Mix 'n' Match Multi-Engine Analytics

Katerina Doka*, Nikolaos Papailiou*, Victor Giannakouris*, Dimitrios Tsoumakos§ and Nectarios Koziris*
*National Technical University of Athens, Greece, {katerina,npapa,vgian,nkoziris}@cslab.ece.ntua.gr
§Ionian University, Greece, dtsouma@ionio.gr

***Abstract*—Current platforms fail to efficiently cope with the data and task heterogeneity of modern analytics workflows due to their adhesion to a single data and/or compute model. As a remedy, we present *IReS*, the *Intelligent Resource Scheduler* for complex analytics workflows executed over multi-engine environments. IReS is able to optimize a workflow with respect to a user-defined policy relying on cost and performance models of the required tasks over the available platforms. This optimization consists in allocating distinct workflow parts to the most advantageous execution and/or storage engine among the available ones and deciding on the exact amount of resources provisioned. Our current prototype supports 5 compute and 3 data engines, yet new ones can effortlessly be added to IReS by virtue of its engine-agnostic mechanisms. Our extensive experimental evaluation confirms that IReS speeds up diverse and realistic workflows by up to 30% compared to their optimal single-engine plan by automatically scattering parts of them to different execution engines and datastores. Its optimizer incurs only marginal overhead to the workflow execution performance, managing to discover the optimal execution plan within a few seconds, even for large-scale workflow instances.**

I. Introduction

Big Data analytics have become indispensable to organizations worldwide as a means of extracting significant value out of the enormous amounts of data that stream into their businesses. That, in turn, offers organizations an unprecedented competitive advantage: The ability to identify new opportunities, take educated decisions based on historical facts, render their operations faster and more cost efficient and keep customers satisfied [1]. The volume, velocity and variety of Big Data pose new challenges to analytics, entailing a high degree of parallelism in both storage and computation: Modern datacenters host huge volumes of data over large numbers of nodes with multiple storage devices and process them using thousands or millions of cores.

In the landscape of Big Data analytics, multiple and diverse execution engines and datastores have emerged as platforms of choice for specific computation types and data formats (e.g., [2], [3], [4], etc.). To alleviate the burden of building and maintaining such systems, many of them are currently either offered as-a-service by the most prevalent Cloud providers (e.g., [5], [6], [7]) or packaged in precooked VM or container images for ease of deployment [8]. Still, although many approaches in the relevant literature manage to optimize the performance of single engines by automatically tuning a number of configuration parameters [9], [10], they bind their efficacy to specific data formats and query/analytics task types.

However, one size does not fit all: No single execution model is suitable for all types of tasks and no single data model is suitable for all types of data. Indeed, modern workflows have evolved into increasingly long and complex series of diverse operators, ranging from simple Select-Project-Join (SPJ) and data movement to complex NLP-, graph- or custom business-related tasks, with varying data formats (e.g., relational, key-value, graph, etc.) and shrinking delivery deadlines [11]. Time constraints aside, analysts may be equally interested in other execution aspects, such as cost, resource utilization, fault-tolerance, etc., and thus need to be able to impose various – and often multi-objective – optimization policies, adding another degree of complexity to an already convoluted problem.

Multi-engine analytics have recently been proposed as a promising solution that can optimize for this complexity [12] and are gaining ground ever since. Cloud vendors currently offer software solutions that incorporate a multitude of processing frameworks, data stores and libraries to facilitate the management of multiple installations and configurations [13], [14], [15]. One of the most compelling, yet daunting challenges in such a multi-engine environment is the design and creation of a *meta-scheduler* that automatically allocates tasks to the right engine(s) according to multiple criteria, deploys and runs them without manual intervention.

Recent works along this line are either proprietary tools with limited applicability and extension possibilities for the community (e.g., [16]) or focus more on the translation of scripts from one engine to another, being thus tied to specific programming languages and engines (e.g., [17], [18]). Contrarily, we would ideally opt for an open-source, engine-agnostic solution that could easily be extended to new engines and implementation languages.

To that end, we present *IReS*, an open-source *Intelligent Multi-Engine Resource Scheduler* that integrates multiple execution engines and datastores into the optimizing, planning and execution of complex analytics workflows[1]. IReS adops a black-box approach on the analytics operators. This facilitates the handling of any kind of task, ranging from low- (e.g., join, sort, etc.) to higher-level operators (e.g., machine learning, graph processing, etc.) that run on any state-of-the-art, centralized or distributed system (e.g., Map-Reduce, BSP, RDBMSs, NoSQL, distributed file-systems,

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

etc.). Moreover, the engine-agnostic approach allows for easy addition of new operators and engines. All that IReS requires is a description of the analytics tasks and data via an extensible meta-data framework, as well as a model of the cost and performance characteristics of the required tasks over the available platforms. Consequently, utilizing a DP-based, state-of-the-art planner, the platform is able to map distinct parts of a workflow to the most advantageous store, indexing and execution pattern and decide on the exact amount of resources provisioned in order to optimize any user-defined policy. The resulting optimization is orthogonal to (and in fact enhanced by) any optimization effort within an engine. In this paper we thoroughly describe the architecture of IReS and delve into the design and implementation details of its inner modules. Our key contributions are:

►A multi-engine planner that selects the most prominent workflow execution plan among existing engines, datastores and operators and elastically provisions the correct amount of resources, consulting the cost and performance models of the various operators.

►A modeling methodology that provides performance and cost metrics of the available analytics operators for different engine configurations. These metrics are collected from actual executions of the operators both offline (training phase) and online (refinement phase). The resulting models are utilized in multi-engine workflow optimization.

►An extensible meta-data description framework for operators and data, which allows IReS to automatically discover all alternative execution paths of an abstractly described workflow by matching operators that perform similar tasks.

►An extensive evaluation of our open-source prototype operating over various real-life and synthetic workflows chosen to include diverse datasets and computation types under realistic conditions. The results attest the ability of IReS to efficiently decide on the optimal execution plan based on the optimization policy and the available engines within a few seconds, even for large-scale workflow graphs, adapt to changes in the underlying infrastructure and temporal degradations with minimal overhead and, most importantly, speed-up the fastest single-engine workflow executions up to 30% by exploiting multiple engines.

II. IReS Architecture

IReS focuses on the highly efficient and user-customizable execution of analytics workflows. This is made possible through the transparent modeling, monitoring and scheduling that involves different execution engines and storage technologies. Our system is able to handle all types of analytics workflows by adaptively choosing to execute each subpart in a (possibly different) deployed engine. IReS assigns sub-tasks to the most advantageous technology available and ensures resource and dataflow scheduling in order to enhance performance: If a single engine is used, enhancement will be achieved through optimized and elastic resource allocation

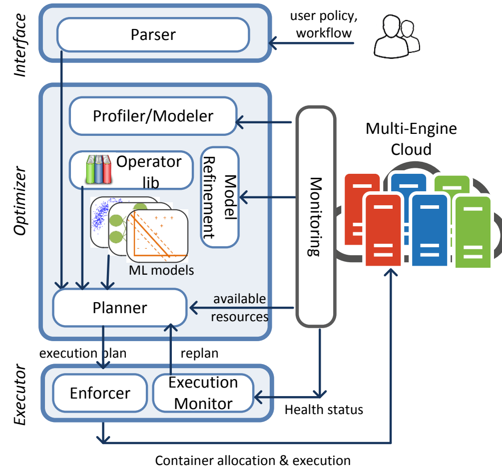

Figure 1: Architecture of the IReS platform

(e.g., execute on the right cluster size, etc.); if multiple ones are required, enhancements will relate both to single-engine optimization and to workflow management that decides on the best execution plan and data placement (e.g., first execute subtask A in Spark, store intermediate results in a NoSQL engine and then run subtasks B and C in parallel, having the final results written in HDFS).

The central notion behind IReS is to utilize detailed models of the costs and performance characteristics of analytics operators over multiple execution engines. The models are stored and updated in an IReS library. Whenever a new workflow is run atop IReS, these models are used in order to intelligently assign and orchestrate workflow parts to the underlying engines according to the user optimization policy. The architecture of the IReS platform is depicted in Figure 1. IReS comprises of three layers, the *interface*, the *optimizer* and the *executor* layer. In the following, we describe in more detail the role, functionality and internals of these layers, delving into the specifics of the most important modules.

A. Interface Layer

The interface layer is responsible for handling the interaction between IReS and its users. A user should be able to accurately define execution artefacts such as operators, data, workflows, etc., along with their inter-dependencies, properties and restrictions using a common meta-data description framework. Based on this framework, the *parser* module is parses the user-provided workflow as a dependency graph and validates the user-defined policy.

The main challenges of defining such a framework are *extensibility* and *abstraction*. Users should be granted the ability to define custom meta-data for fine-grained operator and dataset description. This freedom supports the effortless addition of new engines and operators, as opposed to the rigidity of having a predefined set of meta-data fields. Moreover, users should be able to specify the data and operators that compose their workflow at any desired abstraction

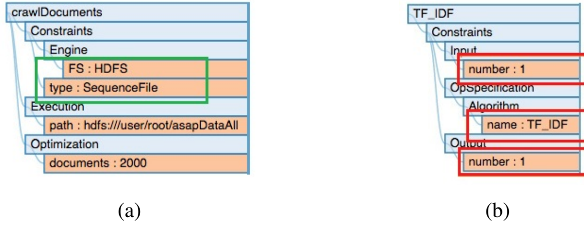

Figure 2: Meta-data descriptions of (a) a dataset of crawled web pages and (b) an abstract tf-idf operator.

level on its various steps, ranging from the fine-grained definition of specific implementations/engines to the coarse-grained description of the general functionality regardless of the platform. It is IReS that will remove this abstraction, examine alternative execution paths of the same conceptual workflow and select the most beneficial one, according to the user-defined policy.

The main entities of our framework are *data* and *operators*, which need to be accompanied by a set of meta-data, i.e., properties that describe them. Data and operators can be either *abstract* or *materialized*. Abstract operators and datasets are defined and used when composing a workflow, whereas materialized ones refer to specific implementations and existing datasets and are usually provided by the operator developer or the dataset owner respectively. Materialized operators along with their descriptions are stored in the *operator library*, as depicted in Figure 1.

The meta-data accompanying operators (e.g., input types, execution parameters, invocation scripts, etc.) and data (e.g., schemata, location of objects, etc.) are organized in a generic tree format. To avoid restricting the user and allow for flexibility, only the first levels of the meta-data tree are pre-defined. Users can add their ad-hoc subtrees to define custom data or operator properties. 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* 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. 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). In general, we pre-define the following the meta-data fields:

Constraints: This sub-tree contains all the information that is required to match (a) abstract operators to materialized ones and (b) data to operators. Mandatory fields include specifications of operator inputs/outputs, algorithms, engines and anything considered necessary in the abstract/materialized matching of operators.

Execution: This sub-tree provides the execution parameters of a materialized operator, such as the path of a dataset or the execution arguments of an operator script.

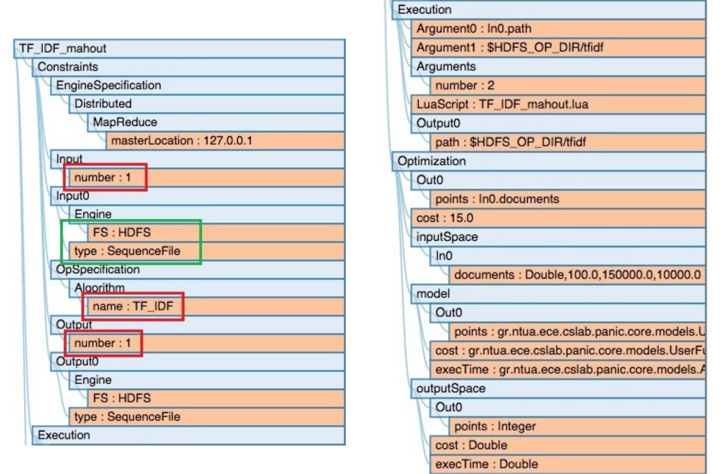

Figure 3: Meta-data description of a materialized tf-idf operator, implemented in mahout/Hadoop

Optimization: This optional part of the meta-data holds additional information that assists in the optimization of the workflow. This information could include, for instance, a cost function provided by the developer of the operator or instructions on how to create one by profiling over specific metrics (e.g., execution time, required RAM, etc.).

As an example, let us assume an analyst wants to perform tf-idf over a corpus of documents crawled from the Internet. First, she needs to describe the input dataset, `crawlDocuments`, as depicted in Figure 2.a: It is a sequence file stored in HDFS, following the path specified by the `Execution` field. The information under `Optimization` notifies the system of the number of documents contained in the dataset. Then, she needs to specify the operation to be performed. In its abstract form, the `TF_IDF` operator (see Figure 2.b) needs only define one input parameter, the implemented algorithm (under `opSpecification.Algorithm`) and an output parameter. In short, `TF_IDF` defines a format that any tf-idf implementation of the specific functionality needs to follow.

Additionally, a materialized tf-idf operator includes all information required in order to perform the operation on an execution engine. In `TF_IDF_mahout` (see Figure 3), the operator calculates tf-idf over Mahout/Hadoop; it thus includes Hadoop-specific information about the input, output and the engine. The input and output in this case have specific types and an engine specification (under `Engine`). The operator itself also has an `EngineSpecification`, indicating its execution location.

To discover the actual implementations that comply with the description of both the abstract operator and the dataset provided by the user, we employ a tree matching algorithm to ensure that all meta-data constraints are met, i.e., all compulsory fields are consistent. This is performed during the planning and optimization phase, described subsequently. In our example, `TF_IDF_mahout` matches `TF_IDF` in

the fields designated by the red rectangles. Moreover, the `crawlDocuments` dataset can be used as input to `TF_IDF_mahout` as is, as the matched greed rectangles suggest. Thus, `TF_IDF_mahout` is considered when constructing the optimized execution plan.

B. Optimizer Layer

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 this layer is the *planner*, 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 and whether data need to be moved to/from their current locations and between runtimes (if more than one is chosen).

Such a decision must rely on the characteristics of the analytics task in hand which are modeled and stored within IReS. The initial model of an operator results from the offline profiling of it using a *profiler* that directly interacts with the pool of physical resources and the monitoring layer in-between. Moreover, 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. This mechanism allows for dynamic adjustments of the models and enables the planner to base its decisions on the most up-to-date knowledge.

►**Profiler/Modeler**: While accurate models exist for SQL operations over an RDBMS, which includes its own cost-based optimizer, this is not the case for other analytics operators (e.g., machine learning, graph processing, etc.) and modern runtimes (be it distributed or centralized): Only a very limited number of operators and engines has been studied, while most of the proposed models entail knowledge of the code to be executed [19], [20], [21]. Moreover, there is no trivial way to compare or correlate cost estimations derived from different engines at a meta-level.

To that end, we adopt an engine-agnostic approach that treats materialized operators as “black boxes”, assuming no prior knowledge of their internals, and models them using profiling in an offline mode, as well as machine learning over actual runs.

The profiling mechanism adopted builds on prior work [22]. Its input parameters fall into three categories: (a) *data specific*, which describe the data to be used for the operator profiling (e.g., the type of data and its size), (b) *operator specific*, which relate to the algorithm of the operator (e.g., the number of output clusters in k-means), and (c) *resource specific*, which define the resources to be tweaked during profiling (e.g., cluster size, main memory, etc.)

The output of each run is the profiled operator’s performance and cost (e.g., completion time, I/O operations, average memory, 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 specified by the user/developer. The collected metrics are then used to create estimation models [23], making use of neural networks, SVM, interpolation and curve fitting techniques for each operator running on a specific engine. The cross validation technique [24] is used to maintain the model that best fits the available data.

►**Model Refinement** Upon execution of a workflow, the currently monitored execution metrics provide feedback to the existing models in order to refine them and capture possible changes in the underlying infrastructure (e.g., hardware upgrades) or temporal degradations (e.g., due to unbalanced use of engines, collocation of competing tasks, surges in load etc.). This mechanism contributes to the adaptability of IReS, ameliorating the accuracy of the models while the platform is in operation.

►**Planner** This module, in analogy to traditional query planners, intelligently explores all the available execution plans and discovers the optimal one with respect to the user-defined optimization objectives. Algorithm 1 describes the optimization process, which relies on dynamic programming (DP) to select the optimal execution plan.

The algorithm receives as input the abstract workflow graph, expressed as a DAG of operator and dataset nodes *G(Datasets,Operators)*. It maintains a $dpTable$ structure, responsible for storing the best execution plan for each different format of a dataset node (e.g., csv, json, etc.). The planner processes all abstract operators of the workflow following a DAG topological order, using a depth-first search (line 11). This ordering ensures that when an operator is being processed, all its predecessors in the DAG have already been processed and thus the $dpTable$ always contains the optimal plans per input.

For each abstract operator, the IReS library is explored to find all matching materialized operators, i.e., operators that share the same meta-data (line 12). To speedup this procedure we use string labelled and lexicographically ordered meta-data trees. This data structure allows for efficient, one pass tree matching. The complexity of matching two meta-data trees with up to t nodes is $O(t)$. We further improve the matching procedure by indexing the IReS library operators using a set of highly selective meta-data attributes (e.g., algorithm name). Only operators that contain the correct attributes are considered as candidate matches and are further examined by the above algorithm.

When all operator matches have been discovered, the process consults the input and output specifications of the materialized operators and adds the required move/transform operators (lines 22-25). Those operators are needed in order to connect operators of different engines and input/output configurations. Here, we make the assumption that operator alternatives have a 1-1 relationship (we do not yet consider the possibility of one operator being equivalent to a combination of 2 or more operators) and that only

ALGORITHM 1: *Optimizer*

1 //$G(Datasets, Operators)$: abstract workflow graph
2 //$Datasets$: set of datasets
3 //$Operators$: set of abstract operators
4 //$target$: target dataset
5 **for** $d \in Datasets$ **do**
6   //initialize dpTable
7   **if** $d.isMaterialized()$ **then**
8   **if** $d == target$ **then**
9   **return** 0;
10         $dpTable[d].insert(d, 0)$;
11 **for** $o \in Operators$ *following DAG topological ordering* **do**
12     $MOperators = findMaterializedOperators(o)$;
13   **for** $mo \in MOperators$ **do**
14         $inputCost = 0$;
15   **for** $in \in mo.getInputs()$ **do**
16             $minCost = \infty$;
17   **for** $tin \in dpTable[in]$ **do**
18   **if** $tin.matchWithOperatorInput(mo)$ **then**
19   **if** $tin.getCost < minCost$ **then**
20                         $minCost = tin.getCost$;
21   **else**
22   **if** $tin.checkMove(mo)$ **then**
23                         $moveCost = tin.getCost + tin.moveCost(mo)$;
24   **if** $moveCost < minCost$ **then**
25                             $minCost = moveCost$;
26             $inputCost += minCost$;
27         $operatorCost = estimateCost(mo)$;
28         $cost = inputCost + operatorCost$;
29   **for** $out \in o.getOutputs()$ **do**
30             $tout = outputFor(mo, out)$;
31             $dpTable[out].insert(tout, cost)$;
32 **return** $dpTable[target].getMinCost()$;

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 estimator models for each one of the materialized operators (line 27). 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.

To study the complexity of the *Optimizer* algorithm, let us assume that a workflow contains op number of abstract operators, with at most m materialized operators matching an abstract one. Moreover, let us assume that each operator has k inputs at maximum. For each intermediate dataset, our $dpTable$ will contain at most m records, each generated from one of the m materialized operators that match the abstract one that produces it. Therefore, the inner loop of

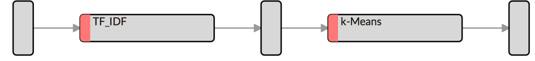

Figure 4: Abstract tf-idf, k-means workflow.

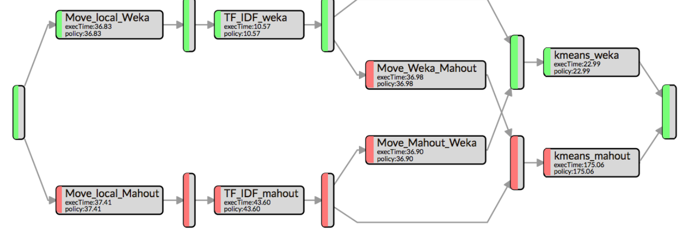

Figure 5: Materialized workflow and optimal plan.

Algorithm 1 (line 17 onwards) will run at most m times. Thus, the worst case complexity of our optimizer is:

$$\mathcal{O}(op \cdot m^2 \cdot k)$$

Figure 4 depicts an abstract workflow which performs tf-idf feature-extraction over a corpus of documents and clusters the output using the k-means clustering algorithm. Assuming each operator has 2 implementations, using either the mahout or WEKA libraries (running in Hadoop and Java respectively) we have the possible alternative execution plans of Figure 5. The planner automatically adds the necessary `move/transform` operators in order to transfer intermediate results between the two engines (i.e., match the output of an operator to the input of the subsequent one).

Let us assume an optimization policy that targets execution time minimization. Intuitively, small datasets run faster in a centralized manner while distributed implementations outperform the centralized ones for bigger datasets. Indeed, the WEKA implementation is estimated to be the fastest for both steps, due to the small input size and is thus included in the selected execution path, marked in green.

►**Resource Provisioning** Apart from deciding on the specific implementation/engine of each workflow operator, the planner of IReS provisions the correct amount of resources to execute the workflow conforming as much as possible to the user-defined optimization policy. This policy may involve the execution time or any user-defined cost function. The resource provisioning process builds on the MOEA framework [25] and relies on the NSGA-II genetic algorithm [26] to supply resource-related parameters (e.g., #cores, memory) from the local minima of the trained models. NSGA-II is the most prevalent evolutionary algorithm that has become the standard approach to generating Pareto optimal solutions to a multi-objective optimization problem. The estimated parameter values are passed as arguments to the workflow execution during run-time.

C. Executor Layer

The *executor layer* is the layer that enforces the optimal plan over the physical infrastructure. Its main responsibilities

include the execution of the ensuing plan, a task undertaken by the *enforcer*, and the assurance of the platform's robustness, carried out by the *execution monitor*.

The enforcer adopts 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 primitives as understood by the specific runtimes and storage engines. Such actions might entail code and/or data shipment.

Our current working prototype relies on YARN [27], a cluster management tool that enables fine-grained, container-level resource allocation and scheduling over various processing frameworks. Apart from requesting from YARN the necessary container resources for each workflow operator, the enforcer needs to pay special attention to the workflow execution orchestration. To that end, IReS extends Cloudera Kitten [28], a set of tools for configuring and launching YARN containers as well as running applications inside them, in order to add support for the execution of a DAG of operators instead of just one.

The execution monitor captures faults and failures occurring on-the-fly through real-time monitoring. Thus, it ensures the robustness and availability of the system by employing two mechanisms:

- A mechanism that monitors the health status of the underlying infrastructure by periodically executing customizable and parametrized health scripts in all cluster nodes. The health status (HEALTHY/UNHEALTHY state per cluster node) is reported back to the IReS server.
- A mechanism that checks the availability of all services (i.e., engines and datastores) needed for the enforcement of an execution plan (ON/OFF status).

This information is used during the phases of both planning and execution of a workflow. During planning, unavailable engines are excluded when constructing the optimal execution plan and resources are provisioned exclusively taking into account the currently available ones. During the execution of a workflow, failures are detected in real-time. The remaining workflow is re-planned and the new plan is enforced. We should note here that our system does not discard results of tasks that have been successfully executed. Contrarily, it takes advantage of any intermediate materialized data, effectively reducing the part of the workflow that needs to be re-scheduled.

III. Experimental Evaluation

In this section we experimentally evaluate IReS to showcase its ability to optimize the execution of an analytics workflow with respect to a user-defined policy by mapping parts of it to the most beneficial compute or data engines. Apart from the gains in workflow performance, which constitute the intuition that inspired IReS, the experiments aim to prove that the overhead of the IReS decision making process is affordable, the resource provisioning strategy caters for the user needs and the system improves its accuracy as it operates, being adaptable to any short- or long term change in the characteristics of the supported engines.

Our system prototype has been implemented in Java and is open-source. In our experiments, IReS controls a cloud-based deployment of several runtime engines and data stores[2] over 16 virtual machines of an Openstack cluster hosted in our lab. All the supported engines have been tuned according to best practices.

Throughout the experiments we make use of three workflows, one of each of the three categories which we consider as the most representative of modern, real-life workflows, namely *text analytics*, *graph analytics* and *relational analytics*. Two of them are driven by real business needs and have been specified in the context of the eu-funded ASAP project[3]. 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). More precisely:

Graph analytics: The workflow involves the processing of anonymized call detail records (CDR), residing in HDFS, to calculate the influence score of a subscriber on a telecommunications network. This is achieved by treating CDR data as a graph, where each customer (i.e., phone number) represents a vertex and each call corresponds to an edge, and applying Pagerank over them. Pagerank has been implemented in Spark, Hama and Java.

Text analytics: The workflow starts by performing tf-idf on web content that resides in HDFS; the outputs are then clustered using k-means. Both operators are chosen between scikit and MLlib running centrally or over Spark respectively.

Relational analytics: The workflow contains 3 synthetic SQL queries (Figure 6.d) which join tables residing in different stores. For this workflow, we use data produced by the popular TPC-H [29] benchmark generator. We make the assumption that the small tables containing legacy data (customer, nation, region) are stored in PostgreSQL, the medium ones (part, partsupp) in MemSQL, taking advantage of the collective memory of the cluster and the large ones (lineitem, orders) in HDFS, since their size can not be accommodated by any of the former.

A. Efficiency of Workflow Execution Plan

In this set of experiments, assuming the optimization objective of minimizing execution time, we plan and execute all three test workflows in a multi-engine environment using IReS and plot the execution time of the chosen plan for

[2]Hadoop 2.7.0, Spark 1.6.0, Hama 0.7.1, scikit-learn 0.17.1, MemSQL 5.0, Postgres 9.5.3

[3]ASAP (Adaptive, highly Scalable Analytics Platform) envisions a unified execution framework for scalable data analytics. www.asap-fp7.eu/

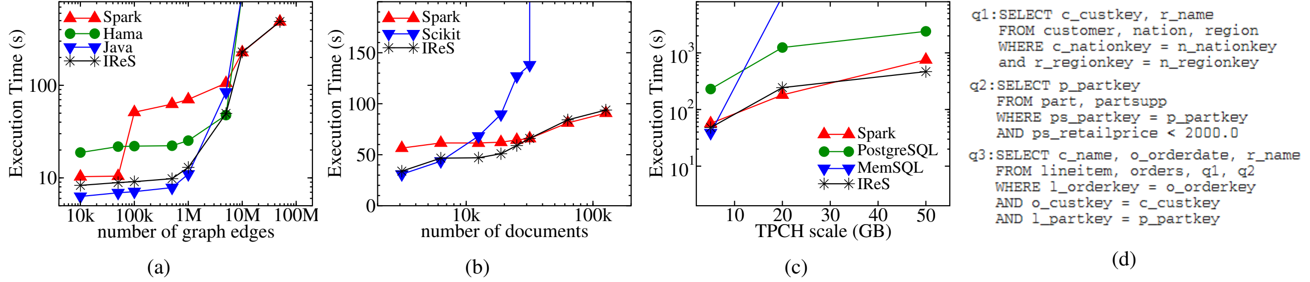

Figure 6: Execution times for the (a) graph, (b) text and (c) relational analytics workflows vs. various input sizes when running on single- and multi-engine (through IReS) environments. (d) The sql query of the relational analytics workflow.

various sizes of the input dataset. These measurements are compared against the time required to run each workflow in its entirety using exclusively a single engine. The goal is to confirm that the execution plan chosen by IReS is at least as efficient as the fastest single-engine choice (with some small overhead) and can in fact speed up the single-engine execution combining different engines in the same plan.

Figure 6.a depicts the execution times of the graph analytics workflow (which consists of a single operator, i.e., pagerank) when run in Java, Hama and Spark as well as the execution times of the plan adaptively selected by IReS for each input size. As expected, a centralized, Java-based, implementation outperforms its alternatives for small-scale graphs. However, this approach fails as the input size grows larger than the available main-memory of a single node. In contrast, a distributed, Spark-based implementation incurs overheads for small graphs but proves scalable when handling larger input sizes. The Hama-based implementation, which relies on a distributed main-memory execution model, proves better for medium scale datasets that can fit in the aggregate cluster memory but also fails for larger graph sizes. We observe that IReS successfully chooses the most efficient operator implementation for each input dataset size. Furthermore, the IReS workflow optimization and YARN-based execution incur a small overhead of a couple of seconds. This overhead is visible for small input sizes but is alleviated for longer running operators.

Figure 6.b refers to the text analytics workflow, proving that the centralized scikit implementation achieves better performance than Spark only for small datasets (less than 10K documents in our case). Using trained cost estimators, IReS selects the proper engines for executing the workflow, depending on the input data size. We also note that IReS performs hybrid executions by combining operators of different engines for a range of input sizes. Indeed, from 10k to about 40k documents IReS maps tf-idf to scikit and k-means to Spark and manages to outperform even the fastest single-engine execution by up to 30%. In these cases, IReS automatically inserts the required move/transform operators.

Figure 6.c depicts the execution performance of the relational analytics workflow. While PostgreSQL can provably

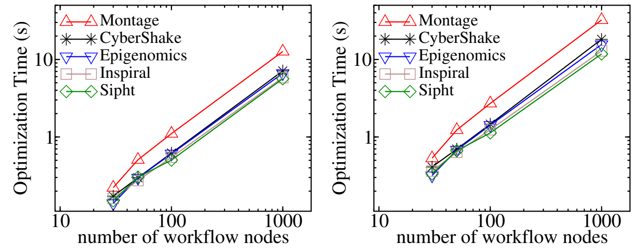

Figure 7: Workflow optimization times for 4 and 8 engines, using various workflow types of ranging size.

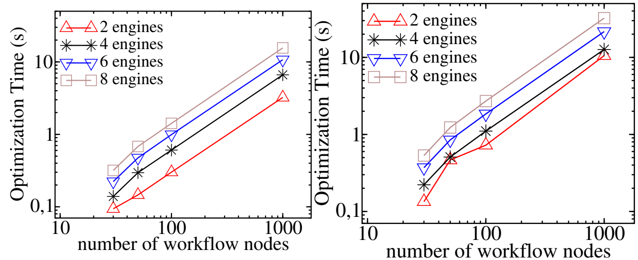

Figure 8: Workflow optimization times for Montage and Epigenomics graphs, using various number of engines and ranging the workflow size.

perform well for small datasets, the cost of data transfer from other engines is prohibitive. MemSQL fails to execute the workflow for sizes larger than 2GB due to intermediate results exceeding the available cluster memory. IReS executes each workflow query in the engine where its tables reside (q_1 in PostgreSQL, q_2 in MemSQL and q_3 in Spark), minimizing the required data movements and thus achieving a constantly good performance, regardless of the data size. In fact, the workflow execution starts to accelerate as the dataset scales to larger sizes (50G), for which the planning and movement overhead is amortized by the pure task execution speed-up.

B. Workflow Planner Performance

In this section we experimentally evaluate the performance of our multi-engine workflow planner with respect to the workflow complexity and the number of alternative implementations of a workflow operator. To provide a reproducible experimental set-up and comparable results we use the Pegasus workflow generator [30]. The produced workflow graphs fall into five scientific workflow categories (i.e.,

Montage, CyberShake, Epigenomics, Inspiral and Sipht) and contain patterns derived from diverse scientific application domains such as astronomy, biology, gravitational physics and earthquake science. They include massively parallel workflows that process large amounts of data, pipelined applications that split up input datasets and operate on different chunks in parallel as well as workflows that have a relatively fixed structure and perform identical analyses on multiple input datasets.

Figure 7 depicts the time required by our planner to optimize all five Pegasus workflow categories. In this experiment we range both the number of the workflow nodes and the number of alternative execution engines (denoted as m in our evaluation of the planner's complexity). The first graph of Figure 7 plots the planner's execution time for 4 engines, while the second for 8 engines, i.e., the IReS operator library contains 4 and 8 alternative implementations of each of the abstract workflow operators respectively. While most of the Pegasus graphs show similar behaviour, the Montage workflow graph is more connected, having multiple nodes with high in- and out-degrees. This results in a $2\times$ increase in planning times, which is theoretically confirmed by our planner's algorithmic complexity ($\mathcal{O}(op \cdot m^2 \cdot k)$). Indeed, performance is linearly affected only by the number of inputs k of each operator. We also note that our planner demonstrates almost linear complexity when ranging the number of workflow nodes between 30 and 1000. In the extreme case of 1000-node workflows the time required to produce the optimal execution plan is less than 10 seconds in all runs. This allows us to expect that the IReS planner can handle even the most complex multi-engine workflow scenarios with an almost negligible overhead compared to the total execution time of the analytics workflow itself.

To further test the impact of the number of available engines on the workflow planning performance, we measure the time required to optimize and plan the Montage and Epigenomics workflows, which we consider the most representative ones based on the previous experiment, while ranging the number of alternative execution engines for *each* workflow operator between 2 and 8 (Figure 8).

As expected, the existence of multiple operator implementations affects the performance of the planning process. However, the IReS planner manages to handle even the extreme cases of 100-node workflows with up to 8 engines within a couple of seconds. The majority of real-life workflows though are far from being that abundant, as our experience in the ASAP project also suggests. An average 10-node workflow, even under the immoderate assumption of 8 alternative operator implementations, can be optimized and scheduled for execution with IReS in the sub-second time-scale. This also holds for all of the real-life workflows utilized throughout this section, which require planning times in the order of milliseconds.

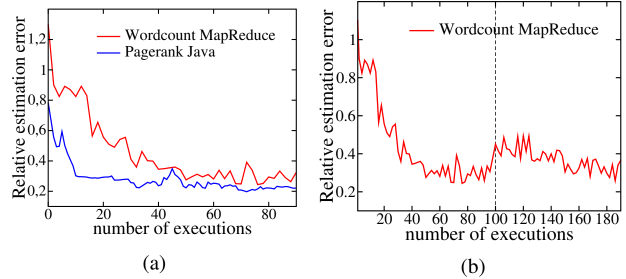

Figure 9: Relative execution time estimation error w.r.t. the number of executions (a) in normal IReS operation (b) when an infrastructure change occurs after 100 executions.

C. Operator Modeling

In this section, we test the ability of IReS to accurately estimate the cost and performance of various operators as well as its adaptability to changes in operator characteristics due to temporal degradation or infrastructural modifications. In this set of experiments we run single-operator workflows, for the sake of simplicity. Apart from the `Pagerank` operator, we introduce, from the field of text analytics, an operator that counts distinct words in a corpus of documents - `Wordcount`. Figure 9.a depicts the relative performance estimation error achieved for `Wordcount` over MapReduce and `Pagerank` using a centralized Java implementation. We iteratively execute the operators with different input sizes, number of resources (i.e., CPUs, RAM) and application specific parameters (i.e., number of iterations), uniformly selecting from a set of possible setups. The models are refined with each operator execution. In the beginning of the experiment there is no knowledge of the operator performance and therefore the models present high estimation errors. However, in both cases the relative execution time estimation error drops bellow 30% after only 50 runs. The accuracy of IReS keeps on improving smoothly after that, as more sample execution points are gathered.

The adaptability and reusability of our machine learning models is tested by enforcing a sudden infrastructure change. Figure 9.b plots the relative execution time estimation error for the Wordcount MapReduce operator when after 100 runs the cluster undergoes an upgrade, where all the HDDs that form the HDFS substrate that stores the data are substituted by SSDs. This affects the execution time estimations of the Wordcount operator (assuming that no I/O information has been modeled and used for estimating the operator performance). As depicted in Figure 9.b there is a temporal degradation of the relative error due to the fact that IReS still uses the same models, which capture the characteristics of the previous infrastructure. Although the relative error increases from 30% to 50% right after the change, it is still more beneficial to use the existing models than to discard them and start from scratch, as the relative error of assuming no knowledge would be almost 100%. Besides, as more execution measurement are acquired the relative

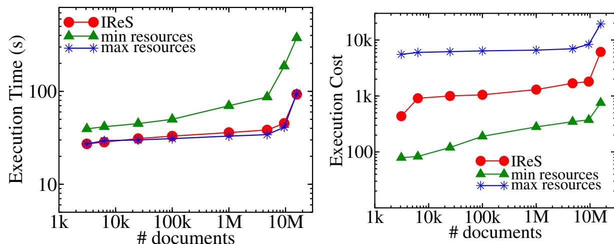

Figure 10: Execution time and cost vs. input size.

error decreases again and the models regain their accuracy, adapting seamlessly to the new cluster state.

D. Resource provisioning

In this last set of experiments we demonstrate the effectiveness of our resource provisioning mechanism by letting IReS decide on the amount of resources to be allocated in a cluster of 32 cores and 54GB RAM in total when executing the Spark (MLlib) implementation of the tf-idf operator. We assume an optimization policy of minimizing the workflow (i.e., operator) execution time. In Figure 10 we plot the time needed to execute the workflow as well as the cost of the allocated resources for various input sizes and 3 different strategies: a) static selection of the maximum available cluster resources (denoted as *max resources*), b) static allocation of the minimum resources required (denoted as *min resources*) and c) dynamic resource allocation through IReS. The execution cost can be considered as the amount of money spent on renting Amazon VMs or simply a function of the utilized resources. To express the execution cost we adopt a simplified version of [31], namely $\#VM \cdot cores/VM \cdot MM/VM \cdot t$, where $\#VM$ is the number of VM instances, $cores/VM$ is the number of cores per VM, MM/VM is the main memory per VM (in GB) and t is the execution time. This is the metric we plot in the second graph of Figure 10.

Intuitively, when running a task in a distributed environment the execution time decreases as more resources are utilized - yet, more resources result in a higher execution cost. Contrarily, settling with the minimum resources necessary to execute an operator cuts corners at the cost of performance. IReS manages to achieve workflow execution times as low as the max resources strategy, yet incurring an execution cost that lies in-between the two static strategies, provisioning just the right amount of resources according to the size of the input data: As the input dataset scales, more resources are provisioned by IReS in order to sustain low execution times, thus the execution cost approaches the one incurred by max resources.

IV. Related Work

In the ever evolving Big Data landscape, the reconciliation and/or combination of the different data models and programming paradigms open up new and promising fields of research. The first attempts along this line lie in the field of data management and aim to provide a unified query language or API over various datastores. SparkSQL [32], part of the Apache Spark project [3], and PrestoDB [33], powered by Facebook, are two production systems that provide a query execution engine with connectors to various external systems (e.g., PostgreSQL, MemSQL, Hive, etc.). However, to perform any operation on external data they both need to fetch and distribute them internally, missing out on many engine-specific optimizations.

Other approaches, like SQL++ [34] and Apache Drill [35], focus more on providing extended SQL querying capabilities over different, possibly schema-less data stores, without assuming any planning or optimization mechanisms. QUEPA [36], the most recent effort on data integration over polystores, offers advanced exploration capabilities through record linkage. However, it too lacks mechanisms that optimize query execution.

Recent research works like the Cascading Lingual project [37], CloudMdsQL [38] and BigDAWG [39] try to optimize query resolution over heterogeneous environments by pushing query processing to the datastores that manage the data as much as possible. They mostly provide rule-based optimizations while considerable effort is devoted to the translation between the involved storage engines' native query languages. All of the above approaches, unlike IReS, focus solely on storing and querying Big Data, rather than performing any complex analytics workflow on them.

In the field of workflow management, HFMS [16] aims to create a planner for multi-engine workflows, but 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. Contrarily, IReS, an early prototype of which has been demonstrated in [40] and [41], is a fully open-source platform that targets both low and high level operators.

Musketeer [17] and Rheem [18] also address multi-engine workflow execution, acting as mediators between an engine's front- and back-end. They first map a user's workflow to an internal representation and then apply a set of rule-based optimizations before sending it for execution. They focus more on the translation of scripts from one engine to another, being thus tied to specific programming languages and engines. Contrarily, IReS is engine agnostic, treating operators as black boxes. This allows for extensibility to new engines and easy addition of new operators regardless of their implementation language.

V. Conclusions

In this paper we presented IReS, a sophisticated meta-scheduler for multi-engine environments. IReS optimizes and plans complex analytics workflows by performing a mix 'n' match of diverse runtimes and data stores and by deciding on the exact amount or resources to be allocated in order to conform as much as possible to the user-defined optimization criteria, be it execution time, resource

consumption or any custom function of measurable execution metrics. This functionality relies on the cost and performance estimations of the available operators over the deployed engines.

The IReS prototype already supports a number of compute and data engines and has been extensively evaluated in optimizing and scheduling a variety of diverse, business-driven workflows that fall into the fields of text, graph and relational analytics. The experiments showcase (a) the performance gains of the IReS mix 'n' match strategy, which reach 30% with respect to statically scheduled, single-engine workflows, (b) the efficiency of the optimizer, which designates the optimal execution plan in the sub-second time scale for realistic, medium-sized workflows, (c) the effectiveness of the resource provisioning strategy, which perfectly matches any user-provided policy and (d) the adaptability of the system, which manages to ameliorate its accuracy with every execution and recover from unexpected changes within a few tens of extra runs.

Acknowledgement

This work has been supported by the European Commission in terms of the ASAP FP7 ICT Project (619706). N. Papailiou has received funding from IKY fellowships of excellence for postgraduate studies in Greece - SIEMENS program.

References

[1] T. H. Davenport and J. Dyché, "Big Data in Big Companies," *International Institute for Analytics*, 2013.

[2] "Apache Hadoop," http://hadoop.apache.org/.

[3] "Apache Spark," https://spark.apache.org/.

[4] "Apache Hama," https://hama.apache.org/.

[5] "Amazon EMR," http://aws.amazon.com/elasticmapreduce/.

[6] "Google Cloud Platform," https://cloud.google.com/hadoop/.

[7] "Microsoft Azure HDInsight," https://azure.microsoft.com/en-us/services/hdinsight/.

[8] "Docker Hub," https://hub.docker.com/.

[9] H. Herodotou *et al.*, "Starfish: A Self-tuning System for Big Data Analytics." in *CIDR*, 2011.

[10] H. Lim, H. Herodotou, and S. Babu, "Stubby: A Transformation-based Optimizer for Mapreduce Workflows," *VLDB*, 2012.

[11] M. Ferguson, "Architecting A Big Data Platform for Analytics," *A Whitepaper Prepared for IBM*, 2012.

[12] D. Tsoumakos and C. Mantas, "The Case for Multi-Engine Data Analytics," in *Euro-Par 2013: Parallel Processing Workshops*. Springer, 2014.

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

[14] "Hortonworks Sandbox 2.1," http://hortonworks.com/products/hortonworks-sandbox/.

[15] "Running Databases on AWS," http://aws.amazon.com/running_databases/.

[16] A. Simitsis, K. Wilkinson, U. Dayal, and M. Hsu, "HFMS: Managing the Lifecycle and Complexity of Hybrid Analytic Data Flows," in *ICDE*. IEEE, 2013.

[17] I. Gog *et al.*, "Musketeer: All for One, One for All in Data Processing Systems," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 2.

[18] D. Agrawal *et al.*, "Rheem: Enabling multi-platform task execution," *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, 2016.

[19] S. Babu, "Towards Automatic Optimization of MapReduce Programs," in *ACM symposium on Cloud computing*, 2010.

[20] Z. Zhang *et al.*, "Automated Profiling and Resource Management of Pig Programs for Meeting Service Level Objectives," in *Conference on Autonomic Computing*. ACM, 2012.

[21] B. Sharma, T. Wood, and C. R. Das, "HybridMR: A Hierarchical MapReduce Scheduler for Hybrid Data Centers," in *ICDCS*. IEEE, 2013.

[22] I. Giannakopoulos *et al.*, "PANIC: Modeling Application Performance over Virtualized Resources," in *2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, 2015, pp. 213–218.

[23] Y. Jin, "Surrogate-assisted evolutionary computation: Recent advances and future challenges," *Swarm and Evolutionary Computation*, 2011.

[24] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *IJCAI*, vol. 14, no. 2, 1995, pp. 1137–1145.

[25] "FrameworkA Free and Open Source Java Framework for Multiobjective Optimization," http://moeaframework.org/.

[26] K. Deb *et al.*, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[27] V. K. Vavilapalli *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[28] "Kitten: For Developers Who Like Playing with YARN," https://github.com/cloudera/kitten.

[29] "TPC-H benchmark," *http://www.tcp.org/hspec.html*.

[30] S. Bharathi *et al.*, "Characterization of scientific workflows," in *Workshop on Workflows in Support of Large-Scale Science*. IEEE, 2008, pp. 1–10.

[31] H.-L. Truong and S. Dustdar, "Composable cost estimation and monitoring for computational applications in cloud computing environments," *Procedia Computer Science*, vol. 1, no. 1, pp. 2175–2184, 2010.

[32] M. Armbrust *et al.*, "SparkSQL: Relational data processing in Spark," in *Proceedings of the 2015 ACM SIGMOD*. ACM, 2015, pp. 1383–1394.

[33] "Presto," http://www.teradata.com/Presto.

[34] K. Ong, Y. Papakonstantinou, and R. Vernoux, "The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases," *CoRR*, vol. abs/1405.3631, 2014.

[35] "Apache Drill," https://drill.apache.org/.

[36] A. Maccioni, E. Basili, and R. Torlone, "QUEPA: QUerying and Exploring a Polystore by Augmentation," in *SIGMOD*, 2016, pp. 2133–2136.

[37] "Cascading Lingual," www.cascading.org/projects/lingual/.

[38] B. Kolev *et al.*, "CloudMdsQL: querying heterogeneous cloud data stores with a common language," *Distributed and Parallel Databases*, pp. 1–41, 2015.

[39] J. Duggan *et al.*, "The BigDAWG Polystore System," *ACM Sigmod Record*, vol. 44, no. 2, pp. 11–16, 2015.

[40] K. Doka *et al.*, "IReS: Intelligent, Multi-Engine Resource Scheduler for Big Data Analytics Workflows," in *Proceedings of the 2015 ACM SIGMOD*. ACM, 2015, pp. 1451–1456.

[41] N. Papailiou *et al.*, "Robust and Adaptive Multi-Engine Analytics using IReS," in *Proceedings of BIRTE*, 2016.