IReS Platform v.1

Deliverable no.: 3.2

27/04/2016

ASAP is funded by the European Commission DG-INFSO

Seventh Framework Programmed, Contract no.: 619706

Deliverable Title	IReS Platform v.1
Filename	D3.2_revised.pdf
Author(s)	K. Doka, N. Papailiou, C. Mantas, V. Giannakouris, D.Tsoumakos
Date	27-04-2016

Start of the project: 01/03/2014
Duration: 3 years
Project coordinator organization: FORTH

Deliverable title: IReS Platform v.1
Deliverable no.: 3.2

Due date of deliverable: 31/08/2015
Actual submission date: 31/08/2015
Revision and resubmission date: 27/04/2016

Dissemination Level

X	PU	Public
	PP	Restricted to other programme participants (including the Commission Services)
	RE	Restricted to a group specified by the consortium (including the Commission Services)
	CO	Confidential, only for members of the consortium (including the Commission Services)

Deliverable status version control

Version	Date	Author
0.1	15/07/2015	K. Doka
0.2	30/07/2015	K. Doka, N. Papailiou, C. Mantas
1.0	10/08/2015	K. Doka, V. Giannakouris
2.0	24/08/2015	K. Doka, D. Tsoumakos
3.0	27/08/2015	K. Doka
4.0	20/04/2016	K. Doka, N. Papailiou, D. Tsoumakos

Abstract

This deliverable is a report on the first version of the Intelligent, Multi-engine Resource Scheduling (IReS) platform. This version incorporates the prototypes of all the core modules of the IReS architecture, including the Modeling, Decision Making and Execution modules. The report first gives a quick overview of the final IReS platform architecture and then delves into the implementation details of each involved module. Internal and external interfaces are specifically defined and finally the profiling, modeling and decision making modules are evaluated in terms of performance, efficacy and accuracy of the produced models.

Keywords

Workflow planning, modeling, profiling, compute engine, data store

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ML	Machine Learning
M-R	Map-Reduce
RDMS	Relational Database Management System
SPJ	Select-Project-Join
TF/IDF	Term Frequency/Inverted Document Frequency
VM	Virtual Machine

1 Introduction

1.1 IReS Overview

The demand for near-real-time, data-driven analytics has given rise to diverse execution engines and data stores that target specific data and computation types. Many of these systems are now offered as a service by IaaS providers, enabling a very wide deployment range. There also exist approaches in the literature that manage to optimize their performance (e.g., [12] [30]) by automatically tuning a number of configuration parameters. Yet, these schemes assume strictly single-engine environments (mainly the Hadoop ecosystem), thus considering specific data formats and query/analytics task types.

However, modern workflows have become increasingly long and complex and may include multiple operators, ranging from simple Select-Project-Join (SPJ) to complex Machine Learning (ML) or custom operators that operate on diverse data types (e.g., relational, key-value, graph, etc.) generated from different resources. What is more, analysts need to be able to execute them under varying constraints and policies (e.g., optimize performance or cost, require different fault-tolerance degrees, etc.). There currently exists no single platform that can optimize for this complexity [41] .

Sensing this trend, cloud software companies now offer software distributions in pre-cooked VM images or as a service. These distributions incorporate different processing

frameworks, data stores and libraries to alleviate the burden of multiple installations and configurations (e.g., [4] [13] [15] [35]). Yet, such multi-engine environments lack a meta-scheduler that could automatically match tasks to the right engine(s) according to multiple criteria, deploy and run them without manual intervention.

To address multi-engine analytics workflow optimization we designed and developed the Intelligent Multi-Engine Resource Scheduler (IReS), an integrated, open source platform for managing, executing and monitoring complex analytics workflows. IReS is a core component of the ASAP system architecture and its main task is to "mix-and-match" diverse execution engines and data stores in order to optimize a workflow with respect to multiple, user-defined criteria [41] .

To that end, IReS incorporates a modeling framework that constantly evaluates the cost and performance of data and computational resources under various configuration setups in order to decide on the most advantageous store, indexing and execution pattern.

A tree-based metadata language that describes operators in abstract and instantiated forms enables the search and matching of operators that perform a similar task in the planning phase. Afterwards, a decision making module chooses among the different equivalent execution plans (i.e., on different engines, resulting in equivalent output) the one that best fits the given policy based on cost and performance models. The chosen plan is scheduled and enforced, taking into account the available resources.

IReS [7] is a fully open-source platform[1] (under the Apache License version 2.0) that targets both low-level (e.g., join, sort, etc.) as well as high-level (e.g., machine learning, graph processing) operators, treating them as black boxes. The extensibility of the platform is one of the main goals of IReS. Thanks to the generic methods it relies upon, IReS can easily include additional operators and engines.

1.2 Purpose of the Document

This document serves as a report on the first version of the IReS platform and accompanies its prototype implementation. Its purpose is to delve into the implementation details of the core architectural modules, specify the internal and external APIs for inter- and intra-platform communication and present an experimental evaluation of the platform's accuracy of operator/engine modeling, efficacy of profiling and performance of decision-making.

1.3 Document Structure

D3.2 is structured as follows:

- Chapter 2 gives a brief overview of the finalized architecture of the IReS platform, including all refinements over the initial architecture as described in D3.1. Moreover, it specifies the external API, through which IReS communicates with external modules of the ASAP system.
- Chapter 3 gives details on the current status of the implementation of all modules involved in the first version of the IReS platform. Moreover, it defines the internal APIs, through which the intra-platform communication is performed.
- Chapter 4 describes the infrastructure where the IReS platform has been deployed, the specific runtimes and data stores it currently supports and the implemented and ready-to-use operators that populate the IReS operator library so far. These operators are used to formulate some basic workflows driven by the use-case scenarios of D8.2 and D9.2. These workflows are also presented here.
- Chapter 5 experimentally evaluates the core modules of IReS, namely the profiling, the modeling and the decision making modules. Experiments focus on performance and scalability aspects as well as accuracy of the models.
- Chapter 6 concludes the deliverable and outlines our next steps.

[1] Source available in https://github.com/project-asap/IReS-Platform

2 IReS Architecture and external API

This chapter revisits the IReS architecture, as defined in D3.1, and finalizes it after refining some parts. Moreover, the way external components of the ASAP system communicate with IReS is specified through a well-defined API.

2.1 General Architecture

Figure 1 depicts the final architecture of the IReS platform as well as its interaction with external components, being developed in WP 4, 5 and 6. IReS comprises of three layers, the *interface*, the *optimizer* and the *executor* layer.

The *interface layer* is responsible for communicating with the application UI in order to receive the input that is necessary for its operations. It consists of the **job parser** module, which identifies execution artifacts such as operators, data, their dependencies and accompanying metadata. Moreover, it validates the user-defined policy. All this information must be robustly identified, structured in a dependency graph and stored.

The *optimizer layer* is responsible for optimizing the execution of an analytics workflow with respect to the policy provided by the user. The core component of the optimizer is the **Decision Making** module, which determines the best execution plan in real-time. This entails deciding on where each subtask is to be run, under what amount of resources provisioned, the plan for moving data to/from their current locations and between runtimes (if more than one is chosen) and defining the output destinations. Such a decision must rely on the characteristics of the analytics task in hand and the models of all possible engines. These models are produced by the **Modeling** module and stored in a database called **Model DB**. The initial model of an engine results from profiling and benchmarking operations in an offline manner, through the **Profiling** module. This module directly interacts with the pool of physical resources and the monitoring layer in-between. While the workflow is being executed, the initial models are refined in an online manner by the **Model refinement** module, using monitoring information of the actual run. Such monitoring information is kept in the **IReS DB** and is utilized by the decision making module as well, to enable real-time, dynamic adjustments of the execution plan based on the most up-to-date knowledge.

The *executor layer* is the layer that enforces the selected plan over the physical infrastructure. It includes methods and tools that translate high level “start runtime under x amount of resources”, “move data from site Y to Z” type of commands to a workflow of primitives as understood by the specific runtimes and storage engines. Moreover, it is responsible for ensuring fault tolerance and robustness through real-time monitoring.

More details about the role and functionality of each module can be found in D3.1. The following chapter elaborates on the internals of each module as well as on implementation decisions and initial evaluation.

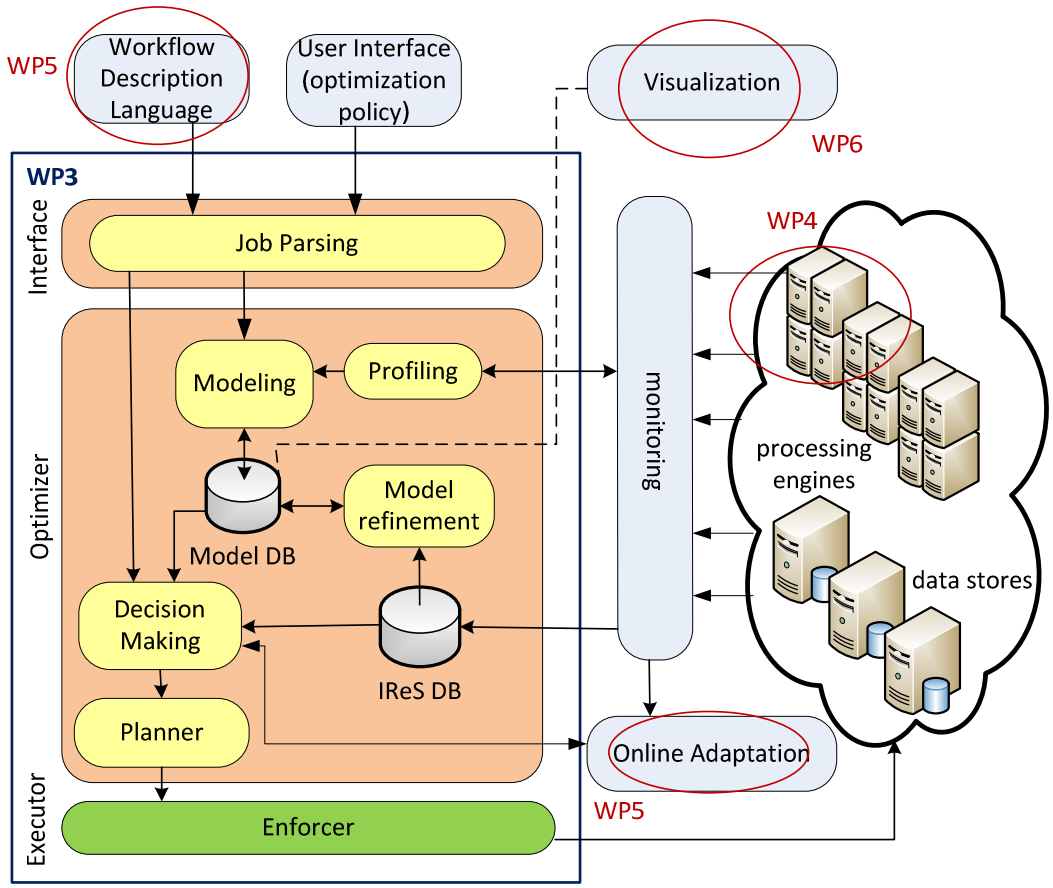


Figure 1 The architecture of the IReS platform

2.2 Workflows

The main functionality of the IReS platform is covered by two workflows, the Profiling and the Planning and Execution workflows.

2.2.1 Profiling Workflow

This workflow, depicted in Figure 2, describes the process that takes place whenever a new materialized operator, accompanied by its metadata description, is added to the operator library of the IReS platform. This operator insertion triggers an offline profiling process to obtain knowledge about its behavior under different configurations.

During profiling, a number of different operator configurations are adaptively selected by the profiler. The operator is then executed under these different setups and a number of metrics (currently 45) are monitored in order to train and create accurate performance models. These metrics include: (a) execution time, (b) all the monitoring metrics reported by the ganglia[2] monitoring tool (e.g., CPU, RAM, iops, network traffic etc.), (c) operator specific metrics (e.g., number of results, output size, etc.). Moreover,

[2] http://ganglia.info/

the developer can provide her own monitoring probes through a well-defined API. These models populate a knowledge base, Model DB, that can be used to facilitate the decision making process.

Profiling is an iterative process, with configurations being selected dynamically, based on the accuracy of the model with respect to the so far collected measures. More details about the implementation of the profiling module follow in section 3.2.

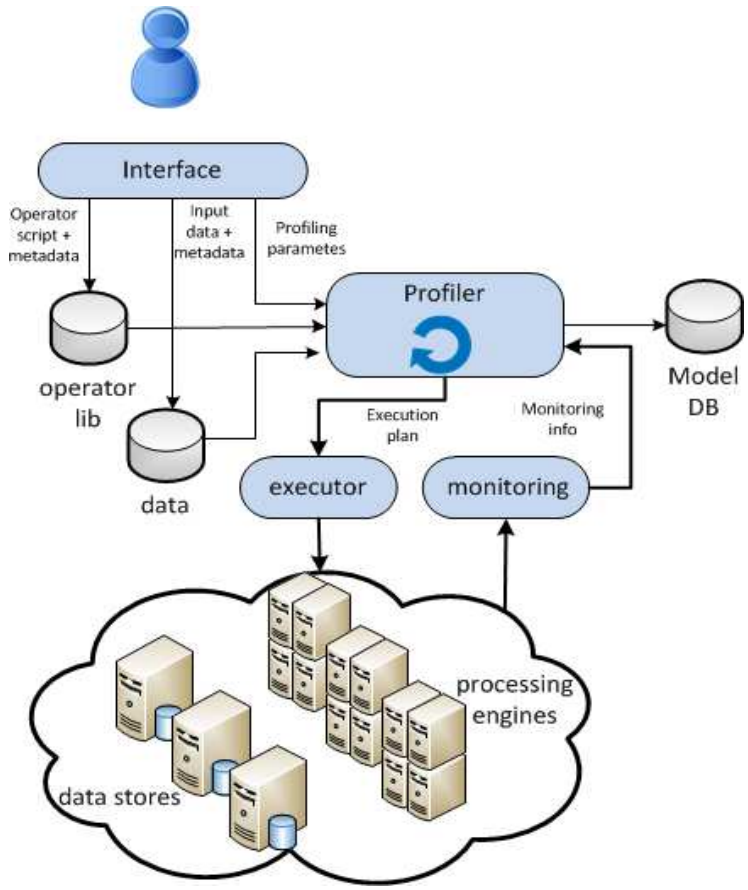


Figure 2 Profiling Workflow

2.2.2 Planning and Execution Workflow

This workflow (see Figure 3) takes place when a new workflow along with the desired optimization policy is provided for execution by the user. This policy can consist of one or a function of multiple operator performance metrics like cost, execution time, etc.

The Decision Making module matches the operators present in the user-provided workflow with the actual implementations of them residing in the platform's operator library and explores the possible alternatives in order to find the plan that best matches the user-defined policy. Details about this process and its complexity are provided in section 3.3.

When this plan is located, it is first validated against the current resources and their load and then executed by the Enforcer module. If necessary, IReS falls back to the execution of another plan (e.g., when the Enforcer detects that the actual plan execution deviates largely from its expected behavior).

Lastly, IReS manages the elasticity of the underlying infrastructure by monitoring the utilization of the engine resources. Based on this monitoring information it can take decisions for allocating and de-allocating computing resources in order to improve the general execution of workflows and operators.

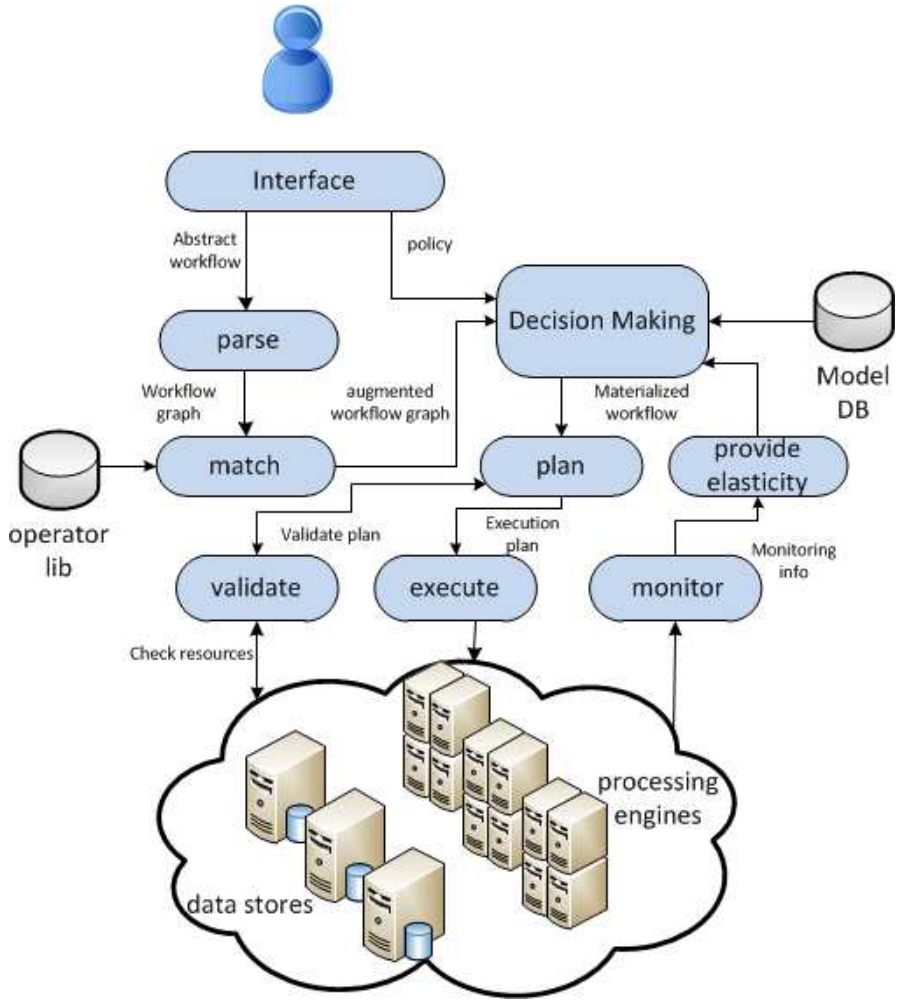


Figure 3 Planning and execution workflow

2.3 External API

The functionality of the IReS platform is exposed to the rest of the ASAP components through a RESTful API, described in detail in Table 1. The intra-IReS API, through which the various modules of the IReS architecture communicate and interact, is presented in the following chapter.

Table 1 The external API of the IReS platform.

Nr.	RESTful API: IReS platform				Description
	Type	Resource URL	Consumes	Produces	

1	GET	/datasets	-	List of datasets	Returns a list of the existing datasets
2	GET	/datasets/{id}	id	Dataset xml	Return the description of a specific dataset
3	PUT	/datasets/edit/{id}	id, xml description	-	If the specific dataset exists in the library, its metadata are replaced with the new ones. Else it gets added along with its metadata description.
4	GET	/datasets/delete/{id}	id	-	Deletes the specified dataset
5	GET	/operators	-	List of operators	Returns a list of the existing materialized operators
6	GET	/ operators /{id}	id	Operator xml	Return the description of a specific operator
7	PUT	/ operators /edit/{id}	id, xml description	-	If the specific operator exists in the library, its metadata are replaced with the new ones. Else it gets added along with its metadata description.
8	GET	/ operators /delete/{id}	id	-	Deletes the specified operator.
9	GET	/ operators /{id}/profile	id	-	Starts the profiling of the specified operator.
10	GET	/ operators /{id}/estimateMetric	id, output metric, list of input metric values	Estimation of output metric	Returns an estimation of the specified output metric for the provided values of the input space metrics.
11	GET	/abstractOperators	-	List of operators	Returns a list of the existing abstract operators
12	GET	/ abstractOperators /{id}	id	Operator xml	Returns the description of a specific abstract operator
13	PUT	/ abstractOperators /edit/{id}	id, xml description	-	If the specific abstract operator exists in the library, its metadata are replaced with the new ones. Else it gets added along with its metadata description.
14	GET	/ abstractOperators /delete/{id}	id	-	Deletes the specified abstract operator
15	GET	/abstractOperators/{id} /checkMatches	id	List of operators	Returns a list of the existing materialized operators that match the specified abstract operator.
16	GET	/abstractWorkflows	-	List of workflows	Returns a list of the existing abstract workflows
17	GET	/ abstractWorkflows /{id}	id	Workflow xml	Returns the description of a specific abstract workflow
18	PUT	/ abstractWorkflows /edit/{id}	id, xml	-	If the specific abstract workflow exists in the library, its metadata

			description		are replaced with the new ones. Else it gets added along with its metadata description.
19	GET	/ abstractWorkflows /delete/{id}	id	-	Deletes the specified abstract workflow
20	GET	/materializedWorkflows	-	List of workflows	Returns a list of the existing materialized workflows
21	GET	/ materializedWorkflows /{id}	id	Workflow xml	Return the description of a specific materialized workflow
22	GET	/ materializedWorkflows /execute/{id}	id	id	Starts the execution of the specified materialized workflow and returns the id of the created running workflow
23	GET	/runningWorkflows	-	List of workflows	Returns a list of the existing running workflows
24	GET	/ runningWorkflows /{id}	id	Workflow xml	Return the description of a specific running workflow
25	GET	/ runningWorkflows /kill/{id}	id	-	Stops the execution of the specified running workflow

3 Implementation

In this section, we describe in detail the current implementation of the IReS platform [7] . We discuss the functionalities provided by the different modules of the platform as well as the intuition behind the architectural and algorithmic decisions made. The code of the IReS platform is open source and can be found in https://github.com/project-asap/IReS-Platform.

3.1 Job Parsing Module

This module[3] is responsible for handling the interaction between the users and the IReS platform. A user should be able to define operators, datasets, workflows, etc. along with their properties and restrictions using a common description framework. The Job parsing module is thus responsible for both defining this description framework and being able to parse and utilize the user provided input.

The main challenges of defining such a metadata description framework are:

- **User extensibility:** Users should be able to define and add their own metadata for operators and datasets. User defined metadata can be used for fine-grained operator description. Using a predefined set of metadata could hinder the extensibility of the platform for supporting new engines and operators.
- **Abstraction:** The IReS platform targets the optimization of multi-engine workflows, examining alternative execution paths of the same conceptual workflow, using various underlying engine and operator implementations. To be able to describe such scenarios, the user should be able to specify the data and operators that compose her workflow in a way as abstract as she desires. The IReS planner and workflow scheduler need to remove that abstraction, find all the alternative ways of materializing the workflow and select the most beneficial, according to the user-defined policy.

3.1.1 Tree-metadata framework

Our proposed metadata framework describes **data** and **operators**. Data and operators can be either **abstract** or **materialized**. Abstract are the operators and datasets that are described partially or at a high level by the user when composing her workflow whereas materialized are the actual operator implementations and existing datasets, either provided by the user or residing in a repository.

Both data and operators need to be accompanied by a set of metadata, i.e., properties that describe them. Such properties include input data types and parameters of operators, location of data objects or operator invocation scripts, data schemata, implementation details, engines etc. The provided metadata can be used to:

(a) Match abstract operators to materialized ones

[3] https://github.com/project-asap/IReS-Platform/tree/master/asap-platform/asap-server

(b) Check the usability of a dataset as input for an operator. If the dataset does not match the operator's input, its metadata can be also used to check for appropriate transform/move operators that can be applied.

(c) Provide optimization parameters like the profiling input/output space (the parameters to take into account and the metrics to measure respectively) or user provided profile functions. This information is based on our black box operator profiling approach described in Section 3.2.

(d) Provide execution parameters like the path of a file in the filesystem or arguments for the execution of the operator script.

To provide such a user extensible metadata framework we opt for a generic tree metadata format. To avoid restricting the user and allow for extensibility, the first levels of the metadata tree are predefined but users can add their ad-hoc subtrees to define their custom data or operators. Moreover, some fields (mostly the ones related to the operator and data requirements) are *compulsory* while the rest are *optional* and user defined. Materialized data and operators need to have all their compulsory fields filled in with information. Abstract data and operators do not adhere to this rule. In general we define the following predefined parts of the meta-data tree:

3.1.1.1 Constraints

This sub-tree contains all the meta-data information that is used to match abstract and materialized operators and datasets. The information contained in this sub-tree should contain input/output specification for operators, algorithm, engine specification and whatever else the user considers that should take part in the abstract/materialized matching of operators. The predefined, compulsory fields of the operator metadata are primarily the number of its inputs and outputs:

```
Constraints.Input.number=<number of inputs>
Constraints.Output.number=<number of outputs>
```

In the above description, the metadata were presented with a key-value representation were the key denotes the path from the root node of the tree to the specified metadata leaf. For each defined input and output the respective specification metadata should be put in the following subtrees:

```
Constraints.Input{id}
Constraints.Output{id}
```

The respective metadata subtrees are automatically matched with the existing datasets in order to check for usability or move/transform operators that should be applied. The respective output metadata specifications are also copied to the metadata of the intermediate output workflow datasets in order to enforce data constraints along the workflow.

3.1.1.2 Execution

This subtree contains all the information required for the execution of a materialized operator. Our execution engine is described in detail in Section 3.4 and utilizes YARN in order to execute a DAG graph of operators. Execution specific metadata like dataset

paths or details about staging in/out files from containers that use their local file system are provided here. This subtree has the following predefined metadata for datasets:

Execution.path=<the path of the dataset>

For operators we have the following metadata:

Execution.LuaScript=<Lua script of the operator>
Execution.Arguments.number=<number of arguments of the execution script>
Execution.Argument{id}=<value of the specific argument>
Execution.Output{id}.path=<the path of the specific output dataset>
Execution.copyToLocal=<list of files that need to be copied in the container before the execution of the operator>
Execution.copyFromLocal=<list of files that need to be maintained after the execution of the operator>

The use of those metadata is further described in Section 3.4, where the Enforcer module of IReS is presented. In general, these metadata give information about the location of execution script for the operator as well as for its arguments. We also give information about stage in and stage out files that are required by the distributed execution of operators using YARN containers.

3.1.1.3 Optimization

This part of the metadata gives information required by the profiler module. They are used to effectively estimate the execution metrics of operators and utilize them to generate execution plans for workflows.

Optimization.inputSpace.{metric name}=<type>
Optimization.outputSpace.{metric name}=<type>
Optimization.model.{metric name}=<UserFunction or Profile>

As can be seen from the above metadata, described in Section 3.2.1, users are able to define the input/output profiling space for each materialized operator. For each of the output metrics the user is able to either provide a user defined function, used for estimation, or state to the system that the metric should be estimated using a profiling procedure.

In the following sections, we give some concrete examples for the metadata of datasets and operators. For better understanding we give both a visual representation of the metadata tree as can be seen from the platform's web interface and also the actual metadata in key-values where the key denotes the path of the specific metadata node.

3.1.2 Dataset metadata description

In this section, we give an example of a dataset description (Figure 4).

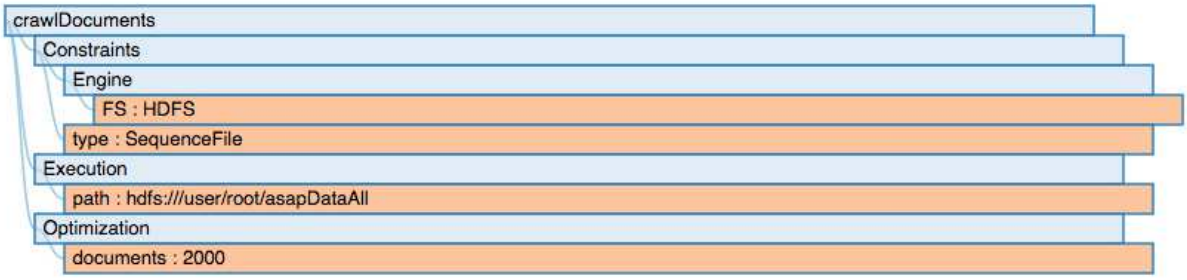


Figure 4: Dataset metadata

Optimization.documents=2000
Constraints.Engine.FS=HDFS
Constraints.type=SequenceFile
Execution.path=hdfs:///user/root/asapDataAll

3.1.3 Operator metadata description

In this section, we give an example of a materialized operator description (Figure 5).

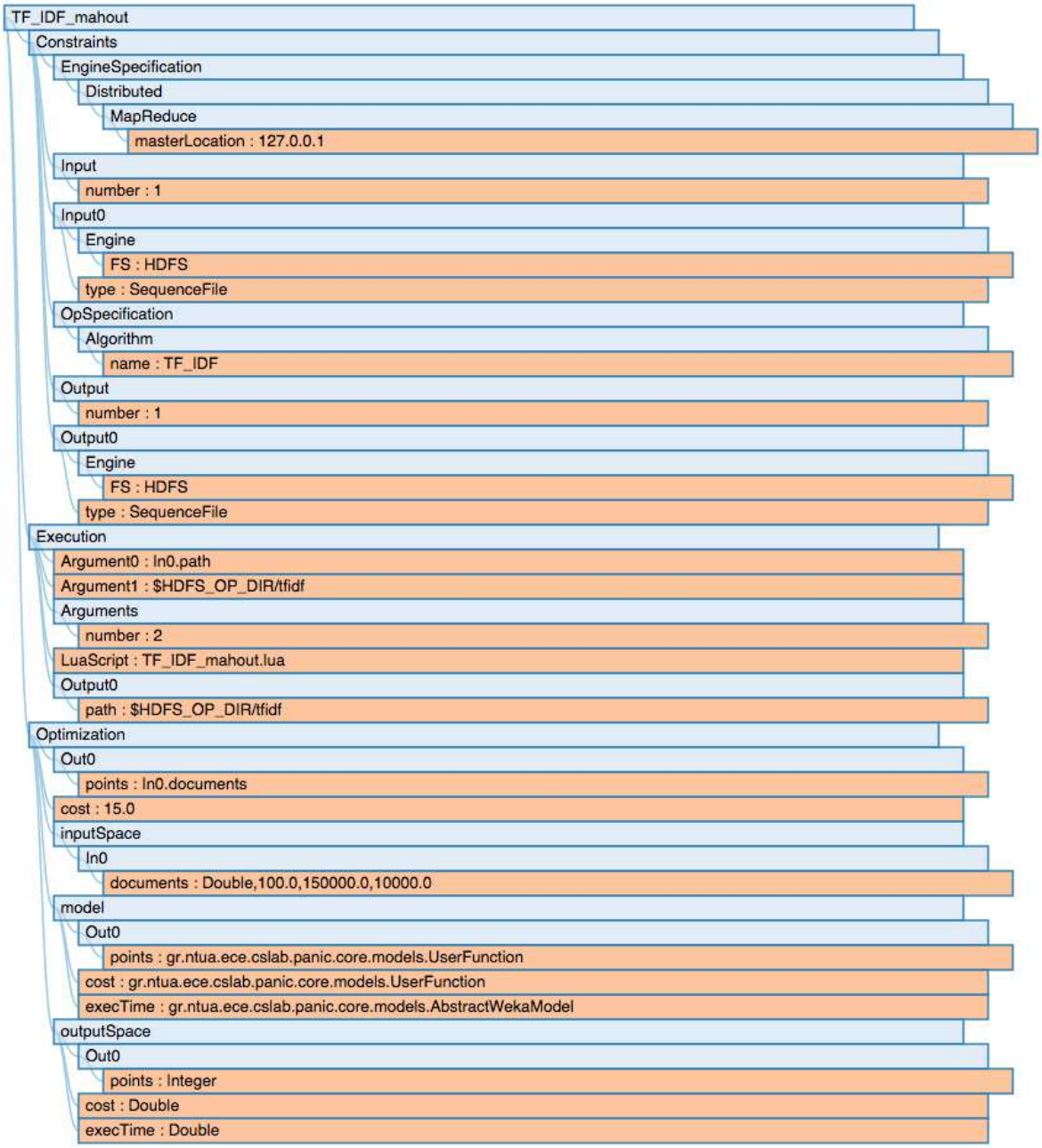


Figure 5: Materialized operator metadata

Constraints.Input.number=1
Constraints.Output.number=1
Constraints.Input0.Engine.FS=HDFS
Constraints.Input0.type=SequenceFile
Constraints.Output0.Engine.FS=HDFS
Constraints.Output0.type=SequenceFile
Constraints.OpSpecification.Algorithm.name=TF_IDF
Constraints.EngineSpecification.Distributed.MapReduce.masterLocation=127.0.0.1
Optimization.inputSpace.In0.documents=Double,100.0,150000.0,10000.0

```
Optimization.outputSpace.execTime=Double
Optimization.outputSpace.Out0.points=Integer
Optimization.outputSpace.cost=Double
Optimization.model.execTime=gr.ntua.ece.cslab.panic.core.models.AbstractWekaModel
Optimization.model.Out0.points=gr.ntua.ece.cslab.panic.core.models.UserFunction
Optimization.Out0.points=In0.documents
Optimization.model.cost=gr.ntua.ece.cslab.panic.core.models.UserFunction
Optimization.cost=15.0
Execution.LuaScript=TF_IDF_mahout.lua
Execution.Arguments.number=2
Execution.Argument0=In0.path
Execution.Argument1=$HDFS_OP_DIR/tfidf
Execution.Output0.path=$HDFS_OP_DIR/tfidf
```

3.1.4 Tree-metadata matching

Apart from materialized operators and datasets the user of the IReS platform can define abstract operators and datasets that are used for creating abstract workflows and can be matched with the existing materialized ones in order to find all possible execution plans. Abstract operators are described using the same tree metadata framework, described in the previous sections. The main difference is that abstract operators can have less metadata attributes than the materialized ones. We also allow users to add regular expressions in the abstract operator metadata. This is done in order for IReS platform to be able to support more generic matching. For example the * symbol under a field means that the abstract operator can match materialized ones with any value in that field.

The matching procedure checks if all the metadata of the abstract operator are present in (match if they are regular expressions) the materialized operator. To make this check efficient, the metadata trees are stored in main memory tree structures. The tree structure used store all children of a metadata node in a sorted list according to their name. Thus, if both metadata trees are stored with ordering we can perform a merge check of both trees in order to find if the operators match. This procedure iterates over the sorted metadata and tries to match the abstract with the materialized ones. To check the matching of two operators we require, in worst case, only one pass over the metadata of both operators. Thus, the matching process is linear to the size of the metadata trees and can be used very efficiently.

3.1.5 Abstract operator description

In this section, we give an example of an abstract operator description (Figure 6).

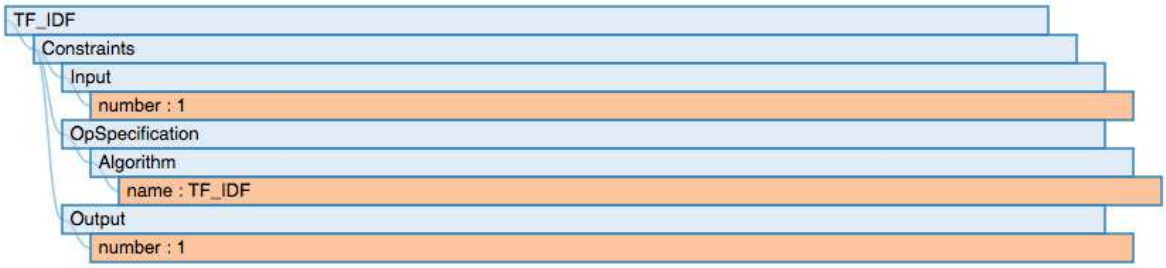


Figure 6: Abstract operator metadata

```
Constraints.Output.number=1
Constraints.Input.number=1
Constraints.OpSpecification.Algorithm.name=TF_IDF
```

As we can see, the abstract operator contains metadata only under the constraints subtree because only those are used for the matching procedure. It mainly targets the matching of the algorithmic operation of the operators as well as the matching of inputs and outputs used. This operator matches with the materialized TF_IDF operator presented in the previous section.

3.1.6 Abstract workflow description

In this section, we present the description of an abstract workflow. The user of the IReS platform has the ability to describe a workflow in an abstract way and the let the system find all possible matches for the operators and generate the materialized workflow that contains all the possible alternative execution plans. An abstract workflow can be created using both materialized and abstract datasets and operators. Materialized datasets are used to define the already existing input datasets of the workflow. Abstract datasets can be used for defining the intermediate results that are created after the execution of a specific operator. These abstract datasets will get concrete specifications from the materialized operator's output specifications when the materialized workflow is generated. Concerning operators, the user can create her workflow using materialized operators that exist in the operator library or abstract operators that match with several of the existing materialized operators.

An example of an abstract workflow is depicted in Figure 7.

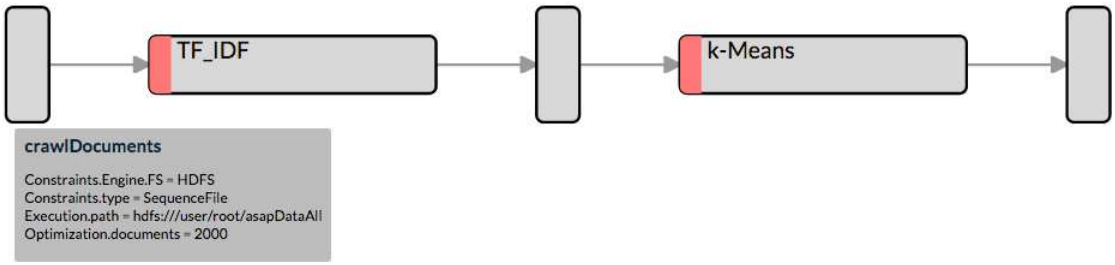


Figure 7: Abstract workflow description

An abstract workflow is defined as a DAG graph that connects a mixture of abstract and materialized datasets and operators. The missing information needed for describing the DAG graph is a set of edges. For example the description of the previous workflow can be created using the following list of edges (d1 is the output of TF_IDF and d2 is the output of k-Means).

```
crawlDocuments,TF_IDF
TF_IDF,d1
d1,k-Means
k-Means,d2
d2,$$target
```

A special tag $$target is used to define which dataset is the final output of the DAG graph.

3.2 Profiling and modeling modules

In this section, we present the techniques used by the IReS platform in order to estimate the performance and cost characteristics of different operators over various engines, offering adaptive and highly extensible analytics execution. The main idea is to provide predictions for each operator's performance by actually running the operator in representative configuration combinations. Although the number of these combinations grows exponentially to the number of inputs, ASAP's Profiler is able to intelligently narrow down the field of profiling scenarios and to maximize the accuracy of the produced models under specific temporal and monetary constraints, as discussed in Section 3.2.2. Using these measurements, we can train surrogate estimator models that can be used to approximate its performance for non-tested configurations. To do so in a generic and extendable way, we propose a black-box operator profiling framework. The code of the profiling and modeling modules is available at the github project repository[4].

3.2.1 Black box profiling approach

In order to provide a generic operator profiling framework, we follow a "black box" approach. According to this, we model each operator as a black box that has user defined

[4] https://github.com/project-asap/IReS-Platform/tree/master/panic

inputs and outputs. The input space of an operator contains all the parameters that affect its performance and need to be varied in order to profile it. For example, the input space of an operator may contain parameters like:

- Amount of resources (e.g. number of VMs, number of CPUs, available RAM, etc.)
- Data complexity (e.g. dataset size in GBs, type, distribution, etc.)
- Operator parameters (e.g. k in k-means, number of iterations, accuracy, etc.)
- As mentioned before, the input space of an operator is user defined, giving the users the capability of defining the parameters that affect the operator's performance. The user should also give the type of each parameter in order for our profiling system to be able to vary it and test different configuration automatically. For example, a parameter like number of VMs is a discrete integer value that can have a minimum and maximum value in order to prune the possible combinations. Concerning data input parameters, like the dataset size or type, the user can provide a set of sample datasets or a dataset generator that can be used in order to test various configurations.
- The output space of an operator can be also described as its optimization space and contains all performance/cost metrics that need to be approximated for the various input configurations. For example, the output space of an operator can contain the following metrics:
- Performance metrics (Execution time, throughput, latency, etc.)
- Cost metrics (monetary or resource usage, e.g., memory utilization, CPU, iops etc.)
- Operator-related metrics (e.g., number of results, output size, etc.)

Currently we profile 45 metrics, including:

- Execution time
- All the monitoring metrics reported by the ganglia[5] monitoring tool (e.g., CPU, RAM, iops, network traffic etc.)
- Operator-related metrics (e.g., #clusters in k-means, output size, etc.).
- Our profiling framework is extensible and allows the user to define the output parameters that she wants to model for a specific operators. Such a parameter can be defined simply by providing a *monitoring probe* that can measure it as the operator executes.

3.2.2 Profiling challenges

The operator profiling is a process that allows the automated execution of operators and monitors their behavior over representative input space configurations. The collected information can form the basic knowledge used to train *surrogate* estimator models that can approximate the operator's behavior (the function that relates the input/configuration space parameters with the output/optimization metrics).

[5] http://ganglia.info/

The main challenge for the Profiler is to intelligently choose the set of profiled configurations. For example, if we have an operator with 3 integer input parameters that range from 1 to 10, there exist 10^3 different deployment configurations. Furthermore, each execution of an operator has a respective temporal and monetary cost in order to be sufficiently profiled. A brute force profiler would need to execute and monitor all those configurations. In such case, the execution time of the profiler could be exponential to the number of inputs, something which is not acceptable. ASAP's Profiler should be able to intelligently narrow down the field of profiling scenarios. Therefore, the Profiler attempts to tackle the problem of generating the most accurate operator profile within a user specified profiling budget of experiments.

The nature of operator profiling is clearly multi-objective, often requiring tradeoffs between diverse and conflicting objectives. While the input parameters, design space, of an application include the number of VMs, their RAM, their disk capacity etc., an application user can be interested in various objectives such as cost, throughput, latency etc. Therefore, the operator can be modeled as a function that maps the design space (number of VMs, RAM, data size, operator parameters, etc.) to the user defined objective space (cost, execution time, etc.). This function represents the operator's profile. Our Profiler will use targeted operator runs, according to a specified financial and time budget, to provide a global surrogate approximation model of the operator's profile function that maps its input space to its optimization/output space.

Many engineering and science problems require expensive experiments or time consuming simulations to generate sample points of the mapping between the input and the output parameters of a system. In such cases, researchers have focused on building accurate surrogate approximation models that, when properly constructed, can mimic the behavior of the system while being computationally cheap to evaluate. Examples of surrogate models include: Kriging models [36] [1] , Splines [5] Artificial Neural Networks [8] , Support Vector Machines [28] etc. The challenge here is how to generate a surrogate model that is as accurate as possible over the domain of interest and at the same time minimize the cost of the performed experiments. Since the system's response behavior is not known upfront and the sample data points are too costly to obtain, the main approach followed is the iterative adaptive sampling of the design space. Each data point obtained is used to update the surrogate approximation model as well as the sampling function. In each iteration, the sampling function selects the next sampling point according to an estimation of its benefit to the surrogate approximation accuracy. This technique is called importance or adaptive sampling and is also known as sequential design.

3.2.3 Profiling approach

In Algorithm 1 (Figure 8), we provide the general methodology used to create a profile for a given operator. The algorithm expects a valid operator/application description A followed by an input domain D, representing the possible setups the operator can be executed with and a list of surrogate models. The profiling process occurs iteratively: while the termination condition is not fulfilled, the domain space is sampled, a new point p is picked and the operator is executed according to p. The deployment produces an optimization vector d, containing the measured outputs, which is then used to train in an

incremental manner all the available surrogate models. The output of the profiling process is the surrogate model which achieves the highest accuracy, according to a user specified metric.

Algorithm 1 Main profiling algorithm

Require: application A, input domain D, models M
Ensure: model m
1: **while** not termination_condition **do**
2: $p \leftarrow$ SAMPLE(D)
3: $d \leftarrow$ DEPLOY(p)
4: **for** m $\in M$ **do**
5: m.train_incrementaly(p,d)
6: **end for**
7: **end while**
8: **return** best_model(M)

Figure 8 Main profiling algorithm

The termination condition can vary. It can be a threshold of sampled points that, if reached, the algorithm terminates. In other cases, it can be related to the achieved accuracy: if the trained model achieves to predict the objective function with error lower than a user defined threshold, the termination condition is reached. As we will present in the following section, the nature of the termination condition is directly entwined with the nature of the sampling algorithm.

3.2.4 Adaptive sampling

The sampling procedure occurs at the beginning of each profiling loop. The sampler receives as input the domain space D of the operator, which determines all the acceptable deployment points. Each point returned by the sampler is used for execution. The operator's output metrics are measured and then an approximation model is trained using the acquired information.

There are many methodologies for sampling a multidimensional space. We can categorize the methods we support in the following categories:

1. Static sampling, where the sampler needs no other information than the domain space characteristics (dimensions and acceptable values) to pick the next sample
2. Adaptive sampling, where the sampler exploits the knowledge obtained by the deployment of previously picked samples.

The static approach does not take into consideration the operator's performance. Typical examples of static sampling are the Random sampler that returns random points and the Uniform sampler which constructs a multidimensional grid in the input space D, and returns points belonging to the grid. We opt for an adaptive sampling approach which exploits the knowledge obtained from each deployment/sample, enabling the sampler to retrieve more samples in regions of the domain space D where the performance appears to have fluctuations or the models have the maximum estimation errors. Equivalently, an adaptive sampler favors areas of D where the operator performance has the most deviations in order to use them to provide more accurate

approximation models. There are no theoretical guarantees for the adaptive sampling technique. In fact the samples are biased and this has proved to beneficial for the specific use case of application profiling [11] .

3.2.5 Approximation models

When a new sample is picked by the sampler and executed, the performance metrics are stored and given as input to an approximation model. The training set of the model consists of the chosen samples along with their output values. After the training process is finished, the model will be able to approximate the objective function for the entire space D. There exist many methodologies for approximating an unknown function. We can categorize them in two major categories: regression based techniques and classification techniques. Algorithms on the former category create an analytical form of the objective function. The classification techniques, on the other hand, do not target to create an analytical function but to classify the points of the domains space in classes. These objects are treated in a similar manner, indicating that the same properties stand for objects in the same class.

In our approach, we utilize the approximation models offered by WEKA [43] , an open source data mining software which implements a variety of machine learning algorithms. Specifically, the supported approximation techniques are the following:

- Gaussian Process, that approximates the objective function using Gaussian distributions
- Multilayer Perceptron, that represents a typical neural network with many hidden layers and neurons
- Linear Regression (Least Median Squares), that implements the methodology introduced at [34]
- Bagging, that executes classification as described in [2]
- Random SubSpace, that constructs a decision tree using the approach presented in [14]
- Regression by Discretization, that enforces regression over a discretized domain of the input space
- RBF Network, which trains a Radial Basis Function Network, as presented at [3]

The accuracy of each one of the aforementioned models is highly affected from the configuration of the model and the nature of the objective function. For example, a linear hyper- plane will be approximated faster using a linear regression method. On the contrary a complex surface which has spikes and valleys is more likely to be approximated more accurately using a non-linear approach. All the available models are trained in parallel by the system, and the model which achieves the best accuracy is eventually chosen.

3.3 Decision making module

In this section, we describe the decision making module[6] and in particular the algorithms used in order to intelligently explore the space of all available execution

[6] https://github.com/project-asap/IReS-Platform/tree/master/asap-platform

plans of a workflow and decide on the execution plan that best fits to the user-provided optimization objectives. In analogy to traditional query optimizers, the IReS Decision Making module tries to approximate the optimum by comparing several alternatives to provide in a reasonable time a "good enough" plan which typically does not deviate much from the best possible result[7]. In the following, the term "optimizer" is used in exactly this sense. The algorithm of our dynamic programming (DP) optimizer is depicted in Figure 9.

ALGORITHM 1: *optimizeAbstractWorkflow*$(G(Datasets, Operators), target)$

```
//G(Datasets, Operators) : abstract workflow graph
//Datasets : set of datasets
//Operators : set of abstract operators
//target : target dataset
for d ∈ Datasets do
    //initialize dpTable
    if d.isMaterialized() then
        if d == target then
            return 0;
        end
        dpTable[d].insert(d, 0);
    end
end
for o ∈ Operators following DAG topological ordering do
    MOperators = findMaterializedOperators(o);
    for mo ∈ MOperators do
        inputCost = 0;
        for in ∈ mo.getInputs() do
            minCost = ∞;
            for tin ∈ dpTable[in] do
                if tin.matchWithOperatorInput(mo) then
                    if tin.getCost < minCost then
                        minCost = tin.getCost;
                    end
                else
                    if tin.checkMove(mo) then
                        moveCost = tin.getCost + tin.moveCost(mo);
                        if moveCost < minCost then
                            minCost = moveCost;
                        end
                    end
                end
            end
            inputCost+ = minCost;
        end
        operatorCost = estimateCost(mo);
        cost = inputCost + operatorCost;
        for out ∈ o.getOutputs() do
            tout = outputFor(mo, out);
            dpTable[out].insert(tout, cost);
        end
    end
end
return dpTable[target].getMinCost();
```

Figure 9 Dynamic programming workflow optimizer

[7] https://en.wikipedia.org/wiki/Query_optimization

Our optimizer explores the space of alternative execution plans generating the materialized workflow graph and selecting the optimal plan with respect to the user optimization policy. It maintains a dpTable structure that is responsible for storing the best execution plan for each different format of a dataset node. Due to the fact that intermediate results can be produced in different formats, e.g. csv, json, etc., we maintain for each different produced format the best plan. Our optimizer processes all abstract operators of the workflow following a DAG topological order, which can be found using a depth-first search. This ordering ensures that when we process an operator all its predecessors in the DAG will have been processed and thus the dpTable will contain the optimal plans for all its inputs.

For each abstract operator, we search the library of available materialized operators to find all matches. As discussed in Section 3.1.4, we use an efficient tree matching algorithm to avoid unnecessary comparisons and follow the hierarchical structure of the tree-based metadata constraints. The optimization process consults the input specifications of materialized operators adding, if this exists in the operator library, the required move/transform operators to utilize all different produced inputs. All such input formats are recorded in the dpTable along with their costs.

Here we make the assumption that operator alternatives have a 1 to 1 relationship (we do not yet consider the possibility of one operator being equivalent to a combination of 2 or more operators) and that only one move/transform operator is used to match consecutive operators with different output/input formats.

Consequently, to estimate operator performance metrics (e.g., cost, execution time) our planner consults the *Model DB* that holds estimator models for each one of the materialized operators. In our current implementation, the planner is configured to optimize one metric or a function of multiple performance metrics that the user is interested in. We are currently investigating methods for optimizing multiple dimensions of performance metrics, such as finding Pareto frontier execution plans. After estimating the operator cost, we add all its output datasets in the dpTable. When all abstract operators have been processed, the optimal cost of the target dataset is returned using the respective dpTable record.

In conclusion, given:

- the performance and cost estimations provided by the models (which may or may not be accurate) and
- the alternative plans that are created under the assumption that equivalent operator implementations have a one-to-one relationship (we do not yet consider the possibility of one operator being equivalent to a combination of 2 operators) and that only one move or transform operator can be used to link operators with different input/output formats,

the DP algorithm of the Decision making module ensures the selection of the optimal alternative plan, which is a very good approximation of the best possible plan.

We now study the complexity of our optimizer. Let us assume that a workflow contains $\boldsymbol{op}$ number of abstract operators and the maximum number of materialized operators that match to an abstract operator is $\boldsymbol{m}$. Let us also assume that operators have a maximum number of $\boldsymbol{k}$ inputs. In detail, for each intermediate dataset, our dpTable will contain at most $\boldsymbol{m}$ records, each generated from one of the $\boldsymbol{m}$ materialized operators, of the abstract operator that has it as output. Therefore, the inner dpTable loop of Figure 9 will run at most $\boldsymbol{m}$ times. Therefore the worst case complexity of our optimizer is:

$$\boldsymbol{O(op * m^2 * k)}$$

The following figures depict two examples of abstract workflows along with their respective materialized workflows that contain their alternative execution plans. More precisely, Figure 11 depicts the alternative execution plans of the abstract workflow of Figure 10, which contains a TF/IDF operation over a corpus of documents followed by k-means clustering. Figure 13 materializes an SQL query over TPC-H data (abstract workflow in Figure 12).

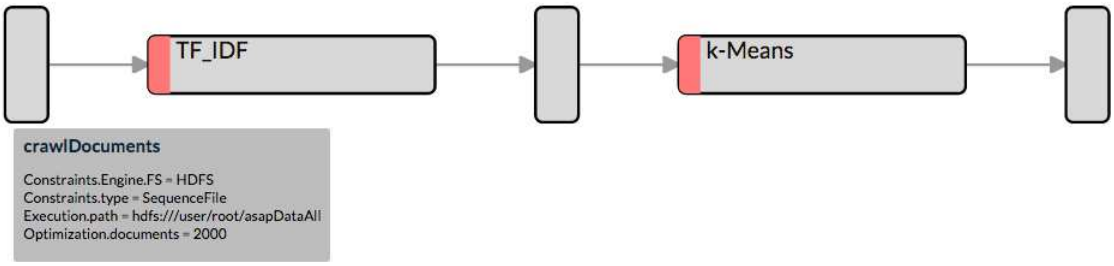


Figure 10: Abstract TF/IDF, k-Means workflow

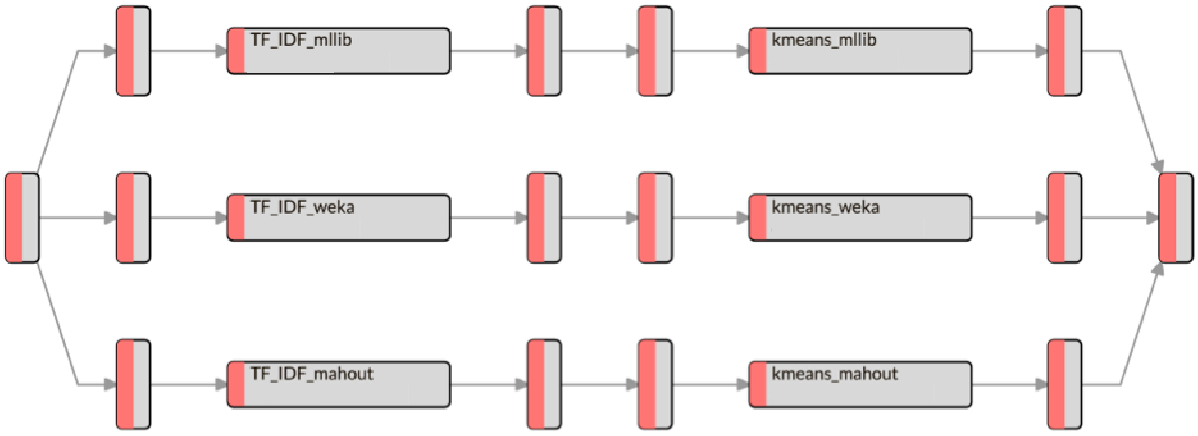


Figure 11: Alternative plans for the TF/IDF, k-Means workflow

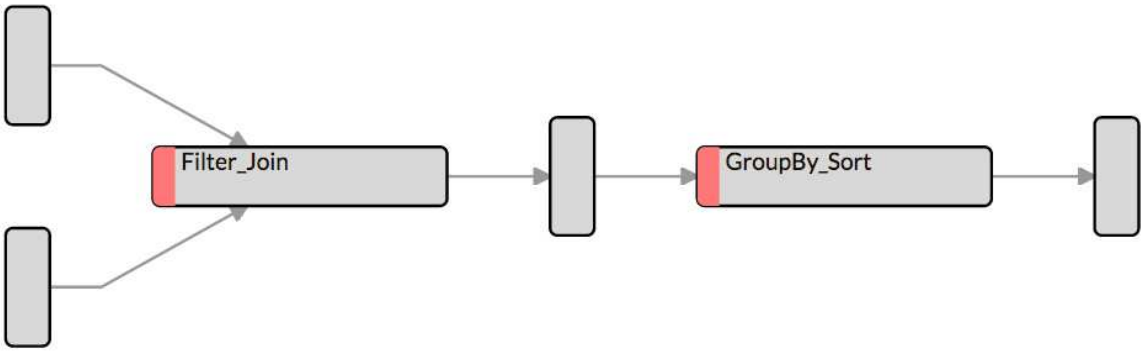


Figure 12: Abstract TPCH SQL query workflow

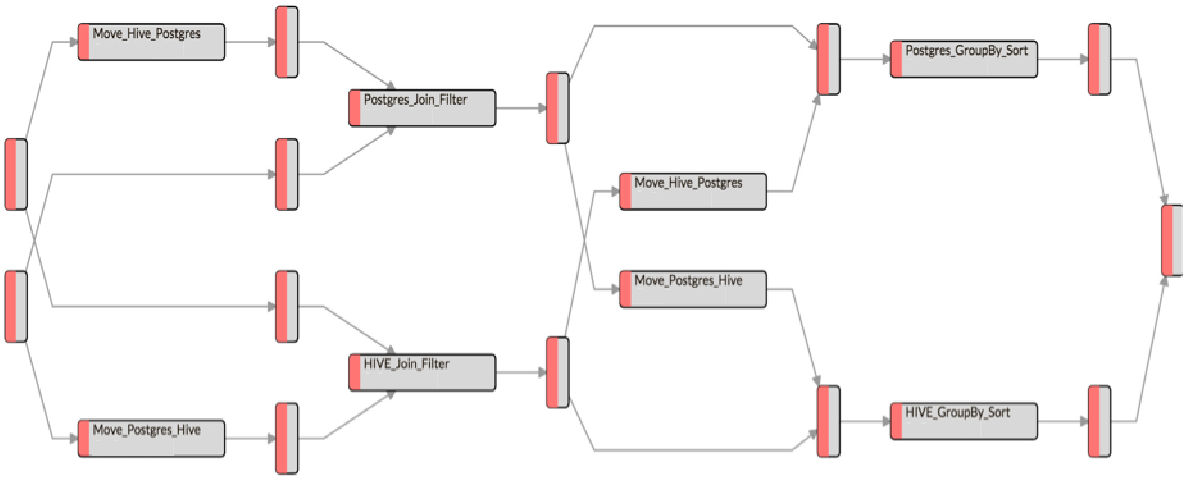


Figure 13: Alternative plans for the TPCH SQL query workflow

We note that, for the materialized TPC-H SQL query workflow depicted in Figure 13, move operators were used in order to transfer intermediate results between the two SQL engines, Hive and PostgreSQL. These move operators were automatically added by the platform in order to match the inputs of the discovered materialized operators. However, observing the TF/IDF and k-means workflow we note that intermediate results were not utilized across different engines due to the fact that the respective move operators were not available in the IReS library.

For our running examples, let us assume a user optimization policy that targets the minimization of execution time. Intuitively, small datasets run faster in a centralized manner while distributed implementations outperform the centralized ones for bigger datasets. In the following figures (Figure 14 and Figure 15 respectively) we see the optimization result for our two example workflows. In the case of the first workflow consisting of TF/IDF followed by k-means, the WEKA implementation is estimated to be the fastest for both steps, due to the small size of the document corpus. The second workflow joins and sorts two TPC-H tables. Since one of the initial tables is large, the join is expected to run faster in Hive than in PostgreSQL. Contrarily, since the output of the

join is relatively small, its sorting is expected to perform better in PostgreSQL. The selected execution plans for each workflow is marked in green.

In the course of the workflow execution, the real-time monitoring information is fed back to the decision making module in order to take into account current running conditions and adapt accordingly. Moreover, our planner considers more than a single final plan to ensure that alternatives will exist in case of failures or other unpredictable circumstances.

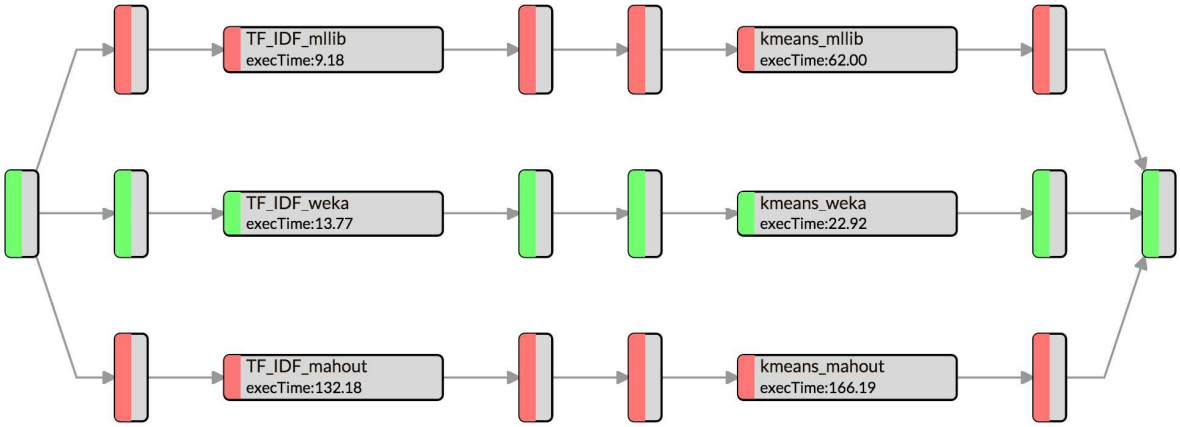


Figure 14: Selected TF/IDF, k-Means execution plan

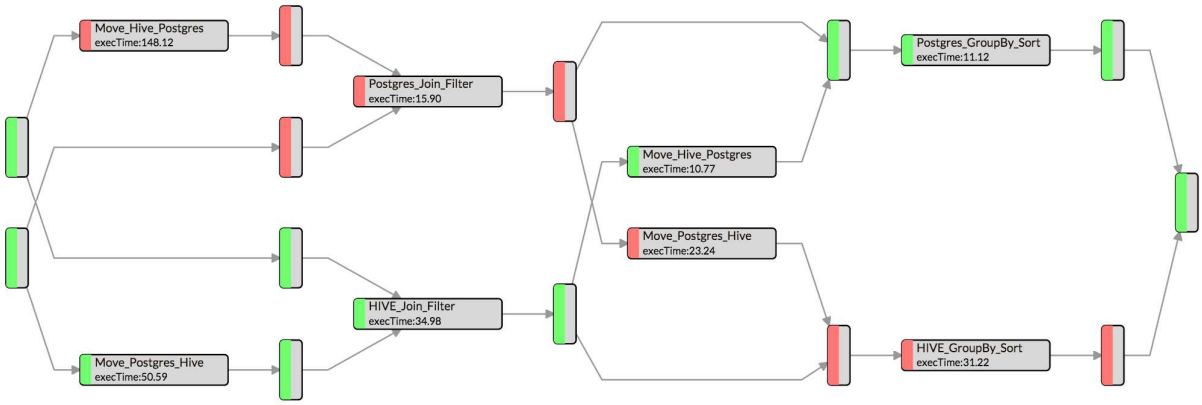


Figure 15 Selected TPC-H SQL query workflow execution plan

3.4 Enforcer module

In this Section, we describe the enforcer module[8] of the IReS platform. This module undertakes the execution of the selected execution plan. In the era of big data, clusters of commodity servers as well as clusters of cloud resources have become the primary computing platform choice. Such clusters power large Internet services and a growing

[8] https://github.com/project-asap/IReS-Platform/tree/master/cloudera-kitten

number of data-intensive applications. Additionally, a large and diverse selection of computing frameworks has been and is being developed in order to take advantage of those cluster resources. In this landscape, where organizations run multiple cluster computing frameworks and in which each framework has its own advantages and disadvantages, a cluster multiplexing approach emerges as the best solution for resource utilization. Resource allocation and scheduling frameworks like Yarn [42] and Mesos[9] have been introduced. Those frameworks target the fine-grained resource allocation, in a container level, as well as the online resource scheduling and sharing between various cluster-computing frameworks.

In order for the IReS platform to be able to fit in this landscape and integrate with the various cluster computing frameworks, we have build our enforcer module on top of the YARN resource scheduler. Our enforcer module requests container resources from YARN in order to launch the execution of operators. It also orchestrates the execution of a DAG graph of operators in order to successfully execute the selected workflow execution plans.

3.4.1 YARN workflow execution engine

In order to provide the above-mentioned functionality, our enforcer module extends the Apache Kitten[10] framework. Apache Kitten is a framework that lets you define the execution of operators on top of YARN. It allows the description of resource configuration (CPU, RAM etc. of the containers launched) as well as the execution configuration of the script or commands that need to be executed inside the allocated container resources. We extend Apache Kitten in order to add support for the execution of a DAG of operators that is required for our workflow execution.

Apache Kitten is a set of tools for writing and running applications on YARN, the general purpose resource scheduling framework that ships with Hadoop 2.2.0. Kitten handles the boilerplate around configuring and launching YARN containers, allowing developers to easily deploy distributed applications that run under YARN. Kitten makes extensive use of Lua tables to organize information about how a YARN application should be executed. Here is how Kitten defines an example of a distributed shell application:

```
distshell = yarn {
        name = "Distributed Shell",
        timeout = 10000,
        memory = 512,
        master = {
        env = base_env, -- Defined elsewhere in the file
        command = {
             base="java -Xmx128m com.cloudera.kitten.appmaster.ApplicationMaster",
             args = {
                "-conf job.xml" },
                }
        },
```

[9] http://mesos.apache.org/

[10] https://github.com/cloudera/kitten

```
        container = {
                instances = 3,
                env = base_env,  -- Defined elsewhere in the file
             command = "echo 'Hello World!' >> /tmp/hello_world"
        } }
```

The *yarn* function of the Lua description provides all the required information for running an operator using YARN. The following fields can be defined in the Lua table that is passed to it, optionally setting default values for optional fields that were not specified:

1. **name** (string, required): The name of this application.
2. **timeout** (integer, defaults to -1): How long the client should wait in milliseconds before killing the application due to a timeout. If < 0, then the client will wait forever.
3. **user** (string, defaults to the user executing the client): The user to execute the application as on the Hadoop cluster.
4. **queue** (string, defaults to ""): The queue to submit the job to, if the capacity scheduler is enabled on the cluster.
5. **conf** (table, optional): A table of key-value pairs that will be added to the Configuration instance that is passed to the launched containers via the job.xml file. The creation of job.xml is built-in to the Kitten framework and is similar to how the MapReduce library uses the Configuration object to pass client-side configuration information to tasks executing on the cluster.
6. **env** (table, optional): A table of key-value pairs that will be set as environment variables in the container. Note that if all of the environment variables are the same for the master and container, you can specify the **env** table once in the yarn table and it will be linked to the subtables by the yarn function.
7. **memory** (integer, defaults to 512): The amount of memory to allocate for the container, in megabytes. If the same amount of memory is allocated for both the master and the containers, you can specify the value once inside of the yarn table and it will be linked to the subtables by the yarn function.
8. **cores** (integer, defaults to 1): The number of virtual cores to allocate for the container. If the same number of cores is allocated for both the master and the containers, you can specify the value once inside of the yarn table and it will be linked to the subtables by the yarn function.
9. **instances** (integer, defaults to 1): The number of instances of this container type to create on the cluster. Note that this only applies to the **container/containers** arguments; the system will only allocate a single master for each application.
10. **priority** (integer, defaults to 0): The relative priority of the containers that are allocated. Note that this prioritization is internal to each application; it does not control how many resources the application is allowed to use or how they are prioritized.
11. **tolerated_failures** (integer, defaults to 4): This field is only specified on the application master, and it specifies how many container failures should be tolerated before the application shuts down.

12. **command/commands** (string(s) or table(s), optional): **command** is a shortcut for **commands** in the case that there is only a single command that needs to be executed within each container. This field can either be a string that will be run as-is, or it may be a table that contains two subfields: a **base** field that is a string and an **args** field that is a table. Kitten will construct a command by concatenating the values in the args table to the base string to form the command to execute.
13. **resources** (table of tables, optional): The resources (in terms of files, URLs, etc.) that the command needs to run in the container. YARN has a mechanism for copying files that are needed by an application to a working directory created for the container that the application will run in. These files are referred to in Kitten as **resources.**

3.4.2 Execution description

As mention in Section 3.1.3, all materialized operators are accompanied by a set of execution metadata that are used for their actual execution. The main part of the execution description is the lua script that was mentioned in the previous section and is used to describe the execution details of an operator. An example description of an operator using a lua script is presented below:

```
-- The command to execute.
SHELL_COMMAND = "./tfidf_mahout.sh"
-- The number of containers to run it on.
CONTAINER_INSTANCES = 1
-- The location of the jar file containing kitten's default ApplicationMaster implementation.
MASTER_JAR_LOCATION = "kitten-master-0.2.0-jar-with-dependencies.jar"

-- CLASSPATH setup.
CP = "/opt/hadoop-2.6.0/etc/hadoop:/opt/hadoop-2.6.0/etc/hadoop:/opt/hadoop-2.6.0/etc/hadoop:/opt/hadoop-2.6.0/share/hadoop/common/lib/*:/opt/hadoop-2.6.0/share/hadoop/common/*:/opt/hadoop-2.6.0/share/hadoop/hdfs:/opt/hadoop-2.6.0/share/hadoop/hdfs/lib/*:/opt/hadoop-2.6.0/share/hadoop/hdfs/*:/opt/hadoop-2.6.0/share/hadoop/yarn/lib/*:/opt/hadoop-2.6.0/share/hadoop/yarn/*:/opt/hadoop-2.6.0/share/hadoop/mapreduce/lib/*:/opt/hadoop-2.6.0/share/hadoop/mapreduce/*:/contrib/capacity-scheduler/*.jar:/opt/hadoop-2.6.0/share/hadoop/yarn/*:/opt/hadoop-2.6.0/share/hadoop/yarn/lib/*"

-- Resource and environment setup.
base_resources = {
  ["master.jar"] = { file = MASTER_JAR_LOCATION }
}

base_env = {
  CLASSPATH = table.concat({"${CLASSPATH}", CP, "./master.jar", "./tfidf_mahout.sh"}, ":"),
```

```
}

-- The actual distributed shell job.
operator = yarn {
  name = "TF/IDF using mahout library",
  timeout = -1,
  memory = 2048,
  cores = 2,

  container = {
    instances = CONTAINER_INSTANCES,
    env = base_env,
    resources = {
    ["tfidf_mahout.sh"] = {
       file = "/opt/asap-server/asapLibrary/operators/TF_IDF_mahout/tfidf_mahout.sh",
      type = "file",              -- other value: 'archive'
      visibility = "application",  -- other values: 'private', 'public'
           }
    },
    command = {
           base = SHELL_COMMAND,
    }
  }
}
```

4 Infrastructure and deployment

This chapter describes the testbed on which the IReS platform has been deployed and tested. This includes technical details about the physical infrastructure, the runtimes and data stores that have been installed and the operators that have been implemented and imported to the IReS operator library. The engines and operator implementations that have so far been considered form an initial set that allows us to execute some of the workflows defined by our use case partners (see D8.2 and D9.2). This set will be enriched in the course of the project with additional engines and operators to accommodate the ASAP related use cases of web and telecommunication analytics, as well as general purpose analytics workflows.

4.1 Engines

This section is a presentation of the Data Analytics engines that have been tested thus far with the IReS platform. A more complete presentation of the various software ecosystems available by the industry and academia is available in D3.1.

4.1.1 Hadoop and MapReduce

Hadoop [24] is the de facto standard in batch Big-Data processing. It started as framework implementing a distributed file system and a Map-Reduce execution engine, running over that filesystem. In the last 10 years of its lifetime, it has evolved in an large ecosystem of open-source software solutions for big data processing, offering products for a diverse set of needs from NoSQL Databases [18] [16] to in-memory stream processing [20] . We will focus only on the products of the Hadoop Ecosystem that we have used with the IReS platform. Those are mainly MapReduce Libraries ran over Yarn [17] . Their main characteristics and design principles will be presented.

MapReduce [26] is the core execution engine that most of the analytics operations of Hadoop use to access the filesystem and manipulate the stored data. Any operation using it needs to be expressed in steps of Map-Combine-Shuffle-Reduce phases. This phased execution also necessitates the spilling of intermediate data (after the Combine phase) to disk. This approach although very powerful and adaptive, carries the inherit tradeoff of high administrative overhead due to the intermediate disk access.

That being said the combination of HDFS-MapReduce that is in the core of Hadoop has been the go-to choice for most all of batch processing for Web-Scale datasets.

In more modern Hadoop installations the responsibility for managing the computing resources for the execution of MapReduce jobs lies not within the library itself but with the Yarn [14] resource negotiator.

4.1.1.1 HDFS

HDFS ("the Hadoop Distributed File System") is the basic layer of the Hadoop Framework. It is developed in Java, runs in userspace and shares some design principles with GFS [10] . HDFS, like GFS, was created to run on top of commodity hardware. Its most crucial design goal is scalability to a high number (100s) of nodes, each offering

local computation and storage. It also offers redundancy for fault tolerance and ad-hoc node additions.

It was originally used to store files that are large (GBs to TBs), immutable and sequentially read and written. Those are still the use cases that are most fitting for HDFS, but many others have been introduced in its lifetime.

HDFS was also mostly used in bare-metal installations with an emphasis on IO performance. Lately, it has been used often in virtualized environments too.

The architecture model of HDFS benefits from the use of a large number of nodes with dedicated storage. The client receives metadata information from a "Namenode" server (that is replicated for fault tolerance) and then can stream the file data in parallel from, possibly, a large number of "Datanodes", thus leveraging their aggregate sequential IO performance. When used for large files and parallel sequential computations, this approach is very effective in handling large amounts of bulk data efficiently.

However it does not offer good performance in handling requests for random seeks of small amounts of data. Those use cases incur a significant performance penalty for using HDFS and are more suited to main-memory cached storage.

In the use cases and operations we have tested with IReS, HDFS is indeed used to read/write large, sequential, write-once files. We believe those are the most common and well-suited use cases of HDFS.

4.1.1.2 Mahout

Mahout is a widely used ML library for Hadoop MapReduce. It has evolved as a stable and dependable tool for Big Data analysis.

Mahout takes the aim of bringing conventional ML algorithms to the Big Data era, and it does so with the use of MapReduce library and the M-R paradigm. There is a recent attempt to bring some of Mahout's operations to Spark, but it is in a very experimental stage.

Mahout inherits the basic performance limitations of MapReduce. Namely, it needs a long time (in the order of minutes) to initialize a job, it suffers an administrative overhead and also, it uses HDFS which is inefficient for handling smaller input sizes.

4.1.1.3 HIVE

Hive brings SQL-like Relational operations to an HDFS back-end. It is a very popular tool because it allows developers to access Big-Data datasets stored in the distributed filesystem with familiar SQL operations. However HIVE uses MapReduce as its back-end. Therefore, contrary to conventional RDMS, HIVE cannot provide interactive performance for its queries. At the same time the user can use custom M-R code in conjunction with HIVE queries to express complex data manipulations easily.

HIVE is mainly used in Data Warehousing applications where a simple interface to access large amounts of data is necessary and interactive response times are not.

The use case we built for HIVE is running a typical SQL query over tabular data stored in HDFS as large text files.

4.1.2 Spark and MLib

Spark is a relatively new execution engine. It is based on the RDD abstraction which is:

"a read-only, partitioned collection of records. RDDs can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs" [45] .

RDDs are suited for in-memory computations and are mainly comprised from data that are retrievable from a filesystem and a timeline of mutations that need to be performed on them.

Spark's RDD abstraction is a very powerful and promising concept and can overcome many of the limitations of the Map-Reduce paradigm. In theory RDDs can be orders of magnitude faster for iterative, graph [19] and streaming computations [38] .

Spark promises graceful fall back to secondary storage for datasets that cannot fit to memory and integration to Scala, Python and Java.

Spark is in active development with a release cycle of a few months and much effort in the open source community is aimed at enriching its libraries and ensuring seamless and efficient operation. However many parts of its implementation and documentation (particularly for Python and Java) are not as mature as other ML libraries. More importantly, in our experiments so far an order of magnitude difference is not observed for datasets of few GBs.

We are using Spark as an alternative to Hadoop and Mahout for both batch and iterative operations.

4.1.3 WEKA

WEKA (Waikato Environment for Knowledge Analysis) suite contains textbook implementations for most popular ML Algorithms. It runs as a centralized java engine and does not include multi-thread capabilities. It is thus a very popular tool for people to train in ML and verify the results they obtained from more complex execution engines. WEKA's library is also a good building block for custom implementations. WEKA can also be used for processing smaller datasets that do not have the volume requiring a distributed implementation.

We have used WEKA for text manipulation and unsupervised learning.

4.2 Clusters

For the testing and development of the IReS platform and the various analytics engines used by the operators, we have set up, two clusters with different configurations.

The first one is hosted and managed by IMR and the other one is hosted by ICCS. The two setups are diverse in hardware and this presents the opportunity to examine the performance characteristics of the underlying software platforms in more than one setting.

4.2.1 IMR Cluster

IMR has allocated a cluster for the needs of ASAP. It consists of 4 server-grade physical nodes. Each one of those is equipped with a 3rd generation i5 CPU (@ 2.90 GHz) and

16GB of physical memory and an array of two HDDs on RAID-0. The operating system is Debian 6 (squeeze) Linux.

For the time being, the two software platforms running on this setup are Hadoop and Weka. The distribution of Hadoop is CDH 4.6.0 (a popular bundling of Hadoop by Cloudera) which uses Hadoop version 2.0.0 over MapReduce scheduler (not the newer YARN resource negotiator). Hadoop is so configured so that all the machines run as workers and one of them runs the master.

There are also plans to set-up spark on this cluster too.

Weka is on version 3-6-12.

4.2.2 ICCS Cluster

The second cluster used for WP3 is hosted by ICCS and runs on a private OpenStack installation. It consists of 11 Virtual machines with 8GB of RAM, and 4 virtual CPU cores each. The volumes used by the VMs are stored on SSD storage. Ubuntu Linux 14.04 is the operating system. The Hadoop and Spark installation on this cluster is configured so that the master runs alone in one of the VMs and the rest run as workers. Hadoop, Spark, Hive and Weka are set up on this cluster.

The version of Hadoop used is 2.7.0, over Yarn, as it is packaged by Apache. The version of Spark is 1.4.1, running in “standalone” mode.

The version of Hive is 1.1.1 and the version of Weka is 3-6-12.

4.3 Operator library

This section enumerates the operator implementations that have been imported to the IReS platform library, going through the whole process of the Profiling Workflow, as described in Section 2.2.1. The operators are so far classified in two major categories, the *analytics operators*, that is, the operators that perform the core analytics jobs over the data provided, and the *auxiliary operators*, namely the operators that are added by the Decision Making module to match data to operator input constraints. Auxiliary operators serve as "glue" between different engines and include move and transformation operations.

4.3.1 Analytics operators

The following subsections describe the implementations of the analytics operations examined, profiled and modeled so far by the IReS platform. These operators have been chosen to be in line with the ASAP use cases as described in deliverables D8.2 and D9.2. Section 4.4 outlines the basic web and telecommunication analytics workflows that are constructed utilizing these operators.

4.3.1.1 TF/IDF

TF/IDF analysis (abr. term frequency/inverse document frequency) is a method of converting a set of text documents into weighted vectors. The coordinates of those vectors signify the importance and relevance of a specific term (word) relative to the document it appears in. This metric is proportional to the frequency that a term appears

in the document and inversely proportional to the frequency it appears in all of the documents.

TF/IDF is a straight-forward approach in judging word relevance in a specific document context. It is the basis and core component of plain text search, but its (numeric) output can be used for a variety of post-processing like document summarization, latent semantic indexing and cluster analysis for a corpus of documents.

While TF/IDF analysis has a lineal complexity, it can be parallelized. Moreover there is more than one approach in computing document vectors.

Input: A set (corpus) of human-readable text documents in some form.

Output:

1) A set of document vectors (usually in sparse vector representation - keeping only the indexes and values of the non-zero coordinates)
2) A term<->term_id dictionary (optional). This maps the term IDs - which are the indexes in the document vector coordinates - into their corresponding text terms

Algorithm: There are two common approaches in computing the Term Frequencies and the IDF metric. Both have lineal complexity, but in real life present very different performance characteristics and cannot be parallelized in the same way and with the same results.

- **Bag-of-Words approach:** This is the most straightforward approach that maps a single term to a single term id and computes the TF and IDF and TF/IDF based on those IDs. This ensures a 1:1 mapping of terms and their IDs and allows for the ability to strictly argue about a words presence in a document. Standard optimizations include discarding terms with low document frequencies, discarding terms with very high document frequencies ('stopwords' like and/or/a etc.) and stemming of the terms so as to reduce the total count of available terms and group those with the same lexicographical stem to a single one ID. Figure 16 shows the effect of stemming on term count and thus the dimensions of the coordinates of the algorithm's output vectors (basic stopwords are removed in both cases).

  A fundamental tradeoff of this approach is that for large sets of, possibly multilingual, documents the number of terms available increases to the 10s of thousands and thus computing storing and distributing (for parallel computations) the dictionary becomes a bottleneck. Consider that in order to achieve a 1:1 mapping of terms; a synchronization step is necessary for distributing the dictionary to all the computing nodes.

- **Feature Hashing Approach:** In this approach, which is also known as "hashing trick", we are willing to accept some term ID collisions. The terms ids are the hashes of the terms themselves and thus we are able to pick the exact dimensions of the output vectors as the hash-bucket size. The tradeoff presented here is, on the one hand, more accurate document representation and, on the other hand, lower storage requirements as well as much higher performance. In practice this approach yields much better performance results and can be much easier to

parallelize. Stemming is also orthogonal to this approach and can also be used as an added enhancement.

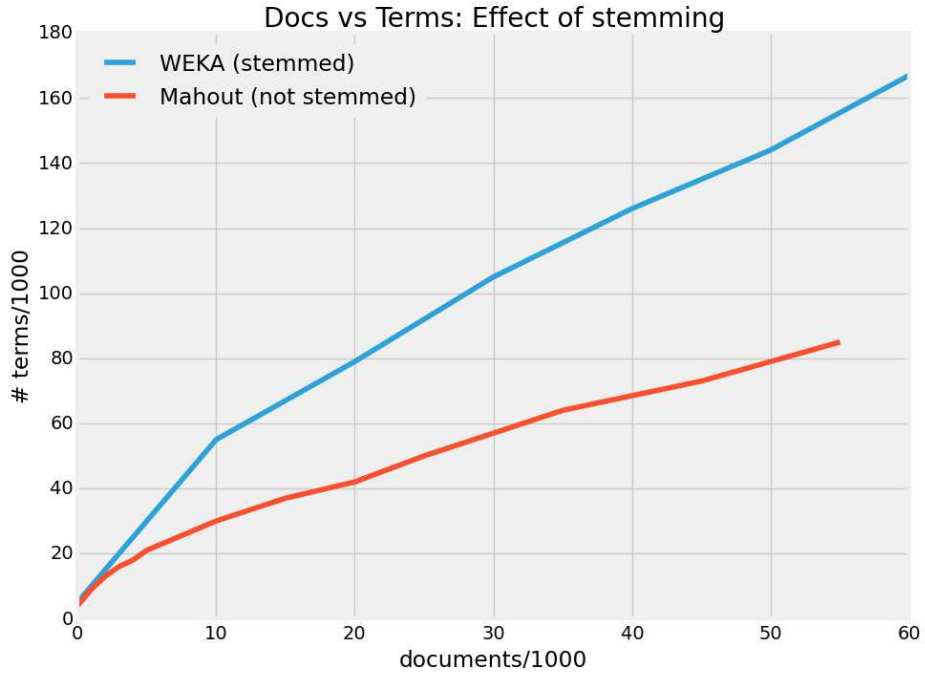


Figure 16 The effect of stemming of document vector dimensions

4.3.1.2 K-Means

K-Means is an unsupervised learning algorithm for classifying the members of a vector collections according on their geometrical distances. K-Means is the most commonly used clustering algorithm for ML analyses.

K-Means is used for any type of clustering on data that are either purely numerical or have been derived from textual sources (e.g. through TF/IDF or Word2Vec).

Input: A set of (numerical) vectors. These can be in either dense or sparse representation.

Algorithm: K-Means requires a random initialization step in order to choose starting point of the cluster centroids. It is iterative, performing many passes on the data and adjusting an estimation of the cluster centroids on each step.

Theoretically, this process is NP-Complete for finding a globally optimum choice of cluster centroids. However all of the popular implementations can be used with a fixed number of iterations and various optimizations of the algorithm (e.g. in choosing the start state in more sophisticated manner than purely random). The complexity in those cases is theoretically O(ikdn) where “i” is the number of iterations, “k” is the number of centroids we wish to compute, “d” is the cardinality (dimensions) of the input vectors and “n” is the input count.

The various implementations of K-Means do fall under this general theoretical complexity bounds, but have important differences in their practical performance. The reason for their differences in performance is the way they handle sparse vectors, the method they use to parallelize the computation and how they handle intermediate data between iterations.

4.3.1.3 Word2Vec

Word2Vec [44] is an unsupervised feature extractor for text processing and natural language processing (NLP) purposes. In particular, Word2Vec is a neural network that turns raw text data into numerical vectors that deep neural networks can understand. In most use cases, given enough data it can extract meaningful, semantic information from text and return accurate predictions about word's meaning based on past occurrences. The most well-known application of Word2Vec is the case of finding word synonyms. We provide an implementation of Word2Vec for Spark in Scala [37] .

Input: A corpus of raw text data

Output: A set of words in vector format of same size as the dictionary. Each vector is a distributed representation of the word.

Algorithm: As aforementioned, the main abstraction of Word2Vec is a Feed Forward Neural Network (FFNN) model with its architecture consisted as follows: The input layer w(t) represents words using 1-of-K coding, where K the vocabulary size. The hidden layer computes probability distribution over all words in the vocabulary. The result is an output layer with dimensionality K. In their last paper about distributed representations of words, Tomas Mikolov et al. proposes two log-linear models: **Continuous Bag-of-Words** and **Continuous Skip-gram** [31] [32] .

- **Continuous Bag-of-Words model:** The continuous Bag-of-Words model aims to predict a word based on its context, that is, the surrounding words. For example, given a sequence of words w_{i-2} w_{i-1}... w_{i+1} w_{i+2} in the input layer the model tries to predict w_i . The model is characterized as a Bag-of-words because its prediction is not affected by the order of the other words in the document.
- **Continuous Skip-gram model:** The skip-gram model tries to find distributed word representation in order to predict the surrounding words in a sentence of a document. Specifically, it's goal is to minimize the average log-probability $p(w_{t+j}/w_t)$. Furthermore, the probability $p(w_{t+j}/w_t)$ is defined by the *Softmax Function*[11]. In order to speed up training of Word2Vec, *Hierarchical-Softmax* can be applied, that is, an efficient approximation of the traditional Softmax. The main advantage of Hierarchical-Softmax is that it needs to evaluate only log2(W)nodes using a binary tree representation instead of W nodes, where W the number of words.

4.3.1.4 LDA (Latent Dirichlet Allocation)

Latent Dirichlet Allocation (LDA) [23] is an unsupervised learning algorithm for topic modeling - clustering that infers topics from a collection of documents. LDA can be seen as a clustering algorithm as it contains such characteristics like cluster centers (topics)

[11] https://en.wikipedia.org/wiki/Softmax_function

and feature vectors (topics and documents). In contrast with traditional clustering algorithms such as K-Means, LDA generates results using statistical inference instead of algebraic distances (e.g. the Euclidean distance). We provide two implementations of LDA, one in Apache Spark [29] and one in Gensim [9] .

Input: (a) A collection of documents in Sparse-Vector format, (b) k - Number of topics, (c) max iterations

Output: A set of k Sparse Vectors, that is, the cluster centers (topics)

Algorithm: LDA is an algorithm that automatically discovers topics in a collection of documents. It takes as input a corpus and a k value (number of topics). At the initial step, it assigns each word in a topic in a semi-random manner using the Dirichlet distribution. Then, it updates the topics iteratively, passing each word in each document. The updates are based on each word's frequency across topics and documents. More precisely, for each document d and word w of d, LDA computes *P(t |d)* for each topic t and *P(w |d)*. Then, it assigns the word w a new topic with a probability *P(t |d)*P(w |d)* . This procedure is applied repeatedly with a user-specified number of iterations.

4.3.2 Auxiliary operators

In order to implement multi-engine workflows we had to write custom code to move data from one engine's format to the other.

We have implemented this functionality for the output vectors of all the TF/IDF operators. This process was more than a simple stateless transformation, because none of the output vectors' term numbering was compatible with another engine. Spark In particular uses hashed term IDs, so its vector space was non-consecutive, which was violating the assumptions the engines had for their input.

We wrote custom programs in java, able to make an in-memory re-map of terms and term ids in order to produce a result compatible with the targeted engine. By using Hadoop's API, we are able to transform vectors from any of Spark's, WEKA's and Mahout's formats to any of the others.

Input:

- a set of vectors, either in a local .arff file (WEKA) or a Hadoop SequenceFile [21] of SparseVectors (Mahout) or a Hadoop text file of SparseVectors (Spark).
- A dictionary of {term → term ID} in the same format (in either a SequenceFile or a text file - local or on HDFS).

Output:

- a set of vectors of the specified format that respects the assumptions of the output engine
- a dictionary {term → term ID} with the same assumptions

Algorithm: An in-memory mapping of {input term ID → output term ID} is populated on-the-fly, based on the input dictionary. All of the dataset vectors are re-mapped and stored in the corresponding format, based on this mapping

4.4 Workflows

The operators described above cover a diverse set of tasks of varying complexity and execution parameters. The user can assemble any workflow she wants using any of the operators that reside in the IReS operator library. To evaluate our platform though, we have created 3 abstract workflows, inspired by the ASAP use cases as described in D8.2 and D9.2. These cover complex data manipulations in the areas of business analytics on telecommunication data and web data analytics, provided by WIND and IMR respectively. The input datasets for these workflows consist of anonymized telecommunication traces and web content data (WARC files).

A short description for each workflow follows:

4.4.1 Web analytics - Clustering

The workflow starts by selecting a subset of the initial web content. Feature-extraction (e.g., TF/IDF) is performed on these documents; the outputs are clustered using k-means clustering (chosen among weka, mahout and MLlib running centrally or over Hadoop or Spark respectively).

4.4.2 Telco analytics - Peak Detection

The workflow involves processing of anonymized CDR data (residing in an RDBMS) via clustering along time and space in order to detect peaks in load, according to a set of criteria. The results of this phase enrich a database (relational) that contains peaks detected in previous runs. The dataset of peaks is used to discover clusters of calls that occur with or without regularity.

4.4.3 SQL workflow

A sample workflow that showcases a simple join operation between two datasets residing in different stores, namely PostgreSQL and Hive, followed by a sorting operation. For this workflow, we use synthetic data produced by the popular TPC-H [40] benchmark generator.

5 Results and evaluation

In this chapter we evaluate the performance and accuracy of the core IReS modules, namely profiling, modeling and decision making/planning. The evaluation of this section has been performed using the ICCS cluster, as described in Section 4.2.2.

5.1 Profiling

In order to investigate the performance characteristics of the operators and the underlying implementations we have worked with, a number of exhaustive profiling rounds were performed.

For each experiment we have kept a number of information metrics (45 in total) that could be useful in modeling the operators' performance. However it is not necessary that all of those metrics will contribute in a performance model.

The metrics collected include:

1. The operator's execution time
2. Input and output sizes (where applicable)
3. Input count (e.g. Number of documents, vectors, etc.)
4. Cardinality of the output (for vectors)
5. Date of the experiment
6. Operator specific parameters (like the number of clusters for clustering operations)
7. A timeline of system metrics (CPU, RAM usage, network traffic, IOPS, etc.) for the whole cluster, periodically pulled from the monitoring system (ganglia)(see Figure 17).

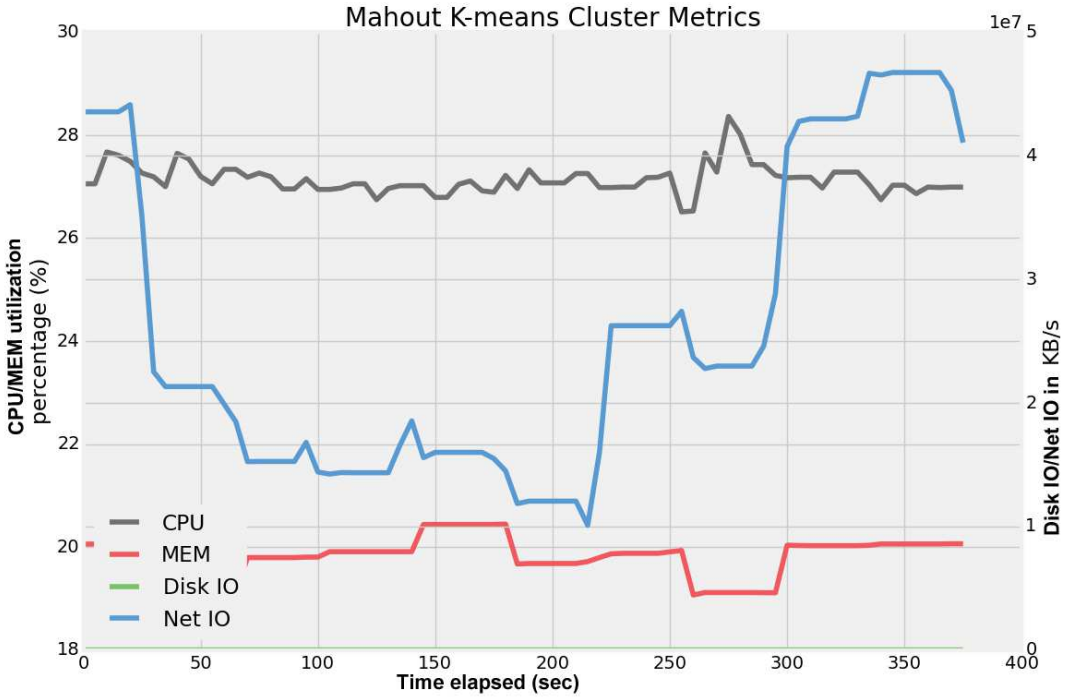


Figure 17 Monitoring Metrics

5.1.1 Results overview

In analyzing the performance results obtained by the profiling experiments, we tried to identify how the execution parameters affect the execution time of each one of the execution engines. The initial intuition regarding engine performance was that:

- The centralized implementation would only be quicker for smaller datasets that fit in-memory of a single node.
- Mahout would be quicker for non-iterative operations with few intermediate steps. Also, it could be a better fit for very large input sizes, in which cases the intermediate processing result might not fit in-memory.
- Spark would be significantly preferable for iterative operators since it avoids persisting intermediate results to disk. However, for very large inputs, keeping data in-memory could be problematic and spilling intermediate outputs to disk would significantly tax performance.

After running the full set of experiments we realized that none of the above intuitions were entirely correct.

The implementation particulars in all of the experiments seem to play a more important role than the basic operational principles of Spark and Mahout. As for WEKA, it follows that it is indeed preferable only for smaller input sizes.

All of the results presented here for Spark and Mahout were obtained in the larger ICCS cluster but those for Mahout were also verified qualitatively in the IMR cluster. For performance reasons the experiments on WEKA were ran natively on one of IMR's machines.

5.1.2 TF/IDF

The implementations of TF/IDF that we examined have fundamental qualitative differences between them. WEKA and Mahout follow a conventional bag-of-words approach that maps a single integer id to each term in the document corpus (apart from those it does not keep as features due to their low document frequency). Their output is not only a set of document vector (which were used for K-Means) but also a dictionary with the aforementioned mapping. This procedure, conventionally, needs at least two phases to complete:

- Dictionary construction, document frequency calculation and feature selection
- Document Vector creation

Spark on the other hand uses a feature hashing approach (a.k.a "hashing trick" [22]). It is thus much quicker in that it can create the document vectors without previously constructing a dictionary. In fact, by itself the TF/IDF for MLib does not output a dictionary that maps terms to term IDs.

In order for the two approaches to be more comparable and for Spark's output to be more usable later where one might want to use the dictionary, we have added a dictionary construction job in the TF/IDF operator for Spark.

Performance Analysis:

WEKA: TF/IDF shows linear performance on all engines since it is I/O bound. The centralized implementation is limited by the capacity of a single machine, and is thus preferable only for smaller datasets. For more than 60K documents the preprocessing phase would fail and we were thus unable to test it for larger data sizes. However up until that point it enjoyed a marginal advantage over Mahout. This might be due to WEKA's lower output dimensions (as seen in Figure 18).

Mahout, Spark: The approach used by Mahout is much slower than the one used by Spark for this operation. Both have linear performance but Spark seems to enjoy an advantage of 9x faster performance.

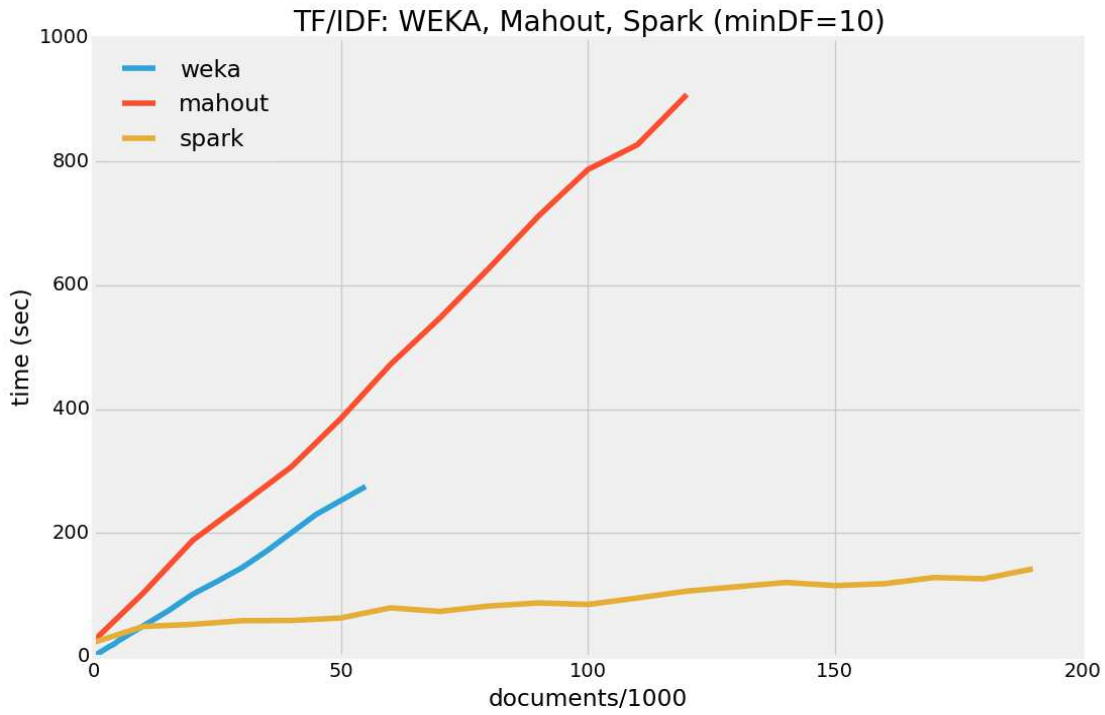


Figure 18 TF/IDF Engine Comparison

Output Data Analysis:

Since the output data of this operator were used as input to other operations (namely, K-Means clustering) it makes sense that we review its output.

The output of this operation is a set of document vectors and a term dictionary.

All of the engines choose a sparse representation of the vectors, so their output is comparable. However, the dimensions of the output vectors for WEKA and Mahout equals to the count of the dictionary entries, while for Spark it is the size of the hash bucket (which the user can specify). Therefore, for Mahout and WEKA it makes sense to:

1. Examine the dimension sizes of the output for various input sizes
2. Try to affect these dimensions explicitly or implicitly
3. Examine how the output size changes with those dimensions

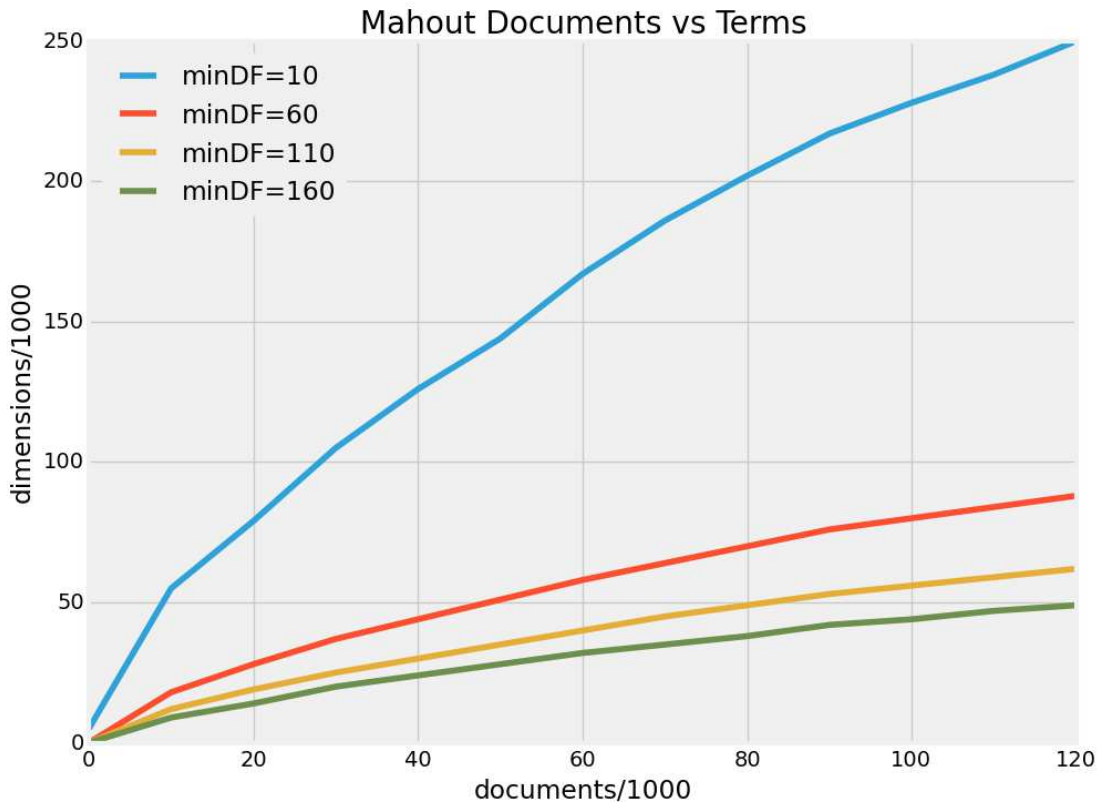


Figure 19 Documents vs Term count

In Figure 19 we can see that although new terms are introduced with larger document collections, this trend is not linear. This is to be expected. More interestingly, we can see that raising the minDF parameter (the minimum number of documents a term needs to be found in so as to be considered a feature) has a dramatic effect to the total term count.

However, as is obvious in Figure 20 this does not translate to any observable difference in output size. Another interesting observation (not visible in the latter figure) is that this particular operator actually inflates (by approx. 1.6:1) the output size in comparison to the input. This is true for Mahout and Spark, but not true for WEKA (~1.1:1 ratio). We suspect this would not be the case if the document corpus was comprised of larger documents.

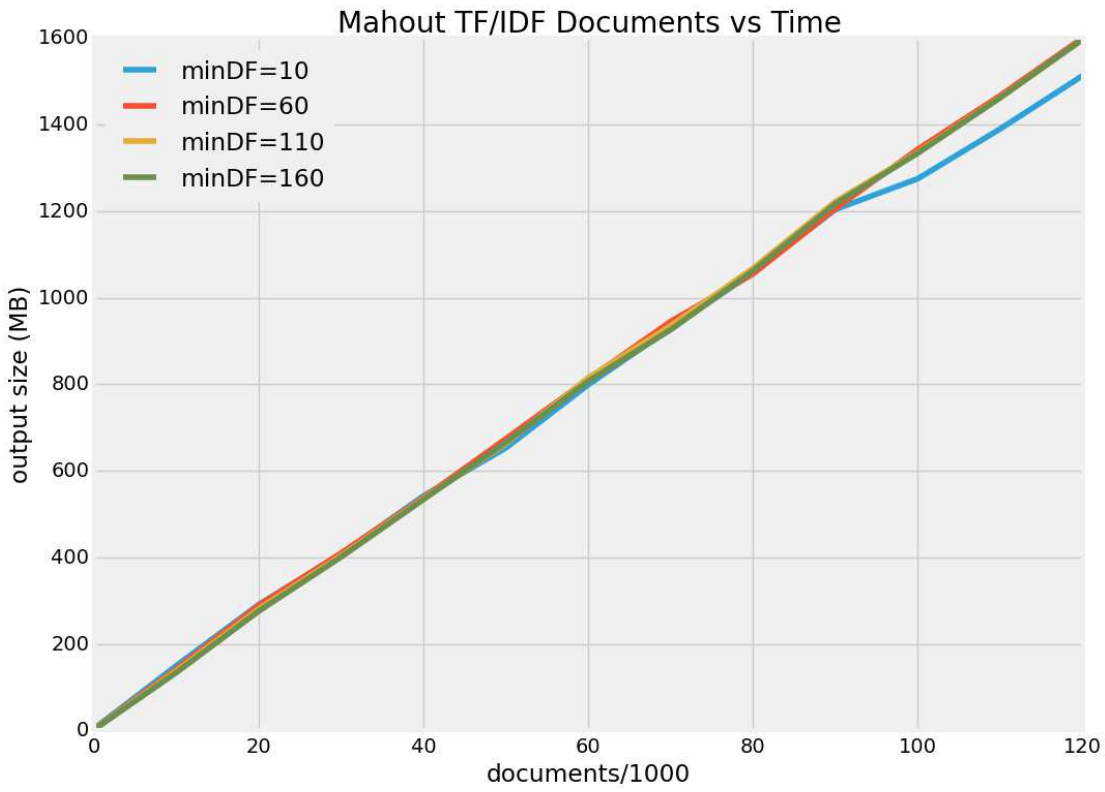


Figure 20 TF/IDF input vs output sizes

5.1.3 K-Means

Performance Analysis

The particular implementation used (WEKA) was single threaded and its performance fell behind Mahout and Spark long before the memory capacity became an issue. We plan to examine an alternative custom thread-based on C and the Cilk framework [25] . As for WEKA, for input sizes larger than about 3000 documents (~30MB), it becomes slower than the respective implementation on the other engines.

For medium sized data and smaller (<50K documents) Mahout was not slower than Spark. Contrary to expectations, Spark was faster for large and very large datasets and higher cluster counts. This difference became apparent without coming close to the point where secondary memory spills would become necessary

Spark offers no order-of-magnitude difference compared to mahout for K-means. However, it is observably faster for larger datasets.

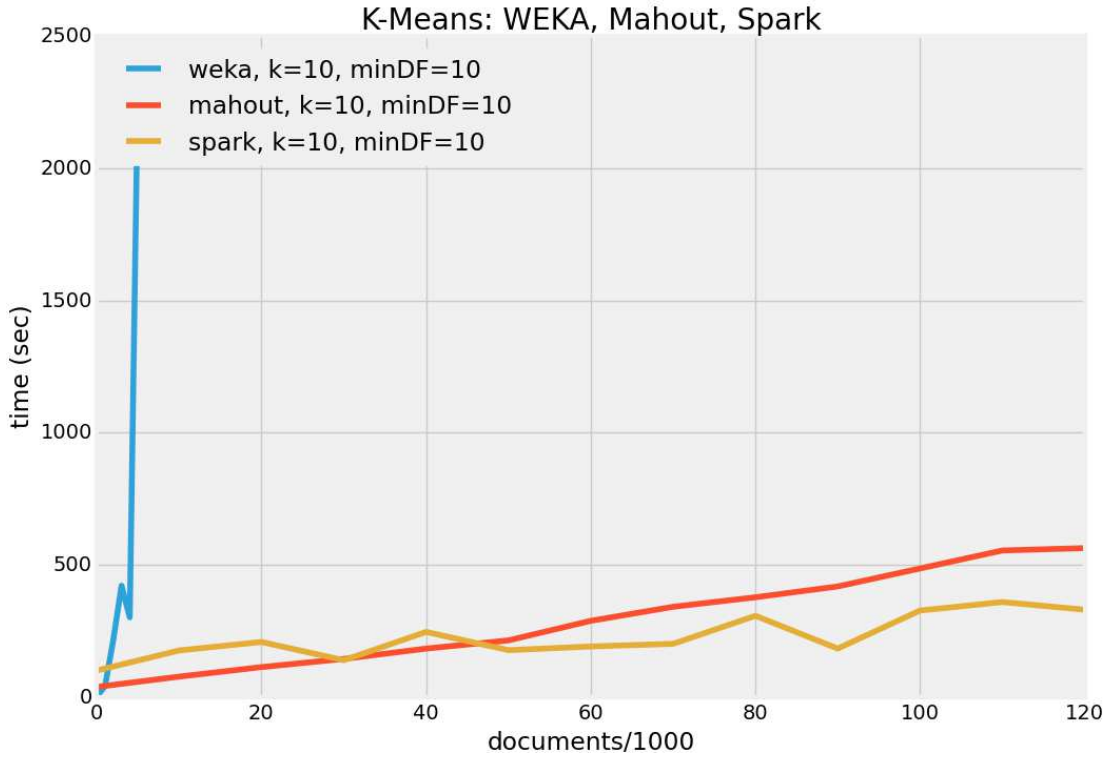


Figure 21 K-Means Engine Comparison

Output Data Analysis

The K-Means operators examined emit as output the cluster centroids in the form of a set of terms that have the highest value in the corresponding coordinates. Quantitatively, this means that the output is relative to the number of centroids asked, and is thus close to zero - compared to the input size. Qualitatively there are some important differences that are beyond the scope of this analysis

5.1.4 LDA

In order to analyze LDA's performance, we run through several benchmarks using two implementations. One on Spark (Distributed) [29] and one on Gensim (Centralized) [9] . Results are plotted in Figure 22.

Gensim: LDA implementation on a single machine, runs with a linear correlation between the number of documents and the execution time. Examining the results we observe that Gensim has a better performance than Spark on smaller datasets but as the input size grows it continuously gets slower.

Spark: On the other hand, Spark seems to be slower on the small datasets. This is reasonable since Spark needs to do some preparation before processing, such as distributing data over the nodes in the cluster and RDD allocation. Subsequently, as the input size gets bigger it scales up better than Gensim centralized implementation.

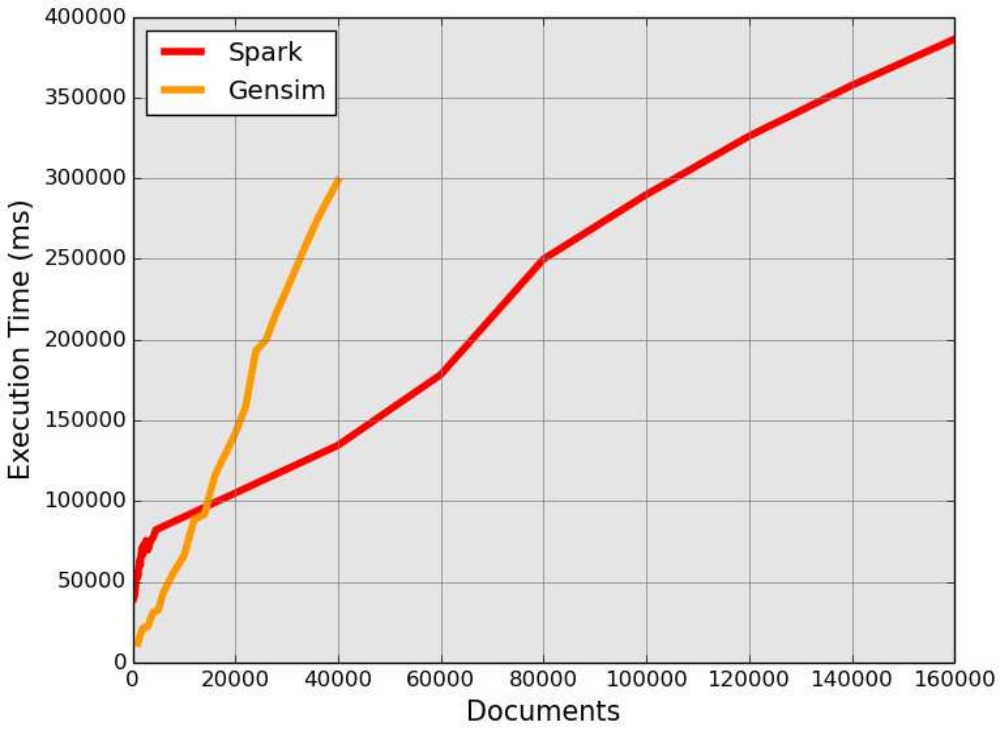


Figure 22 Scaling the performance of the two LDA implementations in Spark for for varying number documents.

5.1.5 Word2Vec

Concerning Word2Vec, we run through some benchmarks on Spark's Scala API. A plot of execution time and number of documents is collocated below. It is clear from Figure 23 that Word2Vec has a linear performance with respect to the number of input documents.

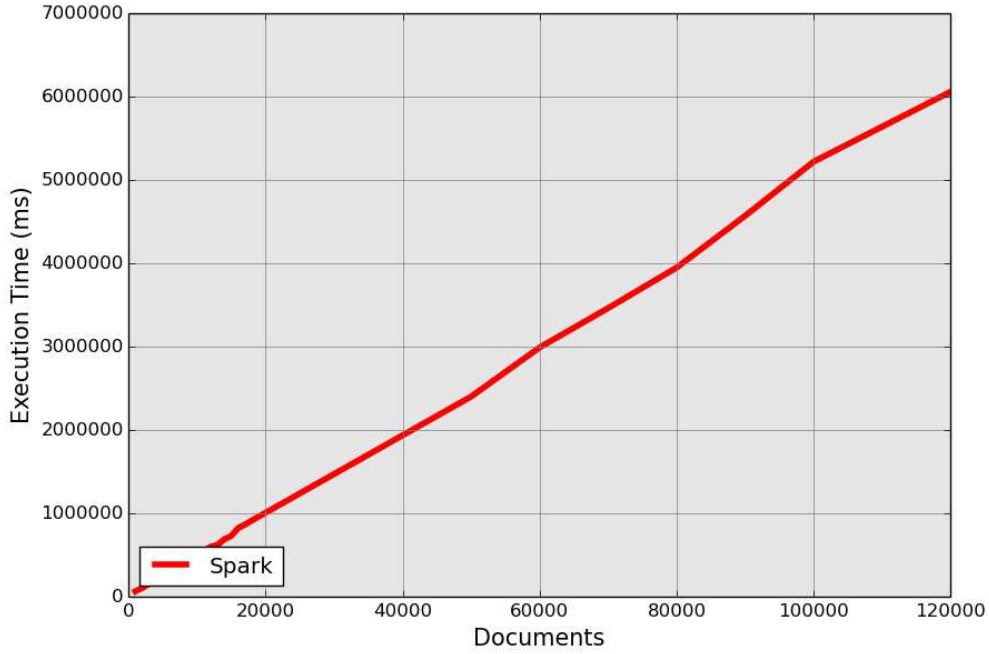


Figure 23 Performance of Word2Vec with respect to the number of input documents.

5.1.6 Auxiliary operators

Performance analysis

The Move operations for vectors all present a similar behavior (Figure 24). Their running time seems to be exponentially related to the input size. This is related to the implementation and possibly to the fact that between engines, the mover needs to re-calculate the term IDs since they are not consistent between engines. As the data structure that keeps the input and output term ID mapping, seeking in it becomes more expensive, and thus each item takes more time to be processed. We could speed up this process with a parallel (possibly Map-Reduce) implementation.

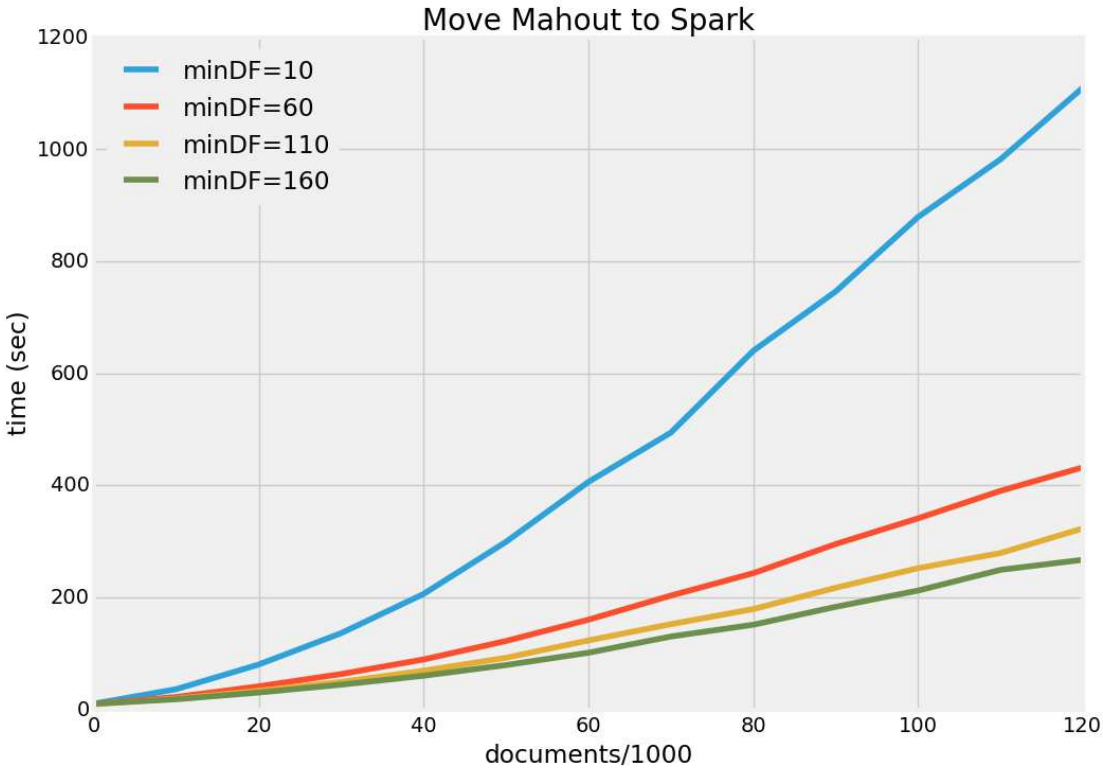


Figure 24 Mahout-to-Spark Mover

5.2 Modeling

5.2.1 Introduction

In the Profiling Section we present some results of several benchmarks we run on our platform operators in order to investigate how they react to different input data sizes, number of documents and other parameters. In this section we present how we take advantage of these results and use them in order to model benchmark data in IReS platform. Using this data, IReS analyzes the performance of each operator using several machine learning algorithms. Then it selects the best model, that is, the model with minimum error and makes accurate predictions about the performance of each operator. Furthermore, it provides detailed visualizations of benchmark data.

5.2.2 Data Modeling

The operators in the library are modeled as follows. Using data obtained from profiling stage, we represent each operator as a feature vector. Each dimension represents a metric such as execution time, input size, CPU usage etc. For example, a vector gained from a Word2Vec run has the format *V = [d, exec_time, input_size, min_df, iterations, vector_size, metrics]*. The features are explained below:

- d: The number of input documents
- exec_time: Execution time
- input_size: The input size in KB
- min_df: The limit of minimum document frequency for each term

- iterations: Number of maximum iterations the algorithm will perform
- vector_size: The dimensionality of the vector, that is, the vocabulary size
- metrics: System metrics such as CPU usage, Memory usage, disk I/Os etc.

Example vector: *[1000, 69244.0, 8833233, 5, 1, 100, (metrics in JSON format)]*

5.2.3 Machine Learning Models

In order to make better decisions in workflow planning and optimization, IReS uses Machine Learning models to predict operator performance, i.e. the execution time. Currently we use PANIC [27] , a framework for application performance and modeling in the cloud. PANIC works with WEKA [43] , an open-source machine learning library. For each operator in the Operator Library, IReS takes the benchmark data and trains all the following WEKA models that PANIC framework contains:

1. Bag Classifier
2. Discretization
3. Gaussian Curves
4. Isotonic Regression
5. Bag Classifier with Least Squares
6. Linear Regression
7. Multi-layer Perceptron
8. RBF
9. Random Committee
10. Random Sub-Spaces

After training the above mentioned models, it evaluates them and selects the model that fits best in the data, that is, the model with the minimum error. For example, as we can see in Figure 25, IReS chose Least Squares model for the Scala implementation of Word2Vec operator in Spark.

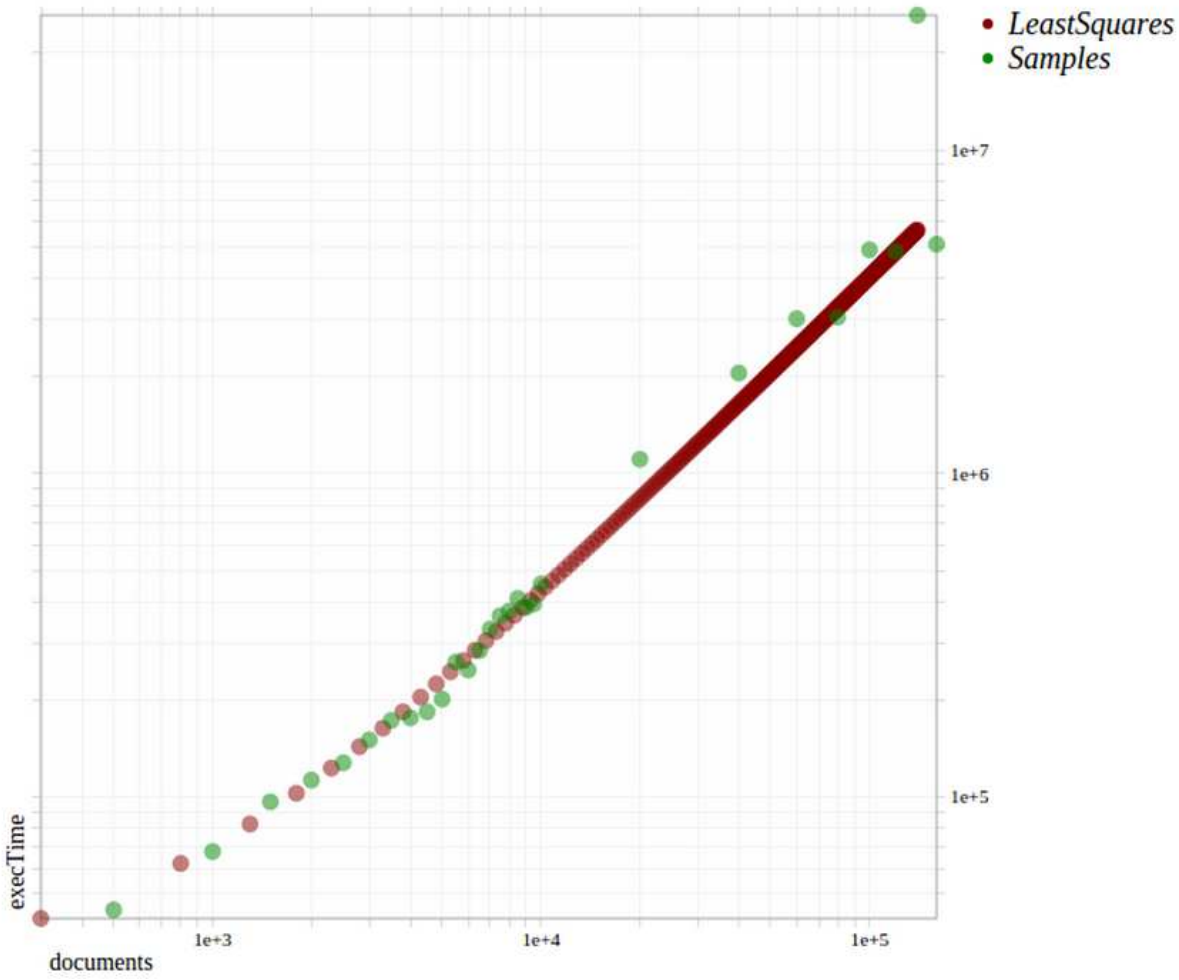


Figure 25 Model and sample values of the Word2Vec operator implemented in Scala over Spark

5.2.4 Data Visualization

Another component of IReS data modeling is data visualization. For every operator's data obtained from past runs, scatter plots are provided. The variables in the plots are user-specified in the description file of each operator. Figure 26 represents a visualized description of the K-Means operator in MLlib and Figure 27 plots the ML-Perceptron model of K-Means in Spark with three variables/dimensions.

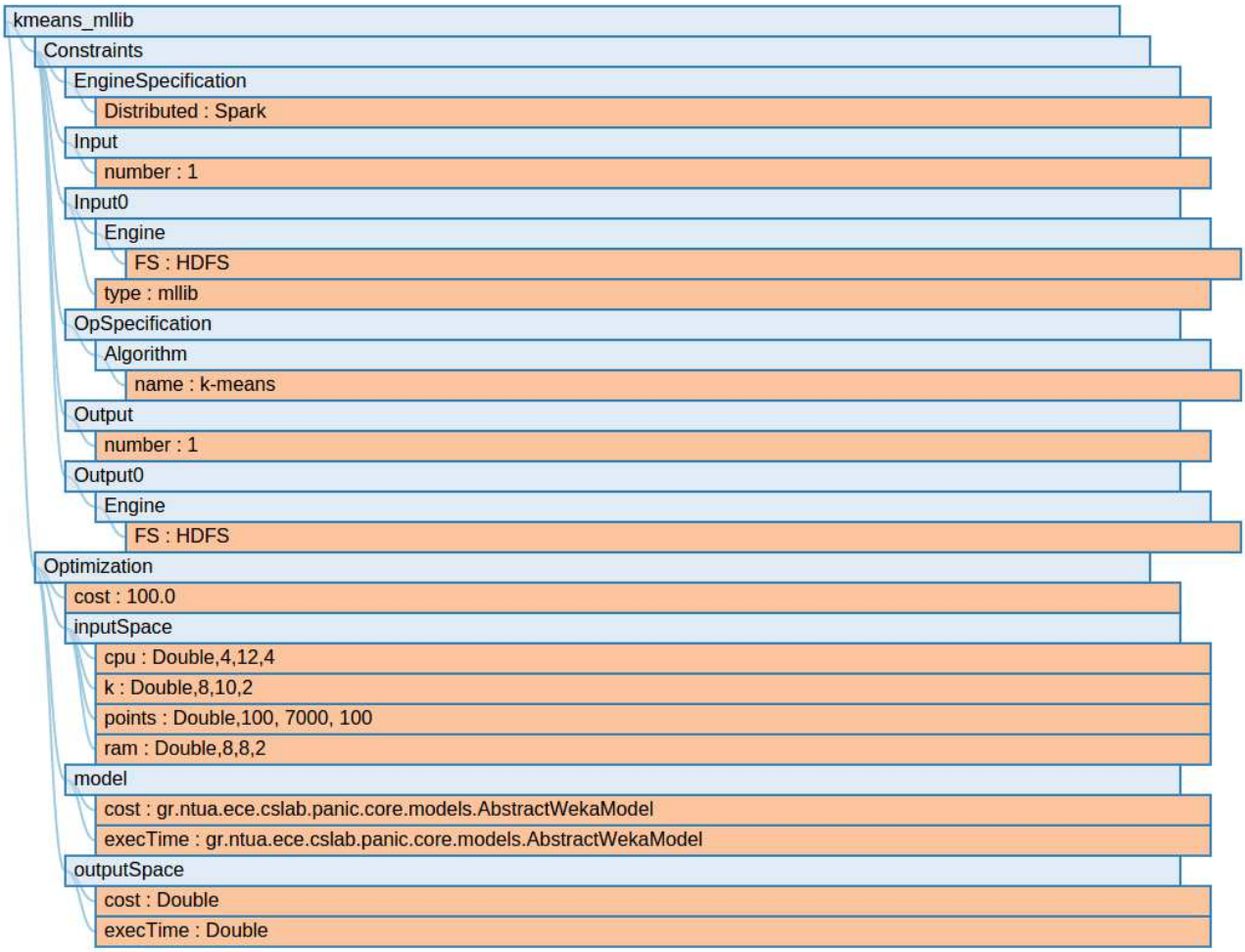


Figure 26 Metadata description of the k-means operator in MLlib

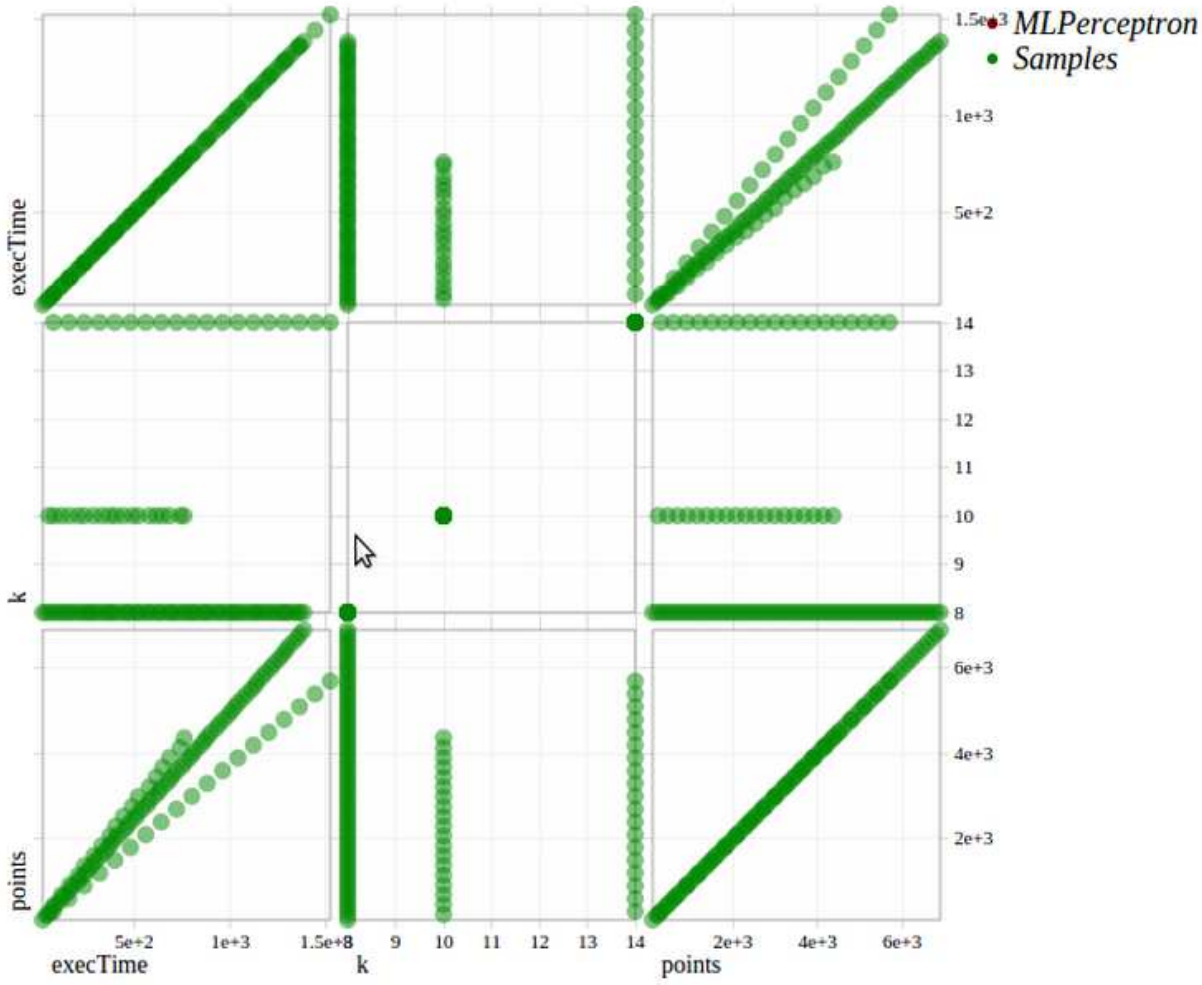


Figure 27 Visualization of the 3-dimensional ML-Perceptron model of the k-means operator in MLlib

Finally, Figure 28 shows how IReS visualizes the Isotronic Regression and MLPerceptron models of the execution time of a sort operator, implemented in Hive, taking into account the number of cluster nodes, the number of unique keys in the dataset and the number of cores per node.

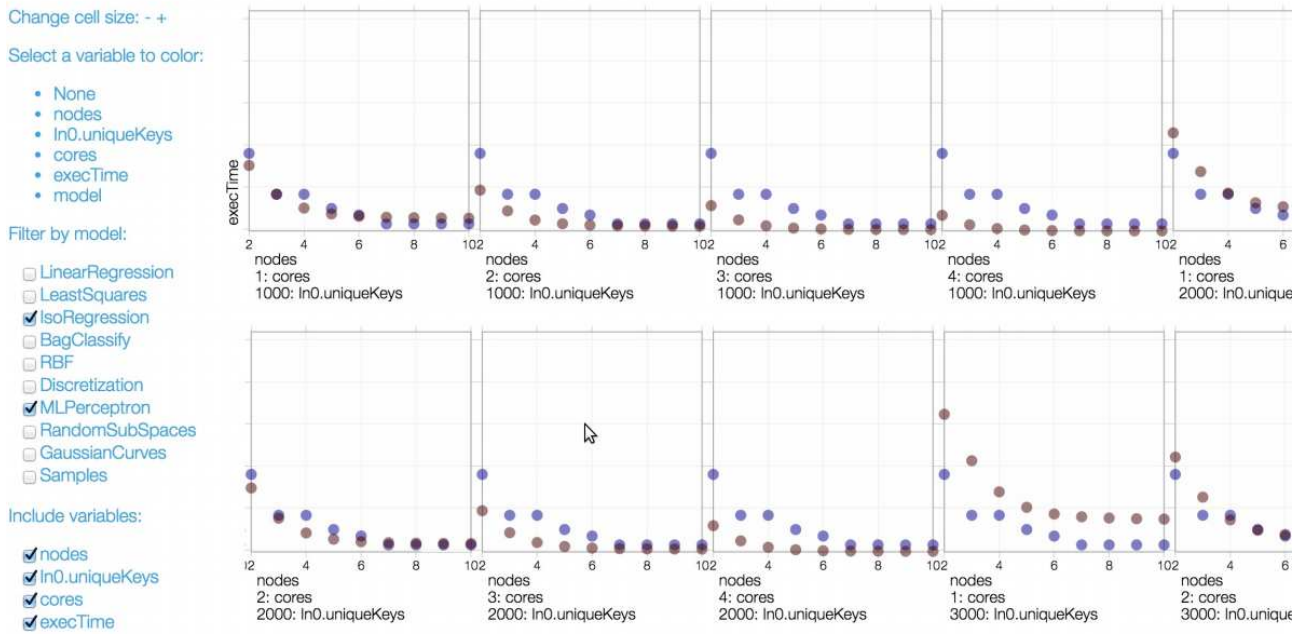


Figure 28 Visualization of the 4-dimensional IsoRegression and MLPerceptron models of a sort operator implementation in Hive.

5.3 Decision Making

In this Section, we present the performance of our decision making module. As presented in previous sections, this module is responsible for exploring the alternative execution paths according to the existing matches between abstract and materialized operators and select the one that best conforms to the user-defined optimization policy.

The above operations are executed in one optimization step by our Decision making module. Figure 29 depicts the time required for our Decision making module to materialize and optimize abstract workflows with variable sizes. To test the efficiency of our module we range the number of the abstract workflow nodes from 2 to 20. We also range the number of materialized operators that match with one of the abstract operators. This is also a very important parameter because it increases the number of possible execution paths. We note that our optimization approach manages to both materialize and optimize the tested workflows in less than a second. For the majority of the tested workflows the optimization time was less than 500 ms. Thus, our optimizer proves efficient and we can expect it to handle all our use case workflows.

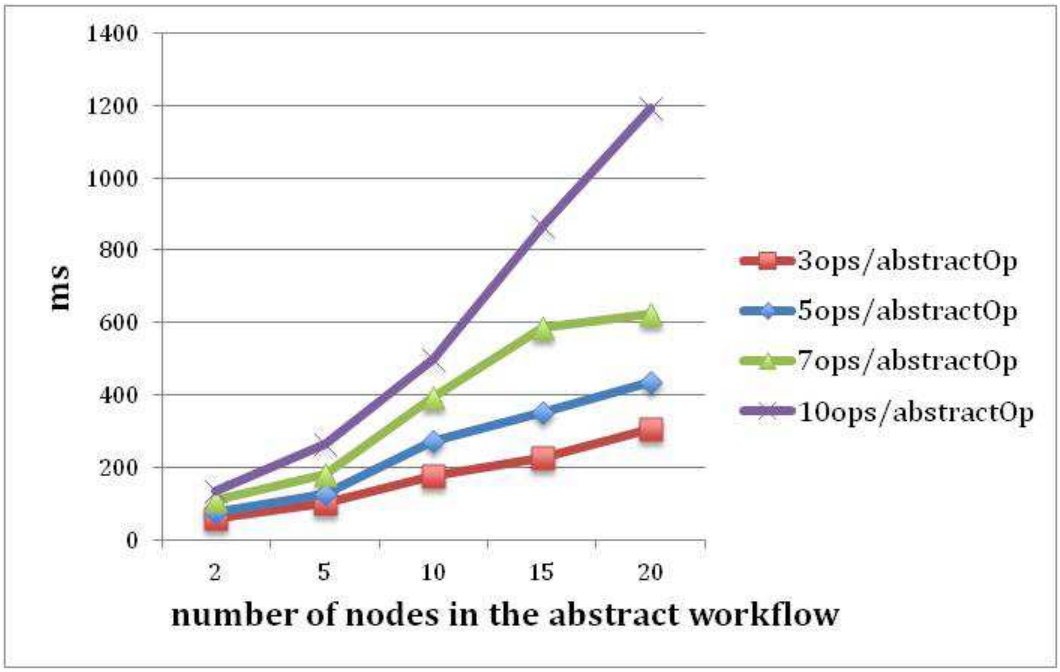


Figure 29: Decision Making execution time for variable number of workflow nodes

Figure 30 depicts the performance of our Decision making module with respect to the number of matches between abstract and materialized operators. This parameter affects the number of possible execution plans and is crucial for the optimizer's complexity. The experimental results show that the optimization time scales well as we range the possible matches. In all cases we observe acceptable response times for the workflow optimization.

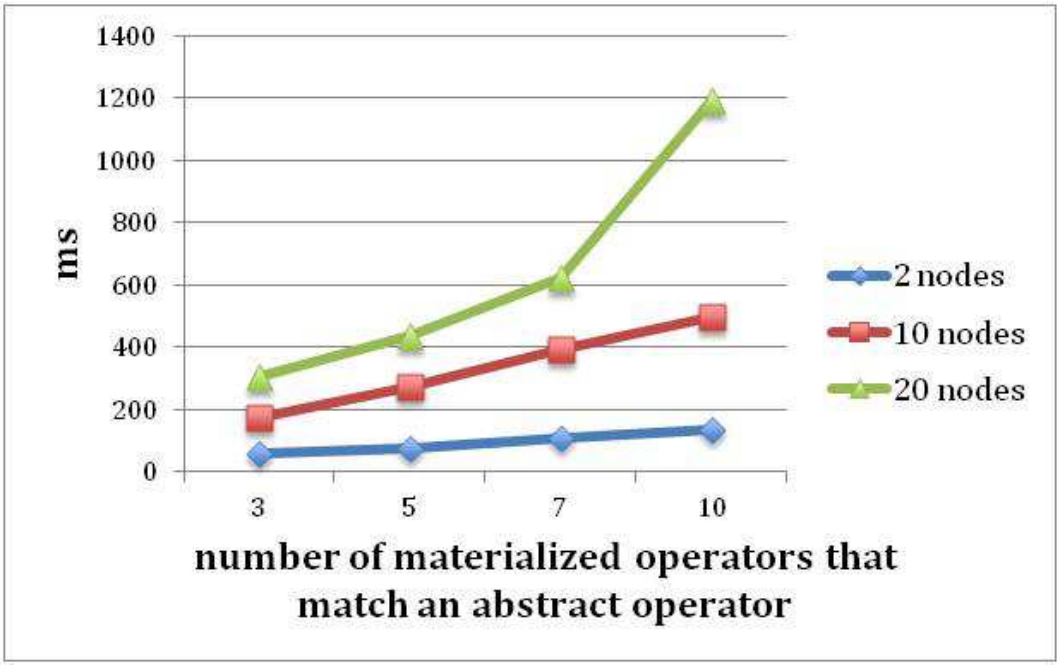


Figure 30: Decision Making execution time for various numbers of materialized operator matches

6 Conclusion

This deliverable describes the first version of the IReS platform prototype. More specifically, it finalizes the IReS system architecture and provides details on the implementation of its various components. The functionality of IReS is accessible to external components via a Restful API, which has been defined. The IReS operator library has been populated with a number of analytics as well as auxiliary operators, which have been profiled, modeled and used in the composition of user-driven workflows. The profiling, modeling and decision making modules have been evaluated in terms of performance and accuracy. The results prove the efficacy of the IReS platform and its ability to (a) create accurate models within a limited time budget and (b) discover the optimal execution plan of a medium-sized workflow (consisting of up to 20 nodes) in less than a second on average.

References

[1] Blei, D. M., Ng, A. Y. and Jordan, M. I. Latent dirichlet allocation. the Journal of machine Learning research 3 (2003): 993-1022.

[2] Breiman, L. Bagging predictors. Machine Learning, 24(2):123–140, 1996.

[3] Broomhead, D. S. and Lowe, D. Radial basis functions, multi-variable functional interpolation and adaptive networks. Technical report, DTIC Document, 1988.

[4] Cloudera Distribution CDH 5.2.0. http://www.cloudera.com/content/cloudera/en/downloads/cdh/cdh-5-2-0.html.

[5] De Boor, C., et al. A practical guide to splines. Vol. 27. New York: Springer-Verlag, 1978.

[6] Dean, J. and Ghemawat, S. MapReduce: simplified data processing on large clusters. Communications of the ACM 51.1 (2008): 107-113.

[7] Doka, K., et al. IReS: Intelligent, Multi-Engine Resource Scheduler for Big Data Analytics Workflows. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015.

[8] Funahashi, K. On the approximate realization of continuous mappings by neural networks. Neural networks 2.3 (1989): 183-192.

[9] Gensim Latent Dirichlet Allocation, https://radimrehurek.com/gensim/models/ldamodel.html

[10] Ghemawat, S., Gobioff, H. and Leung, S. T. The Google file system. ACM SIGOPS operating systems review. Vol. 37. No. 5. ACM, 2003.

[11] Giannakopoulos, I., Tsoumakos, D., Papailiou, N. and Koziris, N.: PANIC: Modeling Application Performance over Virtualized Resources. In Proceedings of the 2015 IEEE International Conference on Cloud Engineering (IC2E 2015), 9-13 March, Tempe, AZ, USA.

[12] Herodotou, H. et al. Starfish: A Self-tuning System for Big Data Analytics. In CIDR, 2011.

[13] Heroku add-ons. https://addons.heroku.com/.

[14] Ho, T. K. The random subspace method for constructing decision forests. IEEE Transactions on Pattern Analysis and Machine Intelligence, 20(8):832–844, 1998.

[15] Hortonworks Sandbox 2.1. http://hortonworks.com/products/hortonworks-sandbox/.

[16] http://cassandra.apache.org/

[17] http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[18] http://impala.io/

[19] http://spark.apache.org/graphx/

[20] http://spark.apache.org/streaming/

[21] http://wiki.apache.org/hadoop/SequenceFile

[22] https://en.wikipedia.org/wiki/Feature_hashing

[23] https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation

[24] https://hadoop.apache.org/

[25] https://software.intel.com/en-us/intel-cilk-plus

[26] https://wiki.apache.org/hadoop/MapReduce

[27] I. Giannakopoulos, D. Tsoumakos, N. Papailiou and N. Koziris: PANIC: Modeling Application Performance over Virtualized Resources. In Proceedings of the 2015 IEEE International Conference on Cloud Engineering (IC2E 2015), 9-13 March, Tempe, AZ, USA.
[28] Joachims, T. Making large scale SVM learning practical. (1999).
[29] Latent Dirichlet Allocation http://spark.apache.org/docs/latest/mllib-clustering.html#latent-dirichlet-allocation-lda
[30] Lim, H., Herodotou, H. and Babu, S. Stubby: A Transformation-based Optimizer for Mapreduce Workflows. VLDB, 2012.
[31] Mikolov, T., et al. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013).
[32] Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems. 2013.
[33] Papailiou, N., et al. Graph-Aware, Workload-Adaptive SPARQL Query Caching. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015.
[34] Rousseeuw, P. J. and Leroy, A. M. Robust regression and outlier detection. 1987.
[35] Running Databases on AWS. http://aws.amazon.com/running_databases/.
[36] Simpson, T. W., et al. Kriging models for global approximation in simulation-based multidisciplinary design optimization. AIAA journal 39.12 (2001): 2233-2241.
[37] Spark - Word2Vec, https://spark.apache.org/docs/latest/mllib-feature-extraction.html#word2vec
[38] spark.apache.org/streaming/
[39] Thusoo, Ashish, et al. Hive: a warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment 2.2 (2009): 1626-1629.
[40] TPC-H benchmark. http://www.tcp.org/hspec.html.
[41] Tsoumakos, D. and Mantas, C. The Case for Multi-engine Data Analytics. In Proceedings of the 2013 Workshop on Middleware for HPC and Big Data Systems (MHPC'13, part of Euro-Par 2013), Aachen, Germany, August 27, 2013.
[42] Vavilapalli, V. K., et al. Apache hadoop yarn: Yet another resource negotiator. Proceedings of the 4th annual Symposium on Cloud Computing. ACM, 2013.
[43] WEKA, http://weka.wikispaces.com
[44] Word2Vec, https://code.google.com/p/word2vec/
[45] Zaharia, M., et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

Appendix A

K. Doka, N. Papailiou, D. Tsoumakos, C. Mantas and N. Koziris: IReS: Intelligent, Multi-Engine Resource Scheduler for Big Data Analytics Workflows. In Proceedings of the 2015 ACM SIGMOD/PODS International Conference on Management of Data (Demo Track), May 31- June 4, 2015, Melbourne, Australia.

Appendix B

I. Giannakopoulos, D. Tsoumakos, N. Papailiou and N. Koziris: PANIC: Modeling Application Performance over Virtualized Resources. In Proceedings of the 2015 IEEE International Conference on Cloud Engineering (IC2E 2015), 9-13 March, Tempe, AZ, USA.

IReS: Intelligent, Multi-Engine Resource Scheduler for Big Data Analytics Workflows

Katerina Doka
Computing Systems Lab
National Technical University of Athens, Greece
katerina@cslab.ece.ntua.gr

Nikolaos Papailiou
Computing Systems Lab
National Technical University of Athens, Greece
npapa@cslab.ece.ntua.gr

Dimitrios Tsoumakos
Department of Informatics
Ionian University
Corfu, Greece
dtsouma@ionio.gr

Christos Mantas
Computing Systems Lab
National Technical University of Athens, Greece
cmantas@cslab.ece.ntua.gr

Nectarios Koziris
Computing Systems Lab
National Technical University of Athens, Greece
nkoziris@cslab.ece.ntua.gr

ABSTRACT

Big data analytics tools are steadily gaining ground at becoming indispensable to businesses worldwide. The complexity of the tasks they execute is ever increasing due to the surge in data and task heterogeneity. Current analytics platforms, while successful in harnessing multiple aspects of this "data deluge", bind their efficacy to a single data and compute model and often depend on proprietary systems. However, no single execution engine is suitable for all types of computation and no single data store is suitable for all types of data. To this end, we demonstrate *IReS*, the *Intelligent Resource Scheduler* for complex analytics workflows executed over multi-engine environments. Our system models the cost and performance of the required tasks over the available platforms. *IReS* is then able to match distinct workflow parts to the execution and/or storage engine among the available ones in order to optimize with respect to a user-defined policy. During the demo, the attendees will be able to execute workflows that match real use cases and parametrize the input datasets and optimization policy. The underlying platform supports multiple compute and data engines, allowing the user to choose any subset of them. Through the inspection of the produced plan, its execution and the collection and presentation of numerous cost and performance metrics, the audience will experience first-hand how *IReS* takes advantage of heterogeneous runtimes and data stores and effectively models operator cost and performance for actual and diverse workflows.

Categories and Subject Descriptors

H.4.m [**Information Systems Applications**]: Miscellaneous

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2735377.

Keywords

Multi-Engine Optimization, Cost Modelling, Profiling, Big Data, Analytics Workflows

1. INTRODUCTION

Big data analytics have become a necessity for the majority of industries [17], taking the lead in risk assessment, business process effectiveness, market analysis, etc. [23, 16]. Enabling engineers, analytics experts and scientists alike to tap the potential of vast amounts of business-critical data has grown increasingly important. Such data analysis demands a high degree of parallelism in both storage and computation: Modern datacenters host huge volumes of data, stored over large numbers of nodes with multiple storage devices and process them using thousands or millions of cores.

The demand for near-real-time, data-driven analytics has given rise to diverse execution engines and data stores that target specific data and computation types (e.g., [1, 4, 2, 13, 3, 10]). Many of these systems are now offered as a service by IaaS providers, enabling a very wide deployment range. There also exist approaches in the literature that manage to optimize their performance (e.g., [20, 22]) by automatically tuning a number of configuration parameters. Yet, these schemes assume strictly single-engine environments (mainly the Hadoop ecosystem), thus considering specific data formats and query/analytics task types.

However, modern workflows have become increasingly long and complex [19]. Specifically, workflows may include multiple data types (e.g., relational, key-value, graph, etc.) generated from different resources. What is more, they are executed under varying constraints and policies (e.g., optimize performance or cost, require different fault-tolerance degrees, etc.). Finally, workflow operators can be greatly diverse, from simple Select-Project-Join (SPJ) and data movement to complex NLP-, graph- or custom business-related operations. There currently exists no single platform that can optimize for this complexity [27].

Sensing this trend, cloud software companies now offer software distributions in pre-cooked VM images or as a service. These distributions incorporate different processing frameworks, data stores and libraries to alleviate the burden of multiple installations and configurations (e.g., [5, 9,

8, 12]). Yet, such multi-engine environments lack a *meta-scheduler* that could automatically match tasks to the right engine(s) according to multiple criteria, deploy and run them without manual intervention. A recent attempt along this line [25, 26] focuses more on lower-level database operators, emphasizing on their automatic translation from/to specific engines via an XML-based language. Yet, this is a proprietary tool with limited applicability and extension possibilities for the community.

To address multi-engine analytics workflow optimization we present the *Intelligent Multi-Engine Resource Scheduler (IReS)*, an integrated, open source platform for managing, executing and monitoring complex analytics workflows[1]. Its goal is to provide adaptive, cost-based and customizable resource management of the diverse execution and storage engines available. IReS incorporates a modelling framework that constantly evaluates the cost, quality and performance of data and computational resources in order to decide on the most advantageous store, indexing and execution pattern.

To that direction, our system handles existing open-source execution models (e.g., Map-Reduce, Bulk Synchronous Parallel) as well as state-of-the-art centralized and distributed storage engines (RDBMSs, NoSQL, distributed file-systems, etc.) in order to have a broad applicability and increased performance gains. IReS is able to optimize workflows consisting of tasks that range from simple group-by, aggregation or complex joins between different data sources to machine-learning tasks and queries on graph data in combination with relational data. In the current implementation, the system bases its operation on the following elements:

- A profiling and modelling engine that benchmarks operator performance and cost for different engine configurations. Outputs are collected via budget-constraint executed benchmarks. The learned models are stored and utilized for the planning phase of the workflow.
- A JSON-based metadata framework that describes operators in abstract and instantiated forms, enabling search and matching of operators that perform a similar task in the planning phase.
- A decision-making and enforcing process that chooses among different equivalent workflow execution plans (i.e., on different engines, resulting in equivalent output) based on cost and performance models and schedules the execution.

The resulting optimization is orthogonal to (and in fact enhanced by) any optimization effort within a single engine. Unlike [25, 26], IReS is a fully open-source platform that targets both low (e.g., join, sort, etc.) as well as high level (e.g., machine learning, graph processing) operators, treating them as black boxes. The generic profiling/modelling method it relies upon allows for easy addition of new operators and engines.

Our demonstration of the IReS system will showcase its ability to i)model operator performance according to different engines and their resources and ii)adaptively decide on which operator version to run based on the optimization policy and the available engines. The demonstration platform will integrate Hadoop [1], Hama [2], Spark [4] and PostgreSQL [11] with HDFS [7], HBase [3] and Elasticsearch [6] and operate upon real-life and synthetic workflows chosen to include diverse datasets and computation types. The participants will have a rich interaction with IReS, controlling policy and input aspects, while being able to evaluate the advantages of multi-engine optimization by inspecting generated plans and output.

[1]IReS is a central component of the *ASAP (Adaptive, highly Scalable Analytics Platform)* EU-funded project. ASAP envisions a unified, open-source execution framework for scalable data analytics. `http://www.asap-fp7.eu/`

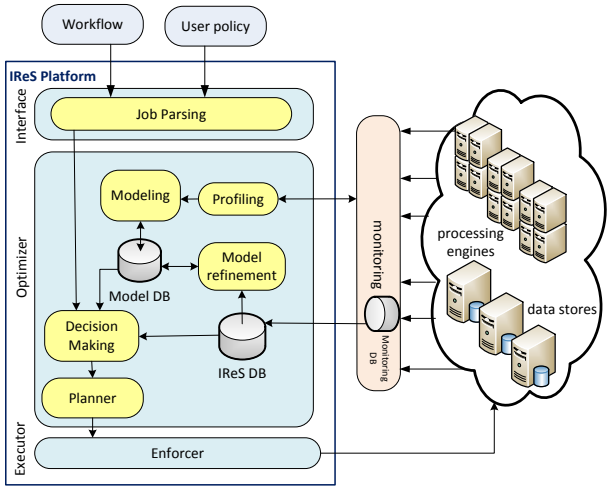


Figure 1: Architecture of the IReS platform

2. ARCHITECTURE

IReS focuses on highly efficient and user-customizable execution of analytics tasks (or workflows). This is made possible through the transparent modeling, monitoring and scheduling that involves different execution engines and storage technologies. Consequently, our system is able to execute all types of analytics workflows by adaptively choosing to execute each sub-part of the workflow to a (possibly different) deployed engine. The IReS platform assigns sub-tasks to the most advantageous technology(-ies) available and ensures resource and dataflow scheduling in order to enhance performance: If a single engine is used, enhancement will be achieved through optimized resource allocation and elasticity modeling (e.g., execute on more VMs, or on smaller cluster with larger main memory, etc.); if multiple ones are required, enhancements will relate both to single-engine optimization and to workflow management that decides what is the best execution plan and data-flow (e.g., execute sub-task 1 first, intermediate results should be stored on a NoSQL engine and then sub tasks 2 and 3 run in parallel and write final results to HDFS files).

The central notion behind the IReS platform is to create detailed models of the costs and performance characteristics of various analytics operations over multiple execution engines. These models are then used to match the user optimization policy with the available execution engines.

The architecture of the IReS platform is depicted in Figure 1. IReS comprises of three layers, the *interface*, the *optimizer* and the *executor* layer.

The *interface layer* is responsible for communicating with the application UI in order to receive the input that is necessary for its operations. It consists of the *job parser* module, which identifies execution artifacts such as operators, data, their dependencies and accompanying metadata. Moreover,

```
{"joinOp": {
  "constraints": {
    "input1": {
      "data_info": {
        "attributes": ["attr1", "attr2"]}},
    "input2": {
      "data_info": {
        "attributes": ["attr1", "attr2"]}},
    "output1": {
      "data_info": {
        "attributes": ["attr1", "attr2"]}},
    "op_specification": {
      "algorithm": {
        "join": {
          "join_condition":
            "input1.attr1=input2.attr1"}}}}}}
```

Figure 2: Metadata description of the abstract join operator

it validates the user-defined policy. All this information must be robustly identified, structured in a dependency graph and stored.

The *optimizer layer* is responsible for optimizing the execution of an analytics workflow with respect to the policy provided by the user. The core component of the optimizer is the *Decision Making* module, which determines the optimal execution plan in real-time. This entails deciding on where each subtask is to be run, under what amount of resources provisioned, the plan for moving data to/from their current locations and between runtimes (if more than one is chosen) and defining the output destinations. Such a decision must rely on the characteristics of the analytics task in hand and the models of all possible engines. These models are produced by the *Modeling* module and stored in a database called *Model DB*. The initial model of an engine results from profiling and benchmarking operations in an offline manner, through the *Profiling* module. This module directly interacts with the pool of physical resources and the monitoring layer in-between. While the workflow is being executed, the initial models are refined in an online manner by the *Model refinement* module, using monitoring information of the actual run. Such monitoring information is kept in the IReS DB and is utilized by the decision making module as well, to enable real-time, dynamic adjustments of the execution plan based on the most up-to-date knowledge.

The *executor layer* is the layer that enforces the optimal plan over the physical infrastructure. It includes methods and tools that translate high level "start runtime under x amount of resources", "move data from site Y to Z" type of commands to a workflow of primitives as understood by the specific runtimes and storage engines. Moreover, it is responsible for ensuring fault tolerance and robustness through real-time monitoring.

In the following, we describe in more detail the role, functionality and internals of the most important modules of the platform.

Job Parsing Module: This module takes as input the user-defined workflow, formulated in a dependency graph format and expressed in a way that allows for various levels of abstraction using a metadata framework. Moreover, the module takes as input the user optimization parameters, which could translate to performance, cost, availability, etc.

The main challenge of defining a workflow description metadata framework is the fact that it requires to be abstract at the user level. The user should be able to describe the data and operators that comprise her workflow in a way as abstract as she desires. The IReS planner and workflow scheduler need to remove that abstraction, find all the alternative ways of materializing the workflow and select the most beneficial, according to the user-defined policy.

Our proposed metadata framework describes *data* and *operators*. Data and operators can be either *abstract* or *materialized*. Abstract are the operators and datasets that are described partially or in a high level by the user when composing her workflow whereas materialized are the actual operator implementations and existing datasets, either provided by the user or residing in a repository.

Both data and operators need to be accompanied by a set of metadata, i.e., properties that describe them and can be used to match (a) abstract operators to materialized ones and (b) data to operators. Such properties include input data types and parameters of operators, location of data objects or operator invocation scripts, data schemata, implementation details, engines, etc. The metadata defined for each object have a generic tree format (JSON). To avoid restricting the user and allow for extensibility, the first levels of the metadata tree are predefined, while users can add their ad-hoc subtrees to define their custom data or operators. Moreover, some fields (mostly the ones related to the operator and data requirements) are *compulsory* while the rest (e.g., known cost models, statistics, etc.) are *optional*. Materialized data and operators need to have all their compulsory fields filled in with information. Abstract data and operators do not adhere to this rule. Apart from having empty fields, they can also support regular expressions (e.g., the * symbol under a field means that the abstract object matches materialized ones with any value of that field).

Let us take a `join` operator on a single attribute as an example. In its abstract form, the `joinOP` operator (see Figure 2) needs only define two input parameters, the condition under which they are joined and an output parameter. Each of the input parameters and the output are abstract `data_info` objects with two attributes: "attr1" represents the field of the join predicate while "attr2" represents the second available field in each `data_info` object. The `op_specification` field of this operator specifies its operation, a single join algorithm, and defines the join condition (in this case an inner join). In short, the abstract join operator defines a format that any join operator implementing the specific functionality needs to follow.

The materialized operators include, on top of that, all information required in order to perform the operation on an execution engine. In `join_1` (see Figure 3.a), the operator executes the join over Hadoop; it thus includes Hadoop-specific information about the input, output and the engine. The inputs and output in this case have specific attribute types and an engine specification (under `engine`) containing the location of the data and information about their structure. The operator itself also has an engine specification (`engine_specification`) indicating its execution location. The example in Figure 3.b describes `join_2`, which joins an HBase and a relational table and outputs the result to HDFS. It runs as a local Java process.

To discover the actual implementations that comply with the description of an abstract operator provided by the user, we employ a tree matching algorithm to make sure that all metadata constraints are met, i.e., all compulsory fields are consistent. This is performed by the decision making mod-

Figure 3: Metadata descriptions of the two materialized join operators

ule, described subsequently. In our example, both `join_1` and `join_2` match `joinOP` and are thus considered when constructing the optimized execution plan.

Apart from the compulsory fields, which are necessary for the matching of abstract to materialized operators, the metadata descriptions of the materialized joins both contain the optional `optimization` field, which holds additional information that assists in the optimization of the workflow. In the case of `join_1`, a cost function is provided by the developer of the operator while for `join_2` the platform is instructed to create one by profiling over specific metrics (execution time and required RAM in our case).

Modelling Module: This module is responsible for constructing models on a per operator–engine combination basis. The relevant literature review [18, 28, 24] has revealed that models already exist for a very limited number of operators and engines and some of them entail knowledge of the code to be executed. Contrarily, we treat materialized operators as "black boxes", assuming no prior knowledge of their internals, and model them using profiling in an offline mode, as well as machine learning over actual runs.

Profiling Module: The profiling module functions in an operator-agnostic way, having no prior knowledge other than the profiler input parameters. These parameters fall into three categories:

- Data specific parameters: These parameters describe the data to be used for the operator profiling, e.g., the type of data and its size.
- Operator specific parameters: These parameters relate to the algorithm of the operator, e.g., the number of output clusters in k-means.
- Resource specific parameters: These parameters define the resources to be tweaked during profiling, e.g., cluster size, storage size, main memory, etc.

The output of each run is the profiled operator's performance and cost (e.g., completion time and I/O operations, average memory and CPU consumption, etc) under each combination of the input parameter values for specific user-defined optimization metrics, such as cost in $ or I/O, latency, throughput, etc. Both the input parameters as well as the output metrics are given by the user/developer.

The aim of the profiling module is to create a surrogate estimation model [21], including neural networks, SVM, interpolation and curve fitting techniques, for each operator running over a specific engine. To that end, we need to sample the operator function by running automated experiments for various values of each of the input parameters and measure the outputs. To create the most accurate surrogate within a budget of experiments, adaptive sampling techniques are adopted to select the combinations of values to be used as input of each run.

Decision Making Module: This module performs the intelligent exploration of all the available execution plans and the discovery of the optimal execution plan according to the user-defined optimization objectives. Initially, it transforms the abstract workflow representation, described as a DAG graph, into a materialized workflow DAG graph that contains all the alternative paths of materialized operators that match the abstract workflow. To do so, for each abstract operator, it searches the library of available materialized operators to find all matches. Our decision module is using an efficient tree matching algorithm to avoid unnecessary comparisons and follow the hierarchical structure of the tree-based metadata constrains. When all operator matches are discovered, the decision making module intelligently consults the input and output specifications of the materialized operators and adds the required move/transform operators. Those operators are needed in order to connect operators of different engines and input/output configurations and generate the final materialized workflow DAG graph.

To find the optimal execution plan, our decision module uses a dynamic programming planner that explores the materialized workflow DAG in order to find the plan that best matches the user optimization policy. To estimate operator performance metrics, our planner consults the profiler module that holds surrogate estimator models for each one of the materialized operators. In our current implementation, our planner can be configured to optimize one metric or a function of multiple performance metrics that the user is interested in. We are currently investigating methods for optimizing multiple dimensions of performance metrics, like finding Pareto frontier execution plans.

In the course of the workflow execution, the real-time monitoring information is fed back to the decision making module in order to take into account current running conditions and adapt accordingly. Moreover, our planner considers more than a single final plan to ensure that alternatives will exist in case of failures or other unpredictable circumstances without having to run the whole decision making process from scratch. These alternatives include the top-k (instead of the best) plans according to the user's optimization preferences or a sample of the multi-dimensional space covering different environments.

Enforcer Module: The enforcer module undertakes the execution of the ensuing plan. First, the enforcer needs to validate the plan by checking the availability of resources and data, the load of the engines, etc. After ensuring that everything is correct, it enforces the plan actions by translating the plan steps to standard, low-level API calls. Such actions might entail code and/or data shipment if necessary. In case of faults and failures occurring on-the-fly, an alternative plan will substitute the current.

3. DEMONSTRATION DESCRIPTION

Our system is controlled by a comprehensive web-based GUI that attendees will utilize. The basic interaction dimensions include input parametrization, operator model visualization, execution plan inspection and execution output evaluation. The GUI controls a cloud-based deployment of several runtime engines and data stores over 16 virtual machines of an Openstack cluster hosted in our lab.

Workflows and Datasets: The users will have the opportunity to test the IReS platform either using one of four predefined workflows or assembling their own, using operators from the ASAP operator library. A diverse set of operations of varying complexity and execution parameters is covered including basic SQL queries (selections, projections, joins), ML algorithms (classification and clustering) as well as NLP methods (named entity recognition).

Three of the predefined workflows represent real use cases driven by business needs. These cover complex data manipulations in the areas of business analytics on telecommunication data and web data analytics, provided by a large telecommunications company and a well-known web archiving organization respectively. The input datasets for these workflows consist of anonymized telecommunication traces and web content data (WARC files). Subsets of those datasets can be used for each of the available workflows. A short description for each workflow follows:

Web analytics - Clustering: The workflow starts by selecting a subset of the initial web content indexed by Elasticsearch. Feature-extraction (e.g., tf-idf) is performed on these documents; the outputs are clustered using k-means clustering (chosen among weka, mahout and MLlib running centrally or over Hadoop or Spark respectively).

Web analytics - Named Entity Recognition: A subset of the dataset (obtained via a query over Elasticsearch as before) undergoes named entity extraction. The results are joined with the YAGO external ontology database [15] to find possible matches and output them.

Telco analytics - Peak Detection: The workflow involves processing of anonymized CDR data (residing in an RDBMS) via clustering along time and space in order to detect peaks in load, according to a set of criteria. The results of this phase enrich a database (relational or graph DB)

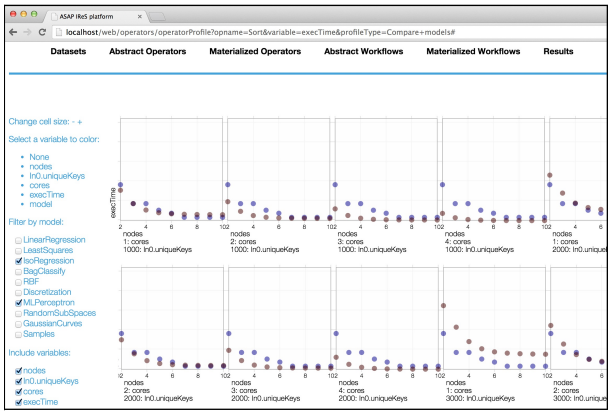


Figure 4: IReS web application GUI - Materialized Operator models

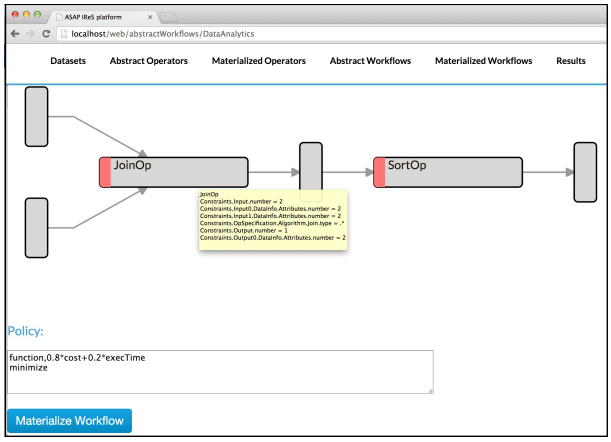


Figure 5: IReS web application GUI - Abstract Workflow

that contains peaks detected in previous runs. The dataset of peaks is used to discover clusters of calls that occur with or without regularity.

Synthetic workflow: A sample workflow that showcases a simple join operation between two datasets residing in different stores, namely PostgreSQL and HBase, followed by a sorting operation. For this workflow, we use synthetic data produced by the popular TPC-H [14] benchmark generator.

User defined workflow: The users will have the opportunity to construct custom workflows by utilizing the current library of operators and datasets.

Interface: Through the platform's front-end, users are able to inspect available operators and datasets, construct the workflow they want to execute or choose one of the pre-defined ones, specify the input parameters, review the proposed execution plan and monitor its progress and output. Our proposed interface consists of 6 sections, namely: *Datasets*, *Abstract Operators*, *Materialized Operators*, *Abstract Workflow*, *Materialized Workflow* and *Results*.

In the *Datasets* tab, the user can browse through the available datasets and view their metadata. In the *Abstract Operator* tab, the user can chose an existing abstract operator and customize it by changing its accompanying metadata. Among others, the user can specify the engine(s) on which an operator will be run or the storage where the data will be saved. The engines supported by IReS include JVM,

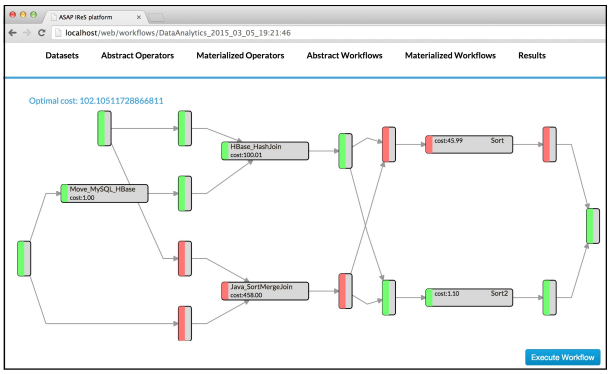


Figure 6: IReS web application GUI - Materialized Workflow tab

Hadoop MapReduce [1], Spark [4], Hama [2] for processing and HDFS [7], HBase [3], Elasticsearch [6], PostgreSQL [11] and local file system for storage.

The *Materialized Operator* tab visualizes the materialized operator models that have been created offline and are stored in the Model DB. The user can plot the modelled metrics (e.g., execution time) versus various parameters (e.g., number of nodes, dataset size, etc.) for a plethora of machine learning models (Figure 4).

The *Abstract Workflow* tab gives the user the opportunity to view the predefined workflows and choose one of them or create one of her own by combining abstract operators and datasets. Either way, the workflow is visualized in its abstract form as a graph, consisting of operator and data nodes (Figure 5). Moreover, the user can specify the policy for which the platform will optimize the execution plan. The supported choices include minimizing cost, execution time or a function of them.

After selecting the abstract workflow and its input parameters, the user is able to move forward to the *Materialized Workflow* tab (Figure 6). A preview of the materialized plan is presented here and the user can inspect the platform's choices for each of the operators and the intermediate results, along with an estimation of the execution cost and performance. At this stage, the user should be able to validate the strategy that will be followed in order to optimize for the chosen attributes. It is also possible to go back and change the input parameters if the user wants to override some of the system's decisions.

The *Results* section offers a live preview of the execution so far. For each finished workflow stage, a summary of its execution aspects is presented, including execution time, resources allocated to each operator, resources actually used, operator throughput and I/O and network costs (if applicable). The cost of each operation as calculated by its cost model is also shown. For the execution steps that have not yet been concluded, an approximation for the anticipated performance and cost measures is presented, if possible via previous knowledge of the operator.

4. ACKNOWLEDGEMENTS

This work has been supported by the European Commission in terms of the ASAP FP7 ICT Project under grant agreement no 619706. Nikolaos Papailiou has received funding from IKY fellowships of excellence for postgraduate studies in Greece - SIEMENS program.

5. REFERENCES

[1] Apache Hadoop. `http://hadoop.apache.org/`.
[2] Apache Hama. `https://hama.apache.org/`.
[3] Apache HBase. `http://hbase.apache.org/`.
[4] Apache Spark. `https://spark.apache.org/`.
[5] Cloudera Distribution CDH 5.2.0. `http://www.cloudera.com/content/cloudera/en/downloads/cdh/cdh-5-2-0.html`.
[6] elasticsearch. `http://www.elasticsearch.org/overview/elasticsearch/`.
[7] Hadoop Distributed File System. http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[8] heroku add-ons. `https://addons.heroku.com/`.
[9] Hortonworks Sandbox 2.1. `http://hortonworks.com/products/hortonworks-sandbox/`.
[10] monetdb. `https://www.monetdb.org/`.
[11] Postgresql. http://www.postgresql.org/.
[12] Running Databases on AWS. `http://aws.amazon.com/running_databases/`.
[13] Stratosphere Project. `http://stratosphere.eu/`.
[14] TPC-H benchmark. *http://www.tcp.org/hspec.html*.
[15] YAGO2s: A High-Quality Knowledge Base. `http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/`.
[16] The Power of Combining Big Data Analytics with Business Process Workflow. CGI Whitepaper, 2013.
[17] 84% Of Enterprises See Big Data Analytics Changing Their Industries' Competitive Landscapes In The Next Year . Forbes Magazine, 2014.
[18] S. Babu. Towards automatic optimization of mapreduce programs. In *ACM symposium on Cloud computing*, 2010.
[19] M. Ferguson. Architecting a big data platform for analytics. *A Whitepaper Prepared for IBM*, 2012.
[20] H. Herodotou et al. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, 2011.
[21] Y. Jin. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 2011.
[22] H. Lim, H. Herodotou, and S. Babu. Stubby: A Transformation-based Optimizer for Mapreduce Workflows. *VLDB*, 2012.
[23] A. Pariyani, U. G. Oktem, and D. L. Grubbe. Process risk assessment uses big data, 06-03-2013. http://bit.ly/1vDlTVk.
[24] B. Sharma, T. Wood, and C. R. Das. Hybridmr: A hierarchical mapreduce scheduler for hybrid data centers. In *ICDCS*. IEEE, 2013.
[25] A. Simitsis, K. Wilkinson, U. Dayal, and M. Hsu. HFMS: Managing the Lifecycle and Complexity of Hybrid Analytic Data Flows. In *ICDE*. IEEE, 2013.
[26] A. Simitsis, K. Wilkinson, and P. Jovanovic. xPAD: A Platform for Analytic Data Flows. In *SIGMOD 2013*.
[27] D. Tsoumakos and C. Mantas. The Case for Multi-Engine Data Analytics. In *Euro-Par 2013: Parallel Processing Workshops*. Springer, 2014.
[28] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Conference on Autonomic computing*. ACM, 2012.

PANIC: Modeling Application Performance over Virtualized Resources

Ioannis Giannakopoulos*, Dimitrios Tsoumakos§, Nikolaos Papailiou* and Nectarios Koziris*
* *Computing Systems Laboratory, School of ECE, National Technical University of Athens, Greece*
{*ggian, npapa, nkoziris*}*@cslab.ece.ntua.gr*
§ *Department of Informatics, Ionian University, Corfu, Greece*
dtsouma@ionio.gr

***Abstract*—In this work we address the problem of predicting the performance of a complex application deployed over virtualized resources. Cloud computing has enabled numerous companies to develop and deploy their applications over cloud infrastructures for a wealth of reasons including (but not limited to) decrease costs, avoid administrative effort, rapidly allocate new resources, etc. Virtualization however, adds an extra layer in the software stack, hardening the prediction of the relation between the resources and the application performance, which is a key factor for every industry. To address this challenge we propose *PANIC*, a system which obtains knowledge for the application by actually deploying it over a cloud infrastructure and then, approximating the performance of the application for the all possible deployment configurations. The user of *PANIC* defines a set of resources along with their respective ranges and then the system samples the deployment space formed by all the combinations of the resources, deploys the application in some representative points and utilizes a wealth of approximation techniques to predict the behavior of the application in the remainder space. The experimental evaluation has indicated that a small portion of the possible deployment configurations is enough to create profiles with high accuracy for three real world applications.**

***Keywords*-application profiling; application performance; cloud applications; performance modeling**

I. Introduction

Cloud Computing has brought forth a new computing paradigm, in which virtualized resources can be allocated and freed on demand. Cloud-based deployments offer multiple advantages including (but not limited to) decreased costs, less administrative burden, bigger flexibility, seemingly infinite resources to harness on a pay-as-you-go manner. As a direct consequence, almost all every-day users are using at least one cloud-based application, while the amount of businesses that take advantage of the Cloud's offerings are ever increasing [1]. This trend is particularly observed in modern compute and data intensive platforms (e.g., Hadoop, NoSQL DBs [2]), which are now the basis of every analytics application/job. As most of these engines are designed to scale horizontally, deploying them over virtualized infrastructures seems like a natural fit.

Predicting application performance is a well-known research and business problem ([3], [4], [5], [6]). Building a reliable model of application behavior offers engineers and analysts a wide range of advantages: Most importantly, it allows for better resource allocation both at deployment and during runtime. This translates to happier customers (minimizing delays, downtimes, denial of service, etc) and better cost management.

As applications become more complex, so does building accurate models of their behavior. This issue is exacerbated by the fact that many applications are now deployed over cloud infrastructures. Applications now run over virtualized resources that: 1) are highly heterogeneous between different providers and inside a single provider; 2) are shared with an unknown number of other applications; 3) have performance that is abstracted from the underlying physical layer that the user understands. As a consequence, performance varies greatly among different providers, different deployments or even different times of day. This is especially true for resource-demanding platforms that require large amounts of CPU and memory/disk I/O and scale horizontally at runtime. These engines are heavily utilized nowadays for storing and analyzing Big Data for almost any conceivable reason.

In this work, we present *PANIC* (**P**rofiling **A**pplications **I**n the **C**loud), an application profiling system that focuses on this important element: It models performance based on the amount and type of virtualized resources allocated to the application. The main goals of *PANIC* are: (i) to provide a generic methodology, applicable for any kind of cloud application deployed, (ii) to offer customizable tradeoff between performance accuracy and profiling cost and (iii) to be customizable, in the sense that a user can choose from a pool of resources for which the profiling will be executed.

The main idea of *PANIC* is to provide predictions for the application performance by actually deploying the application in representative resource combinations and approximate the performance for the rest, non deployed combinations. The system confronts the application performance, which should be a countable quantity, as a high dimensional function and it utilizes a set of approximation techniques (from regression to machine learning classifiers) to identify the behavior of the application. Since the possible combinations of resources grows exponentially with the complexity of the application, *PANIC* exploits sampling techniques in

order to pick representative configurations, deploy the application according to them and, eventually, provide predictions for all the possible configurations. Through our experimental evaluation where we deploy three typical cloud applications, we demonstrate that our system is capable of providing an accurate application profile by deploying only 10% of the possible deployment setups.

II. The problem and some assumptions

Let us assume a two-tier application, consisting of a web server and a database server, deployed over an IaaS provider. We also assume that we can predict the application load and we can deploy each tier in the following possible setups:

Table I: Possible setups of a typical Web Application

Tier 1 - Web server	RAM (MB)	512, 1024, 2048
	Cores	1, 2, 4, 8, 16
Tier 2 - Database Server	Storage (GB)	5, 10, 20, 30, 100

We assume that both the Web and Database Servers will run in a single host each. The following question arises: what performance will the application achieve for different choices of the offered resources for each tier, for a specific load? Answering this question leads to an accurate performance profile of the application that, in turn, delineates the application behavior in general.

In the general case, assume that we have an application described by n dimensions. Each dimension is denoted as $d_i, i \in [1, n]$ and it can be related to exactly one application Tier. The Cartesian product of the dimensions forms the space of the possible setups (denoted as D): $D = d_1 \times d_2 \times ... \times d_n$. The performance of the application is a countable size indicating the ability of the application to fulfill its objectives (e.g. achieved throughput, latency, etc.). The performance space is denoted as P and it is a single dimensional space (our work is easily extensible to support multidimensional performance spaces). Hence, the profile of the application (denoted as p) is defined as a function $p : D \rightarrow P$, indicating the achieved performance for each acceptable deployment setup.

Since the function p is unknown and it cannot be estimated for the entire input space, we address its estimation as a typical function approximation problem. Specifically, we want to estimate the function $\widehat{p} : D \rightarrow P$ with respect to keeping $\sum_{d \in D} |\widehat{p}(d) - p(d)|$ minimum. The approximation process involves sampling D (let D_s represent the set of sampled points) and calculating the values $p(d) \forall d \in D_s$. D_s, along with the respective values $p(d_i), d \in D_s$ are given as input to our approximation algorithms, which in turn create the function $\widehat{p}$. We utilize a large number of approximation techniques, from regression to machine learning and classification algorithms. Since the estimation of $p(d)$ entails the actual deployment of the application, it is obvious that the needed time to estimate $\widehat{p}$ is dominated by $|D_s|$: assuming that the deployment time is constant regardless of the deployment setup, the number of deployments will eventually determine the execution time. Furthermore, the points that are going to be picked into D_s have a huge impact on the accuracy of $\widehat{p}$.

III. Our approach

In Algorithm 1, the general methodology used for creating a profile for a given application is provided. The algorithm expects a valid application description A followed by an input domain D, representing the possible setups the application can be deployed into and a list of applicable models. The profiling process occurs iteratively: while the termination condition is not fulfilled, the domain space is sampled, a new point p is picked and the application is deployed according to p. The deployment produces a performance metric d which is then used to train in an incremental manner all the available models. The output of the process is the model which achieves the highest accuracy, according to a user specified metric.

Algorithm 1 Main profiling algorithm

Require: application A, input domain D, models M
Ensure: model m

```
while not termination_condition do
    p ← SAMPLE(D)
    d ← DEPLOY(p)
    for m ∈ M do
        m.train_incrementaly(p,d)
    end for
end while
return best_model(M)
```

The termination condition can vary. In many cases, it can be a threshold of sampled points that, if reached, the condition is true and the algorithm terminates. In other cases, it can be related to the achieved accuracy: if the trained model achieves to predict the objective function with error lower than a user defined threshold, the termination condition is reached. As we will present in the following section, the nature of the termination condition directly relates to the nature of the sampling algorithm.

A. Sampling

The sampling procedure occurs at the beginning of each profiling loop. The sampler receives as input the domain space D of the application, which is composed of all the acceptable deployment points. If the termination condition in Algorithm 1 is related to the number of sampled points, then the sampler receives as input a positive number $0 < s \leq 1.0$ indicating the maximum number of points that the sampler should return, as a portion of the number of points in D. Each point returned by the sampler will be used for

deployment, the application performance will be measured and then an approximation model will be trained using the acquired information.

There are many methodologies for sampling a multidimensional space; We can categorize the methods we support in the following categories: (i) Static sampling, where the sampler needs no other information than the domain space characteristics (dimensions and acceptable values) to pick the next sample, (ii) Adaptive sampling, where the sampler exploits the knowledge obtained by the deployment of previously picked samples.

The static approach does not take into consideration the application performance. Typical examples of static sampling are the *Random* sampler, that returns random points and the *Uniform* sampler which constructs a multidimensional grid in the input space D, and returns points belonging to the grid. The adaptive approach, on the other hand, exploits the knowledge obtained from each deployment/sample, enabling the sampler to return more samples in regions of the domain space D where the performance appears to have fluctuations. Equivalently, an adaptive sampler will favor areas of D that affect the application performance more. In Algorithm 2 we provide the *Greedy Adaptive Sampling Algorithm*.

Algorithm 2 Greedy Adaptive Sampling Algorithm

Require: input domain D, chosen samples L, number K
Ensure: sample s
1: **if** $|L| < K$ **then**
2: s = borderPoint(D)
3: **else**
4: max = 0
5: **for all** $t_1 \in L$ **do**
6: **for all** $t_2 \in L$ **do**
7: a = find_midpoint(t_1, t_2, D)
8: **if** $|t_1 - t_2| > max$ and $a \notin L$ **then**
9: max = $|t_1 - t_2|$
10: $s = a$
11: **end if**
12: **end for**
13: **end for**
14: **end if**
15: **return** s

The algorithm expects as input the domain space D, the list of all the previously picked samples L and a positive number K. At first the algorithm returns K points from the border of the hyperplane defined by all points $d \in D$. The border points are picked with the notion that if the application is deployed using the highest and lowest available resources for each dimension (in combination, between different dimensions), the objective function will most likely present its highest and lowest values in the respective points. Furthermore, the K border points are also equidistant in order to avoid high gathering of points in a small region of the input space. When the number of chosen points exceeds K, the algorithm then utilizes the knowledge obtained from the first samples. Specifically, the distances[1] between all the points are calculated and the midpoint between each couple is estimated. The midpoint is defined as follows: assuming 2 points $p_1, p_2 \in D$, the midpoint p_{med} is the point whose values for each dimension equal to the average values of points p_1, p_2 to the respective dimensions. If such point does not exist (e.g. such resource combination is not applicable), the geometrically closest point is returned. The eventually picked midpoint is the result of the most distant points, as long as this point was not previously picked.

B. Approximation models

When a new sample is picked by the sampler and deployed, the performance metric for the deployment is stored and given as input to an approximation model. The training set of the model consists of the chosen samples along with their performance values. After the training process is finished, the model will be able to approximate the objective function for the entire space D.

There exist many methodologies for approximating an unknown function. We can categorize them in two major categories: regression based techniques and classification techniques. Algorithms on the former category create an analytical form of the objective function. The classification techniques, on the other hand, do not create an analytical function but rather classify the points of the domains space in classes; These objects are treated in a similar manner, indicating that the same properties stand for objects in the same class.

In our approach, we utilize the approximation models offered by *WEKA*[7], an open source data mining software which implements a variety of machine learning algorithms. WEKA provides a handful of approximation models including, but not limited to: (i) Multilayer Perceptron, that represents a typical neural network with many hidden layers and neurons, (ii) Linear Regression (Least Median Squares), that implements the methodology introduced at [8], (iii) RBF Network, which trains a Radial Basis Function Network, as presented at [9], (iv) Gaussian Process, that approximates the objective function using gaussian distributions, etc.

The accuracy of each of the models is highly affected from its configuration and the nature of the objective function. For example, a linear hyperplane will be approximated faster using a linear regression method; On the contrary a complex surface which has spikes and valleys is more likely to be approximated more accurately using a non linear approach.

[1]The points t_1 and t_2 represent points $p_1, p_2 \in D$ along with their respective performance values r_1, r_2, so $t_1 = (p_1, r_1)$ and $t_2 = (p_2, r_2)$. The norm $|t_1 - t_2|$ in this paper represents the difference $|r_1 - r_2|$, until otherwise stated.

All the available models are trained in parallel by the system, and the most accurate model is eventually picked.

C. System Architecture

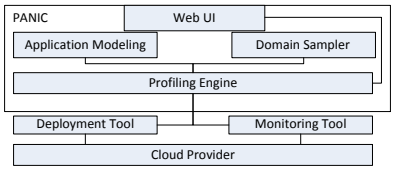


Figure 1: The architecture of PANIC

In Fig 1 we provide the architecture of the system. The core component of our system is the *profiling engine*. It is responsible for the synchronization between different tasks and it orchestrates the different components to achieve the common goal. Each time a new profiling loop is triggered, new application models and a new domain sampler is initialized (according to the user preferences). The sampler will initially create requests for new deployments and the Profiling Engine will forward the request to the Deployment Tool (a tool written for the needs of PANIC). When the execution of the application is terminated, the monitoring tool collects the user specified performance metrics and forwards them to the Profiling Engine. The monitoring system which is used by default is Ganglia[10]. The engine will then retrain the previously initialized Application Models and an accuracy estimation will occur. If the desired accuracy is achieved, then the profiling process is terminated, else a new profiling iteration occurs. The whole process is exported to the user through a Web UI.

IV. Experimental evaluation

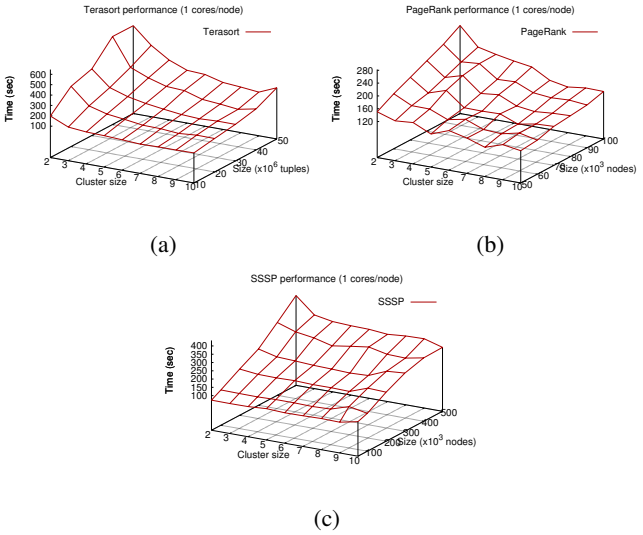
To evaluate the performance of *PANIC*, we have selected a set of distributed analytics jobs/applications that are naturally deployed over large scale virtualized resources. The first benchmark application is TeraSort [11], a well-known benchmark that sorts a set of key values. We test it with datasets of 10M up to 50M key-values (1GB to 5GB of data respectively) and run the TeraSort in Hadoop clusters with different number of nodes and different number of cores per node. The second application is a BSP-based implementation of PageRank [12], a well known graph algorithm implemented over the Apache Hama framework. We utilize 50K to 100K node graphs, each of which has at most 50 outgoing edges and execute PageRank over different cluster sizes as above. Finally, the third application is a BSP implementation of the Single Source Shortest Path (SSSP) algorithm [13], implemented for the Apache Hama framework. For SSSP, we create synthetic graphs consisting of 50k up to 500k vertices and at most 50 edges per node. For all the aforementioned algorithms, the performance metric we seek to predict is execution time.

The domain space for each of the three applications is composed of two dimensions related with the virtualized resources and one dimension related to the application load which, in our case, is intimately related to the size of the input dataset. The dimensions along with their respective values are provided in Table II. To evaluate the efficacy of *PANIC*, all three applications have been deployed for each possible combination. Consequently, the sampling algorithms presented in the previous section were applied and classifiers were trained, allowing us to measure the prediction accuracy.

Table II: Resource Dimensions

Dimension	Values	
Nodes	2, 3, 4, 5, 6, 7, 8, 9, 10	
Cores/node	1, 2, 4	
Dataset size	Terasort (Millions of Key Values)	10, 20, 30, 40, 50
	PageRank (Thousands of Nodes)	50, 60, 70, 80, 90, 100
	SSSP (Thousands of Nodes)	50, 100, 200, 300, 400, 500

A. Raw performance



(a) (b)

(c)

Figure 2: Raw performance

The running times for all three benchmark applications is given in Figure 2. We only provide the execution times of each benchmark application for the single core VM cases; The 2 and 4 cores cases are not provided due to space constraints. In Figure 2a we provide the execution time of the Terasort benchmark with regard to the size of the cluster and the dataset size (measured in millions of key-values). It is obvious that the execution time is inversely proportional to the cluster size and proportional to the dataset size. Furthermore, for large clusters we notice that the execution time decreases less rapidly, because the communication overheads affect more the overall execution time.

The execution time for both PageRank and SSSP are also shown in Figures 2b and 2c respectively. PageRank has a similar behavior to the Terasort case. SSSP, on the other hand, presents a slightly different behavior in terms of scalability. Specifically, when more nodes are added to the Hama cluster, the execution time remains unaffected for smaller dataset sizes (e.g., 50k nodes), while for larger datasets it decreases, but less rapidly than in the other cases. This is due to the larger number of supersteps executed by SSSP. Specifically, for our datasets, each SSSP job requires about 25–30 Hama supersteps while PageRank requires only a third of them. As a consequence, SSSP needs more sequential steps thus more time for synchronization between the BSP workers. Thus, due to this cost, the addition of more workers does not greatly benefit SSSP.

B. Sampling rate

One of the greatest factors that affect the performance of our system is the *sampling rate*. This is defined as the ratio between the number of the chosen points and the total number of acceptable deployments. Lower sampling rates lead to fewer chosen points, offering the classifiers less knowledge for the objective function (the performance of the application). Via the coefficient of determination R^2 [14] we quantify the accuracy of the profiling methods. R^2 declares the degree in which a classifier fits the original data. It is calculated as follows: $R^2 = 1 - \frac{\sum_i (y_i - f_i)^2}{\sum_i (y_i - \overline{y})^2}$ where y_i are the real performance values, f_i are the predicted values and $\overline{y}$ is the mean of the observed data. The closer R^2 gets to 1.0, the better the performed approximation. We also utilize the Mean Absolute Error [15] metric which is defined as: $MAE = \frac{1}{n} \sum_i |f_i - y_i|$.

We applied the sampling methodologies presented in Section III-A and trained all the available models of Section III-B with the chosen points along with the respective performance values for different sampling rates. In Figure 3 we provide the accuracy level of the best model for each sampling rate for all three applications; In Figures 3a, 3c, 3e, R^2 is depicted whereas the ones on the right (3b, 3d, 3f) represent the *MAE*. The best model is defined as the model that presents the highest coefficient of determination. The respective deviation for each application did not overtake 10% of the values of MAE and we do not depict it on the figures.

In our results, we notice that the most accurate models present slightly different behavior for each one of the three applications. In all of them, it is obvious that an increase in the Sampling Rate leads to higher accuracy. This result is expected, since higher sampling rate means that more points are picked, thus the model will obtain more knowledge for the objective function. However, in many cases this might not be the case: The sampler may pick more points but if

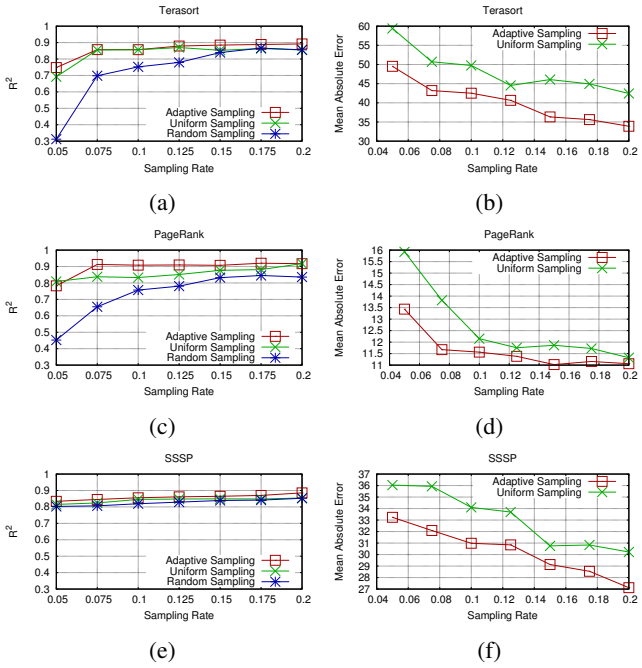


Figure 3: R^2 and MAE for the benchmark applications

they are not representative ones, they may mislead the model and eventually, this may cause lower accuracy. For example, this is the case for Terasort, when increasing the sampling rate from 0.125 to 0.15 for the Uniform sampler. More points are chosen, but very few of them are picked in the regions where the execution time is high, thus the model's accuracy degrades. Such cases are avoided by utilizing Adaptive Sampling.

In conclusion, the provided models in cooperation with the sampling methods enable the system to create an accurate profile of the application even when the sampling rate is less than 10% of the points of the domain space. In terms of accuracy, we achieved R^2 values higher than 0.8 for all the benchmark applications, even when the sampling rate is lower that 10%. Similarly, the accuracy of the trained models increased rapidly in terms of MAE as well for increasing sampling rates. Finally, the profiling process is quite fast: Even when the Sampling Rate is 20%, the total time spent in training is not more than 1.5 seconds. The input space of our experiments consists of 135 discrete points for the Terasort case and 162 points for the SSSP and Pagerank cases. Sampling with 20% of these domains leads to 27 and 32 points respectively, thus the training time of our models is less than 1.5 seconds when there exist 32 points for training.

V. Related Work

Predicting the performance of applications running over virtualized resources concerning the workload is vividly researched in the literature. In [4], Kundu et al. proposed an iterative model training technique for Neural Networks

with which the authors managed to predict the minimum possible Virtual Machine (concerning its resources) which would fulfill their objectives with respect to the SLAs. In an extension of this work, at [3], also utilized Support Vector Machines for the same objective. Their work achieved to highly accurate predictions, however the authors did not address the problem of sampling the input domain space, as we do in this work. Furthermore, Iqbal et al. in [16], propose a method with which, at first, identifies a workload pattern and secondarily builds a model capable to predict the application's capacity (the number of requests it can serve without violating given constraints). This work focuses on web applications and the prediction happens with regression models; *PANIC* on the other hand, provides a wealth of approximation techniques and the it also supports any application able to deployed over a cloud infrastructure.

Similarly, Do et al. in [6] presented a profiling technique which utilizes the Canonical Correlation Analysis, able to identify the relationship between the allocated resources and the application performance. This work targets to predict the performance of a newly allocated Virtual Machine when it is deployed in a specific host running other Virtual Machines. Our work differentiates from this, since our target is to provide an accurate application profile without having any knowledge about the provider. Other works focus on predicting specific application metrics based on I/O workload and access patterns such as [17], [18] and [19]. Our approach differentiates from them, as we propose a system where the user can define application level metrics which indicate the application performance.

VI. Conclusions

In this paper we addressed the problem of predicting the performance of a complex application deployed over virtualized resources. The goal of our work is to propose a system which obtains knowledge about the application by deploying it over a cloud infrastructure, in different deployment setups and then approximating its performance for all the possible setups. The application load, which is a key factor to the performance, is addressed in the same manner as the rest of the resources, contributing to a unified view of all the components that affect the application. The experimental evaluation indicated that such an approach can lead to an accurate prediction of the performance by actually deploying the application for only a very small portion of the deployment space. Furthermore, by utilizing a large number of approximation techniques, our system is able to quickly recognize the behavior of the application by picking the most suitable approximation model enabling the profiling process to terminate faster.

Acknowledgments

This work was supported by the European Commission in terms of the CELAR 317790 FP7 project (FP7-ICT-2011-8). Nikolaos Papailiou has received funding from IKY fellowships of excellence for postgraduate studies in Greece - SIEMENS program.

References

[1] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud computingthe business perspective," *Decision Support Systems*, vol. 51, no. 1, pp. 176–189, 2011.

[2] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. IEEE, 2011, pp. 363–366.

[3] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta, "Modeling virtualized applications using machine learning techniques," *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 3–14, 2012.

[4] S. Kundu, R. Rangaswami, K. Dutta, and M. Zhao, "Application performance modeling in a virtualized environment," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010, pp. 1–10.

[5] A. A. Bankole and S. A. Ajila, "Predicting cloud resource provisioning using machine learning techniques," in *Electrical and Computer Engineering (CCECE), 2013 26th Annual IEEE Canadian Conference on*. IEEE, 2013, pp. 1–4.

[6] A. V. Do, J. Chen, C. Wang, Y. C. Lee, A. Y. Zomaya, and B. B. Zhou, "Profiling applications for virtual machine placement in clouds," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 660–667.

[7] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[8] P. J. Rousseeuw and A. M. Leroy, *Robust regression and outlier detection*, 1987.

[9] D. S. Broomhead and D. Lowe, "Radial basis functions, multi-variable functional interpolation and adaptive networks," DTIC Document, Tech. Rep., 1988.

[10] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[11] O. OMalley, "Terabyte sort on apache hadoop," *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)*, pp. 1–3, 2008.

[12] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." 1999.

[13] S. Pettie, "Single-source shortest paths," *Encyclopedia of Algorithms*, pp. 847–849, 2008.

[14] R. G. D. Steel and J. H. Torrie, "Principles and procedures of statistics: with special reference to the biological sciences," 1960.

[15] R. J. Hyndman and A. B. Koehler, "Another look at measures of forecast accuracy," *International journal of forecasting*, vol. 22, no. 4, pp. 679–688, 2006.

[16] W. Iqbal, M. N. Dailey, and D. Carrera, "Black-box approach to capacity identification for multi-tier applications hosted on virtualized platforms," in *Cloud and Service Computing (CSC), 2011 International Conference on*. IEEE, 2011, pp. 111–117.

[17] Q. Noorshams, D. Bruhn, S. Kounev, and R. Reussner, "Predictive performance modeling of virtualized storage systems using optimized statistical regression techniques," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, 2013, pp. 283–294.

[18] S. Kraft, G. Casale, D. Krishnamurthy, D. Greer, and P. Kilpatrick, "Io performance prediction in consolidated virtualized environments," in *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 5. ACM, 2011, pp. 295–306.

[19] Kraft, Stephan and Casale, Giuliano and Krishnamurthy, Diwakar and Greer, Des and Kilpatrick, Peter, "Performance models of storage contention in cloud environments," *Software & Systems Modeling*, vol. 12, no. 4, pp. 681–704, 2013.