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

This is an example abstract that I will have to modify later on. I will write my abstract at the end, when I am done with my research.

DEDICATION

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

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 another step. Workflows have emerged as a paradigm for representing these computations. Many seminal works on the topic of workflow managing systems began to appear in the mid 2000's**???**, and many workflow systems were developed, such as the e-Science project**?**, Kepler**?** and Taverna**?**.

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 still very limited. Pegasus**?** is a more recent 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. But still, it cannot meet the scalability demands of today's scientific computations.

Orthogonal to the development of workflow management systems, many distributed systems have been designed and developed to meet the growing demands of computation. One of such systems is Apache Hadoop, which is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Apache Oozie**?**

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 Hadoop.

1.1 Our contributions

Chapter 2 of our discussion is also a great contribution to the topic of scientific workflows, since it 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. The system that I have designed and implemented is tied to a specific technology of today, the Hadoop ecosystem. And as Hadoop goes, so goes my system. But the discussions of Chapter 2 will still remain relevant for years to come.

Chapter 2

FOUNDATIONAL DESIGN PRINCIPLES OF THE SYSTEM

In this chapter we discuss some of the reasons for the design decisions taken when implementing the system's functionality. We will discuss the design decisions as they relate to the main functionalities that we intend to implement.

2.1 Skipping unnecessary computations

The system will constantly be managing the computation of actions submitted to it by end-users. 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.

How can we achieve such functionality? How can we know that actions A and B are equivalent? How can we know that an output dataset that we have in our filesystem is the final output of an entire workflow of computations that has been submitted to the system, and that we can skip the computation of not just one action, but of an entire collection of actions? In order to be able to answer these questions, we need to explore what are the best strategies to follow when defining what is an **Action** and what is a **Workflow**.

2.1.1 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 imposible in the most broad sense, since the Halting Problem can be reduced to this problem).

Also, 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 environmental variables.

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. As Figure 2.1 shows, 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 description 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 satisfies the following two simple properties:

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

```
{
   input: ["/path/to/input1", "/path/to/input2"],
   output: "/path/to/output",
   executable: ["/path/to/executable"],
   inputParameters: [ {key: "arg1", value: "Hello"},
                                 {key: "arg2", value: "World!"}]
}
```

Figure 2.1: Hypothetical example of description of an action

2. If two action descriptions will 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 solid 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, we will see later in this chapter how we intend to interpret "all the time" not in a mathematically strict sense, but only in a practical sense. Another challenge that we will have in order to satisfy the second property is that we still have to deal with the issue that the computation performed by that action might not be a **pure function**. 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 we cannot make a guarantee that our measure of equivalence will satisfy Property 2. Because of that, we need to leave on the end user the ultimate responsibility of taking advantage of any computational gains our system could provide. If the end user thinks that the action description of an Action A that he is about to submit to the system might produce a different output to that of an existing output of a previously submitted action B that the system will flag as equivalent to action A, then the user must let the system know that action A computation cannot be skipped.

The description of an action submitted to the system becomes very relevant, since is the starting point to determine equivalence between actions. We could enforce a descriptive schema that is very descriptive (pardon the redundancy), and in that way have more elements to determine the equivalence between two actions, and maybe improve the accuracy of Property 1 of our equivalence measure. But at the same time we might be making the system burdensome to use to the final user, since they will have to invest considerable time writing the descriptions of the actions that they are submitting to the system. A good balance needs to be found here.

2.1.2 Workflows

A **workflow** is essentially a directed acyclic graph (DAG) where nodes correspond to actions 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 actions computations: 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. Handling failures is a discussion that we leave for the next chapter, when we describe the implementation of this chapter's ideas.

2.2 Bounded storage space

2.2.1 The History of Workflows

2.3 A multi-user multi-component system

APPENDIX A

RAW DATA

BIOGRAPHICAL SKETCH