FP7 Project ASAP
Adaptable Scalable Analytics Platform

ASAP D3.3
IReS Platform v.2

WP 3 – Intelligent, Multi-engine Resource Scheduling Platform

Nature: Report

Dissemination: Public

Version History

Version	Date	Author	Comments
0.1	06 Feb 2017	K. Doka, N. Papailiou, V. Giannakouris, G. Mytilinis	Initial Version
0.2	20 Feb 2017	K. Doka, N. Papailiou, V. Giannakouris, G. Mytilinis	First Revision
1.0	27 Feb 2017	K. Doka, N. Papailiou, V. Giannakouris, G. Mytilinis	Final Version

Acknowledgement This project has received funding from the European Union's 7th Framework Programme for research, technological development and demonstration under grant agreement number 619706.

Executive Summary

This deliverable is a report on the final version of the Intelligent, Multi-engine Resource Scheduling (IReS) platform. In this version, new functionality has been implemented while existing modules have been refined and enriched. The IReS platform is presented in full detail in a self-contained manner, while newly designed and implemented features are highlighted. The 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.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation and Overview

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 [25]. 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 data centers 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., [3, 7, 4, 5], 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., [2, 11, 15]) or packaged in pre-cooked VM or container images for ease of deployment [9]. Still, although many approaches in the relevant literature manage to optimize the performance of single engines by automatically tuning a number of configuration parameters [32, 36], 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 [28]. 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 [41] 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 [8, 12, 18]. This is where the ASAP project comes into place: it leverages the power and opportunities offered by multi-engine environments to harvest Big Data through complex analytics workflows.

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. IReS takes over that role exactly within the ASAP project.

IReS is 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 adopts 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, 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. Moreover, IReS can efficiently adapt to the current cluster/engine conditions and recover from failures by effectively monitoring the workflow execution in real-time.

1.2 Purpose of the Deliverable

This deliverable presents the final version of the IReS platform. We present the final system architecture, which has been refined and enriched since Y2 with new modules to better reflect the functionality offered. We delve into the implementation details of the IReS architectural modules, both old and new ones, and provide an extensive experimental evaluation of the platform's accuracy of operator/engine modeling, efficacy of profiling, performance of decision-making and effectiveness of fault-tolerance mechanism.

In Deliverable D3.2 we already presented the first version of the IReS platform, which included the initial version of the following:

- A modeling methodology that provides performance and cost metrics of the available analytics operators for different engine configurations. In the initial version, the metrics were collected offline in order to train all available WEKA models. The resulting models are utilized in multi-engine workflow optimization.
- A multi-engine planner that selects the most prominent workflow execution plan among existing engines, datastores and operators, based on a dynamic programming (DP) algorithm.
- 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.

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

- An execution layer that enforces the selected multi-engine execution plan.

This deliverable, which describes the second version of the IReS platform enriches D3.2 in the following:

- The models of the available operators/engines are constructed using the metrics collected from actual executions of the operators both offline (training phase) and online (refinement phase). Thus, the models are refined with every workflow execution, achieving higher accuracy and capturing temporal performance degradations.

- The planner does not only choose the (near) optimal execution plan but also elastically provisions the correct amount of resources, consulting the cost and performance models of the various operators.

- The execution layer actively monitors the selected multi-engine execution plan, allowing for fine grained resource allocation control and fault tolerance.

- The second version of our open-source prototype has been extensively evaluated 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.

1.3 Structure of the Deliverable

D3.3 is structured as follows:

- Section 2 describes the final architecture of the IReS platform delving into the implementation details of each module. While all modules involved are presented, emphasis has been given to the newly added features of IReS that were not present in the first version of the platform, as described in Deliverable D3.2.

- In section 3 we present how IReS exposes its functionality and interfaces with the rest of the ASAP components. The section also includes a brief installation and usage guide.

- Section 4 includes an extensive experimental evaluation of IReS under various circumstances, using a subset of the project's use case scenarios as well as synthetic workflows. 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.

- Section 5 briefly discusses an emerged side system specifically plans and optimizes SQL-based analytics over multi-engine environments.
- Section 6 concludes the report.

2 Final 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 sub-part of the workflow 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 (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 modules of the platform.

2.1 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 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 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.

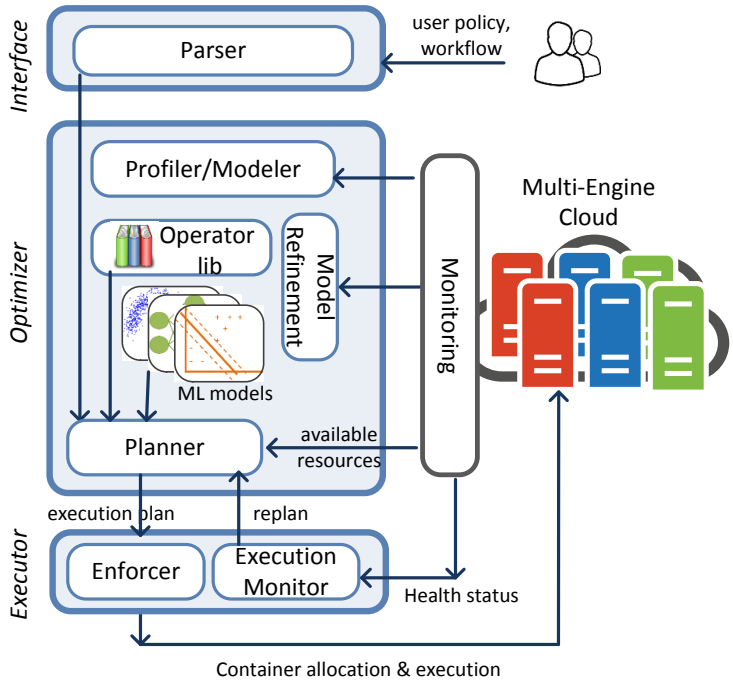
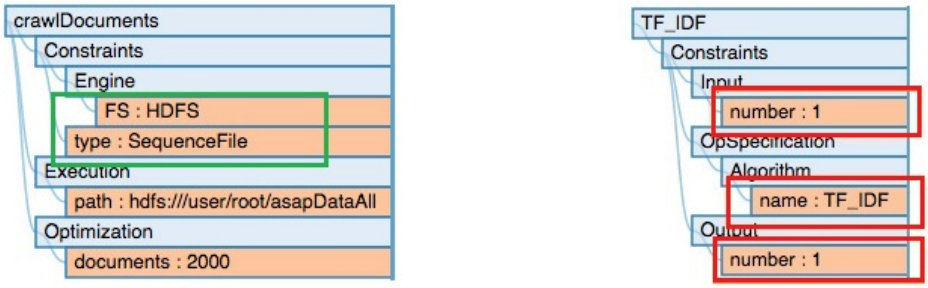


Figure 1: Architecture of the IReS platform

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 predefined. 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 con-



(a) (b)

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

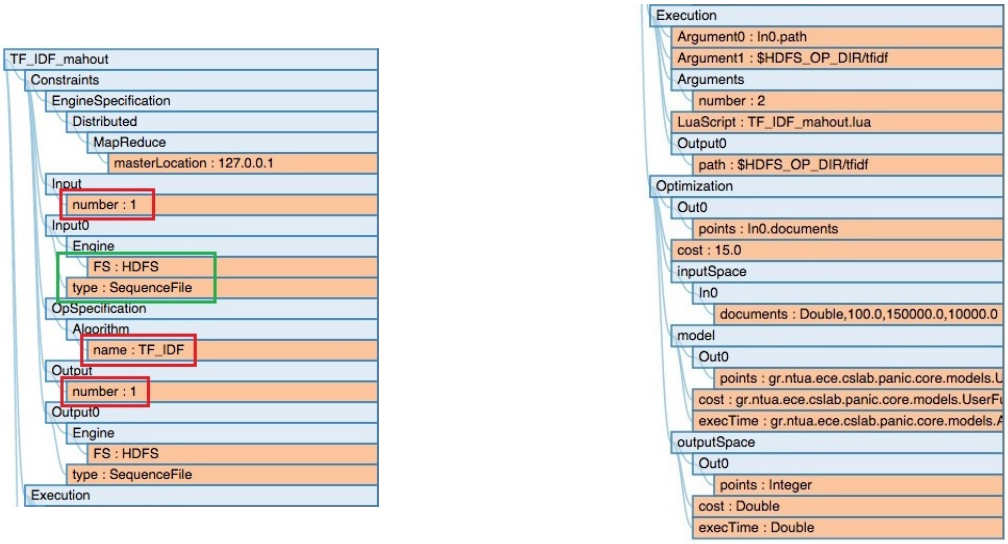


Figure 3: Meta-data description of a materialized tf-idf operator, implemented in mahout/Hadoop

sidered 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.

►**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.

2.2 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.

2.2.1 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 [21, 43, 39].

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 [29]. Its input parameters fall into three categories:

- *data-specific*, which describe the data to be used for the operator profiling (e.g., the type of data and its size)
- *operator-specific*, which relate to the algorithm of the operator (e.g., the number of output clusters in k-means), and
- *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 under each combination of the input parameter values for specific user-defined optimization metrics. Both the input parameters as well as the output metrics are specified by the user/developer. Currently we monitor 45 metrics in total, including:

- The operators execution time
- Input and output sizes (where applicable)
- Input count (e.g. Number of documents, vectors, etc.)
- Cardinality of the output (for vectors)
- Date of the experiment
- Operator specific parameters (like the number of clusters for clustering operations)
- A timeline of system metrics (CPU, RAM usage, network traffic, IOPS, etc.) for the whole cluster, periodically pulled from the ganglia monitoring system [10].

The collected metrics are then used to create estimation models [34], making use of neural networks, SVM, interpolation and curve fitting techniques for each operator running on a specific engine. However it is not necessary that all of those metrics will contribute to the estimation model.

In our approach, we utilize the approximation models offered by WEKA[31], an open source data mining software which implements a variety of machine learning algorithms. Specifically, the supported approximation techniques are the following:

- Gaussian Process, which approximates the objective function using Gaussian distributions

- Multilayer Perceptron, which represents a typical neural network with many hidden layers and neurons
- Linear Regression (Least Median Squares), which implements the methodology introduced at [38]
- Bagging, which executes classification as described in [23]
- Random SubSpace, which constructs a decision tree using the approach presented in [33]
- Regression by Discretization, which enforces regression over a discretized domain of the input space
- RBF Network, which trains a Radial Basis Function Network, as presented in [24]

The cross validation technique [35] is used to maintain the model that best fits the available data.

2.2.2 Model Refinement

This is a new feature introduced in the second version of the IReS platform. 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.

2.2.3 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. In reality, just like traditional query optimizers, our planner 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. In the following, the term “optimizer” is used in exactly this sense.

Algorithm 1 describes the optimization process, which relies on dynamic program-

ming (DP) to select the optimal execution plan.

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 ∈ 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 ∈ Operators following DAG topological ordering do
12     MOperators = findMaterializedOperators(o);
13     for mo ∈ MOperators do
14         inputCost = 0;
15         for in ∈ mo.getInputs() do
16             minCost = ∞;
17             for tin ∈ 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 ∈ o.getOutputs() do
30             tout = outputFor(mo, out);
31             dpTable[out].insert(tout, cost);
32 return dpTable[target].getMinCost();
```

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

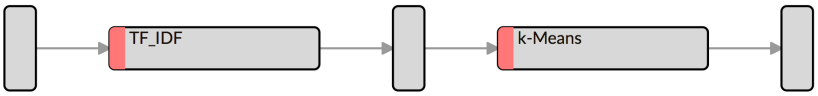


Figure 4: Abstract tf-idf, k-means workflow.

procedure we use string labeled 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 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 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

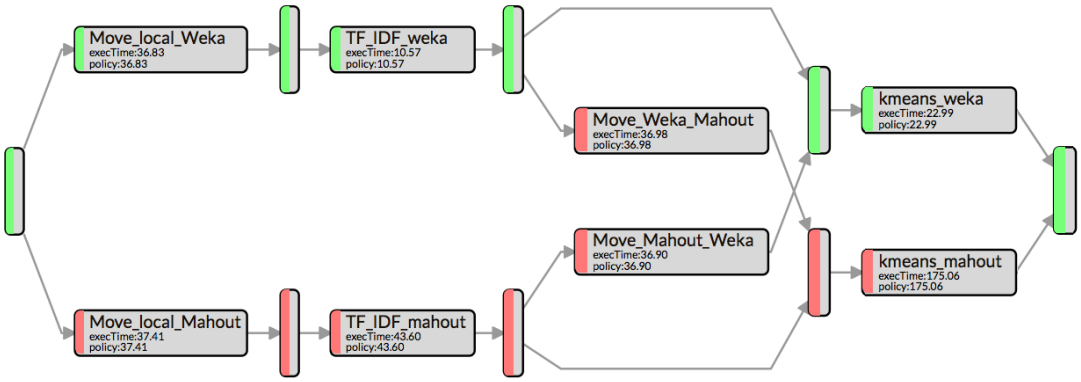


Figure 5: Materialized workflow and optimal plan.

`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.

2.2.4 Resource Provisioning

This is another novel feature of IReS platform v2.0. 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 [1] 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.

2.3 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 a workflow

of primitives as understood by the specific runtimes and storage engines. Such actions might entail code and/or data shipment if necessary.

Our current working prototype relies on YARN [42], 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 [13], 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 newly added feature of this version is the fault-tolerance mechanism. 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.

3 IReS Documentation

This section serves as an installation and execution manual for IReS. The full documentation of IReS is available online[2].

3.1 IReS Installation

To have the IRes platform up and running, 4 steps are required:

1. Clone IReS code to the server
2. Run install.sh
3. Validate installation
4. Start the IReS server

3.1.1 Clone IReS code to the server

For a quick reference of how to use git, click here. Open a terminal (Linux) and navigate to a desired directory where IReS-Platform files will be cloned e.g. asap. Then, clone the project by entering the following command:

```
git clone git@github.com:project-asap/IReS-Platform.git
```

3.1.2 Run install.sh

After successful cloning of the IReS platform, various folders and files can be found inside $IRES_HOME. Among them there exists install.sh. Assuming that the current working directory is $IRES_HOME, executing

```
./install.sh
```

will start building IReS. Upon successful build you will be prompted to provide the path where Hadoop YARN is located in your computer. By doing this, IReS gets connected to Hadoop YARN. Alternatively, executing

```
./install.sh -c $YARN_HOME,$IRES_HOME
```

will make the connection of IReS and YARN, where $YARN_HOME and $IRES_HOME correspond to the absolute paths of YARNs and IReSs home folder.

Assuming that the connections have been established, update the file

[2] https://project-asap.github.io/ASAP-documentation/ires_docs/

```
$YARN_HOME/etc/hadoop/yarn-site.xml
```

with the following property values,

```
yarn.nodemanager.services-running.per-node
yarn.nodemanager.services-running.check-availability
yarn.nodemanager.services-running.check-status
```

These properties enable IReS to run workflows over YARN and monitor cluster resources and services.

3.1.3 Validate installation

If anything goes wrong during the build process of IReS, error messages will be printed out and a log file will be provided.

3.1.4 Start the IReS server

Start the IReS server by running the command

```
./install.sh -r start
```

No exception should be raised. Also, the jps command should print a `Main` process running that corresponds to ASAP server. Open the ASAP server web user interface at `http://your_hostname:1323/web/main`. The IReS home page should be displayed. Run the `hello_world` workflow from the "Abstract Workflows" tab and check what happens not only in IReS web interface but also in YARN and HDFS web interfaces. Make sure that YARN has been started before running any workflow.

3.2 Running a sample workflow

The HelloWorld is a simple workflow constists of just a single operator, designed for demonstration purposes. To run the HelloWolrd follow the next steps:

1. Go to IReS UI: http://ires_host:1323/web/main
2. Go to the Abstract Workflows tab and select the HelloWorld workflow
3. Click on Materialize Workflow button
4. Click on the Execute Workflow button to start the execution

In the figures 6 and 7 below we can see the execution process.

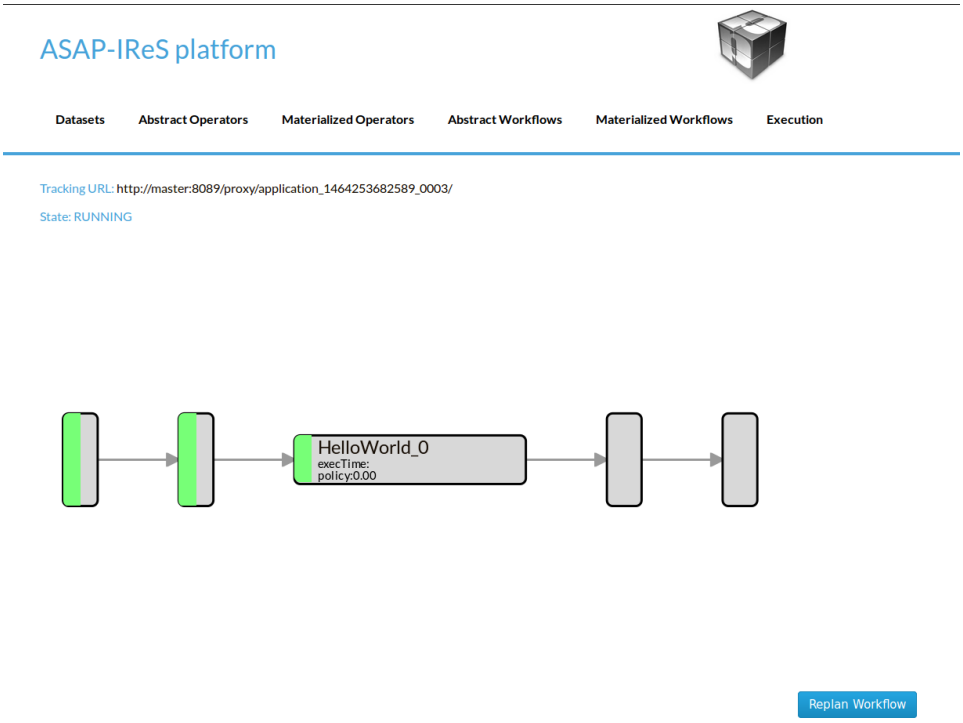


Figure 6: The execution has started.

3.3 Creating a workflow from scratch

This section describes the process of designing a new workflow from scratch. We will create a workflow that consists of a single operator and takes as input a text file and produces as output the number of lines. The basic steps that need to be followed are: (1) The description of the materialized input dataset, (2) the addition of the materialized operators (and their descriptions) to the IReS operator library, (3) the description of the abstract operator, (4) the definition of the workflow and finally (5) the workflow materialization. For steps (2) and (4) there exist alternative ways, which are described in the following.

1. **Dataset definition:** In order to create the workflow input dataset, the dataset definition must be added to the IReS library. Create a file named asapServerLog into the asapLibrary/datasets/ folder and add the following content:

   ```
   Optimization.documents=1
   Execution.path=hdfs\:///user/root/asap-server.log
   Constraints.Engine.FS=HDFS
   ```

   This step assumes that a file named asap-server.log exists in the HDFS.

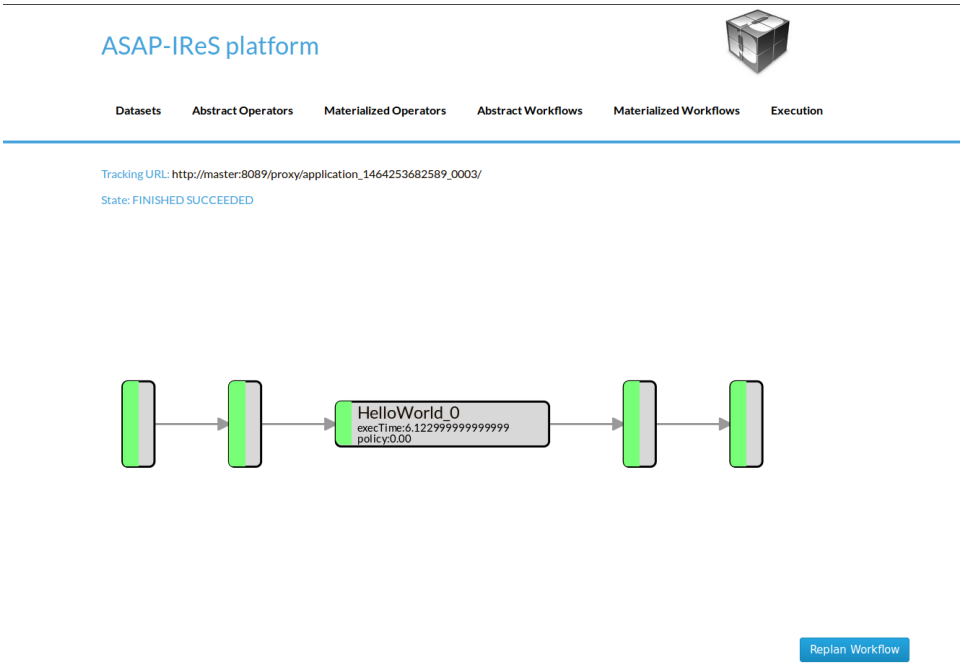


Figure 7: The execution has been completed.

2. a **Materialized Operator Definition (Server-Side):** This is the first alternative way of adding a materialized operator to the IReS operator library. One must essentially create a folder structure in the IReS server, containing a minimum set of files required to describe and run an operator.

   (a) From the bash shell, go to the asapLibrary/operators folder in the IReS installation directory. `cd $ASAP_HOME/target/asapLibrary/operators`

   (b) Then, create a new folder named with the new materialized operators name. `mkdir LineCount`

   (c) Create the description file and enter the information below. A description file should meet the standards of the template provided in the IReS online Documentation[3]. This template contains all the obligatory as well as optional parameters that can be used to describe an operator.

```
$nano description
Constraints.Engine=Spark
Constraints.Output.number=1
```

[3] `https://project-asap.github.io/ASAP-documentation/ires_docs/files/description_template`

```
Constraints.Input.number=1
Constraints.OpSpecification.Algorithm.name=LineCount
Optimization.model.execTime=gr.ntua.ece.cslab.panic.core.models.UserFunction
Optimization.model.cost=gr.ntua.ece.cslab.panic.core.models.UserFunction
Optimization.outputSpace.execTime=Double
Optimization.outputSpace.cost=Double
Optimization.cost=1.0
Optimization.execTime=1.0
Execution.Arguments.number=2
Execution.Argument0=In0.path.local
Execution.Argument1=lines.out
Execution.Output0.path=$HDFS_OP_DIR/lines.out
Execution.copyFromLocal=lines.out
Execution.copyToLocal=In0.path
```

(d) Create the .lua file with the execution instructions

```
$ nano LineCount.lua
operator = yarn {
name = "LineCount",
timeout = 10000,
memory = 1024,
cores = 1,
container = {
instances = 1,
--env = base_env,
resources = {
["count_lines.sh"] = {
file = "asapLibrary/operators/LineCount/count_lines.sh",
type = "file",                -- other value: 'archive'
visibility = "application"  -- other values: 'private', 'public'
}
},
command = {
base = "./.sh"
}
}
}
```

(e) Create the executable named count_lines.sh with the following content

```
$ #!/bin/bash
$ wc -l $1 >> $2
$ chmod +x count_lines.sh
```

(f) Restart the IReS server

```
$ IRES_HOME/asap-server/src/main/scripts/asap-server restart
```

2. b **Materialized Operator Definition (via REST)**: This is the second alternative way to create a materialized operator using the IReS REST API. To do so, create a folder locally and add the required description file as well as all other files needed for the execution. In this case, an extra parameter should be added to the description file which defines the execution command (Execution.command).

   (a) Create inside the folder a file named *description* with the following content:

```
$nano description
Constraints.Engine=Spark
Constraints.Output.number=1
Constraints.Input.number=1
Constraints.OpSpecification.Algorithm.name=LineCount
Optimization.model.execTime=gr.ntua.ece.cslab.panic.core.models.UserFunction
Optimization.model.cost=gr.ntua.ece.cslab.panic.core.models.UserFunction
Optimization.outputSpace.execTime=Double
Optimization.outputSpace.cost=Double
Optimization.cost=1.0
Optimization.execTime=1.0
Execution.Arguments.number=2
Execution.Argument0=In0.path.local
Execution.Argument1=lines.out
Execution.Output0.path=$HDFS_OP_DIR/lines.out
Execution.copyFromLocal=lines.out
Execution.copyToLocal=In0.path
Execution.command=./count_lines.sh
```

   (b) Create the executable named count_lines.sh with the following content

```
$ #!/bin/bash
$ wc -l $1 >> $2
$ chmod +x count_lines.sh
```

   (c) Send the operator via the send_operator.sh script:

   `$ ./send_operator.sh LOCAL_OP_FOLDER IRES_HOST LineCount`

   The script is available at $IRES_HOME/asap-server/src/main/scripts. It can also be downloaded directly[4].

3. **Abstract operator definition:** Create the LineCount abstract operator by creating a file named LineCount in the asapLibrary/abstractOperators folder with the following content:

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

[4] https://github.com/project-asap/IReS-Platform/blob/master/asap-platform/asap-server/src/main/scripts/send_operator.sh

4. a **Abstract workflow definition (Server-Side):** This is the first way to describe the abstract workflow. Create the LineCountWorkflow workflow by creating a folder named LineCountWorkflow in the asapLibrary/abstractWorkflows. The abstract workflow folder should consist of three required components: the *datasets* folder, the *operators* folder and a file named *graph*.

   (a) Create a folder named datasets and copy the asapServerLog file from the asapLibrary/datasets/ folder into it. Then, create an empty file named d1, which represents the abstract output dataset.

   (b) Create a file named graph and add the following content

   ```
   asapServerLog,LineCount,0
   LineCount,d1,0
   d1,$$target
   ```

   This graph file defines the workflow graph as follows: asapServerLog dataset is being given as input to the LineCount abstract operator and LineCount operator outputs the result into d1. Finally, d1 node maps to the final result ($$target).

   (c) Create a folder named operators which will contain the description of the abstract operators involved in the workflow, namely the LineCount operator description as created in step 3.

   (d) Restart the server for changes to take effect

   ```
   $ IRES_HOME/asap-platform/asap-server/src/main/scripts/asap-server
   restart
   ```

4. b **Abstract Workflow Definition (GUI):** Alternatively, the abstract workflow can be defined through the Web UI as follows.

   (a) Go to the *Abstract Workflows* tab and click the *New Workflow* button.

   (b) Then we add the workflow parts one-by-one. First we add the *asapServerLog* dataset from the dataset library. Select the *Materialized Dataset* radio button and enter the dataset name in the comma seperated list text box. Then click the *Add nodes* button to add the dataset node to the workflow graph. Repeat this step to add an output node with name d1. Just enter the name d1 to the text box and click the *Add nodes* button.

   (c) Add the LineCount abstract operator to the workflow. Select the *Abstract Operator* radio button, enter the operator's name (LineCount) in the text box and click again the *Add nodes* button.

   (d) Describe the workflow by connecting the graph nodes defined in the previous steps by entering the following text in the large text box:

   ```
   asapServerLog,LineCount
   LineCount,d1
   d1,$$target
   ```

Click the *Change graph* button.

5. **Workflow materialization:** To materialize the workflow:

   (a) Navigate to the *Abstract Workflows* tab and click on the LineCountWorkflow created in the previous steps.

   (b) Click on the Materialize Workflow button

   (c) Now you can see the materialized workflow. Click on the Execute Workflow button to trigger the execution (see Figure 8). When the execution finishes, navigate to the HDFS to see the output.

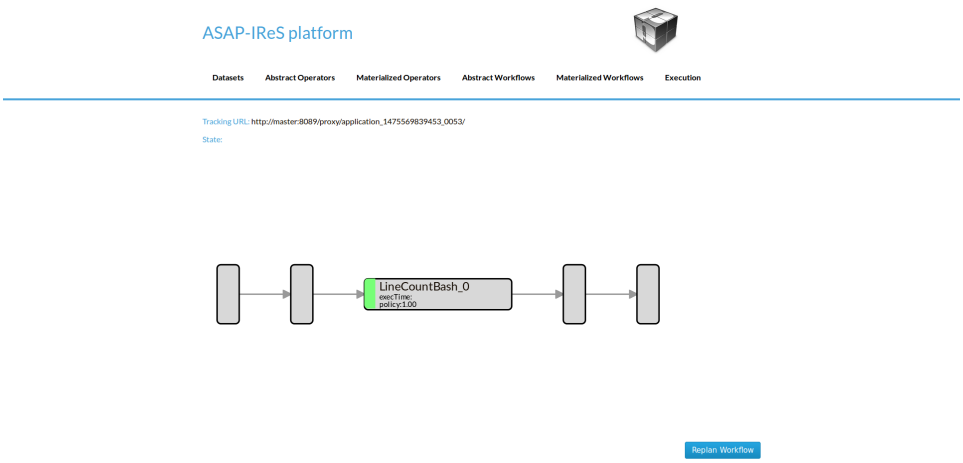


Figure 8: The LineCount workflow.

All resources and sample files described in this section are available online [5].

3.4 Creating a text clustering workflow

This example describes how to define a text clustering workflow consisting of two operators. This workflow takes as input a dataset with raw text files. In the first operator the files are transformed into tf-idf vectors. Then the vectors are given as input to the next operator which performs the clustering using a k-means algorithm. We will use two Cilk-based implementations for this example, and we will create all the required files and directories using the server-side method.

[5] `https://project-asap.github.io/ASAP-documentation/ires_docs/files/LineCountExample.tar`

1. **Dataset definition:** We will use this text file for our example. The following file should exists in the HDFS cluster with name textData. Create the data definition as follows: 1. Create a file named textData in the asapLibrary/datasets folder 2. Add the following content:

```
Constraints.Engine.FS = HDFS
Constraints.type = text
Execution.path = hdfs:///user/asap/input/textData
Optimization.size = 932E06
```

2. **TF-IDF abstract operator definition:** Next, we will define the abstract definition for a TF-IDF operator. First, create a file named tf-idf in the asapLibrary/abstractOperators folder and add the following content:

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

3. **K-Means abstract operator definition:** Create the abstract definition of K-Means operator as follows: Create a file named kmeans in the asapLibrary/abstractOperators folder and add the following content:

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

4. **Abstract workflow definition:** In this step well describe how to connect the two aforementioned operators in order to define the text clustering workflow. 1. Create a folder named TextClustering in the asabLibrary/abstractWorkflows folder 2. Specify the workflow graph by creating a file named graph with the following content:

```
testdir,tfidf_cilk,0
tfidf_cilk,d1,0
d1,kmeans,0
kmeans,d2,0
d2,$$target
```

   Next, we will define the materialized operators. We will use Cilk for our implementations.

5. **TF-IDF materialized operator definition (Cilk):**

   (a) Create a folder named TF_IDF_cilk in the asapLibrary/operators folder.

   (b) Create the description file named description and add the following content:

```
Constraints.Output0.Engine.FS=HDFS
Constraints.OpSpecification.Algorithm.name=TF_IDF
Constraints.Input0.type=text
Constraints.Output0.type=arff
Constraints.Engine=Cilk
Constraints.Output.number=1
Constraints.Input.number=1
Execution.LuaScript=TF_IDF_cilk.lua
Execution.Arguments.number=2
Execution.Argument0=In0.path.local
Execution.Argument1=tfidf.out
Execution.copyFromLocal=tfidf.out
Execution.copyToLocal=In0.path
Execution.Output0.path=$HDFS_OP_DIR/tfidf.out
```

(c) Create the .lua file named TF_IDF_cilk.lua as follows:

```
operator = yarn {
name = "Execute cilk tfidf",
timeout = 10000,
memory = 1024,
cores = 1,
container = {
instances = 1,
--env = base_env,
resources = {
["tfidf"] = {
file = "asapLibrary/operators/TF_IDF_cilk/tfidf",
type = "file",                -- other value: 'archive'
visibility = "application"  -- other values: 'private', 'public'
}
},
command = {
base = "export LD_LIBRARY_PATH=/0/asap/qub/gcc-5/\
lib64:$LD_LIBRARY_PATH./tfidf"
}
}
}
```

(d) Add the tf-idf executable (a link with the corresponding tarball is provided at the end of this section).

6. **K-Means materialized operator definition (Cilk):**

(a) Create a folder named kmeans_cilk in the asapLibrary/operators folder.

(b) Create the description file named description and add the following content:

```
Constraints.Output0.Engine.FS=HDFS
Constraints.OpSpecification.Algorithm.name=kmeans
Constraints.Input0.Engine.FS=HDFS
Constraints.Input0.type=arff
Constraints.Engine=Spark
Constraints.Output.number=1
Constraints.Input.number=1
Execution.LuaScript=kmeans_cilk.lua
Execution.Arguments.number=2
Execution.Argument0=In0.path.local
Execution.Argument1=kmeans.out
Execution.copyFromLocal=kmeans.out
Execution.copyToLocal=In0.path
Execution.Output0.path=$HDFS_OP_DIR/kmeans.out
```

(c) Create the .lua file named kmeans_cilk.lua as follows:

```
operator = yarn {
name = "Execute kmeans",
timeout = 10000,
memory = 1024,
cores = 1,
container = {
instances = 1,
--env = base_env,
resources = {
["kmeans"] = {
file = "asapLibrary/operators/kmeans_cilk/kmeans",
type = "file",                -- other value: 'archive'
visibility = "application"  -- other values: 'private', 'public'
}
},
command = {
base = "export LD_LIBRARY_PATH=/0/asap/qub/gcc-5/lib64\
:$LD_LIBRARY_PATH ; ./kmeans"
}
}
}
```

(d) Add the kmeans executable (provided in the tarball).

7. **Execute the workflow:** After finishing the previous steps restart the server for changes to take effect. Then:

   (a) Go to *Abstract Workflows* and click on TextClustering.

   (b) Materialize the workflow by clicking *Materialize* button.

   (c) Start the workflow execution by clicking Execute button.

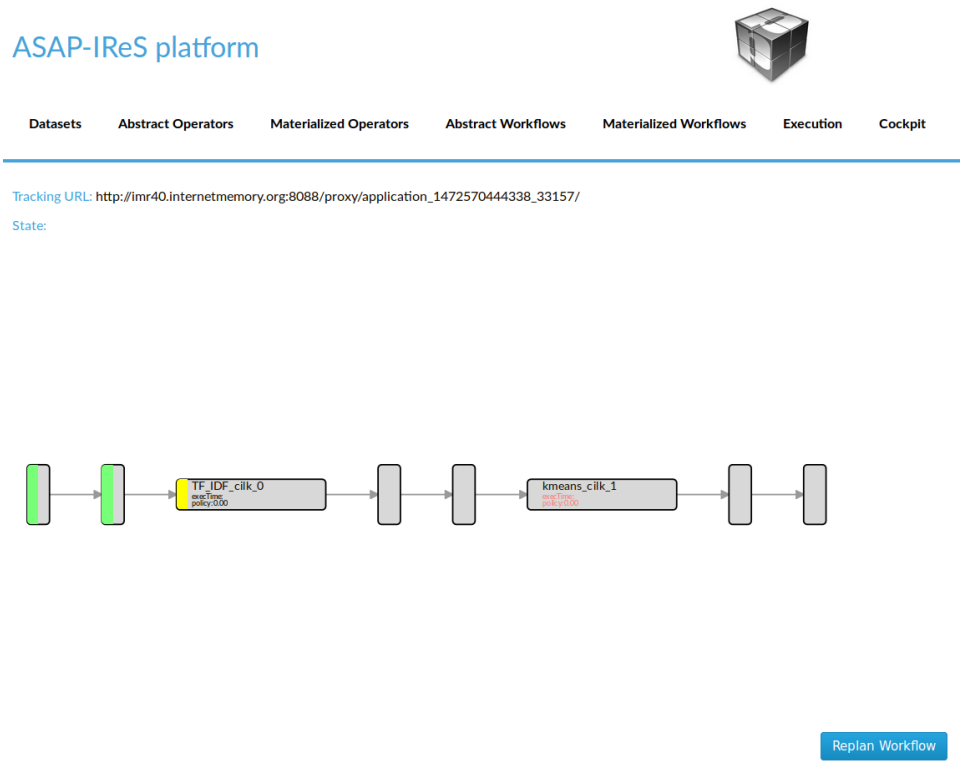


Figure 9: The Cilk text clustering workflow.

The files used in this example can be downloaded from the online IReS Documentation[6].

3.5 External API

The functionality of the IReS platform is exposed to the rest of the ASAP components through a RESTful API. The RESTful API can be accessed online [7]. Also, a list with all the available methods is provided below.

[6] https://project-asap.github.io/ASAP-documentation/ires_docs/files/TextClustering.tar

[7] https://project-asap.github.io/ASAP-documentation/ires_docs/rest_api

4 Evaluation of the IReS platform

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[8]. In our experiments, IReS controls a cloud-based deployment of several runtime engines and data stores[9] over 16 virtual machines of an Openstack cluster hosted in our lab. All the supported engines have been tuned according to best practices.

```
q1:SELECT c_custkey, r_name
   FROM customer, nation, region
   WHERE c_nationkey = n_nationkey
   and r_regionkey = n_regionkey
q2:SELECT p_partkey
   FROM part, partsupp
   WHERE ps_partkey = p_partkey
   AND ps_retailprice < 2000.0
q3:SELECT c_name, o_orderdate, r_name
   FROM lineitem, orders, q1, q2
   WHERE l_orderkey = o_orderkey
   AND o_custkey = c_custkey
   AND l_partkey = p_partkey
```

Figure 10: The sql query of the relational analytics workflow.

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 ASAP. 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). 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

[8] `https://github.com/project-asap/IReS-Platform`
[9] Hadoop 2.7.0, Spark 1.6.0, Hama 0.7.1, scikit-learn 0.17.1, MemSQL 5.0, Postgres 9.5.3

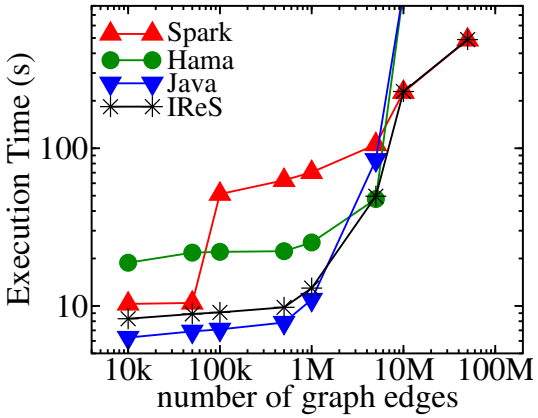


Figure 11: Execution times for the graph analytics workflows vs. various input sizes when running on single- and multi-engine (through IReS) environments.

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 10) which join tables residing in different stores. For this workflow, we use data produced by the popular TPC-H [20] 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.

4.1 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 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 11 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

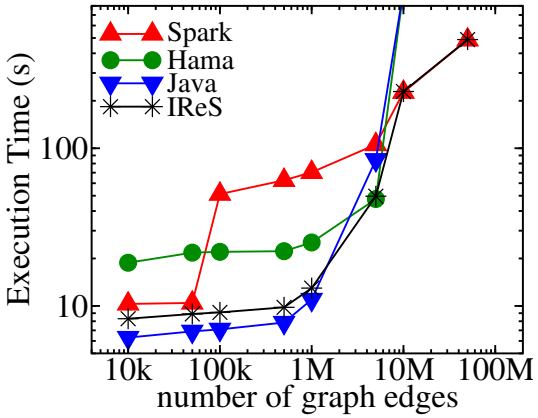


Figure 12: Execution times for the text analytics workflows vs. various input sizes when running on single- and multi-engine (through IReS) environments.

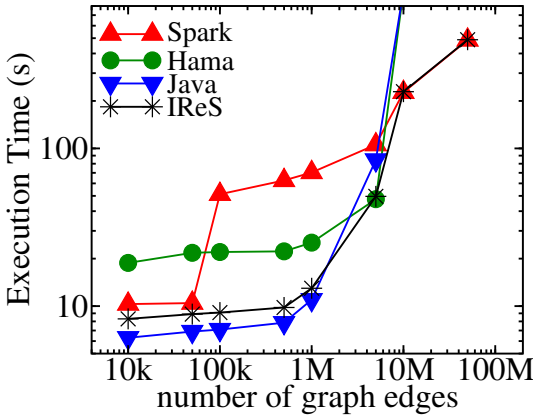


Figure 13: Execution times for the relational analytics workflows vs. various input sizes when running on single- and multi-engine (through IReS) environments.

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 12 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

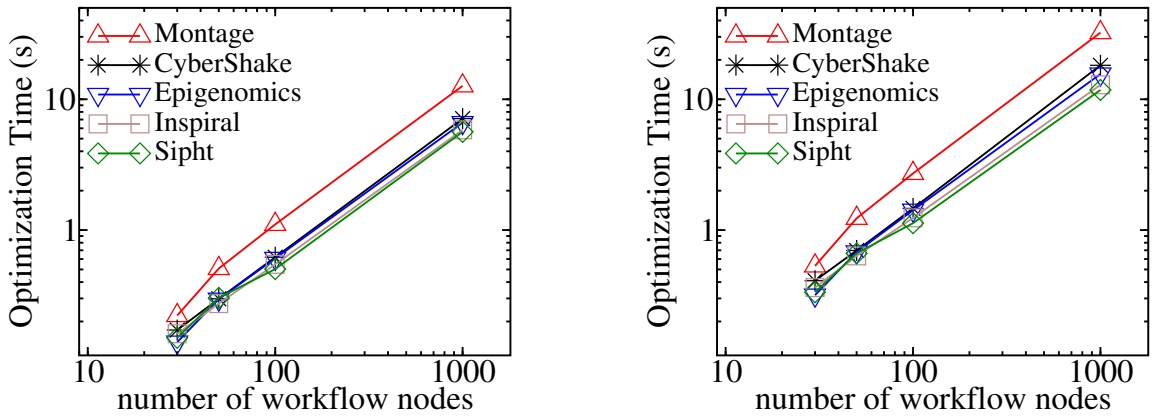


Figure 14: Workflow optimization times for 4 and 8 engines, using various workflow types of ranging size.

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 13 depicts the execution performance of the relational analytics workflow. While PostgreSQL can provably 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.

4.2 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 [22]. 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

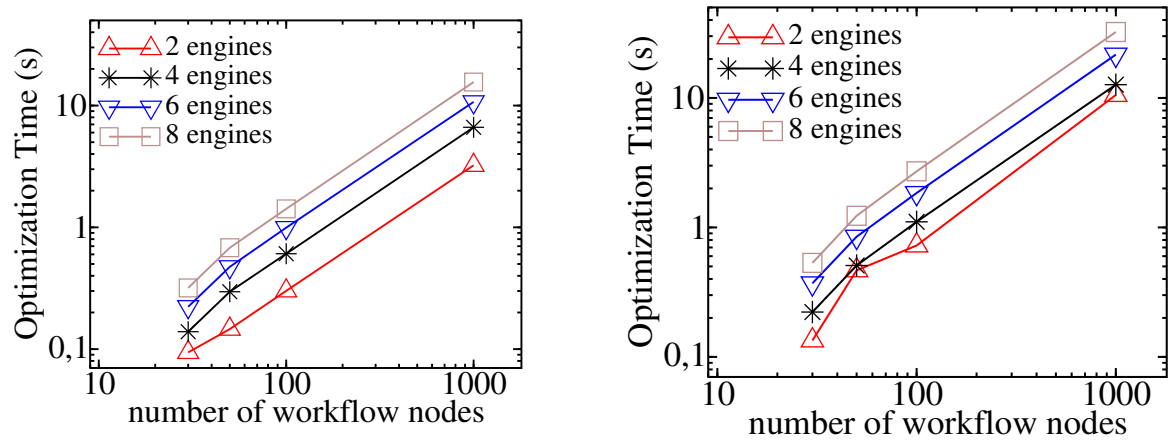


Figure 15: Workflow optimization times for Montage and Epigenomics graphs, using various number of engines and ranging the workflow size.

parallel as well as workflows that have a relatively fixed structure and perform identical analyses on multiple input datasets.

Figure 14 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 14 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 15).

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

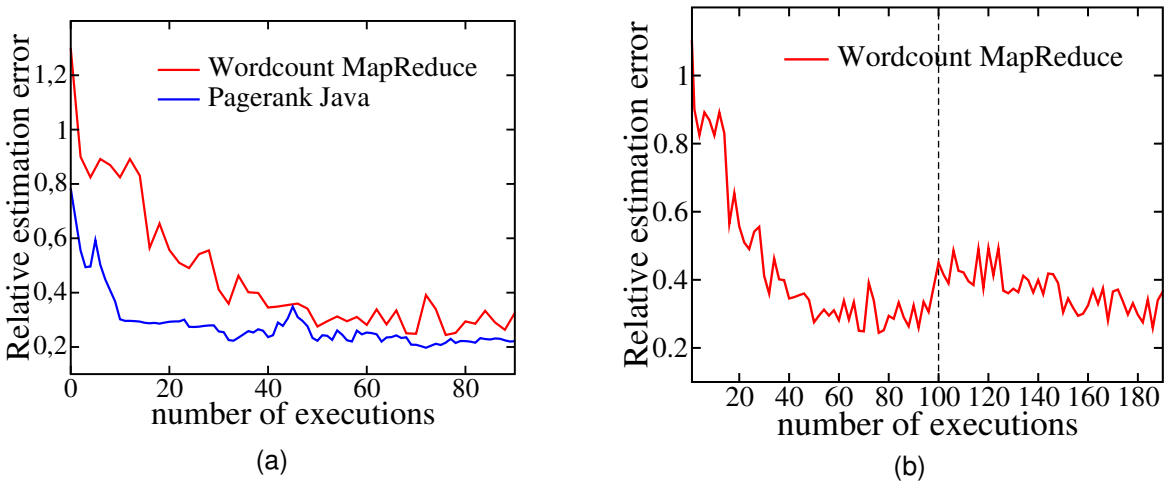


Figure 16: Relative execution time estimation error w.r.t. the number of executions (a) in normal IReS operation (b) when an infrastructure change occurs after 100 executions.

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.

4.3 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 16.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 16.b plots the relative execution time estimation error for the Wordcount MapReduce operator when after 100 runs the clus-

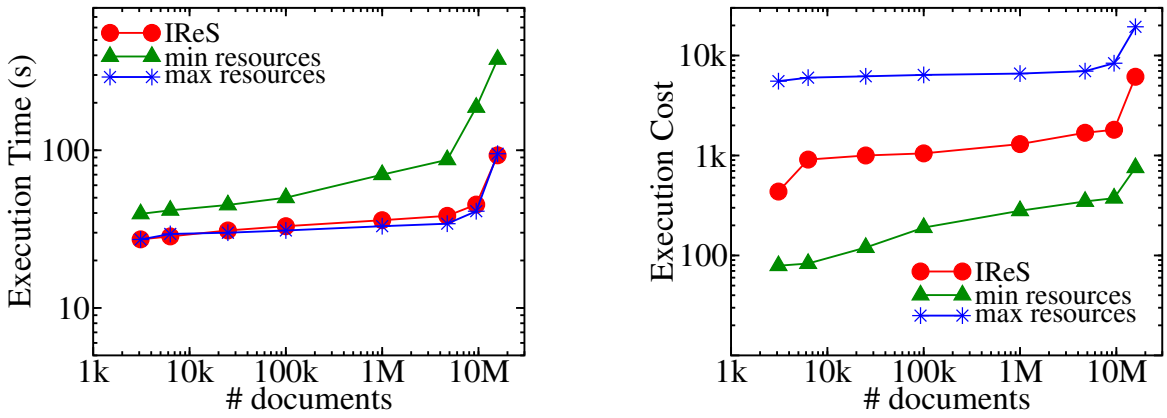


Figure 17: Execution time and cost vs. input size.

ter 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 16.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 error decreases again and the models regain their accuracy, adapting seamlessly to the new cluster state.

4.4 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 17 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 [40], 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 17.

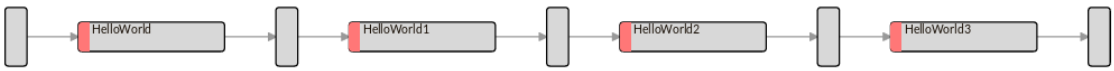


Figure 18: Abstract workflow used in the fault-tolerance evaluation experiment.

Table 1: Operators and available implementations.

Operator	**Engines**
HelloWorld	Python
HelloWorld1	Spark, Python
HelloWorld2	Spark, MLLib, PostgreSQL, Hive
HelloWorld3	Spark, Python

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.

4.5 Fault-tolerance mechanism

In order to validate the effectiveness of the replanning mechanism of IReS, denoted as *IResReplan* hereafter, we compare it to a trivial replanning strategy, where intermediate results are not materialized and the whole workflow is re-scheduled for execution (denoted as *TrivialReplan*). Furthermore, we compare IResReplan to the hypothetic case where no failure happens but one engine, that is normally chosen by the optimal plan, is not available since the beginning of the workflow execution (denoted as *SubOptPlan*). This is equivalent of comparing the execution time of a sub-optimal plan under the absence of failures with the optimal one but when failures occur and IResReplan is used.

We present the following scenario: We consider a workflow of four operators, where each of them can be executed in a set of candidate engines. Figure 18 shows the workflow topology and Table 1 presents the alternative engine options for each operator.

Each of these operators is profiled by the IReS system and a cost model for its

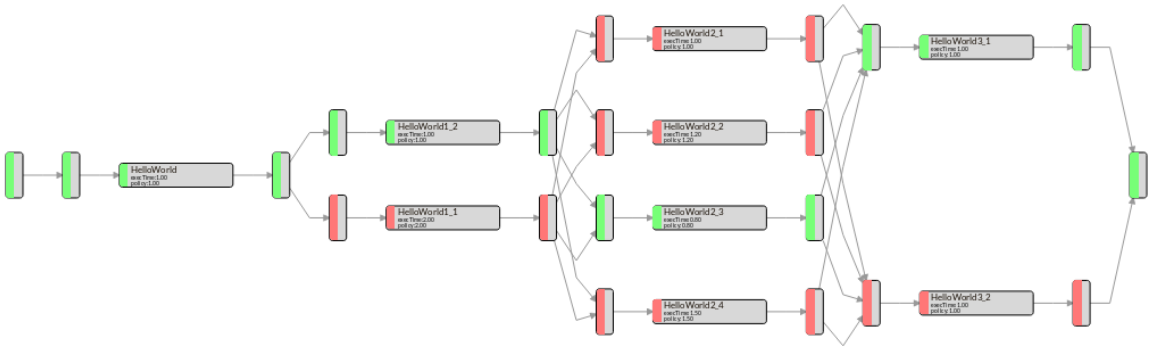
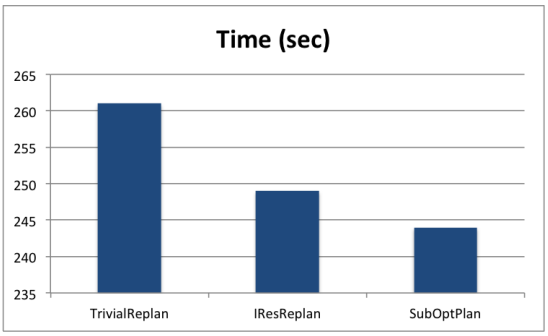


Figure 19: Materialized workflow used in the fault-tolerance evaluation experiment.



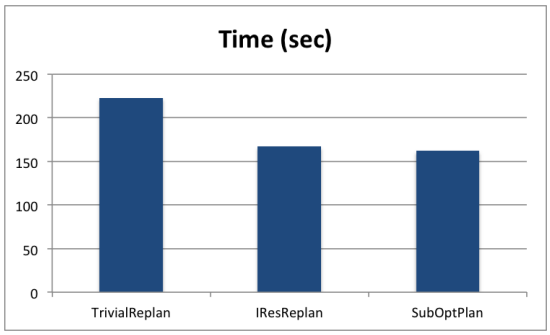
	planning time (ms)
IResReplan	410
TrivialReplan	141

Figure 20: Execution time and planning time when HelloWorld1 fails.

execution has been created. According to this cost model, the planner creates the optimal plan for execution. Figure 19 shows the optimal plan annotated with green color and all the alternatives with red. The black-dashed lines are used to separate different operators. Between two consecutive dashed lines, we can see all the alternatives for the execution of a specific operator.

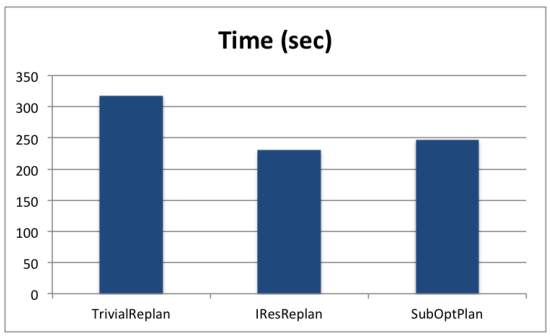
We measure performance in terms of execution and planning time in three different failure cases. In each case, we kill the engine of a different operator. As IResReplan takes advantage of the intermediate results, the more are the tasks that have been successfully executed, the higher the gains we expect.

In the above Figures 20–22, we observe that the IResReplan consistently outperforms the TrivialReplan strategy. Contrarily to IResReplan, the trivial strategy gets worse when many operators have been completed successfully before the failure happens. However, the numbers Tables 20–22 indicate that the replanning process is more expensive in the case of IResReplan. IResReplan makes extra computations in order to identify the successful intermediate results and match the completed part of the workflow with the new graph that the IReS planner provided. Nevertheless, as the



	planning time (ms)
IResReplan	311
TrivialReplan	131

Figure 21: Execution time and planning time when HelloWorld2 fails.



	planning time (ms)
IResReplan	216
TrivialReplan	109

Figure 22: Execution time and planning time when HelloWorld3 fails.

planning time is in the order of milliseconds, the overhead of IResReplan is negligible.

It is also interesting that even under the existence of failures, IReS can be more efficient than another system which makes sub-optimal choices. From the above Figures we can deduce that the further in the execution path the failure happens, the greater the gains of IResReplan are compared to SubOptPlan. That happens because of two reasons:

- When many tasks have been completed successfully, there is a higher probability for IResReplan to have made more optimal choices compared to SubOptPlan. Therefore, the cumulative benefit is maximized.
- Furthermore, in that case, the size of the re-scheduled workflow is smaller and so is the probability to launch a new container. Thus, the overhead of launching new containers is minimized.

5 Side System for SQL Optimizations

SQL emerges as the de facto language for big data due to its extensibility, its ability to naturally represent queries that can be optimized. Thus new platforms constantly seek its support ([19, 6, 17, 27, 37]). These points highly suggest a multi-engine approach that allows SQL analytics over multiple data formats and uses the most appropriate engines. IReS, however, is agnostic of the operator internals and implementation language: An SQL query would be treated as a black-box operator that would be profiled, modeled and handled as a whole, missing out on the opportunity to perform a more fine-grained optimization. Moreover, it frequently proves suboptimal to define a single, global optimization layer, and disregard the capabilities of the underlying engines to locally optimize.

To that end, we have developed an open-source side-system to IRes, *MuSQLE* [10], for high-performance SQL-based analytics over different data sources and execution engines. *MuSQLE* is able to overcome the aforementioned deficiencies and optimize simple or complex SQL queries. Our solution adopts a novel API-based strategy for integrating runtimes: Instead of manual integration, *MuSQLE* utilizes standardized, API-based cost-model and execution endpoints from the participating engines. It is able to optimally disseminate parts of the initial query (including the appropriate data movements between stores) using state-of-the-art planning and letting individual optimizers handle the respective sub-queries. *MuSQLE* utilizes Spark [19] as an executor and orchestrator layer, extending its current functionality as well as providing it with a native cost-model.

In summary, our work makes the following contributions:

- We propose a generic SQL engine API that can facilitate multi-engine query optimization. The API is based on well-documented SQL functionality and can be implemented using generic, engine-agnostic interfaces like JDBC and ODBC.
- We integrate this API into a state-of-the-art query optimizer, allowing for external, multi-engine cost-based query optimization. Our optimizer runs on the logical level, allowing the connected engines to have full control of physical optimization and join execution. This approach avoids the detailed enumeration of all physical operators on the external optimizer and thus further facilitates the integration of a new SQL engine.
- We compile and utilize a cost model for the SparkSQL operators. This model is used within our query planner to achieve query optimization for SparkSQL.
- We present a fully functional system that integrates three popular engines: SparkSQL [19], PostgreSQL [16] and MemSQL [14]. We describe our system architecture and components in detail, as well as extensively evaluate the utility and efficiency of our scheme.

[10] https://github.com/gsvic/MuSQLE

- Our detailed experimental evaluation showcases that MuSQLE can accurately decide on the most suitable execution engine and provides speedups of up to 1 order of magnitude for TPCH queries, leveraging different engines for the execution of individual query parts.

More information about the MuSQLE side-system can be found in [30], which has been added to Appendix B for convenience.

6 Conclusions

In this deliverable we presented the final version of the IReS platform, 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.

Such estimations are obtained from models, initially trained offline through intelligent profiling and refined on-the-fly, as the system operates. This "black box" approach in combination with a generic operator and data description framework proposed, ensures the extensibility of IReS, facilitating the addition of new engines and operators.

Apart from mapping specific nodes and/or subgraphs of the user-provided analytics workflow to the most beneficial compute and data engines, IReS decides on the exact amount or resources to be allocated according to the optimization objectives, thus adding elasticity to the system. The robustness and reliability of IReS is ensured through the constant monitoring of the underlying infrastructure and the partial re-planning of the failed workflow.

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.

References

[1] A Free and Open Source Java Framework for Multiobjective Optimization. http://moeaframework.org/.

[2] Amazon EMR. http://aws.amazon.com/elasticmapreduce/.

[3] Apache Hadoop. http://hadoop.apache.org/.

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

[5] Apache HBase. http://hbase.apache.org/.

[6] Apache Impala. http://impala.io/.

[7] Apache Spark. https://spark.apache.org/.

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

[9] Docker Hub. https://hub.docker.com/.

[10] Ganglia Monitoring System. http://ganglia.info/.

[11] Google Cloud Platform. https://cloud.google.com/hadoop/.

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

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

[14] MemSQL. http://www.memsql.com/.

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

[16] PostgreSQL. https://www.postgresql.org/.

[17] Presto. http://www.teradata.com/Presto.

[18] Running Databases on AWS. http://aws.amazon.com/running_databases/.

[19] Spark SQL. https://spark.apache.org/sql/.

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

[21] Shivnath Babu. Towards Automatic Optimization of MapReduce Programs. In *ACM symposium on Cloud computing*, 2010.

[22] Shishir Bharathi et al. Characterization of scientific workflows. In *Workshop on Workflows in Support of Large-Scale Science*, pages 1–10. IEEE, 2008.

[23] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

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

[25] Thomas H Davenport and Jill Dyché. Big Data in Big Companies. *International Institute for Analytics*, 2013.

[26] Kalyanmoy Deb et al. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[27] Jennie Duggan et al. The BigDAWG Polystore System. *ACM Sigmod Record*, 44(2):11–16, 2015.

[28] Mike Ferguson. Architecting A Big Data Platform for Analytics. *A Whitepaper Prepared for IBM*, 2012.

[29] Ioannis Giannakopoulos et al. PANIC: Modeling Application Performance over Virtualized Resources. In *2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, pages 213–218, 2015.

[30] Victor Giannakouris, Nikolaos Papailiou, Dimitrios Tsoumakos, and Nectarios Koziris. Musqle: Distributed sql query execution over multiple engine environments. *2016 IEEE International Conference on Big Data*, 2016.

[31] Mark Hall et al. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

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

[33] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8):832–844, 1998.

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

[35] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, volume 14, pages 1137–1145, 1995.

[36] Harold Lim, Herodotos Herodotou, and Shivnath Babu. Stubby: A Transformation-based Optimizer for Mapreduce Workflows. *VLDB*, 2012.

[37] 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*, abs/1405.3631, 2014.

[38] Peter J Rousseeuw and Annick M Leroy. *Robust regression and outlier detection*, volume 589. John wiley & sons, 2005.

[39] Bikash Sharma, Timothy Wood, and Chita R Das. HybridMR: A Hierarchical MapReduce Scheduler for Hybrid Data Centers. In *ICDCS*. IEEE, 2013.

[40] Hong-Linh Truong and Schahram Dustdar. Composable cost estimation and monitoring for computational applications in cloud computing environments. *Procedia Computer Science*, 1(1):2175–2184, 2010.

[41] Dimitrios Tsoumakos and Christos Mantas. The Case for Multi-Engine Data Analytics. In *Euro-Par 2013: Parallel Processing Workshops*. Springer, 2014.

[42] Vinod Kumar Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[43] Zhuoyao Zhang et al. Automated Profiling and Resource Management of Pig Programs for Meeting Service Level Objectives. In *Conference on Autonomic Computing*. ACM, 2012.

A Mix ’n’ Match Multi-Engine Analytics

B MuSQLE: Distributed SQL Query Execution Over Multiple Engine Environments

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 pre-cooked 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

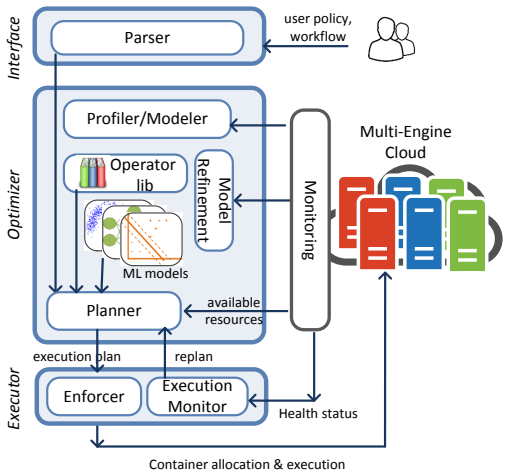


Figure 1: Architecture of the IReS platform

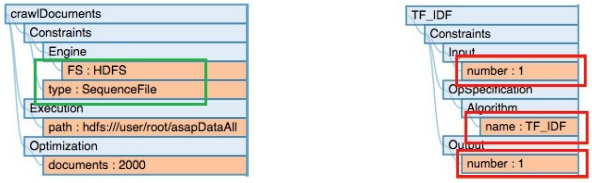
(e.g., execute on the right cluster size, etc.); if multiple ones are required, enhancements will relate both to single-engine optimization and to workflow management that decides on the best execution plan and data placement (e.g., first execute subtask A in Spark, store intermediate results in a NoSQL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



(a) (b)

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.

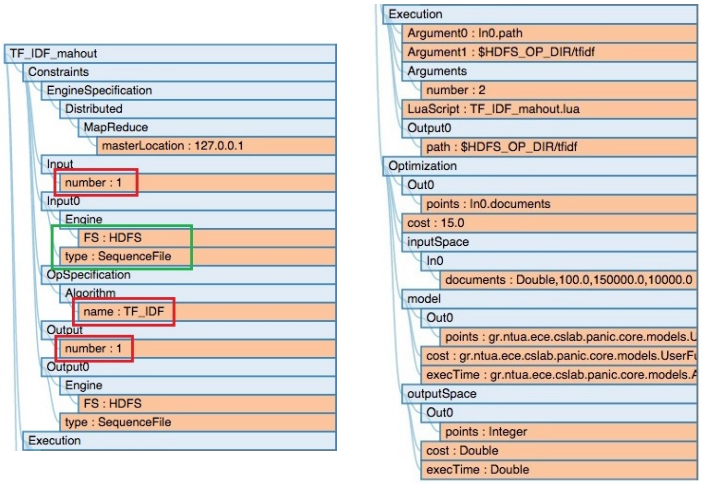


Figure 3: Meta-data description of a materialized tf-idf operator, implemented in mahout/Hadoop

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 ∈ 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 ∈ Operators following DAG topological ordering do
12     MOperators = findMaterializedOperators(o);
13     for mo ∈ MOperators do
14         inputCost = 0;
15         for in ∈ mo.getInputs() do
16             minCost = ∞;
17             for tin ∈ 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 ∈ 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

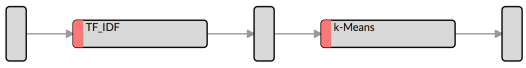


Figure 4: Abstract tf-idf, k-means workflow.

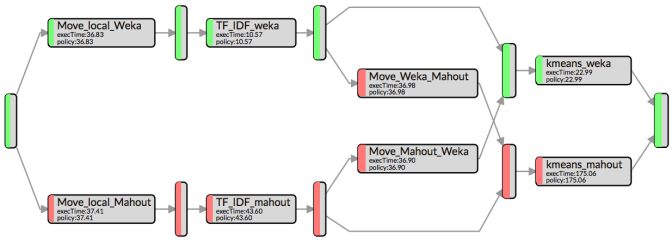


Figure 5: Materialized workflow and optimal plan.

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/

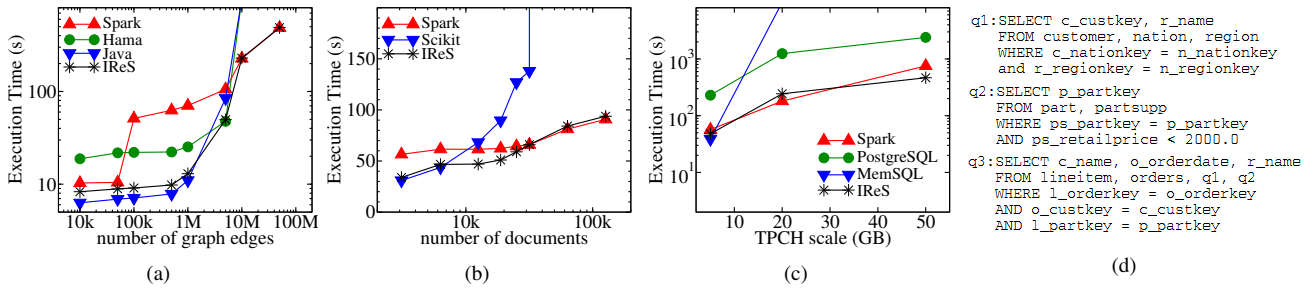


Figure 6: Execution times for the (a) graph, (b) text and (c) relational analytics workflows vs. various input sizes when running on single- and multi-engine (through IReS) environments. (d) The sql query of the relational analytics workflow.

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

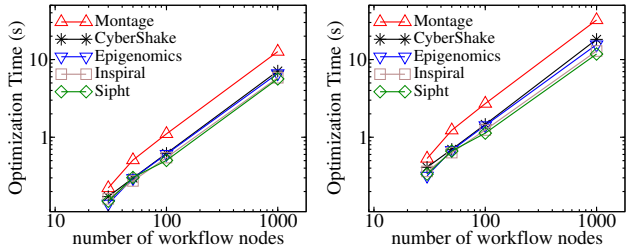


Figure 7: Workflow optimization times for 4 and 8 engines, using various workflow types of ranging size.

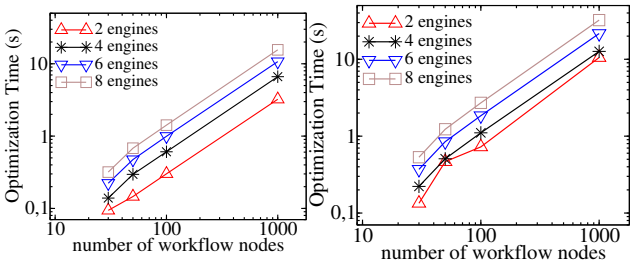


Figure 8: Workflow optimization times for Montage and Epigenomics graphs, using various number of engines and ranging the workflow size.

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.

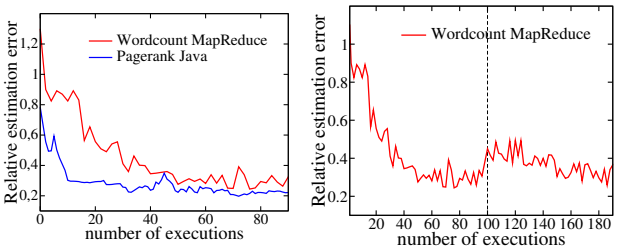


Figure 9: Relative execution time estimation error w.r.t. the number of executions (a) in normal IReS operation (b) when an infrastructure change occurs after 100 executions.

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

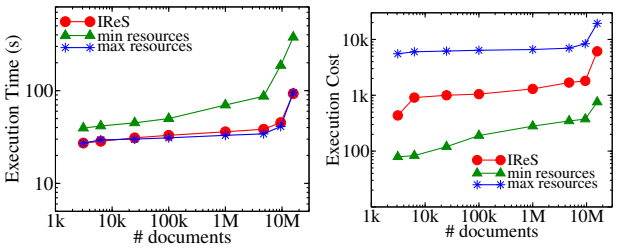


Figure 10: Execution time and cost vs. input size.

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.

MuSQLE: Distributed SQL Query Execution Over Multiple Engine Environments

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

***Abstract*—Multi-engine analytics has been gaining an increasing amount of attention from both the academic and the industrial community as it can successfully cope with the heterogeneity and complexity that the plethora of frameworks, technologies and requirements have brought forth. It is now common for a data analyst to combine data that resides on multiple and totally independent engines and perform complex analytics queries. Multi-engine solutions based on SQL can facilitate such efforts, as SQL is a popular standard that the majority of data-scientists understands. Existing solutions propose a middleware that centrally optimizes query execution for multiple engines. Yet, this approach requires manual integration of every primitive engine operator along with its cost model, rendering the process of adding new operators or engines highly inextensible. To address this issue we present MuSQLE, a system for SQL-based analytics over multi-engine environments. MuSQLE can efficiently utilize external SQL engines allowing for both intra and inter engine optimizations. Our framework adopts a novel API-based strategy. Instead of manual integration, MuSQLE specifies a generic API, used for the cost estimation and query execution, that needs to be implemented for each SQL engine endpoint. Our engine API is integrated with a state-of-the-art query optimizer, adding support for location-based, multi-engine query optimization and letting individual runtimes perform sub-query physical optimization. The derived multi-engine plans are executed using the Spark distributed execution framework. Our detailed experimental evaluation, integrating PostgreSQL, MemSQL and SparkSQL under MuSQLE, demonstrates its ability to accurately decide on the most suitable execution engine. MuSQLE can provide speedups of up to 1 order of magnitude for TPCH queries, leveraging different engines for the execution of individual query parts.**

***Keywords*-Polystore, Multi-Engine Optimization, SQL, Cost-models, Big Data**

I. Introduction

Big Data analytics constitutes a large fraction of modern datacenter workloads with numerous applications in most aspects of business and everyday life. Current methods and tools for Big Data analysis are quite diverse, being dictated by the heterogeneity of use cases that operate over different data formats, computational and functional requirements. This reality has brought forth an abundance of: i) computation languages and models (e.g., SQL, MapReduce, BSP, etc), ii) data store technologies (e.g., NoSQL stores [1], [2], columnstores [3], distributed FSs [4], etc), and iii) execution engines (e.g. [5], [6], [7], [8], etc).

While all these systems have been successful, they still showcase their advantages on a subset of analytics applications. Their striking limitation is that they require specific data formats and query inputs, being able to utilize only their custom execution engine. The need for a *multi-engine* approach, that splits and coordinates workflow execution among multiple engines has been recently recognized and is gaining increasing attention ([9], [10], [11], [12], [13]).

Regarding a standardized querying language, SQL, the mainstay query language for RDBMSs, is generally acknowledged as a top in-demand skill for this new era, with new platforms constantly seeking its support ([14], [15], [16], [13], [17]). SQL emerges as the de facto language for big data due to its extensibility, its ability to naturally represent queries that can be optimized, and the vast number of products and developers that already support it. These points highly suggest a multi-engine approach that allows SQL analytics over multiple data formats and uses the most appropriate engines.

Recent attempts along this line ([9], [18], [11]) tackle the issue by producing federated systems: There exists a custom subsystem where the different engines' operators and cost models are integrated and optimized planning is performed. While this is achieved for the already included systems and operators, this setup lacks extensibility as it requires a lot of manual work in order for one to incorporate a new analytics engine or support new operators on already integrated ones. Moreover, it frequently proves suboptimal to define a single, global optimization layer, and disregard the capabilities of the underlying engines to locally optimize.

In this work we present MuSQLE, an open-source framework[1] for high-performance SQL-based analytics over different data sources and execution engines. MuSQLE is able to overcome the aforementioned deficiencies and optimize simple or complex SQL queries. Our solution adopts a novel API-based strategy for integrating runtimes: Instead of manual integration, MuSQLE utilizes standardized, API-based cost-model and execution endpoints from the participating engines. Our system is able to optimally disseminate parts of the initial query (including the appropriate data movements between stores) using state-of-the-art planning

[1] https://github.com/gsvic/MuSQLE

and letting individual optimizers handle the respective subqueries. MuSQLE utilizes Spark [14] as an executor and orchestrator layer, extending its current functionality as well as providing it with a native cost-model.

In summary, our work makes the following contributions:

- We propose a generic SQL engine API that can facilitate multi-engine query optimization. The API is based on well-documented SQL functionality and can be implemented using generic, engine-agnostic interfaces like JDBC and ODBC.
- We integrate this API into a state-of-the-art query optimizer, allowing for external, multi-engine cost-based query optimization. Our optimizer runs on the logical level, allowing the connected engines to have full control of physical optimization and join execution. This approach avoids the detailed enumeration of all physical operators on the external optimizer and thus further facilitates the integration of a new SQL engine.
- We compile and utilize a cost model for the SparkSQL operators. This model is used within our query planner to achieve query optimization for SparkSQL.
- We present a fully functional system that integrates three popular engines: SparkSQL [14], PostgreSQL [19] and MemSQL [20]. We describe our system architecture and components in detail, as well as extensively evaluate the utility and efficiency of our scheme.
- Our detailed experimental evaluation showcases that MuSQLE can accurately decide on the most suitable execution engine and provides speedups of up to 1 order of magnitude for TPCH queries, leveraging different engines for the execution of individual query parts.

II. Related Work

We now present some of the most relevant approaches in multi engine analytics focusing on SQL-based solutions. We emphasize on the pros and cons of each approach, distinguishing between two categories: *Research works* and *Production-level systems*.

Production Systems: *SparkSQL* [21] provides a complete in-memory query execution engine using Resilient Distributed Datasets (RDDs), allowing remote data manipulation via DataFrames. For example, it is possible to query tables stored in an RDBMS or a main memory store (e.g., MemSQL [20]) as well as Parquet files in a HDFS cluster. The user can query these tables using traditional SQL via the SQLContext or by using methods of the DataFrame interface. However, in such a case, SparkSQL needs to fetch and distribute every external table into its worker nodes in order to perform data operations. As a result, optimization and processing capabilities of the external stores are ignored (e.g., index scans in case of filters).

PrestoDB [16] is a popular system for distributed, analytical query execution over heterogeneous datastores developed at Facebook. It provides a distributed execution model using similar algorithms with SparkSQL for querying data across multiple datastores by providing an engine-specific connector for each external system (e.g., Kafka, Cassandra, Hive, etc.) in order to be integrated to the core system. PrestoDB also needs to fetch each table involved in the query without pushing any operation to the underlying runtimes.

Apache Drill [22] enables SQL-based querying over unstructured, schema-free datastores (e.g., HDFS, MongoDB, Azure). However, Drill does not utilize individual cost models, statistics and estimates in the planning phase. This prevents it from executing one or more subqueries locally even if the local execution would be faster.

Research Works: BigDAWG [13] addresses the problem of query resolution over heterogeneous environments by executing queries using *islands of information*, where each island refers to a data model, a query language and a set of data management systems. In BigDAWG, a query is optimized using either *Single-* or *Multi-Island Planning*. While the system supports native subquery execution inside the underlying engines, it treats each engine as a black-box. As a result, local optimizers are ignored.

A different approach is followed by CloudMdsQL [23] which provides a functional SQL-like language for querying data stored in heterogeneous datastores within a single query, focusing on the ad-hoc usage of each datastore's native query language and engine. CloudMdsQL supports the local execution of a subquery and shipping of intermediate results. As it focuses on integration, it requires custom-made wrappers that translate from source to target language and provide cost models for the optimizer to use. Furthermore, the optimization is more rule-based, using selection and join condition pushdowns. The MuSQLE optimizer is more refined: Statistics injection, when moving an intermediate result into another store, is utilized for increased accuracy.

A different data integration approach is followed by *QUEPA* [24], which focuses on data integration over polystores. QUEPA introduces two new query methods: *Augmented search* and *augmented exploration*. In summary, this approach enables the user to query the polystore without knowing the exact structure of each individual database. Using record linkage, QUEPA will return records enriched with relevant (similar) tuples of other databases of the polystore. In *exploratory mode*, the user is prompted to select the relevant information retrieved that she wishes to explore. However, QUEPA only focuses on integration and does not provide any optimization for the execution phase.

SQL++ [17] provides a semi-structured data model, which combines the traditional SQL language with JSON extensions making it easy to query NoSQL databases by embedding JSON queries inside SQL code. However, SQL++ focuses only on the proposed language without providing query planning optimizations.

MISO [11] focuses on the tuning of the physical design of polystores in order to minimize data movements of

intermediate results between the underlying stores. MISO aims at optimizing the performance of ad-hoc, big data query processing by deciding where data is best to reside in. Yet, MISO also needs to maintain and calibrate its own cost functions for estimating the cost of operations. As a result, for every new engine to be added there is an integration overhead to generate the appropriate cost estimators.

III. Architecture

Figure 1 depicts MuSQLE's architecture. Our system is designed to facilitate the execution of multi-engine SQL queries. Such queries can be executed over tables that reside in multiple engines. The *Metastore* module is responsible for storing the schema and location information for each table. Our *SQL Parser* communicates with the *Metastore* in order to validate a user query and create the query graph. After parsing the query, our *Multi-engine Optimizer*, discussed in Section V, finds the optimal execution plan taking into account logical operator ordering, engine selection for query subgraphs as well as the required intermediate result movements. The generated execution plan is a tree of SQL and move operators. Each SQL operator is bound to a specific engine and refers to a subgraph of the initial query. The move operators handle the transfers of intermediate results between different SQL engines.

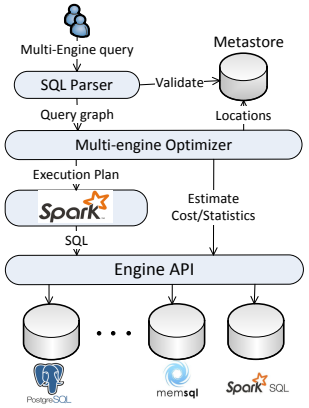


Figure 1: MuSQLE system architecture

In order for our optimizer to interact with the various engines, we introduce an *Engine API*, presented in Section IV. The API contains methods for the estimation of execution cost and intermediate result statistics which are used by our optimizer. It also contains SQL query execution methods as well as methods for retrieving and loading intermediate tables. We opt for utilizing Spark's engine as an execution framework for our multi-engine plans. Using Spark, MuSQLE provides scalable, main-memory based interaction between the connected SQL engines as well as primitives for fault-tolerance. Spark also provides a SQL interface allowing us to also utilize it as an alternative execution engine. We implement our engine API for three state-of-the-art engines: i) PostgreSQL, ii) MemSQL and iii) SparkSQL. The selected engines provide a diverse set, ranging from centralized to distributed execution, row- and column-oriented storage and disk-based to main-memory data indexing and join execution.

IV. Engine API

In this section, we describe in detail our *Engine API* that undertakes the integration between MuSQLE and the execution engine stack. To make our API generic and easy to implement, we devise five basic functions. Our functions are, in most cases, an extension of the already provided SQL functionality and require limited work to be implemented for a new engine. Our API functions are categorized in two groups: *Execution* and *Estimation*.

Execution functions: The following functions are used for executing our multi-engine query plans using Spark:

- `def execute(sqlQuery: String): DataFrame:`

This method sends a SQL query for execution to the specific engine. This is the most basic operation and can be implemented by extending well-known and massively used interfaces like JDBC and ODBC. The result of the query is loaded in a Spark DataFrame and can be subsequently moved to another engine for the execution of a query plan.

- `def loadTable(table: String, df: DataFrame): Unit:`

This method takes as argument a Spark DataFrame, which is an intermediate result table, and loads it in the specific engine. Again, this interface requires limited implementation overhead for most SQL engines.

Estimation functions: These functions are used inside our optimizer to estimate the execution cost and statistics of query subgraphs. In detail, the methods are:

- `def getStats(sql: String): Stats:`

This method is used to obtain an estimation of: i) the execution time of a specific SQL query and ii) the statistics of the result table. This functionality is already provided by many SQL engines in the context of the *EXPLAIN* statement. However, the results returned by different engines do not have a standard format. In most cases, they return the selected execution plan, the number of result rows as well as an execution cost measured in disk or cpu operations. To foster the integration of SQL engines in a multi-engine environment, we believe that such methods should follow a standard output format and return values that are comparable. Especially in the case of engines that use cost-based query optimization, such functionality already exists but is sometimes not exposed by the *EXPLAIN* output. To tackle this problem, we parse the output provided by the *EXPLAIN* statements of both PostgreSQL and MemSQL and use it to implement our API function. For Spark SQL, currently not utilizing a cost-based optimizer, we developed custom cost models for each operation as well as custom statistics, described in Section VI. Details on how we combine the

returned values in order to achieve unbiased query planning are presented in Section V-B.

- `def getLoadCost(stats: Stats): Double:`

This method returns an estimation of the time required to load a table from a Spark DataFrame to the specific engine. The statistics of the table, which contain its number of rows and columns are provided in the Statistics object. Cost functions similar to the following equation can be used for this task: $C_{move} = B_t \cdot T_{eng}$, where B_t is the size of the input table and T_{eng} is the transfer rate for the engine eng.

- `def injectStats(table: String, stats: Stats): Unit:`

This function is required when trying to obtain execution statistics for SQL queries that utilize intermediate results, not present in a specific engine. In a what-if optimization style, our optimizer injects all intermediate result statistics in the required engines, before calling the `getStats` method for queries that contain them. We analyze in detail how this is achieved by our optimizer in Section V. In brief, the `injectStats` method creates a "fake" table using the name and the statistics provided as argument. We have created custom code for doing this operation in all the integrated engines. However, assuming the multi-engine execution scenario is beneficial, such APIs can be easily implemented by the engine developers.

V. Multi-Engine Query Optimization

A. Optimization Algorithm

Finding the optimal join plan for complex queries has always been a major research challenge in optimizing database systems. One of the oldest and most efficient dynamic programming algorithms for join planning is *DPsize* [25], widely used in commercial databases like IBM's DB2. *DPsize* limits the search space to left-deep trees and generates plans in increasing order of size. A more recent approach, *DPccp* [26] and its variant *DPhyp* [27] are considered to be the most efficient, state-of-the-art dynamic programming algorithms for query optimization. They reduce the search space by examining connected subgraphs of the query in a bottom-up fashion. *DPccp* bases its enumeration procedure on finding all *csg-cmp-pairs* in the SQL join graph, where each table is represented by a vertex and join conditions are recorded using edges.

Definition 1: (csg-cmp-pair) Let $G = (V, E)$ be a join graph and S1, S2 two subsets of V such that S1 $\subseteq$ V and S2 $\subseteq$ (V $\setminus$ S1) are a connected subgraph and a connected complement respectively. If there further exists an edge (u, v)$\in$E such that u$\in$S1 and v$\in$S2, we call (S1, S2) a *csg-cmp-pair*.

In essence, *csg-cmp-pairs* are pairs that contain a connected subgraph (csg) of the query graph and one of its connected complement subgraphs (cmp). Each *csg-cmp-pair* corresponds to a 2-way join between the csg and the cmp graphs. This property ensures that the enumeration of all *csg-cmp-pairs* checks all possible 2-way join plans, ensuring the optimality of the selected plan. We extend the *DPhyp* algorithm in order to find the optimal join plan of a multi-engine SQL query. Our main extensions are:

Location-based optimization: One of the major differences of our algorithm compared to *DPhyp* is the structure of the dpTable used. To leverage the strengths of multiple engines, the optimizer needs to take into account the location of each intermediate result. A result with a certain location, while more expensive to generate than another, can be more efficiently used in a subsequent join leading to a better join plan for the query. To cover this case, we change the structure of our dynamic programming table: While in *DPhyp* only one plan is kept for each query subgraph, our dpTable maintains, for each query subgraph, a list of plans that contains the best join plan for each possible location (i.e., integrated SQL engine).

Multi-engine execution cost and statistics estimation: We integrate our generic SQL engine API with the *DPhyp* optimizer, allowing for the discovery of the optimal plan without depending on hand-coded or statically-integrated cost models. This approach separates the task of plan enumeration from the task of cost estimation, facilitating 'out of the box' integration and optimal utilization of new engines.

Algorithm 1 presents the pseudocode of our extended $emitCsgCmp(S_1, S_2)$ function. We maintain the same notation as used in [27], allowing interested readers to easily point to this paper for the details of the baseline algorithm.

ALGORITHM 1: $emitCsgCmp(S1, S2)$

```
1  plans_1 = dpTable[S_1];
2  plans_2 = dpTable[S_2];
3  S = S_1 ∪ S_2;
4  p = ⋀_{(u_1,u_2)∈E,u_i⊂S_i} P(u_1, u_2) ;
5  for (e_1, plan_1) ∈ plans_1 do
6      for (e_2, plan_2) ∈ plans_2 do
7          for e ∈ engines do
8              //execute query in engine e
9              c_1 = 0; c_2 = 0;
10             newPlan_1 = plan_1; newPlan_2 = plan_2;
11             if e_1 != e then
12                 table_1 = newTempTable();
13                 c_1 = e.getLoadCost(plan_1.stats);
14                 e.injectStats(table_1, plan_1.stats);
15                 newPlan_1 = move(newPlan_1, e, table_1);
16             if e_2 != e then
17                 table_2 = newTempTable();
18                 c_2 = e.getLoadCost(plan_2.stats);
19                 e.injectStats(table_2, plan_2.stats);
20                 newPlan_2 = move(newPlan_2, e, table_2);
21             newPlan = newPlan_1 ⋈_p newPlan_2;
22             sql = toSQL(newPlan, e);
23             stats = e.getStats(sql);
24             stats.cost = stats.cost + c_1 + c_2;
25             if dpTable[S][e] = ∅ ∨
                 stats.cost < dpTable[S][e].cost then
26                 dpTable[S][e] = newPlan;
```

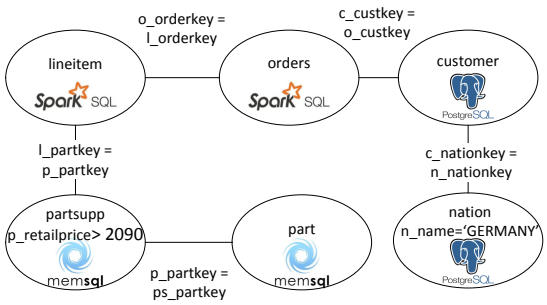


Figure 2: Join graph for Q_e

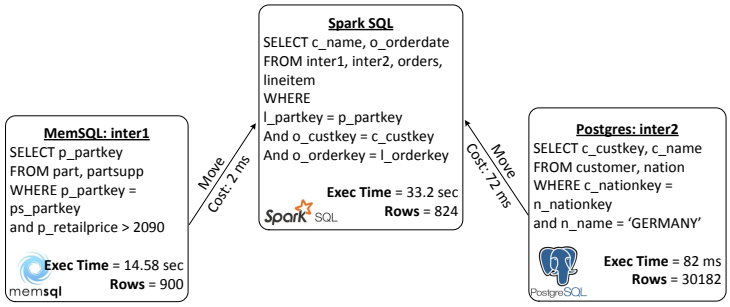


Figure 3: Multi-engine execution plan for Q_e

The $emitCsgCmp(S_1, S_2)$ function is called for each *csg-cmp-pair* and is responsible for checking the cost of the specific join using the optimal plans for S_1 and S_2 as well as the cost model.

As mentioned above, our dpTable is a two-dimensional array having one more dimension corresponding to the engine location of each intermediate result. Therefore, the $emitCsgCmp(S_1, S_2)$ function needs to evaluate the utilization of multiple plans, retrieved in lines 1-2, for both input query subgraphs. Additionally, we must check the cost of using these results in *all* the connected engines. Even if both intermediate results are not in a specific engine, loading them and continuing the query execution in that engine might be beneficial. To check all possible execution plans, we consider all combinations of left plans, right plans and execution engines (lines 5-26). For each of them, we check the locations of the left and the right plan, applying move operations if required. To apply move operators, we call the *getLoadCost* method for engine e in order to estimate the required time (lines 13 and 18). We also inject an intermediate temporary table, in engine e, using the API method *injectStatistics* (lines 14 and 19). This table will be used later on for query estimation.

After checking for move operations, we compose the result plan (line 21) and transform it to a SQL query string (line 22). The method to produce this string recursively iterates through the plan nodes stopping at move operators. This allows us to create a query string that refers to intermediate moved tables with their temporary names. The generated SQL query contains only the join information assigned for execution in engine e and is sent for estimation to the respective engine through the *getExecutionStats* API method. The dpTable record for $S = S_1 \cup S_2$, i.e., the resulting subgraph of the query, is updated only if the estimated cost is lower than the one contained in the dpTable for the specific combination of (S, e) (line 26).

To facilitate the understanding of our algorithm, we present a detailed query optimization example for the following query Q_e:

```
SELECT c_name, o_orderdate
FROM part, partsupp, lineitem, orders,
customer, nation WHERE
p_partkey = ps_partkey AND
c_nationkey = n_nationkey AND
l_partkey = p_partkey AND
o_custkey = c_custkey AND
o_orderkey = l_orderkey AND
p_retailprice > 2090 AND
n_name = 'GERMANY'
```

The above query is based on TPC-H data and returns all customers from Germany that ordered a part with retail price higher than 2090. For this example, we assume that the tables `lineitem` and `orders`, due to their large size, are stored in a Spark cluster, using HDFS files. Accordingly, the `customer` and `nation` tables are stored in a PostgreSQL server and the `part` and `partsupp` tables in a MemSQL cluster. The join graph of Q_e, along with the table location information is depicted in Figure 2. Our optimizer uses the *DPhyp* algorithm to enumerate all *csg-cmp-pairs* of the join graph. One possible *csg-cmp-pair* (S_1, S_2) is the one with $S_1 = \{part, partsupp\}$ and $S_2 = \{lineitem, orders\}$. When *DPhyp* calls the *emitCsgCmp* method for this pair, the optimal plans for both S_1 and S_2 can be found in the dpTable. Algorithm 1 checks all combinations of plans for S_1, S_2 and execution engines. This process results in selecting SparkSQL as the execution engine while moving the result of the optimal plan for S_1 from MemSQL to SparkSQL. In this case, the *toSQL* method (line 22 of Algorithm 1) refers to the intermediate result table coming from MemSQL with its temporary name "inter1" and therefore the remaining SQL query is:

```
SELECT o_custkey, o_orderdate
FROM inter1, lineitem, orders WHERE
l_partkey = p_partkey AND
o_orderkey = l_orderkey
```

This query is sent for estimation to the SparkSQL API. Its estimated cost is added to the move cost of "inter1" and the resulting plan is inserted in the dpTable for $S = \{part, partsupp, lineitem, orders\}$. Later on, the *csg-cmp-pair* (S'_1, S'_2) with $S'_1 = \{part, partsupp, lineitem, orders\}$ and $S'_2 = \{customer, nation\}$ is considered. This time *emitCsgCmp* selects SparkSQL as the execution engine while moving the plan of S'_2 from PostgreSQL to SparkSQL. The *toSQL* method

generates the following query, referring to both "inter1" and "inter2" coming from MemSQL and PostgreSQL:

```
SELECT c_name, o_orderdate
FROM inter1,inter2,lineitem, orders WHERE
l_partkey = p_partkey AND
o_custkey = c_custkey AND
o_orderkey = l_orderkey
```

Again, the estimation API of SparkSQL is called and the resulting plan is inserted to the dpTable, adding the required move cost. After the enumeration of all possible *csg-cmp-pairs*, the optimal plan of the query is discovered. The selected multi-engine plan is depicted in Figure 3 and consists of SQL and move operators. SQL operators are bound to specific engines and their estimated cost and number of results is depicted inside the respective boxes. The tree-based multi-engine plan is executed in a bottom-up fashion, using the Spark processing framework as well as our execution-related, engine API methods.

B. Comparing query estimations of different engines

Query optimization and execution time estimation are challenging tasks and are based on both cardinality estimations and operator cost models. In principle, as long as the cardinality estimations and the cost models of an engine are accurate, a good estimation of the execution time can be obtained for the query in hand. In reality, cardinality estimates are usually computed based on simplifying assumptions like uniformity and independence. Furthermore, cost model functions are oversimplified and do not take into account important parameters (e.g., the server load during query execution). In most cases, cost is measured in primitive operations like disk fetches or CPU cycles. While these measurements are considered to have linear correlation to the actual execution time, the correlation factors depend on hardware-specific parameters like the disk throughput or the CPU speed. A recent survey [28] illustrated that state-of-the-art SQL engines can easily misestimate costs by a factor of 1000 or more.

In this landscape, a major challenge for our system is how to compare and utilize the estimations provided by our user-implemented estimation APIs. To achieve an unbiased optimization procedure, we use the Metastore to record all query estimations, retrieved by the various APIs during query optimization. We also maintain, for each executed query, a detailed log of execution time for both the total query as well as it subqueries executed on different engines. This set of measurements is used to train machine learning models for tweaking the accuracy of the provided APIs. Our models target two challenges:

1) Transforming the costs measured using primitive operations (e.g., disk fetches) to estimations for the execution time. For example, the PostgreSQL EXPLAIN API returns the cost of the query in page fetches. Assuming a linear connection between the disk cost and the execution time, we use the previously described set of measurements to train a linear model that maps disk cost to execution time.

Symbol	Description
D_r	Cost of a single row read
D_w	Cost of a single row write
t_h	Cost of hashing a single value
t_{br}	Cost of broadcasting a single row
C_{cpu}	Cost of a single CPU comparison
$cores$	The number of cores in the cluster
$Part(s)$	The number of partitions of the relation s
$R(s)$	The number of rows of the relation s
S_p	spark.sql.shuffle.partitions

Table I: Cost Model Parameters

2) Due to inaccurate engine predictions or faulty API implementations, an engine can consistently fail to reasonably predict query execution time. To handle this case, we perform an accuracy analysis on top of all our query estimation measurements. This analysis computes the correlation between the estimated and the actual execution times for each engine. The computed correlation is used to adjust our confidence on a specific estimation API. Our optimizer uses a probability, proportionate to the measured correlation, to randomly discard the API estimation results. Therefore, in the case of an API that fails to achieve sufficient correlation to its actual execution times, the entire engine will be discarded from the optimization process.

VI. SparkSQL Cost-Based Query Optimization

Our engine API requires the implementation of cost estimations for each integrated SQL engine. While PostgreSQL and MemSQL provide such functionality through their internal cost-based optimizers, SparkSQL opts for heuristic-based optimization, without implementing cost models for its various operators. In this section, we present a cost model for SparkSQL which can be used to estimate the execution time of a SparkSQL physical plan. We also add support for injection of table statistics and utilize intermediate result cardinalities in order to provide more accurate execution time estimations.

Cost models: We have integrated a set of cost models into SparkSQL, extending the logic and formulas described in [29]. We present the cost models for three important operators: Sort-Merge Join, Broadcast-Hash Join and Exchange. Detailed formulas exist for all operators, yet we omit them due to space limitations. Table I presents the notation used in our cost models. First, we define the number of rounds required for an operation to run. The number of parallel tasks depends on how many CPUs a task uses. This can be set by modifying the `spark.task.cpus` parameter. For the rest of our discussion we assume that this parameter is set to 1. Thus, the number of tasks which can be run in parallel is equal to the number of cores in the cluster. The number of rounds required for a task of p partitions is thus:

$$Rounds(p) = \left\lceil \frac{p}{cores} \right\rceil$$

Exchange: This operation performs a shuffle operation on the data and partitions the results. The operation will be executed into $Part(s)$ tasks, where each task will process $R(s)/Part(s)$ rows. Each row will be hashed on the same column and will be sent to the corresponding partition according to the resulting hash value. The resulting cost is thus:

$$C_{exch}(s) = \frac{R(s)}{Part(s)} \cdot (C_{cpu} + D_w) \cdot Rounds(Part(s))$$

Broadcast-Hash Join: First, each row of the "small" table is hashed on the join condition attribute. This process takes place in the Spark driver node. Then, the hashed relation is broadcasted to all the workers. The cost of this operation equals: $C_{broadcast}(s) = R(s) \cdot (t_h + t_{br})$. After all nodes receive the hashed relation, a local join is performed for each partition of the large relation with the small relation. Thus, the total cost of a broadcast-hash join equals:

$$C_{bhj} = C_{broadcast}(s) + \frac{R(s) \cdot R(l) \cdot C_{cpu}}{Part(l)} \cdot Rounds(Part(l))$$

Sort-Merge Join: This is the SparkSQL distributed implementation of the traditional Sort-Merge Join algorithm. Before the actual join execution, each relation involved is first shuffled and sorted. Thus, for relation s the sorting cost equals: $C_{sort}(s) = R(s) \cdot \log R(s) \cdot C_{cpu} \cdot Rounds(s)$
After the two relations are sorted, they need to be merged. The merge cost is:
$C_{merge}(s,t) = R(s) \cdot R(t) \cdot Rounds(S_p) \cdot C_{cpu}$
Summing up, the cost of a sort-merge join is defined as:

$$C_{smj}(s,t) = C_{exch}(s) + C_{sort}(s) + C_{exch}(t) + C_{sort}(t) + C_{merge}(s,t)$$

VII. Statistics Injection

PostgresSQL Statistics Injection: In PostgreSQL, the $reltuples$ and $relpages$ columns of the pg_class system table represent the number of rows and pages respectively. Even if $relpages$ change manually, the planner checks the actual number of pages using the *RelationGetNumberOfBlocks()* method. Thus, in order to modify the statistics, we used *pg_dbms_stats*[2], an open source framework which provides functionality for "freezing" and modifying table statistics of PostgreSQL.

SparkSQL Statistics Injection: We extend SparkSQL, adding support for statistics injection and cardinality estimation. The existence of statistics lead to better cost estimations as well as to more accurate physical operator selection. For example, when planning queries that include tables stored in external data sources (e.g., MemSQL, PostgreSQL, etc), SparkSQL chooses explicitly the Sort-Merge Join algorithm even though the external table may be small. We solve this problem by *injecting* the input size of an external relation using the MuSQLE optimizer. Therefore, when SparkSQL is integrated with MuSQLE, it is able to adaptively select the use of *Broadcast Hash Join* for small external tables.

[2]https://github.com/ossc-db/pg_dbms_stats

VIII. Spark-Based Query Execution

After generating the multi-engine execution plan described in Section V, we need to execute this plan over a Spark cluster. SparkSQL uses Catalyst as its query optimizer. Thus, we need to transform our query plan into a SparkSQL equivalent using Catalyst's expressions. To do so, we developed a method which takes as input a multi-engine plan (i.e., see Figure 3) and transforms it into a SparkSQL native execution plan. We achieve that by traversing our plan in a bottom-up manner. At each leaf, using pattern matching, we match each operator of our plan (move, SQL) to the SparkSQL's Catalyst equivalent one. Finally, this method returns our execution plan represented in SparkSQL operations.

IX. Experimental Evaluation

In this section, we present a detailed evaluation of our multi-engine SQL platform. We integrate MuSQLE with a diverse set of engines, consisting of three state-of-the-art systems: PostgreSQL, MemSQL and SparkSQL. The selected engines excel on different and complementary data and query use cases, allowing MuSQLE to adaptively provide a combination of their advantages.

A. Cluster Setup and Compared Systems

All our experiments use Virtual Machine resources deployed on a private Openstack cluster, consisting of 8 VM containers. Each VM container has a 26-core Intel Xeon® CPU at 2.67GHz, 48 GB of RAM and two 2TB disks setup with RAID 0. Our experiments run on 13 Virtual Machines each with 8G RAM, 4 VCPUs and 100G of disk storage. The VMs are organized as follows:

- SparkSQL: We setup a Spark cluster consisting of 1 master and 5 worker VMs. The master VM runs the Spark driver(master) and the HDFS namenode. Each of the worker VMs runs a Spark worker and a HDFS datanode.
- PostgreSQL: One Virtual Machine is used for running a PostgreSQL server.
- MemSQL: We use a cluster of 1 master and 5 worker VMs. The master VM runs MemSQL's Master Aggregator while each worker runs a MemSQL leaf node.

There is currently limited work on the multi-engine SQL optimization landscape as mentioned in the related work section (code for the most relevant [11] and [13] was not available till the time of publication). To showcase the advantages of our system, we compare it against the performance of the three underlying SQL engines, if they functioned individually.

B. Datasets & Query Sets

Datasets: We evaluate MuSQLE using synthetic data generated from the popular TPCH benchmark [30]. In order to test the scalability of our system as well as its optimization

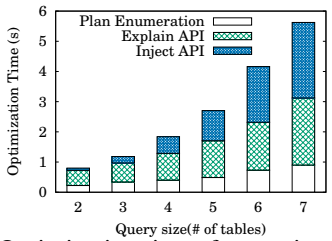


Figure 4: Optimization times for queries with variable sizes, using SparkSQL, PostgerSQL and MemSQL.

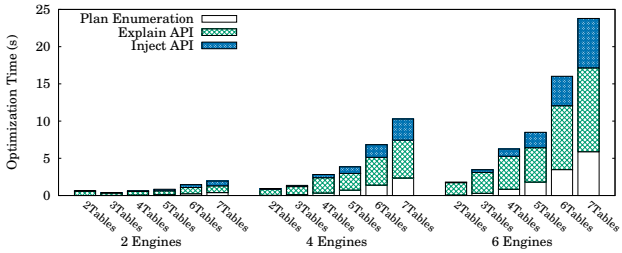
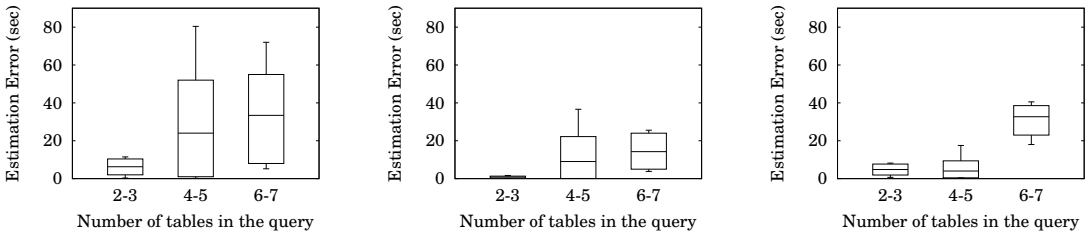


Figure 5: Optimization times for queries with variable sizes, using 2-6 fake SQL engine APIs.



(a) (b) (c)

Figure 6: Absolute estimation errors of execution time for (a) PostgreSQL, (b) MemSQL and (c) SparkSQL.

potential, we use three different TPCH scale sizes: 5GB, 20GB and 50GB.

Query sets: Our system focuses on SQL queries that include multiple tables residing in different engines. To present a detailed evaluation of such query scenarios, we generate a custom query set[3], extending the original TPCH benchmark queries. Our set consists of 18 queries, each one referenced as Qn, $0 \leq n \leq 17$. We classify our queries in two categories: i)**join-only** queries (Q0 - Q8) and ii)**join-filter** queries (Q9 - Q17). Queries of the first category contain multiple joins, producing large output sizes as they combine all information of the primitive tables without applying any filtering operation. In contrast, the second category includes queries with ranging selectivity, containing various filtering predicates. The selected categories can showcase the benefit of pushing the execution of subqueries to the individual engines. In brief, queries with large joins and small selectivity need to transfer large intermediate results between the connected engines and thus present small improvements. When queries have joins or subqueries with high selectivities, it is far more beneficial to push their execution to individual engines, transferring only their small intermediate results.

C. Query Optimization

The time required for optimizing queries that contain a variable number of tables, using our three integrated engines, is presented in Figure 4. As mentioned in Section V, our optimizer utilizes the engine APIs to estimate the cost and statistics of the various intermediate execution plans. External API calls can insert arbitrary large overheads on the optimizer's execution time. To measure this impact we break down the total optimization time into: 1) the plan enumeration time, 2) the time spent in external cost estimation APIs ("EXPLAIN API") and 3) the time spent on statistics injection ("INJECT API"). As depicted in Figure 4, we are able to find optimal plans for all our multi-engine queries within 6 seconds. However, the majority of the optimization time is spent on the external engine APIs. While the actual plan enumeration cost, introduced by our optimizer, is less than 1 sec for all queries, the total optimization time ranges between 1 and 6 seconds. This is largely attributed to the complexity of the external API implementations. For the purposes of this paper, we provided basic implementations for all engine APIs. However, the benefits of multi-engine execution can push individual engine experts to fine-tune the performance of the respective APIs.

To test the impact of adding a larger number of SQL engines on the optimization time, we simulate multiple engine API implementations. All methods of this API insert a delay, randomly selected from the distribution of delays of the actual engine API calls. Figure 5 presents the optimization times required for various numbers of connected engines. We note that the number of engines affects the performance of our optimizer. However, as presented in the following sections, this overhead is usually alleviated by the large improvements on the query execution times.

D. Cost Model Accuracy

Figure 6 presents the estimation accuracy for our integrated engines. We use a box plot in order to capture the average, standard deviation, min and max values of the error. As mentioned in Section V-B, we train regression models in order to translate the cost estimations of MemSQL and PostgreSQL into execution time estimations. As expected, the query estimation error increases with the query size, due to the propagation of erroneous cardinality and cost

[3]https://github.com/gsvic/MuSQLE/blob/master/Queries.scala

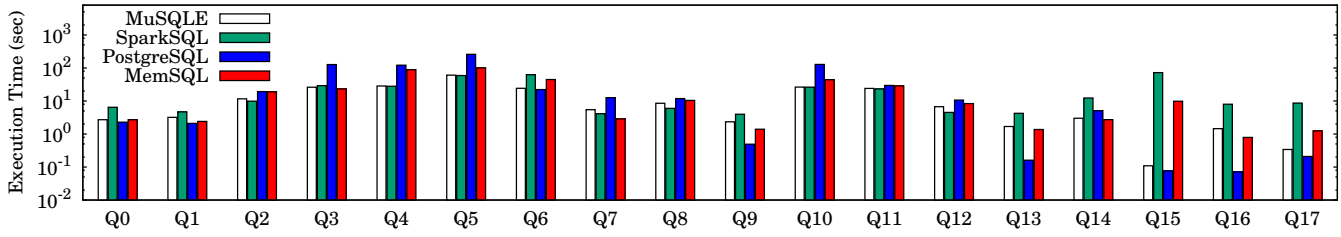


Figure 7: TPCH 5GB, all tables are stored in all engines

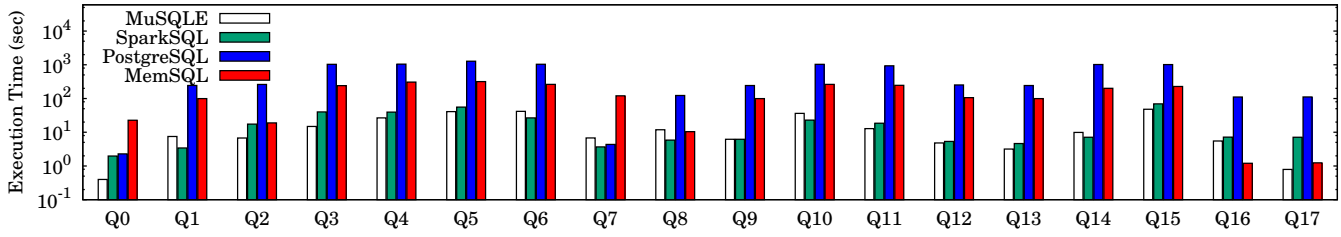


Figure 8: TPCH 5GB, each table is stored in a specific engine

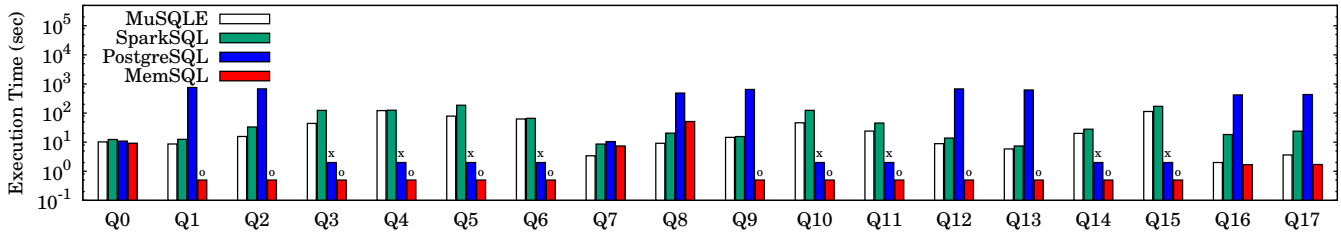


Figure 9: TPCH 20GB, each table is stored in a specific engine.
(**x**: execution time $>$ 20 minutes, **o**: out-of-memory)

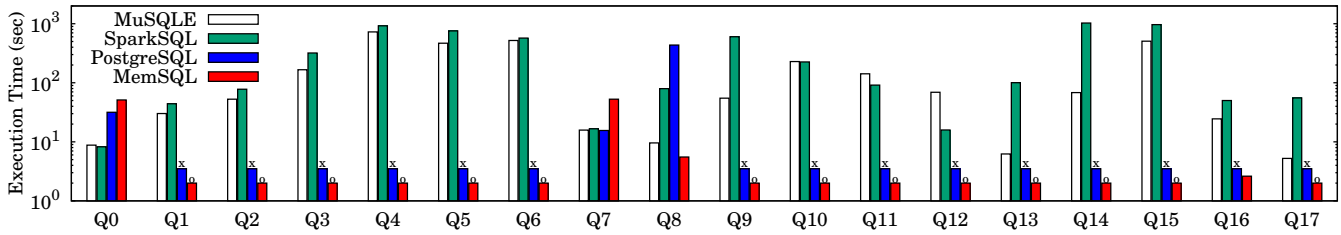


Figure 10: TPCH 50GB, each table is stored in a specific engine.
(**x**: execution time $>$ 20 minutes, **o**: out-of-memory)

estimations. However, we note that our proposed SparkSQL cost model, in conjunction with our proposed statistics, achieves good estimation accuracy.

E. Performance Comparison

To showcase MuSQLE's multi-engine benefits, we run the experiments under the following scenarios:

All tables stored in all engines: In this case, we assume that all tables are stored in all connected engines. Due to the existence of data in all engines, MuSQLE cannot achieve significant performance improvements with respect to the best underlying engine. However, this scenario nicely demonstrates the accuracy of our optimizer in such cases. As presented in Figure 7, our optimizer manages to select the best execution engine for most query cases. For $Q9$, $Q13$ and $Q16$ wrong cost estimates lead to sub-optimal execution plans.

Different Table Locations: In most real-life multi-engine query scenarios, tables will be stored in different engines according to their characteristics. For example, a table with high frequency of updates would be stored in an OLTP database, while a large log table in a plain HDFS file. To test this scenario, we select the following location for the TPCH tables: PostgreSQL stores the small sized tables (customer, nation, region). MemSQL stores the medium sized ones (part, partsupp, supplier), while the larger tables (lineitem, orders) are stored in HDFS.

TPCH 5GB: Figure 8 depicts the respective results for the TPCH 5 GB dataset. The performance improvement for this dataset is not significant due to the small amount of input data. This suggests that, in most cases, the optimal execution plan is to move the tables and execute the whole query in a single engine in order to prevent the intermediate result data movements. However, we note that our system, correctly

estimating the execution times for the different engines, selects the most profitable. For example, the execution of Q12 takes place in PostgreSQL while query Q17 runs in MemSQL. Again, this experiment showcases our optimizer's decision accuracy.

TPCH (20, 50)GB: For larger dataset sizes, all individual engines incur significant overheads when loading external tables. In such cases, plans that perform local processing inside the individual engines while moving small sized intermediate results prove largely beneficial. Specifically, Figures 9 and 10 illustrate that most of the queries could not be completed in MemSQL due to the large intermediate results which lead to an out-of-memory error. In the case of PostgreSQL, it took more than 2000 sec to complete the execution of several queries, requiring over twenty minutes to fetch the external tables. SparkSQL, taking advantage of its distributed execution engine, succeeds in handling all the queries of this scale. However, MuSQLE not only manages to select the most efficient execution engine, but also achieves better response times than SparkSQL by pushing local processing on the other engines. For example, $Q14$ contains one filter on $lineitem$ table. MuSQLE's execution plan pushes a subquery containing the filter into SparkSQL. The small sized intermediate results are then moved in MemSQL, where they are joined with the smaller $part$ and $partsupp$ tables. Similar improvements can be also observed for queries $Q13$, $Q15$, $Q16$ and $Q17$, which represent the cases that MuSQLE outperforms even the best individual engine, resulting in up to one order of magnitude speedups.

X. Conclusions

In this paper we presented MuSQLE, a multi-engine SQL executor. MuSQLE introduces a generic engine API that needs to be implemented for each integrated engine. We extend a state-of-the-art query optimizer, adding support for location based optimization and individual engine cost estimation. We have integrated MuSQLE with PostgreSQL, MemSQL and SparkSQL. Our detailed experimental evaluation proves that MuSQLE can accurately select the best execution engine for a large set of queries and provides speedups of up to 1 order of magnitude, leveraging different engines for the execution of individual query parts.

Acknowledgments

This work has been partly 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.

References

[1] "Apache HBase," http://hbase.apache.org/.
[2] "neo4j," http://neo4j.com/.
[3] "monetdb," https://www.monetdb.org/.
[4] "Hadoop Distributed File System," http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[5] "Apache Hadoop," http://hadoop.apache.org/.
[6] "Apache Spark," https://spark.apache.org/.
[7] "Apache Hama," https://hama.apache.org/.
[8] "Apache Flink," https://flink.apache.org/.
[9] A. Simitsis, K. Wilkinson, U. Dayal, and M. Hsu, "HFMS: Managing the Lifecycle and Complexity of Hybrid Analytic Data Flows," in *ICDE*. IEEE, 2013.
[10] D. DeWitt, A. Halverson, R. Nehme *et al.*, "Split query processing in polybase," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013.
[11] J. LeFevre, J. Sankaranarayanan, H. Hacigumus *et al.*, "MISO: Souping up big data query processing with a multistore system," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2014.
[12] K. Doka, N. Papailiou, D. Tsoumakos, C. Mantas, and N. Koziris, "IReS: Intelligent, Multi-Engine Resource Scheduler for Big Data Analytics Workflows," in *SIGMOD*, 2015.
[13] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik, "The bigdawg polystore system," *ACM Sigmod Record*, 2015.
[14] "Spark SQL," https://spark.apache.org/sql/.
[15] "Apache Impala," http://impala.io/.
[16] "Presto," http://www.teradata.com/Presto.
[17] 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*, 2014.
[18] A. Simitsis, K. Wilkinson, and P. Jovanovic, "xPAD: A Platform for Analytic Data Flows," in *SIGMOD 2013*.
[19] "PostgreSQL," https://www.postgresql.org/.
[20] "MemSQL," http://www.memsql.com/.
[21] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark SQL: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015.
[22] "Apache Drill," https://drill.apache.org/.
[23] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, and J. Pereira, "CloudMdsQL: querying heterogeneous cloud data stores with a common language," *Distributed and Parallel Databases*, 2015.
[24] A. Maccioni, E. Basili, and R. Torlone, "QUEPA: Querying and exploring a polystore by augmentation."
[25] P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang, "Query optimization in the IBM DB2 family," *IEEE Data Eng. Bull.*, 1993.
[26] G. Moerkotte and T. Neumann, "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products," in *VLDB*, 2006.
[27] G. Moerkotte and T. Neumann, "Dynamic programming strikes back," in *SIGMOD*, 2008.
[28] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, 2015.
[29] D. Taniar, C. H. Leung, W. Rahayu, and S. Goel, *High performance parallel database processing and grid databases*. John Wiley & Sons, 2008.
[30] T. P. P. Council, "TPC-H benchmark specification," *Published at http://www. tcp. org/hspec. html*, 2008.

FP7 Project ASAP
Adaptable Scalable Analytics Platform

End of ASAP D3.3
IReS Platform v.2

WP 3 – Intelligent, Multi-engine Resource Scheduling Platform

Nature: Report

Dissemination: Public