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

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 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 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 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.

Before discussing more serious matters regarding workflows, we need to agree on some notational conventions. We will 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 will also use functional notation to represent actions outputs an inputs. If an action a_1 takes as input original dataset o_1, derived dataset d_2 and the output of action a_3 on original dataset o_2, we represent its output as follows: $d_1 = a_1(o_1, d_2, a_3(o_2))$.

Equivalence of actions in the workflow setting

Consider the previous example with the definition of action a_1. What strategy can we devise to efficiently determine if we have dataset d_1 in storage 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

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
```

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 we pop them from the queues to compare them, they correspond to each other. In the next chapter (the implementation chapter) we will see how that assumption is valid for some action types, but not for others.

To analyze the running time of this algorithm, 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 $O(m(M + N))$. If we index the metadata of our stored datasets in a clever way, we can reduce that to a running time of $O(t(M + N))$ with $t << m$.

But this approach will 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. The second reason is that indexing the metadata of a graph datastructure is not a trivial task.

The encryption alternative

Encryption is another strategy that can be used to determine if an action's output already exists in the storage system. Given an action node a_i in a workflow, we can recursively compose the description of action a_i together with the descriptions of its parent actions to produce a hash value that "uniquely" identifies the dataset d_i produced by action a_i as output. We can 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 describing the above situation is highly dependent on the semantic of the description of the actions. Because of that, we leave its discussion to the next chapter. But we have enough elements as to arrive to 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 no cryptographic function is perfect, we need to analyze what would be the probability of a clash between the hash signatures of non-equivalent actions.

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 encryption algorithm, and that D is the number of datasets currently in storage. The question we want to answer is: What is the probability of any two of the 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:

$$1 - p(N) = 1 \times \frac{2^n - 1}{2^n} \times \frac{2^n - 2}{2^n} \times ... \times \frac{2^n - N + 1}{2^n}$$
$$1 - p(N) = \frac{2^n \times 2^n - 1 \times ... \times 2^n - N + 1}{2^{n^N}}$$
$$1 - p(N) = \frac{2^n!}{2^{n^N}(2^n - N)!} \tag{2.1}$$

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 $N = 1,000,000$, we have that 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 partially valid. We only need to add a second step to the process. In the first step we can still use the hash value of an action that is being submitted to the system to quickly find an already existing **tentative** dataset. Then we can compare the workflow that produced the tentative dataset to the workflow of the submitted action to exactly determine if they are equivalent.

2.2 Bounded storage space

Bounded storage space is the constraint that makes the problem we are trying to solve interesting. There are physical limitations and technical limitations that won't allow us to have unlimited space. Therefore, at all moments in time, the system needs to decide which datasets to keep in storage and which datasets to delete. The system also has to enforce its own decisions.

2.2.1 The History of Workflows

The concept of **History of Workflows** becomes useful once we add the constraint of bounded storage capacity to our system. Since we want to optimize the

time of computing future workflows, the decision system is in reality an optimization problem that tries to "guess" which datasets will be more relevant in the future. What strategy should we use in order to optimize the computational time of future workflow submissions? That is an open question that has multiple valid answers. But whatever strategy we use to accurately predict which datasets might be needed in the future will need access to the history of workflows submitted to the system up to that point. Because different strategies might need different things from the history of workflows, it is recommended to provide a flexible API that can be used to query the history of workflows to obtain a diverse set of statistics over specified periods of time and at different resolutions.

2.2.2 The Decision System

The core functionality of the system is the Decision System that determines which datasets need to remain and which datasets need to be deleted from the storage of the system. The Decision System can be run each time a new workflow is submitted to the system, or it can be run only when space occupied by stored datasets reaches a certain threshold of the total capacity of the system. The second option makes more sense, so we will define our interface with that in mind. The Decision System defines an interface that can be implemented by different 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. A set D of datasets currently stored in the file system.

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

4. $t : A \to \mathtt{N}$, 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 we remove from storage datasets of B' instead of B, we would spend less time computing our future workflows.

Literature Review of related problems

This section can be skipped without risking lack of understanding of the rest of the presentation, but for completeness sake we think that is necessary to review multiple efforts in the literature to solve problems similar to the one we have posed in Section 2.2.2.

Is not surprising that none of the related problems have been defined in the same way we have defined ours, so we want to offer some commentary on which of them we think are more relevant to problem in 2.2.2 and why.

2.3 A multi-user multi-component system

Chapter 3

IMPLEMENTATION OF THE SYSTEM

3.1 Birdeye overview

Introduce the system and what it does. To the point that it takes you to the Hadoop ecosystem and how it uses it

3.1.1 The Hadoop ecosystem

3.1.2 Configuration parameters

Talk about the configuration parameters used for the system such as namenode, oozieurl, etc.

3.2 The Workflow Definition Language

We have chosen the JSON format for the definition of workflows because its expressiveness is sufficient for what we need, and it is also very human readable. As shown in Figure 3.1, a workflow is made of a **name**, an **start action id**, an **end action id** and a **list of actions**.

The workflow definition in Figure 3.1 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.

```
{
    "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 3.1: 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).

5. This constraint can be derived from 4, but so that it is not overlooked, we state

the rule explicitly here: The *endAction* cannot be a parent or ascendant of the *startAction*

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

3.2.1 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 determines 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 at the end. The default value for *isManaged* is *True*.

Notice also how action names do not need to be unique. An action name is just a mnemonic resource to understand what the action does. Also depending on the action type, there might be other required attributes too. We currently support three kinds of actions: **Command-line actions**, **MapReduce v1.0 actions** and **MapReduce v2.0 actions**, and in the future we are planning to add support for **Spark actions** and **Sqoop actions**

TODO: Explain how an action's dataset name is determined

Command Line Action

MapReduce v1.0 Action

MapReduce v2.0 Action

3.3 The Action Manager

The Action Manager's purpose is to submit individual actions to the Hadoop cluster for computation. In our current implementation it uses **Apache Oozie** as an intermediary, but there is nothing in the system that restricts us from doing away with **Apache Oozie** in the future.

On its current implementation, the Action Manager is ready to be distributed across different machines. That is, if there are multiple action managers running on different machines, they will not step up on each other toes, because they use the database as a mean of synchronization among them.

The Action Manager works as follows:

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 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 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.

3.3.1 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** we have implemented synchronization using the database as our shared resource and defining a synchronization oriented semantic for each of the different states an action can have.

An action can be in one of the following states: *WAITING*, *READY*, *PROCESSING*, *SUBMITTED*, *RUNNING*, *FINISHED*, *FAILED*, and *KILLED*. (See Table 3.1 for a complete reference).

3.3.2 The Action Scraper

Every certain amount of time, the action scraper will query the database to find available actions and add them to the queue. Available actions are actions that are in the *READY* state, or actions that have been in the *PROCESSING* state for a long time. The reason why we add 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 started processing those actions, but because of some failure the process died before finish processing them.

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

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 ActionScraper found a READY action in the database and has placed it in the actions queue of the actions to be submitted.
SUBMITTED	The action has been taken from the queue and has been submitted 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 3.1: Action State Descriptions.

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 process should be allowed to do it. In order to solve the problem we create an intermediate state $S2$ (*PROCESSING* in our case), and we let all the processes compete to be the first to change the state of o to $S2$. All the loosing processes drop the processing of object o, and the winning process carries on.

3.3.3 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 include in the queue actions that have been in the *PROCESSING* state for a long time makes the design of the Action Submitter more careful. The submitter first attempts to update the state of the action to *SUBMITTED* in the database. 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 point later on. As an area of future improvement, a ceiling should be imposed over the number of times an action fails when submitted to the cluster, otherwise, the system will keep trying to submit the action forever.

3.4 The Workflow Manager

Now that we have explained the **Action Manager**, we are in a better shape to understand the inner workings of 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 will be discussed later when we discuss the **Callback System**.

3.4.1 Datasets

The **Workflow Manager** makes its decision on whether an action needs to be computed or not by exploring the state of the datasets that are the outputs of the action. A dataset is another important entity in our model. 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 3.2 for a complete reference).

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 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.

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. Datasets of leaf actions are never removed by the system. They can be manually removed by the users.
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 dataset will be removed.
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 3.2: Dataset State Descriptions.

The **Workflow Manager** makes the determination if an action needs to be computed as described in Algorithm 2

Algorithm 2 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, PROCESSING, 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)
```

I want to call the attention to three different behaviors of the algorithms described above. 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, notice how 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) do not delete a 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, we still prepare 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 mean that we need to introduce more complex synchronization mechanisms across multiple components of the system. For now we think that the benefits of simplicity will outweight the efficiency gains of trying to improve a situation that we consider will happen rarely.

3.5 The Callback System

Once an action is submitted, three callbacks are provided to the Hadoop cluster so that it can notify back to the system of any relevant event regarding the execution of the action by the cluster. All callbacks are designed in such a way that the state of the action is always the same after multiple calls to the same callback.

3.5.1 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: dataset state is changed from TO_STORE, TO_LEAF or TO_DELETE to $STORED$, $LEAF$ or $STORED\_TO\_DELETE$ accordingly. Also, the size the dataset occupies in the filesystem is also updated on its entry in the database. That size is an important factor used by the optimization algorithm.

3.5.2 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, its state is changed to $STORED\_TO\_DELETE$ regardless of the previous state of the dataset. Also, the state of the action itself is changed to $FAILED$.

3.5.3 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, its state is changed to $STORED\_TO\_DELETE$ regardless of the previous state of the dataset. Also, the state of the action itself is changed to $KILLED$.

3.6 The Dataset Manager

The dataset manager takes care of handling the deletion of datasets from the file system. Its archicture 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 DatasetScraper 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 will take care of removing the datasets from the cluster and updating the state of those datasets in the database.

3.6.1 The Dataset Scraper

Every certain time, the dataset scraper will query the database to find available datasets to be deleted and add them to the queue. Datasets to delete are such that are in the $STORED\_TO\_DELETE$ state, or datasets that have been in the $DELETING$ or $PROCESSING$ state for a long time. The reason why we add 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.

3.6.2 The Dataset Deletor

The Dataset Manager is constantly taking new elements from the queue and passing them to the Dataset Deletor threads that take care of removing the datasets from the file system. 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.

3.7 The Decision Manager

The Decision Manager is the piece of the system that determines which of the datasets currently stored in the system should be deleted. The manager has three main components that work together: A file system 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 thing that happens is that the Decision Manager queries the file system utility to obtain the amount of space currently in use by the datasets managed by

the system. If the space exceeds a certain threshold, then the decision engine fires up 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 then rebuilds the graph of the workflows to which this actions belonged too in a special datastructure that we call simplified workflow history.

2. We pass the simplified workflow history that comprises the last N submitted actions to the decision algorithm, together with the amount of space that we need to free, and the decision algorithm returns a list of datasets that need to be deleted. The sum of the storage space of the returned datasets will be at least the amount of space that needs 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.

3.8 The Decision Algorithms

3.8.1 Most-Commonly-Used Algorithm

Chapter 4

EVALUATION METHODOLOGY OF THE DECISION ALGORITHMS AND RESULTS

In this chapter we propose an evaluation methodology for the decision algorithms for the system. We also report on the evaluation of the two different kinds of algorithms that we have implemented.

Since it will be difficult to obtain enough real world workflows data to do an statistically valid evaluation of the system, we will use a probabilistic generator of workflows with the flexibility to adjust parameters to create different types of workloads. This will give us the opportunity to do a more fine-grained evaluation and compare how the algorithms behave under different types of workloads.

4.1 Evaluation Methodology

To evaluate the the Decision System, our strategy will be to:

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.

4.2 Workflows generator

Since there is not enough real data as to evaluate the behavior of our system, we have designed a workflows generator that probabilistically creates sequences of workflows given certain parameters.

The sequence generator is composed of three probabilistic generators that work together to produce the history of of workflows. See Appendix A.1 for information on how to use the generator. We have also created a Java $COMMAND\_LINE$ action that takes as parameters the size of an output in megabytes, the time of execution in seconds, and a String to act as a differentiator. Both the action's output and the computation time of the action are determined by the parameters passed to it.

4.2.1 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. This list of actions will be used as a pool of actions from which the workflow generator will select actions to compose the workflows.

4.2.2 The Workflow Generator

The **Workflow Generator** is a little more complex piece of computation. Its purpose is to generate a workflow (DAG) using as nodes from the actions created by the Action Generator. You can find a Python implementation of the algorithm in Appendix B. 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 take. 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 node 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

4.3 Ideal Execution Time calculation

The first step is to define what an ideal execution time of a history of workflows is, and also how to compute that execution time given a history of workflows.

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
```

BIOGRAPHICAL SKETCH