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 October 2016 by the
Graduate Supervisory Committee:

Rida Bazzi, Chair
Dijiang Huang

ARIZONA STATE UNIVERSITY

December 2016

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 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 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

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

LIST OF TABLES

LIST OF TABLES

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 compute on the data it receives as input from a previous step. It has been widely adopted to model and express these computations as a directed acyclic graph (DAG) of computations, or a workflow (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 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 their creation time 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)), and there have been efforts in the research of algorithms that choose to save the right intermediate datasets in order to optimize the computation time of the workflows to be computed (see Zohrevandi and Bazzi (2013)). But the research effort has been mainly directed to algorithms 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 in 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 date reuse functionality. There is the need for a system that can manage highly scalable computations and that also knows about the concepts of 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. Pingo also takes care of the management of storage and provenance information of the intermediate datasets. Our 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 Our Contributions

My system has a commonality with previous systems in that it stores intermediate datasets produced by workflows with the purpose of skipping the computations of future workflows. In order to decide which datasets should be kept or not, I have opted

for the approach of constraining the storage space available to the system, keeping the storage space of the datasets that will optimize the computational time of workflows under such constraint. But differing from previous systems, Pingo does it in a "reactive way". That is, most previous systems decide which intermediate datasets to keep on storage with foreknowledge of the definition of the future workflows that will be submitted to the system. In our case, we 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 our work is that, to the best of our 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 we have also designed it to be an scalable multi-user system. There are other smaller contributions scattered throughout the report. For example, we propose a hashing mechanism as an efficient alternative to determining the provenance of datasets produced by actions.

Our presentation of Pingo is divided as follows: Chapter 3 contains a throughout explanation of the tradeoffs and balancing acts that need to be addressed (or at least considered) when designing a system that promises a functionality similar to the one we are attempting in our project. Chapter 2 contains a discussion of previous systems that are capable of managing the computation of scientific workflows. It also contains a section that discusses recent advances in the area of optimization algorithms for the data reuse problem. Chapter 4 provides a detailed description of the implementation of the system. A central section of the chapter is the presentation of two families of decision algorithms that determine which intermediate datasets should be kept in

storage at any given time. In Chapter 5 I propose a methodology to evaluate the performance of the decision algorithms proposed in 4. I also report on the actual results of evaluating Pingo using such methodology. Finally, in Chapter 6 I talk about the new possibilities of research that can improve and add to 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. From a conceptual point of view, workflows can be divided into two types: business workflows and scientific workflows. 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 a directed acyclic graph (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). Such requirement introduces the opportunity to store intermediate datasets to optimize the execution of computations of workflows.

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

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 managing 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 the e-Science project(Deelman *et al.*, 2009), Kepler(Altintas *et al.*, 2004) and Taverna(Oinn *et al.*, 2006). 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 recent 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 Directed Acyclic Graphs (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, storing all the intermediate data generated during workflow executions may cause high storage cost. On the contrary, if we delete all the intermediate data and regenerate them every time when ever needed, the computation cost of the system may also be very high. Because of that, good tradeoffs need to be found to solve the problem. They are sometimes achieved by constraining the amount of storage space available to the system, and other times

by assigning cost to both storage space and computation time and designing decision algorithms that optimize for that cost.

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

FOUNDATIONAL DESIGN PRINCIPLES OF THE SYSTEM

This chapter discusses the design of the **Pingo** system and the reasons behind it. I have organized the discussion from a functional standpoint, discussing the design as it relates to the main functionalities of the system, the most essential functionality being the ability to receive the submission of workflows of actions and to carry out the computations represented by such actions.

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, we need to first precisely define what is an **Action** and what is a **Workflow**.

3.1.1 Actions

The way we define what an **action** is will dictate the possibility of designing practical functionality that will allow us to optimize the time spent computing a workflow.

Simplifying things, we can think of an action as a pure function $f : A \to B$. Suppose that we are asked to compute $f(a)$. If we have previously computed $b = g(c)$, and we can determine that $g = f$ and that $c = a$, then we can skip the computation of $f(a)$ (if it is that we still have b among us. 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 more difficult and has deep theoretical ramifications (it is computationally impossible in the most broad sense, since the Halting Problem can be reduced to this problem).

But comparing our actions to pure mathematical functions is not completely accurate. **?** says that a pure function is a function that "always evaluates the same result value given the same argument values. The function result cannot depend on any hidden information or state that may change while program execution proceeds." In some cases, as we will see in the next chapter, there will be nothing stopping our actions from reading values from the environment, which in many cases is not under the control of the **Pingo** system.

Because of these difficulties, we need to devise a more practical approach to determine the equivalency between two actions. That approach needs to be derived from a closer look to the functionality that we want our ideal system to embody. The system will receive descriptions of actions as structured text. Those descriptions 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. The system uses that descrip-

tion to execute a corresponding computation which will produce an output as a file (or as a collection of files). We will be happy if our measure of equivalency between two actions satisfies the following two simple properties:

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

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

Both properties are challenging to carry out in a real setting. The way we have worded the first property allows for the possibility of partial success that can be assessed using accuracy measures. As we have mentioned before, achieving 100% accuracy is theoretically impossible, and achieving high accuracy is challenging, since there are infinitely many ways to express two equivalent computations. In devising a measure of equivalency between two actions, we will be happy if it can achieve consistent accuracy in at least the most common use cases.

The second property must be satisfied all the time for our measure of equivalency to be considered usable. Still, as will be described later in the chapter, it is convenient to interpret the phrase "all the time" not in a mathematically strict sense, but only in a practical sense. Another challenge in order to satisfy the second property is that computations performed by actions might not be **pure functions**. (For a more throughout discussion on pure functions, see the discussion of Jones (2003) on the Haskell programming language). If the function output depends on hidden information or state that may change while program execution proceeds, or between different executions of the program (i.e.: "randomization" or access to outside environmental variables), then there is no guarantee that the measure of equivalence will satisfy Property 2. Because of that, the end user should have the ultimate responsibility of determining if a task should be recomputed even if there is present in the system an

output dataset of an equivalent task. If the end user thinks that the action description of an Action A to be submitted to the system 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 action A's computation must be executed.

The description of an action submitted to the system becomes very relevant, since it is the starting point to determine equivalence between actions. A description schema that is very descriptive (pardon the redundancy) provides more elements to determine the equivalence between two actions. As such it has the intended consequence of increasing the percentage of actions that can be determined to be equivalent. On the other hand, a very descriptive schema is burdensome to the final user, since in practice it causes the end user to spend more time writing the descriptions of the actions to be submitted to the system. A good compromise needs to be found to this two conflicting aspects.

3.1.2 Workflows

In essence, a **workflow** is a directed acyclic graph (DAG) where nodes correspond to actions or original datasets (datasets not derived from the computation of an action) and a directed edge from a node a to a node b means that the output of action A is used as as input by action B. 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 we already know, can start executing whenever computational resources are available. The fact that the workflow is a DAG (i.e.: has no cycles) guarantees that all actions in the workflow will be computed at some point in time, given that there are no malformed computations or failures. The discussion of how to handle failures belongs to the next chapter.

Before discussing more serious matters regarding workflows, we need to agree on

some notational conventions. We identify actions with the lower letter a (a_1, a_2, ..., a_n); and original datasets with the lower letter o, and derived datasets with the lower letter d. We also use functional notation 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, we represent action a_1's output d_1 as: $d_1 = a_1(o_1, d_2, a_3(o_2))$.

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 dataset d_1 is in storage so that without having to calculate all the computations defined by the workflow where a_1 is? As a starting point, for every dataset d_i kept in storage, we need to keep some accounting with the definition of the workflow that produced d_i. Then we would need to search over those definitions until we 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 algorithm described in 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 a and d_i. If it finds a dataset d_i for whose DAG is equivalent to the DAG that produced a, it returns d_i, 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 queues to compare them, they correspond to each other. The next chapter (the implementation chapter) discusses

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
```

how that assumption is valid for some 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 $O(m(M+N))$. Indexing the metadata of the stored datasets in a clever way, we can reduce that to a running time of $O(t(M+N))$ with $t << m$.

The indexing approach can be problematic to implement for at least two reasons: First, t will increase without bounds as the system ages. The space occupied by the index of the metadata will also increase, making it difficult to keep it all in memory for fast computations. It is true that the index can be purged from time to time so that it only keeps relevant datasets in it, but it would represent added complexity that needs to be implemented. The second reason is that indexing the metadata of a graph datastructure is not a trivial task. (Simmhan *et al.* (2005))

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.

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 the next chapter. But there are enough elements 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 the datasets in the system is an action that can be done in constant time if the datasets are properly indexed. Secondly, since it is impossible to know before hand the set of keys that are going to be hashed, the possibility of perfect hashing is discarded, and at least an upper bound analysis of what would be the probability of a clash between the hash signatures of non-equivalent actions is in place.

An exhaustive mathematical analysis goes beyond the scope of this dissertation, but we do can make some educated estimations. Assume that n is the number of bits of the hash signatures produced by the hashing algorithm, and that D is the number of datasets currently in storage. The question to answer is: What is the probability of any two non-equivalent datasets of having the same hash value? There are 2^n possible hash values. Assuming that all of them have the same probability of occurring, then the probability that all the N hash values are different, $1 - p(N)$ is:

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

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 $4.135580766728708e - 13$. This number can be considered acceptable for most practical purposes, but if for some reason it becomes unacceptable, the mechanism is still valid if we add a verification step to the search process. The first step finds all the datasets whose corresponding hash value is equal to the one of the action under consideration. In the second step, we can compare the workflows that produced each of the candidate datasets to the workflow of the submitted action to exactly determine if they are equivalent. A good hash function allows for the assumption that the number of candidate datasets is 1.

3.2 Bounded storage space

Placing a constraint of 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 to carry out the execution of the decisions.

3.2.1 The History of Workflows

The concept of **History of Workflows** becomes useful if 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 things from the history of workflows. Because of that, there needs to be an API layer that provides varied ways 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 Decision System

Pingo's core functionality is the Decision System that determines which datasets need to remain and which datasets need to be deleted from storage. The Decision System 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 it will be able to complain with the bounded storage constraint as well as the first option, but uses the computational resources less. 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. 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. $s : D \rightarrow \mathtt{N}$, where $s(d_i)$ is the storage space in megabytes that dataset d_i occupies in the file system.

4. $t : A \rightarrow \mathtt{N}$, where $t(a_i)$ is the computational time in seconds that it takes to 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.

5. F, the amount of space that we need to free on the file system.

The algorithm outputs a set $B \subset D$ 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.

Chapter 4

IMPLEMENTATION OF THE SYSTEM

4.1 Birdeye overview

The Pingo system is a group of independent processes that together take care of the management of scientific workflow computations and their outputs. Firstly I give a bird-eye overview of it, and in the remaining of the chapter I explain each of the components with more detail.

In general the process works as follows: An end-user uses JSON to define a workflow of computations and submits the workflow of computations to an endpoint of the system. Also, before submitting the actions to the system, the user needs to have placed the jar files that contain the "executables" that carry those computations in a folder in the distributed file system that is accessible to Pingo. There is a submission manager that takes care of parsing the workflow and analyzing the directed acyclic graph of computations defined in it in order to determine which actions need to be computed. Those actions are then submitted to the database with either one of two states: $WAITING$ or $READY$.

There exists a process called the Action Submitter that is constantly querying the database, pulling actions that are ready to be submitted for computation. This system converts the JSON definition of the action into an xml definition that is understood by Hadoop, and then submits the computation to be carried out by Hadoop (by Hadoop we mean the Hadoop ecosystem). There can be multiple Action Submitter processes working at the same time. To synchronize efforts, each process will use the database and its locking mechanisms guarantying that no action will be submitted

twice for computation to Hadoop.

Another independent piece of the system is the Callback Manager. When an action is submitted for computation to Hadoop, a callback endpoint is provided so that Hadoop can notify back the system with information about the computation: if it finished, failed or was killed, etc. As a redundancy measure, the Callback Manager is constantly polling Hadoop for information regarding each of the actions submitted for computation. Once an action finishes computing, the callback manager takes care of updating the state of the action and of the dataset produced by the action in the database, as well as of updating 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 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 takes care of enforcing 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 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

```
{
    "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

end action id and a **list of actions**.

The workflow definition in Figure 4.1 consists of two actions whose ids are 1 and 2 where action with id 2 must be executed after action with id id finishes. This is expressed by making action 1 a parent of action 2. Among the constraints that are imposed by the system we have the following:

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 directed **acyclic** graph (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 be forced to 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 user needs to have Read/Write permissions to any path it provides, otherwise, the execution of the action will fail when attempting to persist the output. The default value for *isManaged* is *True*.

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**, but 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 that is submitted to be executed by the cluster. Before submitting the action, there must be an action folder existing in the cluster's file system that contains the Java jar file with the action's executables, as well as any jars with the 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 provided in the action JSON description are passed as arguments to the main class. For example, for Command-Line actions the contract is designed in such a way that the Main class will read first in order the configuration parameters given in the field $additionalInput$, and then the rest of the arguments will be the paths to the output of its parent actions, also received 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 identified 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 lineage (the hashes of its parents). Algorithm 2 describes the procedure to produce a unique string. It takes as input parameters p and a, where ps is a list with the generated unique string of parents, and a is the action description. Depending on the action type of action a, Algorithm

```
{
    "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

2 uses a different subroutine to analyze the action's description. Algorithm 3 shows the subroutine for the Command Line Action.

Algorithm 2 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 3 Command Line Action Hashing

```
1: procedure CommandLineActionEncryption(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 fields of the JSON description are list fields. 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 examples of Figure

4.3 represent two actions that are not equivalent. In the case of a MAP_REDUCE action, order in a list field does not matter, and both examples represent actions that are equivalent between themselves. For types of actions where order does not matter in a list description field, their hash subroutine is 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, but there is nothing in the system that restricts it from doing away with **Apache Oozie** in the future.

The Action Manager is ready to be distributed across different machines. That is, if there are multiple action managers running on different machines, they have synchronization mechanisms that allow them to work together. This is a description

of its functionality:

1. It maintains a synchronized queue Q with the actions that need 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** takes new actions from the queue 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 that an action can be in.

An action 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 the queue. Available actions are actions that

Action State	Description
WAITING	It means that 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.

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

Before adding the action to the queue, 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 changed by someone else, and it will immediately drop it.

The synchronization technique described and exemplified in the above paragraph will be used multiple times by different 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 our *READY* state, and $S3$ to our *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 our 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 take care of submitting 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 they 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** makes its decision on whether an action needs to be computed or not by exploring the state of the dataset that is the output of the action. 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. 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).

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.

The **Workflow Manager** processes all the actions of the submitted workflow, **starting from the leaf actions** in a Breadth-First-Search (BFS) 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 4

Algorithm 4 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)                            ▷ D is the dataset that corresponds to 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        ▷ In this way the dataset cannot now be
   marked to be deleted by the Decision Algorithm.
13:                for each child C of action A do
14:                    if C was marked for computation when processed then
15:                        addClaim(C, D)          ▷ Marks in database that action C
   depends on D. No dataset can be deleted if it has a pending claim.
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 A finishes computing.

Secondly, on the Workflow Manager algorithm, there is a command described as $addClaim(C, D)$. What this does is to add 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 that has not been computed yet.

Thirdly, for the sake of correctness of the overall state of the system, we 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, we could make all the children actions of A to depend on A' (the sibling action of A 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 we are planning to change the state of the children of action A 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 is always the same 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. The callback finally 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 takes care of handling 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 Q 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 entity that queries the database for datasets ready to be deleted.

3. The Dataset Manager takes dataset entries inserted to the queue 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 DatasetManager in another process could have began processing those actions, but that process died before finish processing them.

Before adding the dataset to the queue, 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 too in a special data structure that we call 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 free, 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.

4.8 The Decision Algorithms

All the decision algorithms implement the same interface. They take two parameters as input: a data structure called $SimplifiedWorkflowHistory$ (SWH), and the space to free ($spaceToFree$). The Simplified Workflow History provides a very generic API that makes it easy to gather statistics on the history of workflows.

Currently, there exists two different implementations of decision algorithms in Pingo: a most-commonly-used decision algorithm, and an adaptive algorithm.

4.8.1 Most 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 5 defines value as the number of times the dataset is used throughout the history of workflows. That is a very simple definition of value, but provides the most straightforward implementation. Algorithm 5 is a good baseline to compare against.

Algorithm 5 Most-Commonly-Used Algorithm

```
1:  procedure MostCommonlyUsed(SWH, spaceToFree)
2:      datasetCounts ← SWH.datasetCounts()
3:      sortedDatasetCounts ← sortByCount(datasetCounts)
4:      spaceFred ← 0
5:      toDelete ← List()
6:      for count, dataset in datasetCounts do
7:          if spaceFred ≥ spaceToFree then
8:              break
9:          toDelete.append(dataset)
10:         spaceFred ← spaceFred + dataset.space
11:     return toDelete
```

4.8.2 Simple Adaptive Algorithm Family

Algorithms that belong to the $MostValuableAlgorithm$ family are a good starting point, but they have a weakness: they treat the entire history of workflows in the same way. This is troublesome, specially if it is the case that most recent datasets are more likely to be used 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? Finding good answers to such question is an area of further research. I propose a simple adaptive algorithm, described in Algorithm 6. Such algorithm searches over the entire history of workflows to determine some statistics that help decide how far back do workflows usually look back when reusing datasets. After computing such statistics, the algorithm calls the corresponding procedure from the **Most Valuable Algorithm Family** in order to determine the datasets to delete in this smaller workflow history.

Algorithm 6 Adaptive Most-Commonly-Used Algorithm

```
1:  procedure AdaptiveMostCommonlyUsed(SWH, spaceToFree)
2:      recencyList ← List()
3:      n ← length(SWH)
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:     smallerHistory ← SWH.subList(n − μ − 2σ, n + 1)
12:     return MostCommonlyUsed(smallerHistory, spaceToFree)
```

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 different kinds of algorithms that we have implemented.

It is almost impossible to obtain enough real world workflows data to do an statistically meaningful evaluation of the system. A good alternative is to use a probabilistic generator of workflows with the flexibility to adjust parameters to create different types of workloads. That strategy allows for a more fine-grained evaluation and comparison of how the algorithms behave under different types of workloads.

5.1 Evaluation Methodology

The strategy to evaluate the the Decision System is:

1. Probabilistically generate a history H of workflows.
2. Compute the **ideal execution time** of history H
3. Submit the history H to Pingo for computation and record the **actual execution time**
4. Compare the ideal execution time with the real execution time. The higher the ratio of $actual/ideal$, the best the algorithm.

5.1.1 Workflows generator

The workflow generator creates sequences of workflows in a probabilistic way given certain parameters. All the actions are Java $COMMAND\_LINE$ actions that 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 with size as given by the parameters and to stay in a for loop without returning for the amount of seconds 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.1 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 more complex piece of computation. Its purpose is to generate a workflow (DAG) selecting nodes from the pool of actions created by the Action Generator. Appendix B has a Python implementation of the algorithm. Roughly, the algorithm does the following:

1. Selects n nodes from the composition of all previous workflows up to that point

in the sequence 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. Create two normal distributions with parameters provided in configuration. Call one distribution 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 of 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 to imagine. For example, Figure 5.1 shows how to generate tree DAGs with very high probability if the number of parents that an action can have is restricted to only one. Figure 5.2 shows how to generate DAGs with more complex dependency relationships between nodes. Last, Figure 5.3 shows an example of the varieties of graph produced when the variance of the distributions is increased.

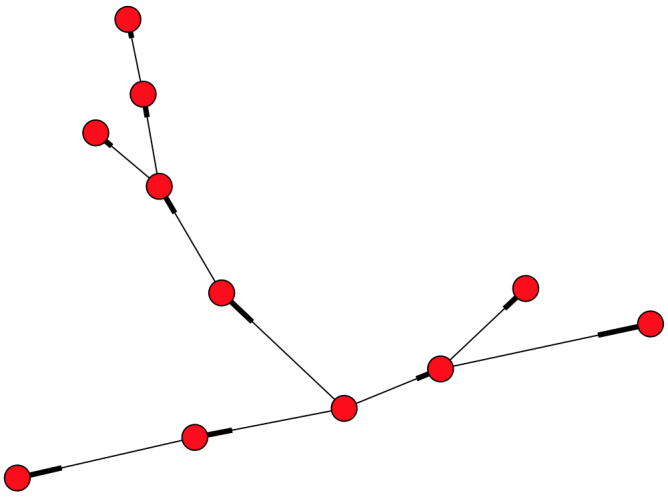

Figure 5.1: Example of a tree workflow that is produced with parameters: nb_children.mean = 2, nb_children.std = 1, nb_parents.mean = 1, nb_parents.std = 0.00001.

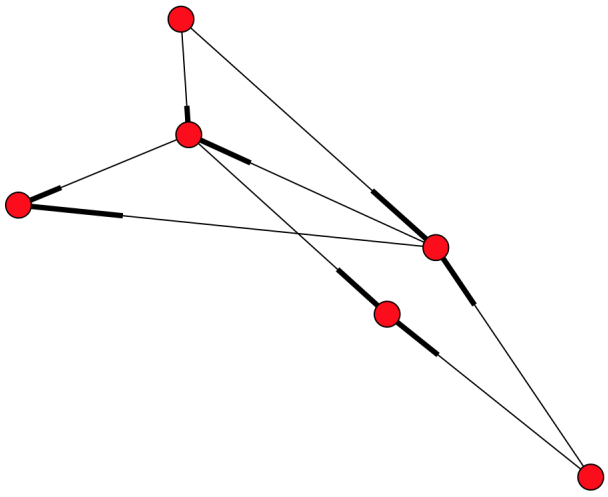

Figure 5.2: Example of DAG that is produced with parameters: nb_children.mean = 2.1, nb_children.std = 0.0001, nb_parents.mean = 2.1, nb_parents.std = 0.0001.

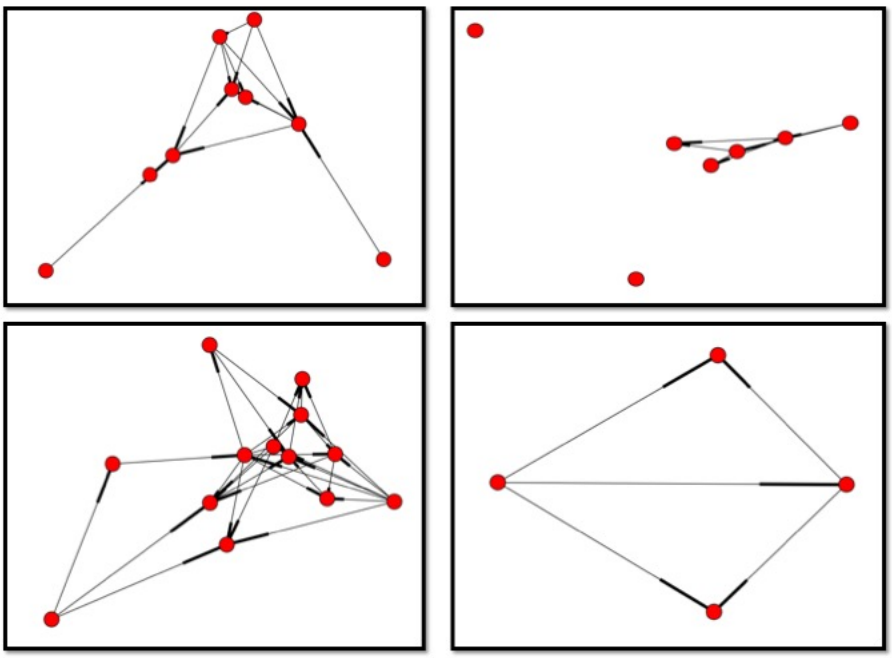

Figure 5.3: Example of four DAGs produced with parameters: nb_children.mean = 2.1, nb_children.std = 4.5, nb_parents.mean = 2.1, nb_parents.std = 4.5.

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 an optimal solution to the problem 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 Section 1 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 /textbfideal 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

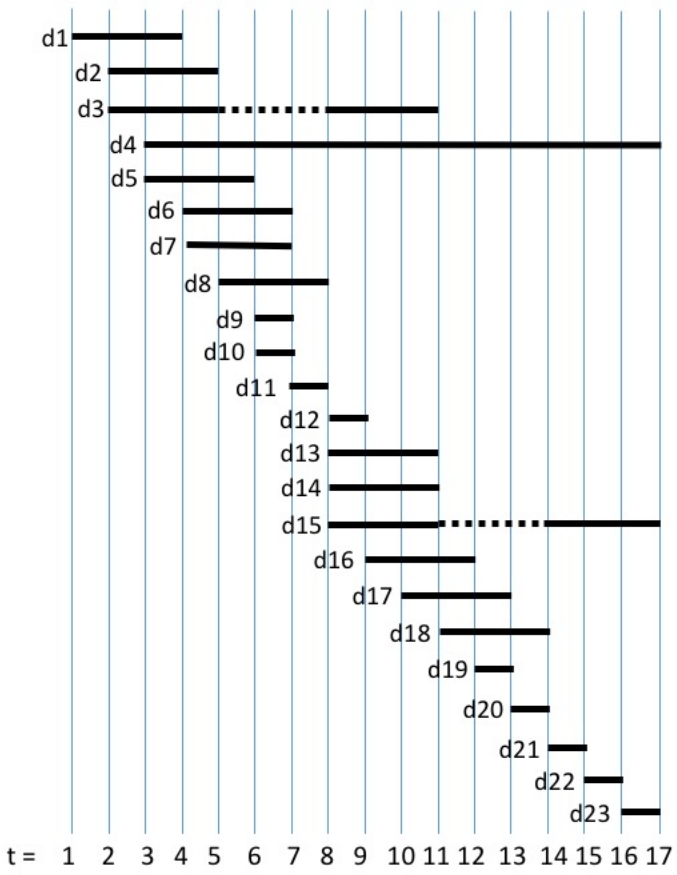

Figure 5.4: Effective life diagram of datasets of hypothetical history of workflows.

$S' = \sum_{d \in M_t} s(d)$. 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.

Algorithm 7 Ideal Computation Time algorithm

1: **procedure** IDEALCOMPUTATIONTIME($S, H = (W_1, ..., W_n), b, e, c$)
2: $totalTime \leftarrow 0$
3: **for** t from 1 to n **do**
4: Let $M_t = \{d \,|\, b(d) \leq t \wedge e(d) \geq t\}$
5: Let $N_t = \{d \,|\, b(d) < t \wedge e(d) \geq t\}$ ▷ $N_t \subseteq M_t$
6: Let $P_t = \{d \,|\, d \in W_t\}$ ▷ $d \in M_t \implies d \in W_t$
7: Let $M_H = (M_1, ..., M_n)$
8: Let $N_H = (N_1, ..., N_n)$
9: Let $P_H = (P_1, , , , , P_n)$
10: **for** t from 1 to n **do**
11: 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).
12: 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.
13: $totalTime \leftarrow totalTime + et$
14: **return** $totalTime$

Algorithm 8 Computation Time Left Subroutine

1: **procedure** COMPUTATIONTIMELEFT(d, P_H, m, n, c)
2: $timeLeft \leftarrow 0$
3: **for** t from m to n **do**
4: **if** $d \in P_t$ **then**
5: $timeLeft \leftarrow timeLeft + c(d)$
6: **return** $timeLeft$

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 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 **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.

*********BEGIN HERE ********** Each experiment was run 5 times using the same configuration parameters and the results were averaged. The first experi-

ment 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, **Computation time percentage** 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.

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

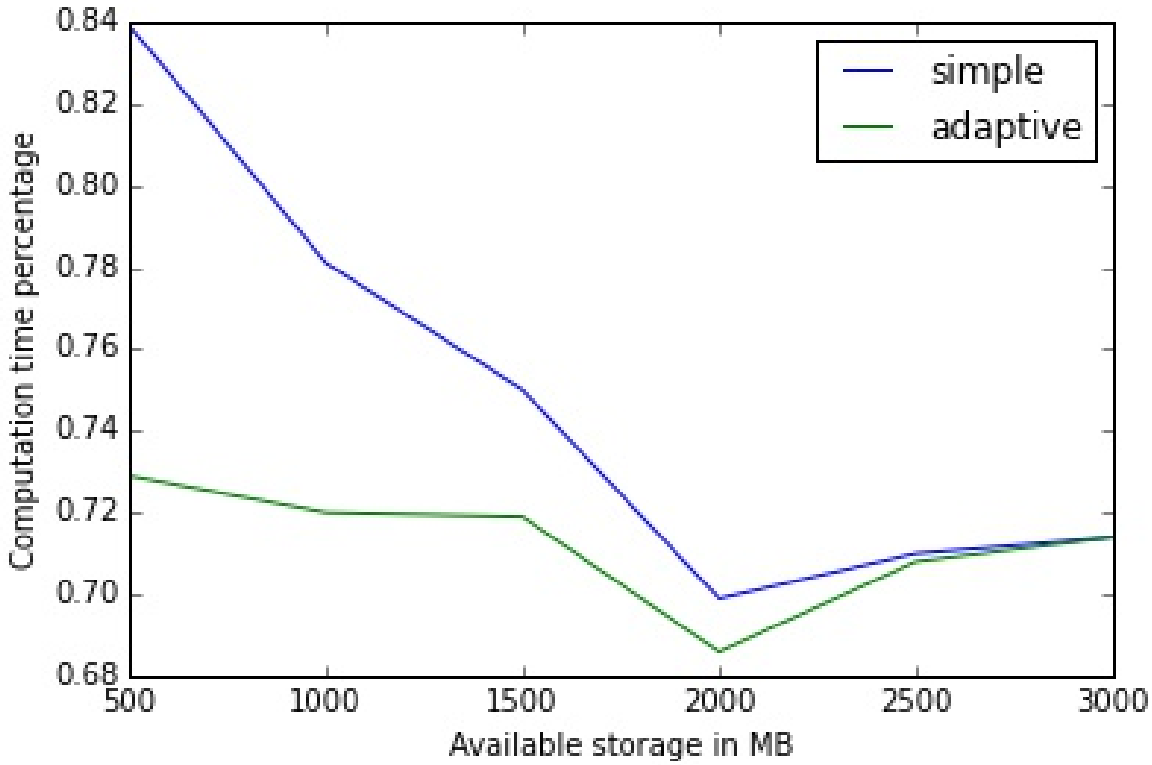

Figure 5.5: Computation time as storage increases

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 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.

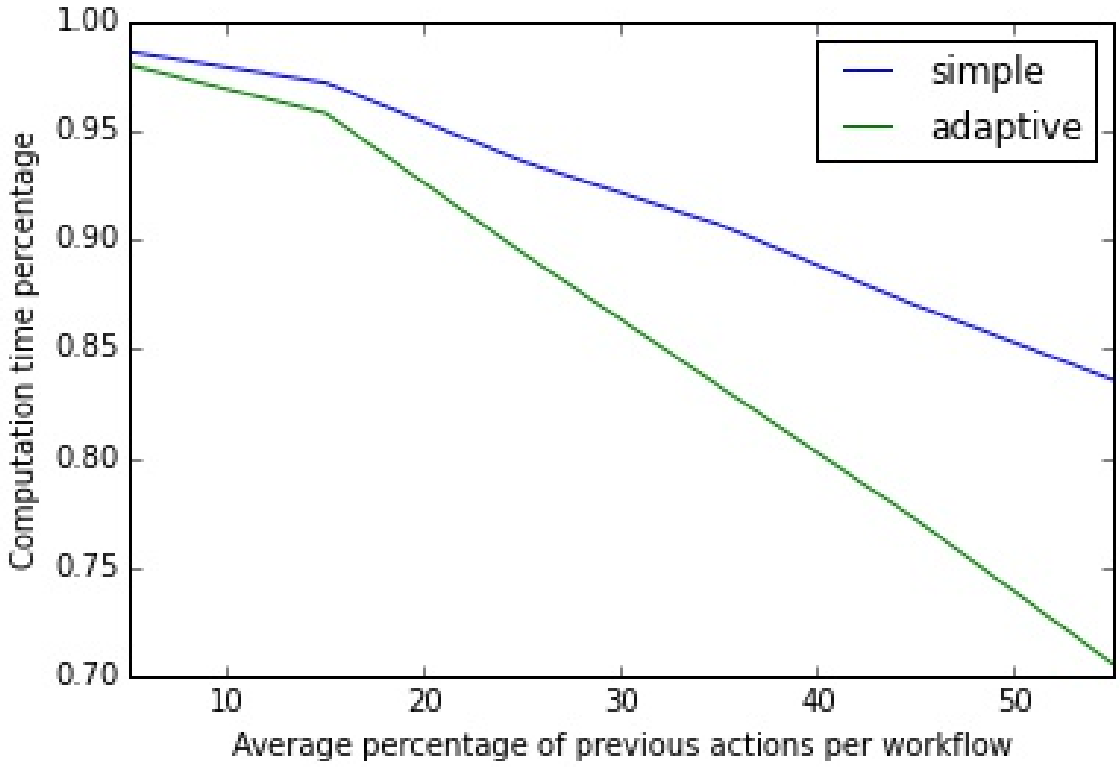

Figure 5.6: Computation time percentage as percentage of actions from previous workflows increases

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 algo-

rithms should be able to detect the rate at which the look back parameter might be changing, and adapt to it. The experiments designed were able to give some good insights into how the different algorithms we tested performed under different conditions. Because of that, I think that those evaluation experiments should serve as the foundation of any future evaluation methodology on the system.

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 as to provide good statistical guarantees on the validity of evaluations on it. 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 previous proposition of gathering a more diverse set of data of workflow submissions.

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 effective 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 other persons. I certainly have attempted to design it with that goal in mind.

REFERENCES

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).

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

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).

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

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).

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).

Jones, S. P., *Haskell 98 language and libraries: the revised report* (Cambridge University Press, 2003).

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

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).

Simmhan, Y. L., B. Plale and D. Gannon, "A survey of data provenance in e-science", ACM Sigmod Record **34**, 3, 31–36 (2005).

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).

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

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, 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

A.1 The Workflows Generator

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