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

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

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

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

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

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

## ABSTRACT

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

## Categories and Subject Descriptors

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

## Keywords

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

## 1. INTRODUCTION

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

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

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

Sensing this trend, cloud software companies now offer software distributions in pre-cooked VM images or as a service. These distributions incorporate different processing frameworks, data stores and libraries to alleviate the burden of multiple installations and configurations (e.g., [5, 9, 8, 12]). Yet, such multi-engine environments lack a *meta-scheduler* that could automatically match tasks to the right engine(s) according to multiple criteria, deploy and run them without manual intervention. A recent attempt along this line [25, 26] focuses more on lower-level database operators, emphasizing on their automatic translation from/to specific engines via an XML-based language. Yet, this is a proprietary tool with limited applicability and extension possibilities for the community.

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

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

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

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

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

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

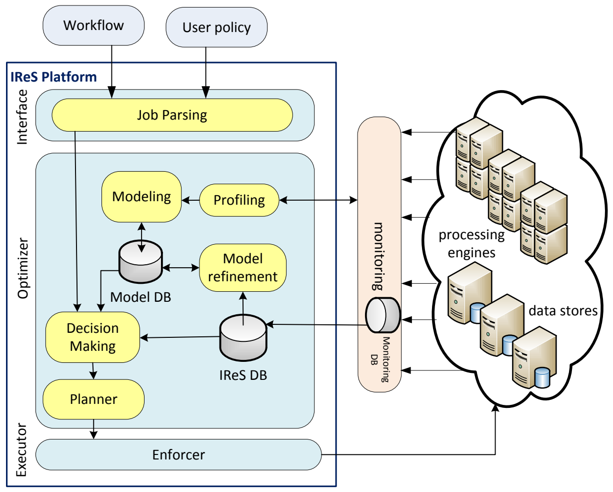


**Figure 1: Architecture of the IReS platform**

## 2. ARCHITECTURE

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

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

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

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

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

**Figure 2: Metadata description of the abstract join operator**

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

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

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

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

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

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

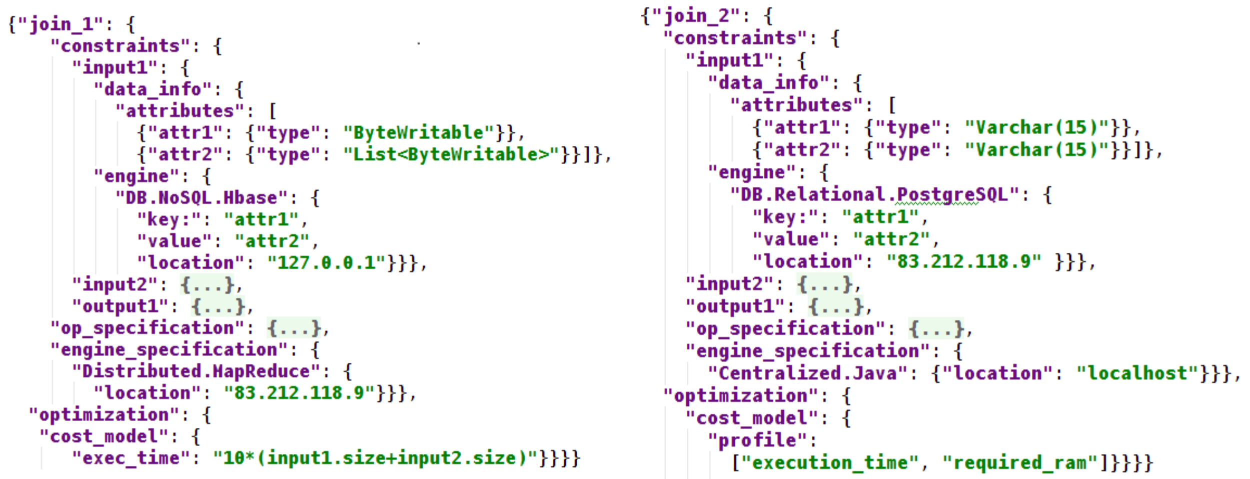
Our proposed metadata framework describes *data* and *operators*. Data and operators can be either *abstract* or *materialized*. Abstract are the operators and datasets that are described partially or in a high level by the user when composing her workflow whereas materialized are the actual operator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

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

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

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

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



```
{"join_1": {
    "constraints": {
      "input1": {
        "data_info": {
          "attributes": [
            {"attr1": {"type": "ByteWritable"}},
            {"attr2": {"type": "List<ByteWritable>"}}]},
        "engine": {
          "DB.NoSQL.Hbase": {
            "key:": "attr1",
            "value": "attr2",
            "location": "127.0.0.1"}}},
      "input2": {...},
      "output1": {...},
    "op_specification": {...},
    "engine_specification": {
      "Distributed.MapReduce": {
        "location": "83.212.118.9"}}},
  "optimization": {
   "cost_model": {
      "exec_time": "10*(input1.size+input2.size)"}}}}
```

(a)

```
{"join_2": {
  "constraints": {
    "input1": {
      "data_info": {
        "attributes": [
          {"attr1": {"type": "Varchar(15)"}},
          {"attr2": {"type": "Varchar(15)"}}]},
      "engine": {
        "DB.Relational.PostgreSQL": {
          "key:": "attr1",
          "value": "attr2",
          "location": "83.212.118.9" }}},
    "input2": {...},
    "output1": {...},
    "op_specification": {...},
    "engine_specification": {
      "Centralized.Java": {"location": "localhost"}}},
  "optimization": {
    "cost_model": {
      "profile":
        ["execution_time", "required_ram"]}}}}
```

(b)

**Figure 3: Metadata descriptions of the two materialized join operators**

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

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

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

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

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

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

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

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

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

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

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

## 3. DEMONSTRATION DESCRIPTION

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

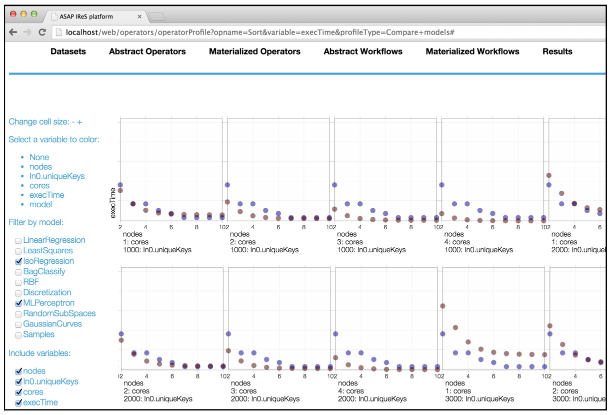
**Workflows and Datasets:** The users will have the opportunity to test the IReS platform either using one of four predefined workflows or assembling their own, using operators from the ASAP operator library. A diverse set of operations of varying complexity and execution parameters is covered including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

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

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

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

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



localhost/web/operators/operatorProfile?opname=Sort&variable=execTime&profileType=Compare+models#

Change cell size: - +

Select a variable to color:

- None
- nodes
- In0.uniqueKeys
- cores
- execTime
- model

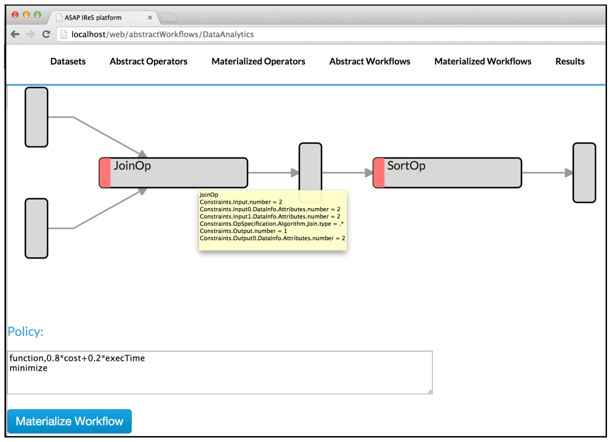
Filter by model:

- [ ] LinearRegression
- [ ] LeastSquares
- [x] IsoRegression
- [ ] BagClassify
- [ ] RBF
- [ ] Discretization
- [x] MLPerceptron
- [ ] RandomSubSpaces
- [ ] GaussianCurves
- [ ] Samples

Include variables:

- [x] nodes
- [x] In0.uniqueKeys
- [x] cores
- [x] execTime

**Figure 4: IReS web application GUI - Materialized Operator models**



localhost/web/abstractWorkflows/DataAnalytics

```
JoinOp
Constraints.Input.number = 2
Constraints.Input0.DataInfo.Attributes.number = 2
Constraints.Input1.DataInfo.Attributes.number = 2
Constraints.OpSpecification.Algorithm.Join.type = .*
Constraints.Output.number = 1
Constraints.Output0.DataInfo.Attributes.number = 2
```

Policy:

```
function,0.8*cost+0.2*execTime
minimize
```

**Figure 5: IReS web application GUI - Abstract Workflow**

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

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

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

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

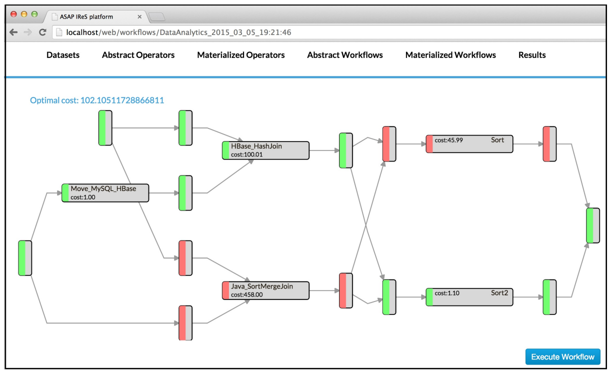


**Figure 6: IReS web application GUI - Materialized Workflow tab**

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

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

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

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

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

## 4. ACKNOWLEDGEMENTS

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

## 5. REFERENCES

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