. If there is an error while submitting the action, then it changes the state of the action back to *READY*, which gives that action the opportunity to be picked again by an **Action Scraper** at some time in the future. As an area of future improvement, a ceiling should be imposed over the

number of times that a failing action is resubmitted to the cluster, or otherwise, the system will keep trying to submit the action forever.

4.4 The Workflow Manager

The **Workflow Manager** receives the workflows submitted to the system and determines which of the actions from the workflow need to actually be submitted to Hadoop for computation. Those actions are inserted into the database and can initially be in one of two states: *WAITING* or *READY*. If they are in a *READY* state, any active **Action Manager** will pick them up and submit them to the cluster for computation. If they are in a $WAITING$ state, it means that some of their dependencies have not been computed yet. The actions will eventually be submitted for execution once their parents finish executing. The process of how actions in the $WAITING$ state are notified that their parents finish executing is discussed later in section 4.5.

4.4.1 Datasets

The **Workflow Manager** queries for the state of the output dataset of an action to determine if the action needs to be computer or not. A dataset entity is an entry of a dataset information in the database; its dataset file is the physical file in the distributed file system. A dataset entry is always linked in the database to its corresponding action definition since the dataset path is the same as the hash of its corresponding action description. Dataset entities can be in one of the following states at any given time: *TO_DELETE*, *TO_STORE*, *TO_LEAF*, *STORED*, *LEAF*, $STORED\_TO\_DELETE$, $PROCESSING$, $DELETING$ and $DELETED$. (See Table 4.2 for a complete reference).

The **Workflow Manager** processes all the actions of the submitted workflow,

Dataset State	Description
TO_DELETE	The dataset file does not exist in the file system, but once it does, its dataset entry will be transitioned to state STORED_TO_DELETE.
TO_STORE	The dataset file does not exist in the file system, but once it does, its dataset entry state will be transitioned to STORED.
TO_LEAF	The dataset file does not exist yet in the file system, but once it does, its dataset entry state will be transitioned to the LEAF state.
STORED	The dataset file is stored in the filesystem and it corresponds to an intermediate action. The dataset file will be stored in the file system until the decision algorithm determines in the future that is not optimal for the system to keep storing it anymore.
LEAF	The dataset file is stored in the filesystem and it corresponds to a leaf action. The system never removes datasets of leafs actions. The end-user can manually remove them.
STORED_TO_DELETE	The dataset file is stored temporarily until all other actions that have claims to it as a dependency finish computing. Once all those actions finish computing, the system removes the dataset.
PROCESSING	The dataset entry is being processed with the purpose of deleting its dataset file. This is a synchronization state.
DELETING	The dataset file is being deleted. This is another synchronization state.
DELETED	The dataset file has been deleted.

Table 4.2: Dataset State Descriptions.

starting from the leaf actions in a Breadth-First-Search manner. If by analyzing the action it determines that the action needs to be computed, it calls the *prepareForComputation* procedure on that action. The *prepareForComputation* procedure first creates an action object P in the *WAITING* state and inserts it to the database. Also, for each children C of action P that also needs to be computed, the system marks on the database that C is depending on P, so that C will need to wait for P's output dataset before being ready to be computed. At last, the procedure adds all the parents of the action P to the queue if they have not already being added.

The **Workflow Manager** makes the determination if an action needs to be computed as described in Algorithm 6

Algorithm 6 Workflow Manager Algorithm

```
1:  procedure WorkflowManager(Q, W)
2:      A ← Q.pop()                                  ▷ A is the next action to be processed
3:      if A.isManaged==False OR A.forceComputation == True then
4:          prepareForComputation(A)
5:      else
6:          D ← dataset(A)
7:          if D == null OR D.state is one of [DELETED, DELETING, PROCESS-
    ING, TO_DELETE, STORED_TO_DELETE] then
8:              prepareForComputation(A)
9:          else
10:             if D.state is one of [STORED, LEAF] then
11:                 if A is a leaf action but D.state == STORED then
12:                     D.state ← LEAF
13:                 for each child C of action A do
14:                     if C was marked for computation when processed then
15:                         addClaim(C, D)
16:                         if addClaim(C, D) fails because D.state has changed then
17:                             prepareForComputation(A)
18:             else
19:                 if D.state is in [TO_STORE or TO_LEAF] then
20:                     prepareForComputation(A)
```

Three different behaviors of the algorithms described above deserve closer attention. First, on the *prepareForComputation* procedure, the system marks on the database that an action C is depending on an action P. This is needed so that the

Callback Mechanism (which will be described later) can find which are the actions depending on action P when action P finishes computing.

Secondly, on the Workflow Manager algorithm, there is a command described as $addClaim(C, D)$. That command adds a claim from child C to the dataset entity D in the database, so that the **Dataset Deletor** system (to be described later) does not delete dataset D while there is an action that depends on it whose computation has not been carried yet.

Thirdly, for the sake of correctness of the overall state of the system, I have introduced an inefficiency in the Workflow Manager's algorithm. Notice that if a dataset D is in TO_STORE or TO_LEAF state, the algorithm still prepares action A for computation. A dataset D is in TO_STORE or TO_LEAF state if its corresponding action is currently computing given dataset. This means that some other workflow submitted to the system is currently computing dataset D. To make the system more efficient, instead of asking the system to recompute action A, one could make all the children actions of A to depend on A' (the equivalent action to action A, but from another workflow), and add a claim from the child actions of A to dataset D. The problem with this approach is that both action A' and dataset D could be having their states changed to something contrary to the current situation at the same time that the state of the children of action A is being changed with outdated information on the states of A' and D. Trying to handle that situation would translate into the introduction of more complex synchronization mechanisms across multiple components of the system. For now the benefits of simplicity outweight the efficiency gains of trying to improve a situation that happens rarely.

4.5 The Callback System

The submission of an action to the Hadoop cluster includes three callbacks. Hadoop can use them to notify back to the system of any relevant event regarding the execution of the action's computations. The three callbacks are designed in such a way that the state of the action remains the same even after multiple calls to the same callback.

4.5.1 The Success Callback

The first thing the Success Callback does is to change from $WAITING$ to $READY$ the state of any child actions of the currently finished action that are not waiting for any other parent action to finish. It also changes the state of the currently finished action to $FINISHED$. The callback also removes any claims the currently finished action may have had over datasets. Finally, the callback updates the state and metadata of the dataset outputted by the currently finished action, changing it from TO_STORE, TO_LEAF or TO_DELETE to $STORED$, $LEAF$ or $STORED\_TO\_DELETE$ accordingly. Also, it updates the size the dataset occupies in the filesystem. That size is an important metric used by the optimization algorithm.

4.5.2 The Action Failed Callback

The Action Failed callback is simpler than the success callback. It removes any claims that the failed action may have had over any datasets. If the action that failed produced any output or partial output, it changes the output dataset's state to $STORED\_TO\_DELETE$ regardless of the previous state of the dataset. Also, it changes the state of the action itself to $FAILED$.

4.5.3 The Action Killed Callback

The Action Killed callback removes any claims that the killed action may have had over any datasets. If the action that was killed produced any output or partial output, it changes that output's dataset state to $STORED\_TO\_DELETE$ regardless of the previous state of the dataset. Also, the state of the action itself is changed to $KILLED$.

4.6 The Dataset Manager

The **Dataset Manager** handles the deletion of datasets from the file system. Its architecture is similar to the architecture of the **Action Manager**:

1. The **Dataset Manager** maintains a synchronized queue L with the datasets that need to be deleted from the cluster. The queue is capacity-bounded and supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element. All operations are thread save.

2. The queue is filled by a **Dataset Scraper** process that queries the database for datasets ready to be deleted.

3. The **Dataset Manager** takes dataset entries inserted to queue L and hands them to a pool of **Dataset Deletor** threads that remove the datasets from the cluster and update those datasets' states accordingly.

4.6.1 The Dataset Scraper

Every certain time, the **Dataset Scraper** queries the database to find available datasets to be deleted and add them to the queue. Datasets to delete are those in

the *STORED_TO_DELETE* state, or datasets that have been in the *DELETING* or *PROCESSING* state for a long time. The reason why it queries for datasets that have been in the *DELETING* or *PROCESSING* state for a long time is to account for the rare case where another **Dataset Manager** in another process could have began processing those datasets, but that process died before finishing to process them.

Before adding the dataset to queue L, the **Dataset Scraper** attempts to update the state of the dataset in the database to *PROCESSING*. If the update fails because the dataset entity has changed in the database after it was queried by the scraper, then the scraper drops the dataset and does not add it to the **Dataset Manager** queue. Otherwise, if the update is successful, the action is added to the **Dataset Manager** queue.

4.6.2 The Dataset Deletor

The **Dataset Manager** constantly takes new elements from the queue to pass them to the **Dataset Deletor** threads that remove them from the distributed filesystem. The deletor first attempts to update the state of the dataset to *DELETING*. If it succeeds, then it actually deletes the dataset from the file system and updates its state to *DELETED*. If it does not succeed, or if some other error occurs while deleting so that it cannot delete, it changes the state of the dataset back to *STORED_TO_DELETE* and stops processing it.

4.7 The Decision Manager

The Decision Manager determines which of the datasets currently stored in the filesystem should be deleted. The manager has three main components that work together: A filesystem utility that determines how much space is available at the

time, an Action Rolling Window system and a Decision Algorithm. The Decision Manager is implemented in such a way that the decision algorithm is a plug and play piece that can be substituted, with different algorithms optimizing for different evaluation metrics.

The first step of the decision process is to query the file system utility to obtain the amount of space currently in use by the intermediate datasets managed by the system. If the space exceeds a certain threshold, then the decision engine begins its process:

1. First it obtains a list of the last N submitted actions to the system from the Action Rolling Window. Using this list of actions, it rebuilds the graph of the workflows to which this actions belonged to in a special data structure called **Simplified Workflow History**.

2. It passes the Simplified Workflow History that comprises the last N submitted actions to the decision algorithm, together with the amount of space that needs to freed, and the decision algorithm returns a list of datasets to delete. The sum of the storage space of the returned datasets will be at least the amount of space to be freed.

3. The Decision Manager changes the state of the datasets returned by the decision algorithm to $STORED\_TO\_DELETE$, leaving in the hands of the Dataset Manager the actual execution of the deletion of the datasets.

Chapter 5

EVALUATION METHODOLOGY OF THE DECISION ALGORITHMS AND RESULTS

This chapter proposes an evaluation methodology for the decision algorithms of Pingo. It also reports on the evaluation of the two families of algorithms introduced in Chapter 3.

It is very difficult to obtain enough real-world workflows logs to do an statistically meaningful evaluation of the system. Bharathi *et al.* (2008) have one of the few works in the area of datasets for the evaluation of the performance of workflow systems. They provide a characterization of workflows from five diverse scientific applications, describing their composition and data and computational requirements. They also created a workflow generator that produces synthetic, parameterizable workflows that closely resemble the workflows that they characterize. Unfortunately, the generator system does not know of the concept of history of workflows. Each workflow that is generated by their system is independent of previously generated workflows. Because of that, their generator is not useful to evaluate the performance of the algorithms proposed in Chapter 3.

An alternative I have found is to create a new probabilistic generator of workflows with the ability to generate histories of workflows, albeit not guaranteed to look like the workflows characterized by Bharathi *et al.* (2008). Even if the data is not ideal, it still provides a well defined alternative that allows the comparison of the performance of the two proposed families of algorithms.

5.1 Evaluation Methodology

The strategy to evaluate the Decision System is:

1. Probabilistically generate a history H of workflows.
2. Submit the history H to Pingo for computation and record the **actual execution time** using the two different algorithm families.
3. Compare the execution time of each family of algorithms.

5.1.1 Workflows generator

The workflow generator creates sequences of workflows in a probabilistic way given certain parameters. The actions of the generated workflows take as parameters the size of an output in megabytes and the time of execution of that action in seconds. The task of those actions is to output a file with random contents and of size specified by the first parameter and to take for such task the amount of time specified by the second parameter.

The sequence generator is composed of two probabilistic generators that work together to produce the history of of workflows. See Appendix A for information on how to use the generator.

The Action Generator

The first of the generators is the **Action Generator**. It takes as input the number of actions to generate and the mean and variance parameters of two normal distributions, one for the size of outputs and another one for the computational time of the action. It generates a list of actions, each one with a unique id and its corresponding randomly generated parameters. The workflow generator uses this list of actions as a pool of actions from where to select the actions to compose the workflows.

The Workflow Generator

The **Workflow Generator** is a little more complex than the Action Generator. Its purpose is to generate a workflow (DAG) selecting nodes from the pool of actions created by the Action Generator. Roughly, the algorithm does the following:

1. It selects n nodes from the composition of all previous workflows up to that point in the history of workflows. It takes good care that if any pair of nodes nodes a, b among the n nodes are related between each other (antecesor/succesor) relationship, then the nodes in between them are also included among the n nodes.

2. It randomly selects $workflow\_size - n$ new actions from the pool of actions that are not nodes in the DAGs of the workflows already generated..

3. It creates two normal distributions with parameters provided in the configuration. Let one distribution be the *childrenDist* and the other one *parentDist*. For each action a selected to be part of the new workflow, if action a was selected from previous workflows, use *childrenDist* to generate the number of children that this action will have. Otherwise if action a is a new action, use, *childrenDist* and *parentDist* to generate the number of children and the number of parents that this action will have, respectively.

4. Use a greedy algorithm to create a directed acyclic graph that satisfies the constraints on the number of children and number of parents a node will have in the best possible way and return the corresponding workflow.

The parameters used to define the structure of the DAGs are good enough to produce most of the varieties of histories of workflows possible to imagine. For example, Figure 5.1 shows how to generate tree DAGs with very high probability if the

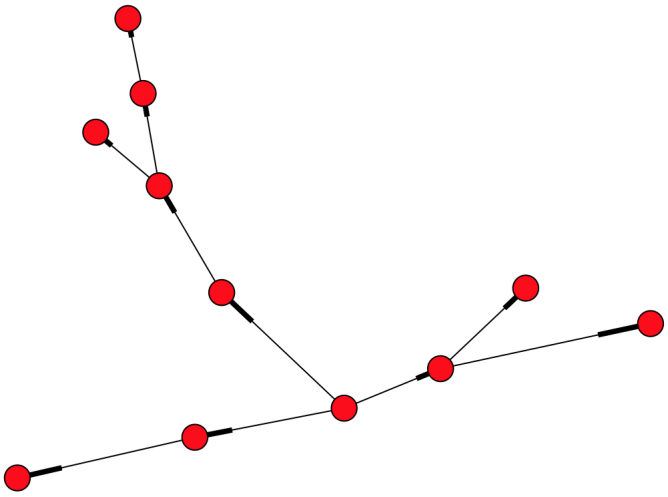

Figure 5.1: Example of a tree workflow that is produced with parameters:

number of parents that an action can have is restricted to only one. (The DAG in Figure 5.1 was generated with parameters: nb_children.mean = 2, nb_children.std = 1, nb_parents.mean = 1, nb_parents.std = 0.00001). Figure 5.2 shows how to generate DAGs with more complex dependency relationships between nodes. (The DAG in Figure 5.2 was generated with parameters: nb_children.mean = 2.1, nb_children.std = 0.0001, nb_parents.mean = 2.1, nb_parents.std = 0.0001.) Last, Figure 5.3 shows an example of the varieties of graph produced when the variance of the distributions is increased. (The DAGs in Figure 5.3 were produced with parameters: nb_children.mean = 2.1, nb_children.std = 4.5, nb_parents.mean = 2.1, nb_parents.std = 4.5).

5.1.2 Ideal Execution Time calculation

Most of the research done in the area of scientific workflows becomes relevant and applicable, not in the context of the **Decision System** of Pingo, but in the context of the metrics to evaluate the **Decision System**. Most of the previous research has focused in finding optimal and near-optimal solutions to the problem

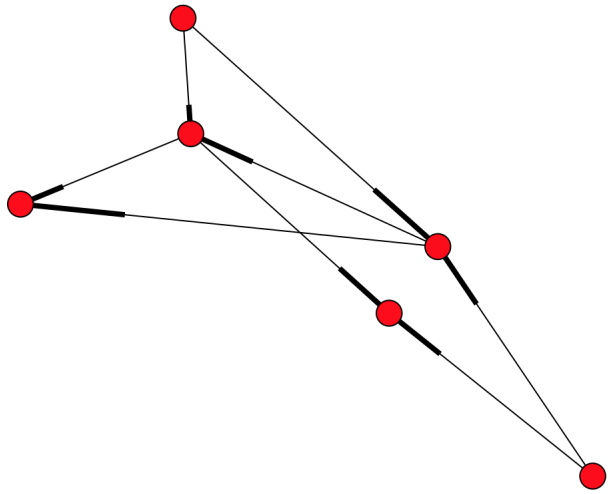

Figure 5.2: Example of a more complex DAG

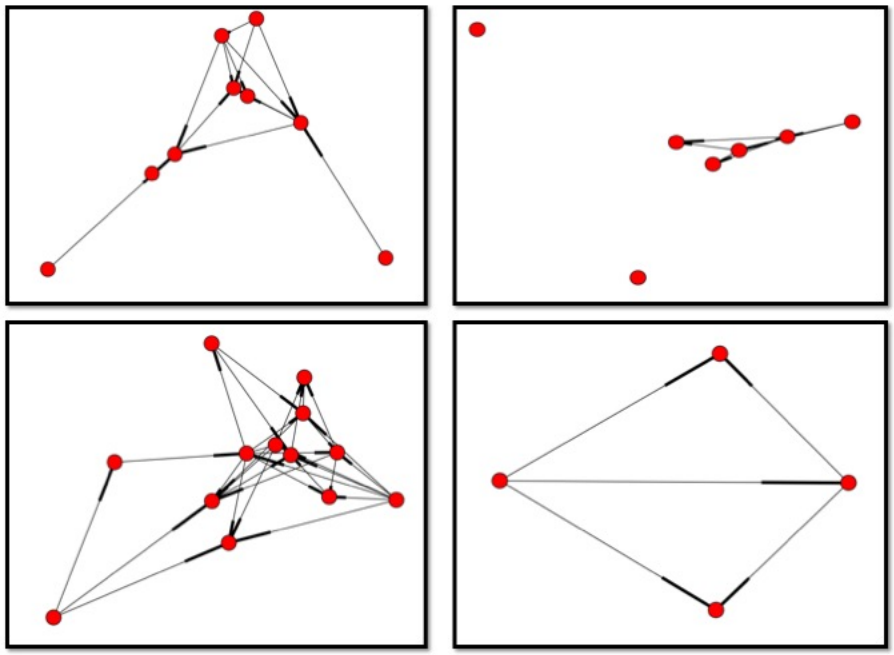

Figure 5.3: Example of four DAGs with variance of distributions increased

of scientific workflows with constrained space, assuming that we know the entire history of the workflows that will be submitted to the system from the beginning. In Pingo, the **Decision System** does not know the end from the beginning. Instead, it only uses the previous workflows submitted to the system to predict how the future workflows might look like. This different approach makes sense in fast-paced research settings where researchers don't know from the start the exact process (and hence the workflow) that they will follow in their research.

For evaluation purposes it is possible to know the end from the beginning and all the research from Chapter 2 becomes more directly relevant to our problem. As a future endeavor, it would be good to do a more throughout exploratory work on the scientific workflows research literature in order to apply the most relevant produced results to the evaluation of the Pingo system.

This section does its little contribution to the evaluation methodology of scientific workflow systems. It defines what is **ideal execution time of a history of workflows**. It also discusses an algorithm on how to compute it.

Consider a sequence $H = (W_1, ..., W_n)$ of n workflows that have been submitted to the system over time, and let S be the storage capacity of the file system. In sequence H time will be a discrete magnitude with n time steps, and we say that $t = i$ corresponds to the time when workflow W_i is submitted to the system. The effective life of dataset d in sequence H is defined as a tuple of time steps $(t1, t2)$, with $t1$ being the time step corresponding to the first workflow that creates dataset d, and $t2$ being the time step corresponding to the last workflow that makes mention of dataset d in H. See Figure 5.4 for an example of the effective life diagram of an hypothetical history of workflows. Time steps are represented by vertical blue lines, and datasets are represented by horizontal black lines. Dotted ranges in the black lines (as in dataset $d3$ and $d15$ in the figure) mean that the datasets were not part of

the workflows submitted at the corresponding time steps.

An ideal system would only keep datasets in storage exactly for the duration of their effective life. Let D_H be the set of all datasets of a history H. In our analysis, let $s : D_H \rightarrow (N)$ be a function where $s(d_i)$ represents the storage that dataset d_i occupies on the filesystem. Let also $c : D_H \rightarrow (N)$ be a function where $c(d_i)$ represents the average time that it takes to compute dataset d_i by its corresponding action. At any given time t there will be a set M_t of datasets in storage, occupying an space $S' = \sum_{d \in M_t} s(d)$. S' must be less than S, otherwise, it is needed to remove some of the datasets present at that time to satisfy space constraint S.

Let $b(d)$ and $e(d)$ be the start time and end time, respectively of the effective life time of dataset d. Since every dataset needs to exist for at least one time step (the time step corresponding to the workflow that created the dataset), it is only possible to consider removing datasets from $N_t \subseteq M_t$, where $d \in N_t$ if $b(d) < t$. If the storage occupied by datasets in N_t is less than $S' - S$, this will lead to a system failure that can only be effectively prevented by increasing S or making the submitted workflow smaller. For simplicity, this analysis ignores such situation.

Determining which datasets to keep and which ones to delete so as to optimize the total execution time of the history of workflows is an NP-Hard problem. For an example reference, see the analysis of the problem of optimizing workflow computations with constrained storage for the case of two workflows as presented by Zohrevandi and Bazzi (2013). Because of that, we relax the constraints a little bit and ignore dependencies among datasets, so that what we call an **ideal solution** will not be an **optimal solution**. Algorithm 7 describes how to compute the ideal computational time of a sequence of workflows. Note that it uses Algorithm 8 as a subroutine.

There are to ways in which Algorithm 7 does not find a global optimal value:

1. It does not take into account dependencies among datasets (as defined by the

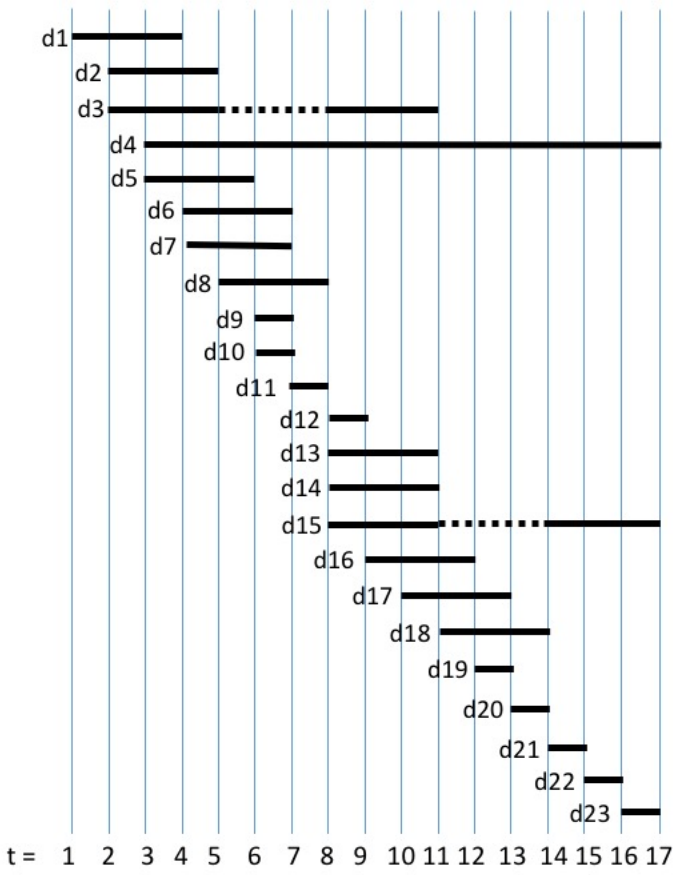

Figure 5.4: Effective life diagram of datasets of hypothetical history of workflows.

DAGs that represent the workflows).

2. At each time step t, it greedily finds a **local** minimum time that is added to the total result.

But most good heuristic algorithms that can be proposed to find an ideal computation time of a sequence of workflows will be valid for our purposes, for as long as they provide results that are always lower than the real computation times taken by Pingo in computing those sequences of workflows. As further research improves the **Decision System** of Pingo so that its prediction capabilities begin to improve, further research will be needed in order to close the gap between the concepts of the

Algorithm 7 Ideal Computation Time algorithm

procedure IDEALCOMPUTATIONTIME($S, H = (W_1, ..., W_n), b, e, c$)
  $totalTime \leftarrow 0$
  for t from 1 to n **do**
    Let $M_t = \{d \,|\, b(d) \leq t \wedge e(d) \geq t\}$
    Let $N_t = \{d \,|\, b(d) < t \wedge e(d) \geq t\}$ ▷ $N_t \subseteq M_t$
    Let $P_t = \{d \,|\, d \in W_t\}$ ▷ $d \in M_t \not\Longrightarrow d \in W_t$
  Let $M_H = (M_1, ..., M_n)$
  Let $N_H = (N_1, ..., N_n)$
  Let $P_H = (P_1, , , , , P_n)$
  for t from 1 to n **do**
    Find subset A of N_t such that $\sum_{d \in A} s(d) \leq S$ and $\sum_{d \in A} computationTimeLeft(d, P_H, t, n)$ is maximum among all possible subsets of N_t. (This is the classic Knapsack problem which has pseudopolynomial solutions and good approximations).
    Let et be the time that workflow W_t will take to compute its actions, assuming that A is the set of datasets currently present in storage.
    $totalTime \leftarrow totalTime + et$
  return $totalTime$

Algorithm 8 Computation Time Left Subroutine

procedure COMPUTATIONTIMELEFT(d, P_H, m, n, c)
  $timeLeft \leftarrow 0$
  for t from m to n **do**
    if $d \in P_t$ **then**
      $timeLeft \leftarrow timeLeft + c(d)$
  return $timeLeft$

ideal and the **optimal** computation time.

5.2 Evaluation Experiments

In evaluation experiments, the main focus becomes evaluating the simple and adaptive algorithm families under different parameters. All the experiments follow the same process: (1) Generate a history of workflows using the generator that we have designed. (2) Submit that history of workflows to the system, one workflow at a time, and see how much computation time does the system take using different algorithms. (3) When submitting the workflows to the system, always wait for the previous submitted workflow to compute in order to submit the next workflow. The

purpose of that practice is to measure the gains in decrease of computation time that the algorithms provide under the most ideal circumstance. (4) Estimate the computation time it would have taken the system if we had used no algorithm.

Each experiment was run 5 times using the same configuration parameters and the results were averaged. The first experiment explores the effect of changing the storage constraint in the total computation time of a history of workflows. It should be expected that as the the storage in the system increases, the computation time reduces. In the second experiment the storage constrain remains fixed. The second experiment explores instead how the different algorithms behave under different kinds of workflows, focusing on changing the percentage of actions from previous workflows that are included in new workflows.

The experiments run in the Hadoop distribution service provided by Amazon Cloud Computing, with the default settings, with the only difference that we also include Apache Oozie among the packages to install (it is not included by default). Because of the limitations in time to run the experiments, and in storage (more storage costs more money), the computations of the actions that we run take only a few seconds, and their output is also relatively small. The main interest running the experiments is to compare how different algorithms will compare against each other under different types of loads. I ran a couple of experiments with more real-life computation time and output sizes and confirmed that the results stay consistent.

5.2.1 How computation time is affected by the amount of storage in the system

This experiment reports how the computation time is affected by the amount of storage space available in the system. The complete report of the parameters used to configure Pingo and the workflows' generator is in Appendix C. The number of actions for the experiment is 300, with an average of 10 actions per workflow,

and each workflow repeating **half** the actions from previous workflows. This means that on average, a workflow produced by the generator uses 5 new actions from the 300 possible actions it can choose, making the length of the generated sequences of workflows to be around 60.

Since each action will take an average of 10 seconds to finish, and its output will have an average size of 10MB, the total average time of our workflows should be 3000 seconds, and the average storage needed to save all of the actions is 3000MB. In the experiment, the available space of the system is modified at increments of 500MB, starting at 500MB.

Figure 5.5 reports the computation time of the runs as a percentage of the computation time of the runs if all computations are performed. There are many interesting comments and conclusions that are obtained from those results. First of all, as expected, the computation time of the history of workflows decreases as the available storage in the system increases. The effect can be more easily seen in the simple algorithm. Another noticeable result is that as the storage capacity of the system increases, the simple algorithm approaches the performance of the adaptive algorithm.

Another important conclusion that is derived from the results reported in Figure 5.5 is that the adaptive algorithm is fairly robust. Its performance is not affected as much by the available storage in the system as it is the case with the simple algorithm. It is remarkable to see how it achieves excellent performance even with storage capacity at 500MB. To place the result in perspective, in the runs with storage capacity at 500MB, the adaptive algorithm performed 6 percent worse than in the runs with storage capacity at 2000MB, where it achieved its best results.

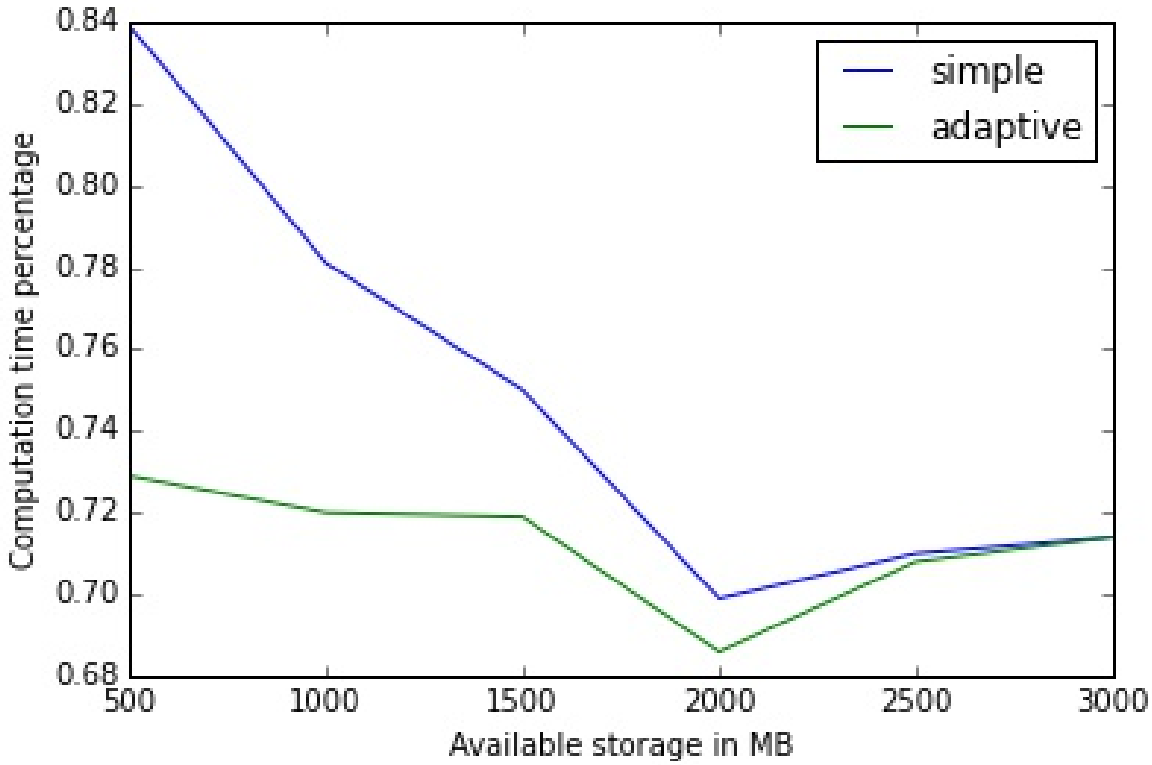

Figure 5.5: Computation time as storage increases

5.2.2 How computation time is affected by the types of workflows in the history of workflows

It is possible to produce many different kinds of workflows by playing with the parameters of the workflow generator. This experiment only focuses in the parameter that determines the percentage of actions from previous workflows submitted to the system that are used by new workflows submitted to the system. In the experiment the parameter changes at 10 percent increments, starting at 5 percent, and ending at 55 percent. The amount of available space of the system is fixed at 500MB. The complete report of the parameters used to configure Pingo and the workflows' generator in Appendix C.

There are no surprises in the results of Figure 5.6. Both, the simple and adaptive

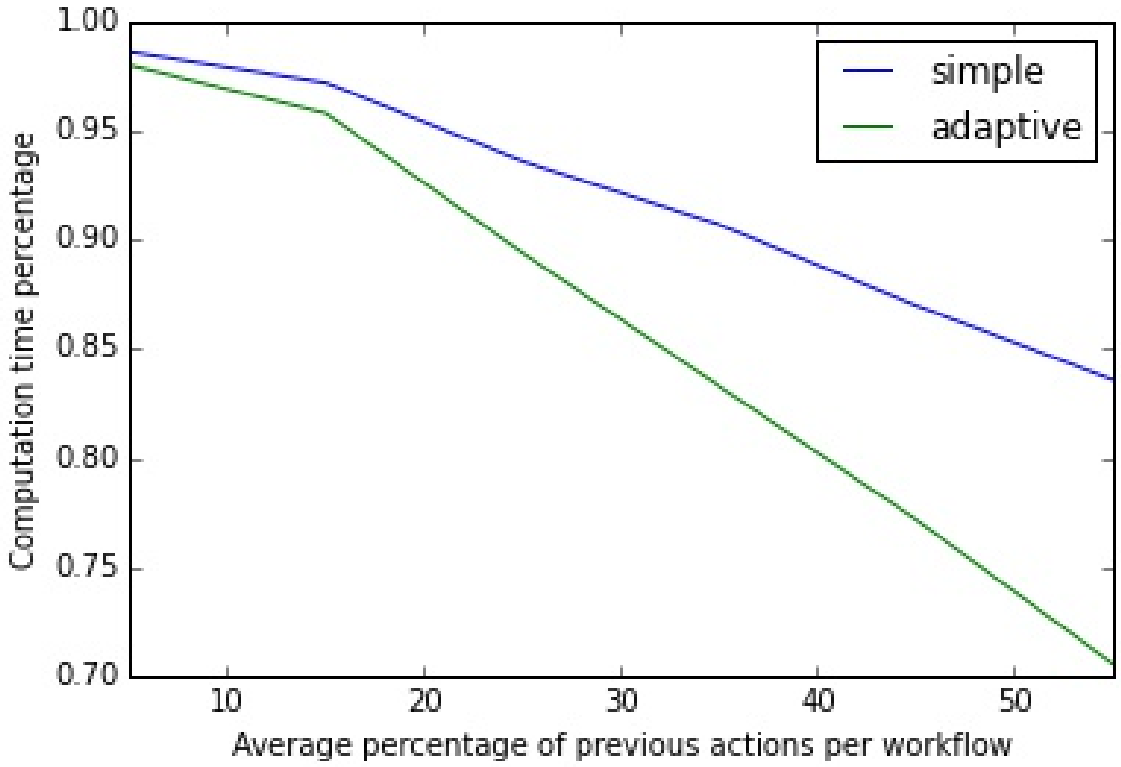

Figure 5.6: Computation time percentage as percentage of actions from previous workflows increases

algorithms decrease their computation time as the percentage of previous actions that a workflow includes increases. This result was expected, since the more previously computed actions a workflow includes, the more opportunities the Pingo system has to skip those computations since is likely that the outputs of those computations are available in storage.

5.3 Conclusions of our evaluations

From the evaluations of the algorithms it can be concluded that the adaptive family of algorithms will perform better in the most basic scenarios than the simple algorithms by themselves. In both algorithms, the definition used to assign value to each action was very simple, and it did not take into account the computation time

of an action, only its frequency of usage. Because of that, there is still room for improvements on the already promising results.

For future evaluations, I propose the creation of more complex adaptive algorithms. The current algorithm works well when the **look back** parameter does not probabilistically change much. In real life scenarios, more complex adaptive algorithms should be able to detect the rate at which the look back parameter might be changing, and adapt to it.

I also want to warn that the evaluation results should be taken with a grain of salt. The workloads used were synthetically generated. More efforts should be done in the future to collect workloads that correspond to real-life scenarios.

Chapter 6

FUTURE RESEARCH

There are many areas where the Pingo system can improve. The first great improvement can happen in the evaluation methodology. All the workflows used to evaluate Pingo were probabilistically generated. The generation system is designed with flexibility in mind so that it can generate different kinds of workflow loads to the system. But there is no substitute to real data. Unfortunately, the amount of real data available to the researcher was not enough and was not completely relevant to the specific problem that Pingo attempts to solve. More real data needs to be gathered in order to produce more accurate evaluations on the performance of the decision algorithms of the system.

Another area of research is the implementation of more sophisticated adaptive decision algorithms that can handle the most diverse types of workloads submitted to the system. In this respect, this area of research is very much interlinked to the previous proposition to gather a more diverse dataset of workflows.

Another important area of future research has to do with the design of the system. As we have seen, the system is nothing more than a composition of smaller independent subsystems that poll data from Hadoop or from a database that keeps the state of actions and datasets. More research is needed on how to tune the parameters that control the frequency of this polling events, so that each independent subsystem carries its own processing computations as effectively as possible without putting to much strain in the underlying database cluster.

I am sure that the avid reader of this report will have identified some other opportunities in which the system can be improved or expanded. I gladly accept any

related commentaries and suggestions about it. The most rewarding news for me as a researcher is that the system I have created is used and expanded and adapted to different needs by others. I certainly have attempted to design it with that goal in mind.

REFERENCES

Adams, I. F., D. D. Long, E. L. Miller, S. Pasupathy and M. W. Storer, "Maximizing efficiency by trading storage for computation.", in "HotCloud", (2009).

Ali, W., S. M. Shamsuddin and A. S. Ismail, "A survey of web caching and prefetching", Int. J. Advance. Soft Comput. Appl **3**, 1, 18–44 (2011).

Altintas, I., C. Berkley, E. Jaeger, M. Jones, B. Ludascher and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows", in "Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on", pp. 423–424 (IEEE, 2004).

Barker, A. and J. Van Hemert, "Scientific workflow: a survey and research directions", in "International Conference on Parallel Processing and Applied Mathematics", pp. 746–753 (Springer, 2007).

Bellare, M. and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols", in "Proceedings of the 1st ACM conference on Computer and communications security", pp. 62–73 (ACM, 1993).

Bharathi, S., A. Chervenak, E. Deelman, G. Mehta, M.-H. Su and K. Vahi, "Characterization of scientific workflows", in "Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on", pp. 1–10 (IEEE, 2008).

Bray, T., "The javascript object notation (json) data interchange format", (2014).

Cheng, J., D. Zhu and B. Zhu, "A new algorithm for intermediate dataset storage in a cloud-based dataflow", in "International Workshop on Frontiers in Algorithmics", pp. 33–44 (Springer, 2015).

Coron, J.-S., Y. Dodis, C. Malinaud and P. Puniya, "Merkle-damgård revisited: How to construct a hash function", in "Annual International Cryptology Conference", pp. 430–448 (Springer, 2005).

Dean, J. and S. Ghemawat, "Mapreduce: simplified data processing on large clusters", Communications of the ACM **51**, 1, 107–113 (2008).

Deelman, E., D. Gannon, M. Shields and I. Taylor, "Workflows and e-science: An overview of workflow system features and capabilities", Future Generation Computer Systems **25**, 5, 528–540 (2009).

Deelman, E., K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation", Future Generation Computer Systems **46**, 17–35 (2015).

Fox, G. C. and D. Gannon, "Special issue: Workflow in grid systems", Concurrency and Computation: Practice and Experience **18**, 10, 1009–1019 (2006).

Gifford, D. K. and J. M. Lucassen, "Integrating functional and imperative programming", in "Proceedings of the 1986 ACM conference on LISP and functional programming", pp. 28–38 (ACM, 1986).

Gil, Y., E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau and J. Myers, "Examining the challenges of scientific workflows", Ieee computer **40**, 12, 26–34 (2007).

Hollingsworth, D. and U. Hampshire, "Workflow management coalition: The workflow reference model", Document Number TC00-1003 **19** (1995).

Islam, M., A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann and A. Abdelnur, "Oozie: towards a scalable workflow management system for hadoop", in "Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies", p. 4 (ACM, 2012).

Liu, J., E. Pacitti, P. Valduriez and M. Mattoso, "A survey of data-intensive scientific workflow management", vol. 13, pp. 457–493 (Springer, 2015).

Merkle, R., "A certified digital signature", in "Advances in Cryptology?CRYPTO?89 Proceedings", pp. 218–238 (Springer, 1990).

Oinn, T., M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin *et al.*, "Taverna: lessons in creating a workflow environment for the life sciences", Concurrency and Computation: Practice and Experience **18**, 10, 1067–1100 (2006).

Ramakrishnan, A., G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers and M. Samidi, "Scheduling data-intensiveworkflows onto storage-constrained distributed resources", in "Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on", pp. 401–409 (IEEE, 2007).

Ramamritham, K. and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems", Proceedings of the IEEE **82**, 1, 55–67 (1994).

Singh, G., M.-H. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz and G. Mehta, "Workflow task clustering for best effort systems with pegasus", in "Proceedings of the 15th ACM Mardi Gras conference: From lightweight mash-ups to lambda grids: Understanding the spectrum of distributed computing requirements, applications, tools, infrastructures, interoperability, and the incremental adoption of key capabilities", p. 9 (ACM, 2008).

Smith, A. J., "Cache memories", ACM Computing Surveys (CSUR) **14**, 3, 473–530 (1982).

Taylor, I. J., E. Deelman, D. B. Gannon and M. Shields, *Workflows for e-Science: scientific workflows for grids* (Springer Publishing Company, Incorporated, 2014).

Turing, A. M., "On computable numbers, with an application to the entscheidungsproblem", Proceedings of the London mathematical society **2**, 1, 230–265 (1937).

Wang, J., "A survey of web caching schemes for the internet", ACM SIGCOMM Computer Communication Review **29**, 5, 36–46 (1999).

White, T., *Hadoop: The definitive guide* (" O'Reilly Media, Inc.", 2012).

Yu, J. and R. Buyya, "A taxonomy of workflow management systems for grid computing", Journal of Grid Computing **3**, 3-4, 171–200 (2005).

Yuan, D., Y. Yang, X. Liu and J. Chen, "On-demand minimum cost benchmarking for intermediate dataset storage in scientific cloud workflow systems", Journal of Parallel and Distributed Computing **71**, 2, 316–332 (2011).

Yuan, D., Y. Yang, X. Liu, G. Zhang and J. Chen, "A data dependency based strategy for intermediate data storage in scientific cloud workflow systems", Concurrency and Computation: Practice and Experience **24**, 9, 956–976 (2012).

Zohrevandi, M. and R. A. Bazzi, "The bounded data reuse problem in scientific workflows", in "Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on", pp. 1051–1062 (IEEE, 2013).

APPENDIX A

SYSTEMS' USER GUIDE

For a complete reference on how to install and use the system, please see the documentation at http://github.com/jadielam/scientific-workflows.

APPENDIX B

IMPLEMENTATION OF WORKFLOW GENERATOR ALGORITHM

```
import numpy as np
import networkx as nx
import random
import string

def workflow_generator(previous_workflows_union, workflow_size,
            nb_previous_actions, total_nb_actions, conf):

    #1. Algorithm to determine which are the previous actions to include.
    top_sort = nx.algorithms.dag.topological_sort(previous_workflows_union)
    previous_actions_indexes = random.sample(range(len(top_sort)),
        min(nb_previous_actions, len(top_sort)))
    previous_actions_to_include = []
    C = list(previous_actions_indexes)
    while len(C) > 1:
        new_C = []

        for i in range(1, len(C)):
            source = top_sort[C[0]]
            target = top_sort[C[i]]
            try:
                shortest_path =
                    nx.algorithms.shortest_paths.shortest_path(previous_workflows_union,
                    source, target)
                if len(shortest_path) > 1:
                    previous_actions_to_include.extend(shortest_path)
                else:
                    new_C.append(i)
            except:
                new_C.append(i)
        C = new_C

    if len(C) == 1:
        previous_actions_to_include.append(top_sort[C[0]])

    #2. Algorithm to determine the parameters of both the previous actions
    #and the new actions to include.
    actions_params = {}
    nb_children_mean, nb_children_std = conf['nb_children']['mean'],
        conf['nb_children']['std']
    nb_parent_mean, nb_parent_std = conf['nb_parent']['mean'],
        conf['nb_parent']['std']

    for action_id in previous_actions_to_include:
```

```
    nb_children = int(abs(np.random.normal(nb_children_mean,
        nb_children_std)))
    actions_params[action_id] = { 'nb_children': nb_children }

new_actions_lower_bound = len(previous_workflows_union.node)
new_actions_upper_bound = min(new_actions_lower_bound + workflow_size
    - nb_previous_actions, total_nb_actions)
new_actions = range(new_actions_lower_bound, new_actions_upper_bound)
for action_id in new_actions:
    nb_children = int(abs(np.random.normal(nb_children_mean,
        nb_children_std)))
    nb_parents = int(abs(np.random.normal(nb_parent_mean,
        nb_parent_std)))
    actions_params[action_id] = { 'nb_children': nb_children,
        'nb_parents': nb_parents}

#3. Create workflow graph
#3.1 Add subgraph from previous workflows
workflow =
    nx.DiGraph(previous_workflows_union.subgraph(previous_actions_to_include))

#3.2 Add new items
for action_id in previous_actions_to_include:
    nb_children = actions_params[action_id]['nb_children']
    i = 0
    j = 0
    index_permutations = np.random.permutation(range(len(new_actions)))
    while i < nb_children and j < len(new_actions):
        tentative_id = new_actions[index_permutations[j]]
        nb_parents = actions_params[tentative_id]['nb_parents']
        if nb_parents > 0:
            actions_params[tentative_id]['nb_parents'] = nb_parents - 1
            workflow.add_edge(action_id, tentative_id)
            i = i + 1
            j = j + 1
            continue
        else:
            j = j + 1
            continue

for action_id in new_actions:
    nb_children = actions_params[action_id]['nb_children']
    i = 0
    j = 0
    index_permutations = np.random.permutation(range(len(new_actions)))
    workflow.add_node(action_id)
    while i < nb_children and j < len(new_actions):
        tentative_id = new_actions[index_permutations[j]]
        nb_parents = actions_params[tentative_id]['nb_parents']
        if nb_parents > 0 and tentative_id != action_id and
```

```
            tentative_id not in nx.algorithms.dag.ancestors(workflow,
            action_id):
            actions_params[tentative_id]['nb_parents'] = nb_parents - 1
            workflow.add_edge(action_id, tentative_id)
            i = i + 1
            j = j + 1
            continue
        else:
            j = j + 1
            continue

#5. Return workflow
return workflow
```

APPENDIX C

PARAMETERS OF EXPERIMENTS

```
{
"nb_actions": 300,
"action_size": {
"mean": 10,
"std": 3
},
"action_time": {
"mean": 10,
"std": 3
},
"workflow_size": {
"mean": 10,
"std": 4
},
"previous_actions": {
"mean": 0.5,
"std": 0.1
},
"nb_children": {
"mean": 2.1,
"std": 4.5
},
"nb_parent": {
"mean": 2.1,
"std": 4.5
},
"workflow": {
"name": "workflow",
"version": "1.0",
"main_class_name": "io.biblia.workflows.job.Main",
"action_folder": "/user/hadoop/examples/apps/scientific-workflows",
"nameNode": "hdfs://ec2-54-80-213-20.compute-1.amazonaws.com:8020"
}
}
```

Figure C.1: Parameters of Workflows' Generator in Experiment 1

```
{
"nb_actions": 300,
"action_size": {
"mean": 10,
"std": 3
},
"action_time": {
"mean": 10,
"std": 3
},
"workflow_size": {
"mean": 10,
"std": 4
},
"previous_actions": {
"mean": [0.5, 0.15, 0.25, 0.35, 0.45, 0.55],
"std": 0.1
},
"nb_children": {
"mean": 2.1,
"std": 4.5
},
"nb_parent": {
"mean": 2.1,
"std": 4.5
},
"workflow": {
"name": "workflow",
"version": "1.0",
"main_class_name": "io.biblia.workflows.job.Main",
"action_folder": "/user/hadoop/examples/apps/scientific-workflows",
"nameNode": "hdfs://ec2-54-80-213-20.compute-1.amazonaws.com:8020"
}
}
```

Figure C.2: Parameters of Workflows' Generator in Experiment 2

BIOGRAPHICAL SKETCH