Multi-Engine Analytics with IReS

Katerina Doka[1], Ioannis Mytilinis[1], Nikolaos Papailiou[1], Victor Giannakouris[1], Dimitrios Tsoumakos[2], and Nectarios Koziris[1]

[1] Computing Systems Laboratory, National Technical University of Athens, Greece.
{katerina, gmytil, npapa, vgian, nkoziris}@cslab.ntua.gr
[2] Department of Informatics, Ionian University, Corfu, Greece.
dtsouma@ionio.gr

Abstract. We present *IReS*, the *Intelligent Resource Scheduler* that is able to abstractly describe, optimize and execute any batch analytics workflow with respect to a multi-objective policy. Relying on cost and performance models of the required tasks over the available platforms, IReS allocates distinct workflow parts to the most advantageous execution and/or storage engine among the available ones and decides on the exact amount of resources provisioned. Moreover, IReS efficiently adapts to the current cluster/engine conditions and recovers from failures by effectively monitoring the workflow execution in real-time. Our current prototype has been tested in a plethora of business driven and synthetic workflows, proving its potential of yielding significant gains in cost and performance compared to statically scheduled, single-engine executions. IReS incurs only marginal overhead to the workflow execution performance, managing to discover an approximate pareto-optimal set of execution plans within a few seconds.

1 Introduction

Over the last two decades, a plethora of diverse execution engines and datastores, both centralized and distributed, have emerged to cope with the challenges posed by the volume, velocity and variety of Big Data and the analysis thereof (e.g., [2, 3, 1, 8], etc.). In the course of time, Big Data analytics platforms become faster, more efficient and sophisticated, but also more specialized, excelling at certain types of data and processing tasks. Moreover, although many approaches in the relevant literature manage to optimize the performance of single engines by automatically tuning a number of configuration parameters (e.g., [26, 31]), they bind their efficacy to specific data formats and workloads.

However, there is no one platform to rule them all: No single execution model is suitable for all types of tasks and no single data model is suitable for all types of data. Thus, the task of architecting an analytics environment that best suits specific scientific or business needs can be frustrating to prospective users. Especially when having to orchestrate long and complex processing pipelines that contain diverse operators and crunch multiple data formats from various sources, analysts can easily go down the wrong technology path. Modern business logic and scientific simulations are encoded into workflows that include a multitude of diverse tasks. Such tasks range from simple Select-Project-Join (SPJ) and

data movement to complex NLP-, graph- or custom business-related tasks over a variety of data formats and origins, such as relational data from a legacy DBMS, key-value pairs from a NoSQL cluster, graph data, etc. To add insult to injury, such workflow execution may involve multiple and often conflicting constraints, including time, cost, resource utilization and other execution aspects.

To help address such challenges, many organizations are deploying multiple platforms to handle different parts of their data and workflows (e.g., Facebook [12], Twitter [11], Uber [27], etc.). Even cloud vendors currently offer software solutions that incorporate a multitude of processing frameworks, data stores and libraries to alleviate the burden of managing multiple installations and configurations (e.g., [5, 6, 10]). The new paradigm of *multi-engine analytics* [36] has recently been proposed as a promising solution that can abstract away the specifics of the underlying platforms and hide the details of how and where each task is executed. This approach aims to unify runtimes and storage backends and promote a declarative way of specifying and executing processing tasks on arbitrary datasets. One of the most compelling, yet daunting challenges in a multi-engine environment is the design and creation of a *meta-scheduler* that automatically allocates workflow subtasks to the right engine(s) according to multiple criteria, deploys and runs them without manual intervention.

Related work mostly revolves around storage engines. Traditional data federation approaches (e.g., [32, 35]) solely focus on SQL analytics, while modern ones [28] consider data lake scenarios where historical data stored in HDFS are combined with operational data residing in OLTP systems or NoSQL stores. Polystores, on the other hand, migrate data across various data stores, creating additional load (e.g., [16, 21, 38]). Very few recent approaches exist for both data and execution engines, which are either proprietary tools with limited applicability and extension possibilities for the community (e.g., [34]) or focus more on the translation of scripts from one engine to another, being thus tied to specific programming languages and engines (e.g., [23, 13]). Contrarily, we would ideally opt for a solution that i) adopts a declarative approach for expressing workflows, ii) is able to accommodate new platforms, implementation languages and tools as they emerge or as business needs change, iii) supports multiple optimization goals, iv) executes workflows in a fail-safe manner and v) is open-source.

To this end, we design and implement *IReS* [19, 20], the open-source *Intelligent Multi-Engine Resource Scheduler*[3] that acts as an "umbrella" for multiple execution engines and datastores, allowing for their seamless integration in the context of a single analytics workflow. IReS handles the whole life cycle of a workflow, from its declarative description to its optimization, planning, execution and monitoring in a transparent to the user way, masking the specifics of the underlying platforms.

Given a high-level description of the analytics tasks and data at hand using an extensible meta-data framework, IReS is able to optimize, schedule, execute and monitor any workflow that contains them. First, IReS models the cost and performance characteristics of the required tasks over the available platforms in

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

an offline manner. Based on the produced, incrementally updatable models, the planner of IReS is able to map distinct parts of a workflow to the most advantageous store and execution engine and decide on the exact amount of resources provisioned in order to optimize any multi-objective, user-defined policy. The resulting optimization is orthogonal to (and in fact enhanced by) any optimization effort within an engine. Finally, IReS executes the optimized workflow, efficiently adapting to the current cluster and engine conditions and recover from failures by effectively monitoring the workflow execution in real-time.

In this paper we thoroughly present IReS and its internals, delving into the design and implementation details of its modules. Our key contributions are:

- A multi-engine planner that selects the most prominent workflow execution plan among existing runtimes, datastores and operators and elastically provisions the correct amount of resources, based on cost and performance estimations of the various operators over the available platforms.
- A modelling methodology that provides performance and cost estimations of the available analytics operators for different engine configurations. The resulting models are utilized by the planner for multi-engine workflow optimization.
- An execution layer that enforces and monitors the selected multi-engine execution plan, allowing for fine grained resource control and fault tolerance.
- An extensible meta-data description framework for operators and data, which allows for declarative workflow description and automatic discovery of all alternative execution paths.
- An extensive evaluation of our open-source prototype operating over both real-life and synthetic workflows. The results prove that IReS is able to i) efficiently decide on the best execution plan based on the optimization policy and the available engines within a few seconds, even for large-scale workflow graphs and multiple optimization objectives, ii) tolerate engine and hardware failures and, most importantly, i) speed-up the fastest single-engine workflow executions by up to 30% by exploiting multiple engines.

2 IReS Architecture

IReS follows a modular architecture, as depicted in Figure 1. IReS comprises three layers, the *Interface*, which serves as the point of interaction with the outside world, the *Optimizer*, which constructs the execution plan that best serves the current business objectives and the *Executor*, which enforces it. 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 of the platform.

2.1 Interface Layer

The Interface layer is responsible for handling the interaction between IReS and its users. It allows users to declaratively describe execution artefacts such as operators, data and workflows, along with their inter-dependencies, properties and restrictions using a unified description framework. It thus enables the user to focus on *what* she wants to achieve rather than *how* to achieve it, abstracting

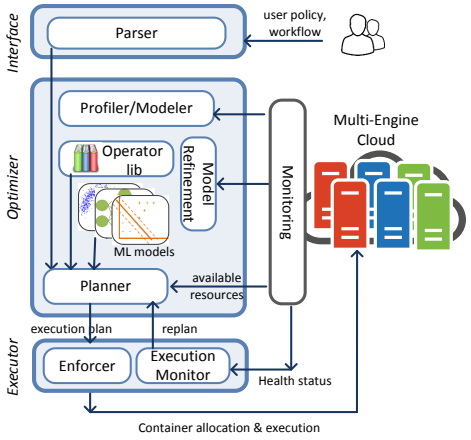

Fig. 1: Architecture of the IReS platform.

away the details and specifics of underlying platforms. The user-provided workflow is parsed as a dependency graph by the *Parser* and is handed over to the IReS Optimizer along with the user-defined optimization policy.

The IReS description framework sets the guidelines for recording all the information necessary to define and plan a workflow. The two core entities of the framework are *data* and *operators*, which are connected in directed graphs to form workflows. Both data and operators need to be accompanied by a set of metadata, 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. For instance, an abstract operator could simply signify that tf-if is applied on one input dataset and provides one output dataset, while a materialized operator is tied to a specific implementation of tf-idf in Spark/MLlib. Materialized operators, along with their descriptions, are stored in the *Operator library* (Figure 1). The role of IReS is to eventually map the abstract operators and datasets to materialized ones based on the compliance of their metadata, in order to provide a tangible execution plan.

The metadata that describe operators, such as input types, execution parameters, etc., and data, such as schemata, location of objects, etc., are organized in a generic tree format. To allow for flexibility, only the first levels of the metadata tree are pre-defined. Users can add their ad-hoc subtrees to define custom data or operator properties. Some fields, mostly the ones related to operator and data requirements, such as the number of inputs/outputs of an operator or the format of a dataset, are *compulsory*, since the workflow planning and execution is impossible without them. The rest, e.g., cost models, statistics, etc. are *optional*: They either serve as extra matching fields for more fine-grained execution planning or they provide additional hints to facilitate the adherence to the workflow optimization policy. Most metadata fields of abstract data and operators are optional, to allow for any desired level of abstraction. Moreover, they support regular expressions. In general, we pre-define the following metadata fields:

```
#Reviews
Constraints.Engine.FS=HDFS
Constraints.type=text
Execution.path=hdfs:///user/ires/data
Optimization.documents=2000
```

```
#tf-idf
Constraints.Input.number=1
Constraints.Output.number=1
Constraints.OpSpecification.Algo=tf-idf
```

Fig. 2: Metadata description of (a) the dataset and (b) an abstract tf-idf operator.

Constraints , which contain the information required to discover all possible execution plans that correspond to the abstractly defined workflow. This discovery entails the matching of (a) abstract operators to materialized ones and (b) data to operators in the corresponding metadata fields. Mandatory fields include specifications of operator inputs/outputs, algorithms implemented and underlying engines.
Execution , which provides (a) the execution parameters of a materialized operator, such as the path of its executable or the required execution arguments, and (b) access info of a materialized dataset, such as its path.
Optimization ,which is optional and holds additional information that assists in the optimization of the workflow. This information may include a performance/cost function provided by the developer of a materialized operator or dataset statistics (e.g., number of documents in a dataset) provided by its owner. In case a performance/cost function is not a priori known, optimization metadata fields provide IReS with instructions on how to create one by profiling over specific metrics, such as execution time, required RAM, etc. More details on the profiling process can be found in Section 2.2.

As a motivating example, imagine an e-commerce website that decides to summarize reviews for a specific product, i.e., perform tf-idf over the corpus of the product reviews, followed by a k-means clustering. First, the input dataset, `Reviews`, needs to be described as depicted in Figure 2(a): It is a text file stored in HDFS, as recorded in the `Constraints.type` and `Constraints.Engine.FS` fields respectively, following the path specified by the `Execution.path` field. The information under `Optimization` tracks down the number of documents contained in the dataset. This information will be used during the planning phase (see Section 2.2) to help obtain more accurate performance and cost estimations for the operators using `Reviews` as input. Then, we need to specify the operations to be performed. In its most abstract form, the `tf-idf` operator (see Figure 2(b)) just needs to define the number of input/output parameters as well as the implemented algorithm (under `OpSpecification.Algo`). The same holds for the `k-means` operator.

We assume that the IReS operator library already contains two materialized tf-idf operators, one in Python/scikit and one in Spark/MLlib, provided by the company developers. The metadata of the materialized operators include all information required in order to execute the operations on an engine. For instance, the description of the `tf-idf_spark` materialized operator in Figure 3(a) states that this is a Spark implementation (l. 8) of the tf-idf algorithm (l. 7) that reads its input from HDFS (l. 3) in the form of a text file (l. 4) and outputs its results in HDFS (l. 6). The operator is executed via a shell script (l. 12), having the path of the input file as an execution argument (l. 10). The `Optimization` metadata designate time and cost as possible optimization objectives of the spe-

```
1  #tf-idf_spark
2  Constraints.Input.number=1
3  Constraints.Input0.Engine.FS=HDFS
4  Constraints.Input0.type=text
5  Constraints.Output.number=1
6  Constraints.Output0.Engine.FS=HDFS
7  Constraints.OpSpecification.Algo=tfidf
8  Constraints.Engine=Spark
9  Execution.Arguments.number=1
10 Execution.Argument0=In0.path
11 Execution.Output0.path=$HDFS_OP_DIR/tfidf
12 Execution.script=tf-idf_spark.sh
13 Optimization.model.execTime=AbstractWekaModel
14 Optimization.inSpace.In0.documents=Int,1000,10000,50000
15 Optimization.inSpace.cores=Double,2.0, 4.0, 16.0
16 Optimization.model.cost=UserFunction
17 Optimization.cost=In0.documents*4096.0
```

```
1  #tf-idf_scikit
2  Constraints.Input.number=1
3  Constraints.Input0.Engine.FS=localFS
4  Constraints.Input0.type=text
5  Constraints.Output.number=1
6  Constraints.Output0.Engine.FS=localFS
7  Constraints.OpSpecification.Algo=tfidf
8  Constraints.Engine=Python
9  Execution.Arguments.number=...
```

Fig. 3: Metadata description of tf-idf operator in (a) Spark/MLlib and (b) python.

cific operator (l. 13, l. 16). Execution time estimations are provided by a model (l. 13), which is constructed by the IReS profiler/modeler as instructed in l. 14-15, through a process which will be described subsequently, while cost estimations are given by a developer-provided function (l. 16-17). The `tf-idf_scikit` materialized operator is described similarly (see Figure 3), indicating Python as the implementation engine (l. 8) and the local filesystem as the input source (l. 3).

To discover the actual tf-idf implementations that match the abstract `tf-idf` operator and comply with the `Reviews` dataset, we employ a tree matching algorithm to ensure that all compulsory fields match. This is performed during planning and optimization, described subsequently. In our example, both `tf-idf_scikit` and `tf-idf_spark` match `tf-idf` in the fields designated in blue. Moreover, the `Reviews` dataset can be directly used as input to `tf-idf_spark`, as the matched metadata fields in red suggest. Thus, `tf-idf_spark` is considered when constructing the optimized execution plan. `tf-idf_scikit` cannot operate on `Reviews` as is, since the fields in red do not match: `tf-idf_scikit` reads from the local filesystem while `Reviews` is stored in HDFS. However, we will see in the next Section how we can circumvent this incompatibility.

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 execution plan that best satisfies the - possibly multiple - user objectives 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 tasks that reside within the IReS *Operator library*. To that end, each operator is modeled and the corresponding models are stored in the IReS *ML models library*. The initial model of an operator results from the offline profiling of it using the *Profiler/Modeler* module, which 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 do exist for relational operators over RDBMSs, which usually include their own cost-based optimizer, this is not the case for other analytics operators (e.g., natural language processing, machine learning, graph processing, etc.) and modern runtimes (be they 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 (e.g., [39, 33]). 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" and models them using profiling and machine learning. 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., type and size, dimensionality, distribution, etc.), (b) *operator specific*, which relate to the algorithm of the operator (e.g., number of output clusters in k-means, number of iterations in pagerank, etc.), and (c) *resource specific*, which define the resources to be tweaked during profiling (e.g., cluster size, main memory, etc.). The sampling of the configuration space follows an adaptive approach, picking the most representative instances of the deployment space to achieve high accuracy given a certain budget of runs.

The output is the profiled operator's performance and cost (e.g., completion time, average memory, CPU consumption, etc.) under each combination of the input parameter values. Both the input parameters as well as the output metrics are specified by the user/developer in the materialized operator's metadata (Figure 3, l. 14-15). The collected metrics are then used to create estimation models, making use of neural networks, SVM, interpolation and curve fitting techniques for each operator running on a specific engine. The cross validation technique [29] is used to maintain the model that best fits the available data.

►**Model Refinement** The offline produced models assure the system a warm start, as they can provide accurate cost and performance estimations from the very first workflow planning. However, as the system is put to use, online measurements of executing tasks can contribute to the increase of the models' prediction accuracy and to the better adjustment to the current conditions. 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 to discover the (near-)optimal one with respect to the user-defined, possibly multiple optimization objectives.

The Planner's input is the abstract workflow graph, expressed as a DAG of operator and data nodes. As a first step, for each abstract operator of the input workflow the Planner needs to explore the IReS Operator library to discover all matching materialized operators, i.e., operators that share the same metadata. To speedup this procedure, we use string labelled and lexicographically ordered metadata trees, which allow for efficient, one pass tree matching. The matching is performed by simultaneously iterating over both trees using a recursive merge procedure to match children nodes. The complexity of matching two metadata 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 metadata attributes (e.g., algorithm name). Only operators that contain the correct attributes are considered as candidate matches and are further examined.

The discovery of the best materialized operator combinations out of all feasible ones that simultaneously optimize more than one execution criterion, e.g., both execution time and memory utilization, translates to a multi-objective optimization problem that, in the case of conflicting objectives, has a possibly large number of pareto-optimal solutions.

A simplistic algorithm would exhaustively try all possible combinations and check their validity by consulting the input/output specifications of the materialized operators and datasets. In case an input dataset can not be used as is by a materialized operator, or if subsequent operators are incompatible in their input/output formats, the Planner searches the IReS library for *auxiliary operators*, which can be interposed to "glue" different engines. Such auxiliary operators include move and transformation operations (e.g., the copyToLocal and copyFromLocal mechanisms of HDFS, which move datasets from HDFS to the local filesystem and vice versa), are provided by the developers and treated as common materialized operators. Thus, invalid combinations are those that contain incompatible elements for which no auxiliary operator exists that can render them compatible.

After the elimination of invalid combinations, the objective functions for each valid one should be evaluated based on the prediction models of the involved operators, including the auxiliary ones. Finally, the algorithm should return the combinations that provide the pareto-optimal solutions that minimize or maximize the objective functions.

While such a naive algorithm would provide optimal solutions, it would only be practical for small workflow instances. Assuming a workflow graph of n abstract nodes and m materialized matches for each of them, the complexity of the algorithm is $\mathcal{O}(m^n)$, meaning that the size of combinations to be checked grows exponentially with the number or workflow nodes. Thus, to be able to accommodate large and complex workflow instances within a reasonable time-frame, we opt for a heuristic planning algorithm, *H-Planner*, which relies on genetic algorithms to find near-optimal solutions.

More specifically, H-Planner uses NSGA-II [18], the most prevalent evolutionary algorithm that has become the standard approach to generating pareto-optimal solutions to a multi-objective optimization problem. All candidate ma-

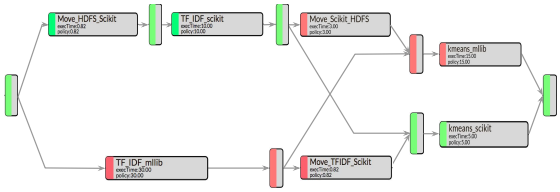

Fig. 4: Materialized workflow and optimal plan.

terialized operators of each abstract one are provided as input to the algorithm. NSGA-II initially creates random permutations of the input, i.e., different combinations of materialized operators, validates them, adding - if necessary - auxiliary operators, and evaluates for each of them a set of scores. This set contains the aggregate of the estimations of all involved operators for each objective function (consulting the models). Combinations that best fit the optimization criteria are selected and their crossover, along with some mutations (i.e., small changes at random), are provided as input to the next iteration of NSGA-II. After a fixed number of iterations or if no significant progress is achieved, the process results in a set of approximate pareto-optimal execution plans.

In the special case of (1) a single optimization objective and (2.a) workflows that exhibit compatibility of input/output operator specification or (2.b) a linear structure, we can employ a more accurate and efficient algorithm, the *DP-Planner*, which relies on dynamic programming (DP) to select the truly optimal execution plan. The aforementioned conditions are deemed necessary to guarantee the principle of optimality: When all workflow operators are compatible, i.e., use the same input/output format and engine, or when the workflow does not contain operators that branch and merge again at any point, we can ensure that optimizing each step of the workflow will result in the optimization of the entire workflow. As Big Data workflows commonly use HDFS as their common data substrate and often follow simple linear structures, we believe that this algorithm can find application in many practical cases.

The abstract workflow of our motivating example performs tf-idf feature-extraction over `Reviews` and clusters the output using k-means. Assuming each operator has two implementations, using either Spark/MLlib or Python/scikit, we have the possible alternative execution plans of Figure 4. Note that the Planner automatically adds the necessary `move` operators in order to match `Reviews`, which resides in HDFS, with `tf-idf_scikit`, which reads from the local filesystem, and 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 prevail for bigger datasets. Indeed, the Python implementation is estimated to be the fastest for both steps, even with the additional cost of transferring data from HDFS to the local filesystem, 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 aims to provision the

correct amount of resources so that the workflow execution conforms as much as possible to the user-defined optimization policy. The possible resource-related parameters that need to be defined include, for instance, the number of cores or the amount of memory which will be allocated to the execution of a materialized operator. The resource provisioning process builds again on the NSGA-II genetic algorithm: The range of possible values for each resource-related parameter is provided as input to the genetic algorithm and various combinations of them are iteratively tested to discover the one that achieves local optima of the trained models. 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 fault tolerance, carried out by the *Execution Monitor*.

►**Enforcer** The Enforcer orchestrates the execution of the materialized operators, over the appropriate platforms and resources, as chosen by the Planner. 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 series of primitives as understood by the specific runtimes and storage engines. Such actions might entail code and/or data shipment, if necessary.

Our working prototype relies on YARN [37], 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 [7], 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. Concisely, each workflow is deployed over the physical resources as a YARN application: An application master container is launched to coordinate all containers required to execute each workflow operator. The number and size of those containers are designated by the Planner's resource provisioning mechanism.

►**Execution Monitor** This module captures failures that might occur, both at node and engine levels, on-the-fly through real-time monitoring. Thus, it ensures the availability and fault tolerance of the system by employing two mechanisms:

– A *node health check mechanism* provided by YARN, which monitors the health status of the underlying infrastructure by periodically executing customizable and parametrized scripts in all cluster nodes. The health check script may include rules for the per node usage of memory, CPU, network, etc. Any node failing to adhere to the script rules is characterized as unhealthy. For instance, a health script may check a node's current memory usage and report the node as unhealthy if it exceeds 95%. The health status (HEALTHY/UNHEALTHY state per cluster node) is reported back to the IReS server. No execution will be scheduled to unhealthy cluster nodes.

Table 1: Operators with their associated engines

Operator	**Compute Engine(version)/Data Engine**
PageRank	Spark(2.1.1)/HDFS, Hama(0.7.1)/HDFS, Java(1.8)/localFS)
k-means	Spark(2.1.1)/HDFS, Python scikit-learn(1.19.0)/localFS
tf-idf	Spark(2.1.1)/HDFS, Python scikit-learn(1.19.0)/localFS
UserProfiling	Spark(2.1.1)/HDFS
Classifier	Spark(2.1.1)/HDFS
tokenization	Spark(2.1.1)/HDFS, Java Stanford CoreNLP(3.9.0)/localFS
stop-word removal	Spark(2.1.1)/HDFS, Java Stanford CoreNLP(3.9.0)/localFS
sentence detection	Spark(2.1.1)/HDFS, Java Stanford CoreNLP(3.9.0)/localFS

– A *service availability check mechanism* that examines the availability of all engines needed for the enforcement of an execution plan. Essentially, a daemon running on the YARN application master container periodically pings the available services and stores their status (ON/OFF) in memory. This information is served, whenever needed, to the IReS Planner. The period of the availability check is customizable, currently set to 5 seconds.

This information is used during the phases of both planning and execution of a workflow: **During workflow planning**, unavailable engines are excluded when constructing the optimal execution plan and resources are provisioned exclusively taking into account the currently healthy ones. **During workflow execution**, engine failures are detected in real-time. When failures affect an operator's execution, the workflow enters a REPLANNING state, which triggers the following steps: (a) The operator currently running is stopped and all subsequent operators pending for execution are cancelled; (b) The YARN application master checks for the existence of any intermediate materialized datasets and determines which part of the workflow needs to be re-scheduled for execution; (c) The Planner is invoked to select the new execution plan of the remaining workflow; (d) The new plan is enforced.

As a checkpointing mechanism, IReS persists in HDFS the output of successfully executed operators. Taking advantage of any intermediate materialized data, it effectively reduces the workflow part that needs to be re-scheduled.

3 Experimental Evaluation

Our prototype is implemented in Java 1.8 and uses the YARN scheduler and the Cloudera Kitten 0.2.0 project for the management of the deployed compute and data engines. IReS is open-source and available on GitHub. In the following experiments, IReS orchestrates a number of runtimes and data stores presented in Table 1. For all engines, we maintain the default configuration. Both our multi-engine framework and the available engines are deployed over a 8-node cluster, where each node features eight Intel(R) Xeon(R) CPU E5405 @ 2.00GHz cores and 8 GB of main memory. In our experiments, we make the assumption that there is only one workflow executing at any given time. The scheduling of multiple concurrent workflows is a subject of future work.

►**Workflows and Data**. For our experiments we use both real and synthetic workflows. Real workflows are driven by actual business needs and have been

specified in the context of the EU-funded ASAP project[4]. They cover complex data manipulations in the areas of *telecommunication analytics* and *web data analytics*, provided by a large telecommunications company and a well-known web archiving organization respectively. The tasks involved include *machine learning (ML)*, *graph processing* and *natural language processing (NLP)* operators (see Table 1) over datasets that consist of anonymized telecommunication traces and web content data. By selecting workflows that cover different and representative areas (ML, graph, text) of Big Data applications, we showcase the general applicability of our framework.

Synthetic workflows are used for stressing our system in terms of workflow size and complexity and evaluating various aspects of it in a controlled manner. They include workflows produced by the Pegasus workflow generator [15], as well as sequential workflow topologies. The Pegasus-generated workflow graphs fall into four scientific workflow categories (Montage, CyberShake, Epigenomics and Inspiral) and contain patterns derived from different scientific application domains, such as astronomy, biology, gravitational physics and earthquake science. All these graphs present complex topologies, where nodes have high in- and out-degrees. Montage is the most complex workflow to process, as it is the most interconnected one, while Epigenomics and Inspiral contain more sequential patterns that render processing easier.

3.1 Execution of Real-life Workflows

In this set of experiments, we use IReS to optimize the execution of real-life applications in a multi-engine environment. We assume a single optimization objective, minimizing execution time. The execution time of the plan produced by IReS is compared against that of the whole workflow exclusively running on a single engine. The goal for IReS is to discover execution plans at least as efficient as the fastest single-engine choice (plus a small overhead). As the combination of different engines within the same plan reveals chances for further optimization, there exist cases where IReS outperforms the fastest single engine alternative.

►**Influencer Detection**. The first application, derived from the telecommunication analytics domain, calculates the influence score of a subscriber on a telecommunications network (to be used in a recommendation system). Data input is in the form of csv call detail records (CDR). For solving the problem, we first model the CDR data as a graph: Each customer (i.e., phone number) represents a vertex and each call corresponds to an edge. This way, we can directly apply the Pagerank algorithm. The available execution engines are: (i) Spark, (ii) Hama and (iii) a centralized Java implementation.

Figure 5.a depicts the execution time of the workflow for various graph sizes when executed over the available engines and when IReS is used. As expected, the centralized (Java) implementation outperforms the others for small-scale graphs. However, as input size grows and exceeds the memory limits of a single node, this approach fails. Distributed platforms expose a different behavior. They

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

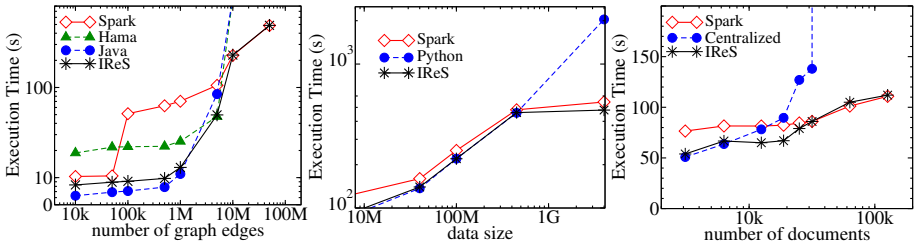

Fig. 5: Execution times for the (a) Influencer Detection, (b) Tourism Observation and (c) Document similarity workflows vs. various input sizes.

incur overheads for small graphs but scale well for larger datasizes. The Hama-based implementation achieves good performance for medium-sized graphs, but as it relies on a distributed main-memory execution model, it also seizes to scale when the graph cannot fit in the aggregate memory of the cluster. The important thing to note is that IReS always adopts the fastest implementation alternative, depending on the input size: For small data sizes, IReS picks Java for executing the workflow; Hama is selected for graphs with over 1M edges while Spark is the platform of choice for large graphs having more than 10M edges. The workflow optimization algorithm and the YARN-based execution incur a small overhead, which is more visible when the overall workflow execution time is limited, i.e., for small data sizes. As data size grows, this overhead is amortized and can be considered almost negligible.

►**Tourism Observation**. This application builds a tourism observation service based on CDR data. The workflow consumes two datasets: (i) the CDRs and (ii) a dataset that maps the GSM cells of the mobile network to geographical regions. Both datasets reside in HDFS. The first operator, called *user_profiling*, joins the two datasets and outputs records that encode the temporal behaviour of users. These vectors are subsequently clustered by a k-means algorithm to discover typical calling behaviours. Both the user profiles and the clusters are finally handed over to a proprietary classification algorithm that labels the calling behaviours and returns the percentage of each label in each spatial region. For example, we can deduce that in New York, 75% of the callers are residents and 25% commuters. For all the operators, IReS chooses between two alternative implementations: (i) a centralized Python code based on the scikit-learn library (ii) a Spark job based on the MLlib library.

Figure 5.b plots the workflow execution time as the size of the CDR dataset increases. The centralized implementation performs better than Spark for CDR datasets smaller than 1GB while Spark scales better as size grows. IReS always adapts to the fastest engine. When the join selectivity of the *user_profiling* operator is high, leading to small join results, IReS opts for a hybrid execution plan, selecting the Spark implementation for the first operator and the centralized Python implementations for the remaining two. This way, for large datasets, IReS manages to outperform even the best single-engine execution.

►**Document Similarity**. This application, which falls into the web data analytics category, aims to cluster similar documents together. The input dataset, provided by a large, European web archiving organization, comprises text files

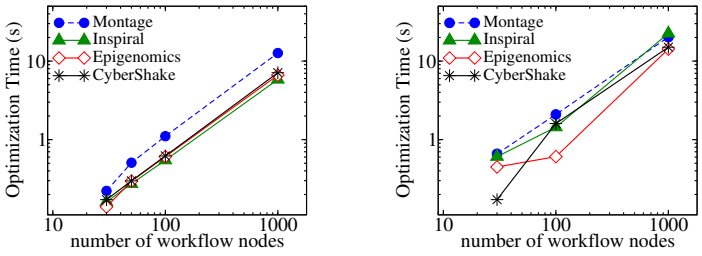

Fig. 6: Optimization time vs. workflow size, 4 engines/operator: (a) Single-objective optimization with DP-Planner (b) 2-objective optimization with H-Planner.

of 40 KB each. As a first step, documents go through an NLP pipeline consisting of sentence detection, tokenization and stop-words removal. Documents are vectorized according to a vocabulary and the tf-idf metric is computed for each of them. Finally, k-means clustering is used to group similar document-vectors. For each operator, two implementations are available: (i) a distributed, Spark-based implementation, and (ii) a centralized implementation (a Java-based implementation using the Stanford CoreNLP library for the NLP operator and a Python-based implementation using scikit-learn for the tf-idf and k-means operators).

Figure 5.c presents the performance results of the Document Similarity workflow when varying the number of input documents. We observe that the centralized implementation outperforms Spark only for small datasets (less than 10K documents in our case). Using our trained cost estimators, IReS selects the proper engines and always performs as good as the fastest engine does. What is more, for a range of input sizes, IReS picks hybrid plans that combine different execution engines. Indeed, from 10K to 40K number of input documents IReS maps the tf-idf operator to Python and the *k-means* operator to Spark. This way, it outperforms even the fastest single-engine execution by up to 30%.

3.2 Workflow Planner

In this section we experimentally evaluate the performance of the IReS multi-engine workflow planner when performing single- or multi-objective optimization with respect to: (i) the optimality of yielded plans, (ii) the number of available alternative implementations of each operator and (iii) the workflow complexity. For these experiments, we use the synthetic, Pegasus-generated workflows. This gives us the flexibility to create arbitrarily large workflow graphs and test complex topologies that are more difficult to obtain in practice. To perform optimization, IReS requires performance profiles for every operator available in the IReS library. As these operators are part of a synthetic workflow, models have to be simulated. To every targeted optimization objective M_i (e.g., execution time, cost, memory utilization, etc.), we randomly assign a domain of the form $R_i = (0, N_i]$,i.e., a range where M_i can obtain a value from. The M_i value of each materialized operator is selected uniformly at random in R_i. To further stress the Planner in terms of the number of possible execution plans that need to be processed, we make the assumption that all operators are compatible in their inputs/outputs. As a side effect, no auxiliary operators are required.

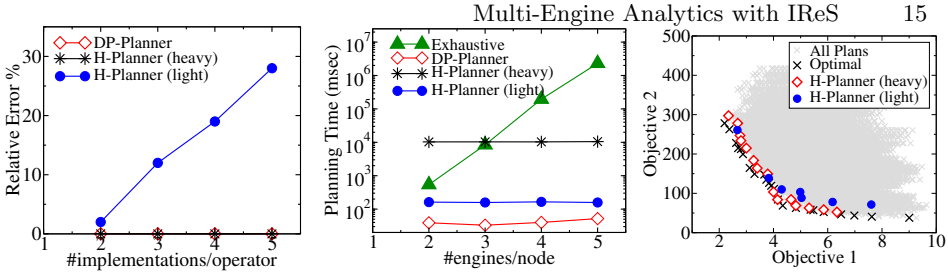

Fig. 7: (a) Relative error and (b) planning time of various algorithms for a synthetic 10-node workflow (single-objective). (c) Pareto frontiers (multi-objective).

In Figure 6.a we assess the performance of the DP-Planner algorithm for the single-objective optimization problem, while Figure 6.b presents the performance of the H-Planner algorithm for the 2-objective optimization problem, when ranging the number of workflow nodes from 30 to 1000, considering 4 alternative execution engines per operator. Here we should note that DP-Planner discovers the exact optimal plans, while H-Planner trades optimality for speed, returning approximate, near-optimal solutions.

Both planning algorithms exhibit similar behaviour, scaling linearly with the graph size. The CyberShake, Epigenomics and Inspiral workflows of the same size require comparable times to be processed by each of the algorithms. Only Montage deviates in the single-objective case, exhibiting slightly increased planning times due to its dense structure: The Montage graph is more connected, having multiple nodes with high in- and out-degrees. Since the complexity of DP-Planner is $\mathcal{O}(op \cdot m^2 \cdot k)$, k being the number of inputs of each operator, the high in-degrees of Montage have a linear impact on optimization time.

For both the single- and multi-objective optimization cases, even under the extreme scenario of 1000-node workflows, the overhead of IReS is less than 12 seconds in all runs. An average 30-node workflow, 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. This allows us to expect that the IReS Planner can handle the most complex multi-engine workflow scenarios under any single- or multi-objective optimization policy with negligible overhead compared to the total execution time of the analytics workflow itself.

In the next experiment, we investigate the quality of the execution plans that IReS selects in both the single- and multi-objective policy scenarios, as well as the cost for discovering them for all planning algorithms employed. As a baseline, we implement an exhaustive algorithm that enumerates all possible execution plans of a workflow and selects the optimal one(s). The complexity of it is exponential to the number of workflow nodes and highly affected by the number of available engines for each operator. As the cost of constructing the optimal solution for the Pegasus graphs is prohibitive, for this experiment we use a smaller synthetic graph of 10 nodes. This is the largest graph size on which the exhaustive algorithm successfully runs in our machine. The synthetic graph has been created by removing nodes from the smallest available Montage graph

until we end up with 10 nodes. We assume that all operators have the same number of alternative implementations, which ranges from 2 to 5.

Figure 7.a plots the relative error between the cost of the selected and the cost of the optimal plan as the number of operator alternatives increases, when using three algorithms: DP-Planner and two variations of H-Planner, the *light* one that runs for 100 generations and the *heavy* one that runs for 1000 generations. The more the generations, the longer it takes for the algorithm to execute but the closer the results are to the optimum. As expected, DP-Planner always achieves optimal results. Zero relative error is also observed for the heavy version of H-Planner, since it performs a more extensive search to discover the optimal plan. The light version of H-Planner, configured to run for an order of magnitude less generations, deviates from the optimum, but still less than 30%. Moreover, in that case, the number of alternative implementations has a monotonically increasing effect on the relative error.

Figure 7.b presents the corresponding planning times. We see that the exhaustive algorithm, denoted as *Exhaustive*, very soon becomes unaffordable even for small graphs: For 5 implementations per operator, the algorithm needs almost an hour to investigate all possible plans. The fastest planning algorithm is DP-Planner, requiring at most 35msec for discovering the optimal plan (in the case of 5 alternatives/engine). The light version of H-Planner follows, with a constant planning time of around 110ms, regardless of the number of alternatives per engine. The heavy version of H-Planner is 2 orders of magnitude slower than the light one. Even so, it requires less than 10 seconds to provide results, time certainly affordable especially when the quality of results is the desideratum.

Next, we consider two optimization objectives, *Objective 1* and *Objective 2*, under the same experiment configuration. In this case the Planner returns a set of approximate pareto-optimal execution plans. DP-Planner is unable to handle multi-objective planning, thus only the two variations of H-Planner are evaluated. Figure 7.c plots all possible execution plans in terms of their Objective 1 and Objective 2 values in grey, considering 4 alternatives/operator. The black marks depict the pareto optimal frontier [18], as designated by the exhaustive algorithm. The heavy version of H-Planner produces the plans in red while the light version the plans in blue. Both H-Planner variations result in plan sets that lie close to the pareto optimal ones and cover their entire frontier, with H-Planner heavy producing results closer to optimal. To quantify the quality of the two H-Planner versions, we calculate the *Hausdorff distance* [25] between resulting plan sets and the pareto optimal one (as calculated by the exhaustive algorithm). The Hausdorff distance is a metric that measures how far two subsets of a metric space are from each other and is defined as the greatest of all distances from a point in one set to the closest point in the other set. More formally, assuming A is the approximate plan set and P the pareto optimal one, the Hausdorff distance between the two sets is $d_H(A,P) = \max_{\forall a \in A}\{\min_{\forall p \in P}\{d(a,p)\}\}$, where $d(a,p)$ is the Euclidean distance between points a and p. For the experiment of Figure 7.c, it holds that $d_H(A,P) = 15.6$ for H-Planner heavy and $d_H(A,P) = 60.7$ for the H-Planner light. The plan set produced by the heavy version of the H-

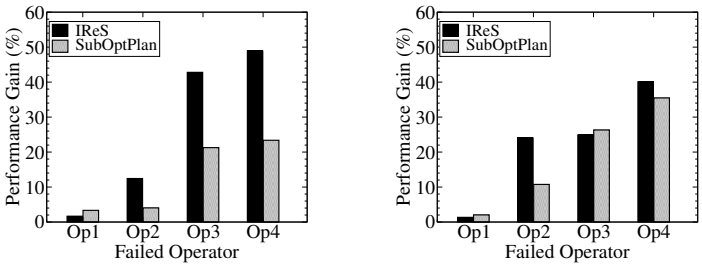

Fig. 8: Total execution time in the presence of failures. (a) *Eng1* is the fastest for all operators. (b) The fastest engine alternates between *Eng1* and *Eng2*

planner is 4× closer to the pareto-optimal than the one produced by the light version. Experiments with ranging the number of alternative implementations per operator show no qualitative difference.

3.3 Fault-tolerance mechanism

In this section, we test IReS under the presence of failures and evaluate its resilience. To have a better control over the experiment, we assume a synthetic, sequential workflow of four operators, `Op1` to `Op4`, each having two implementations (*Eng1, Eng2*). Each operator/engine combination has a performance model drawn from a uniform distribution.

We conduct four experiments, simulating the failure of `Op1` to `Op4` respectively, by disabling the engine selected by IReS during operator execution. As explained in Section 2.3, IReS will cancel the execution of the failed operator and all subsequent ones, it will determine the subgraph of the initial workflow that needs to be re-planned and will issue a request to the Planner for a new plan. The IReS fault-tolerance mechanism is compared to two alternative strategies: (a) *TrivialReplan*, which does not materialize intermediate datasets and thus requires re-scheduling of the whole workflow and (b) *SubOptPlan*, which represents the hypothetical case where the initial execution plan had been selected excluding the failed engine, i.e., the workflow execution is ab initio suboptimal, but not affected by the engine failure.

Figure 8 plots the gain in workflow execution time achieved by the IReS and SubOptPlan strategies compared to TrivialReplan when each of the `Op1` to `Op4` fails. In Figure 8.a, *Eng1* is the fastest alternative for all operators. Since IReS does not need to re-execute the successfully completed operators, the workflow performance ameliorates as failures occur later in the execution path. Contrarily, the performance of workflows executed with TrivialReplan naturally degrades as the position of the failed operators moves towards the end of the workflow, since larger parts of the workflow need to be re-executed. Thus, IReS exhibits increasing performance gains that reach 50%. The performance gain of SubOptPlan, which always runs the whole workflow in *Eng2*, is always worse than that of IReS, with the exception of `Op1` failure: When `Op1` fails, IReS can not take advantage of any intermediate result and has to re-schedule the whole workflow from scratch. The new plan coincides with the SubOptPlan plus the replanning overhead. When failures occur at later stages, IReS exploits the performance

gains of the operators successfully executed on *Eng1*, thus the performance gain compared to SubOptPlan increases.

In the setup of Figure 8.b, we randomly pick which of *Eng1* or *Eng2* is the best alternative for each operator. Once again, IReS plans always outperform the TrivialReplan ones, with an increasing performance gain as failures occur later in the execution. Compared to SubOptPlan, we see that there are still cases where IReS performs better (failure of `Op2` and `Op4`). When the selected engine of `Op1` or `Op3` fails, SubOptPlan slightly outperforms IReS due to the extra cost that the replanning and launching of new containers incurs.

4 Related Work

Data federation approaches have a long tradition, having extensively studied query execution across multiple data sources for decades [32, 35, 17]. However, these approaches focus solely on SQL analytics and fail to optimize query execution over datasets split between multiple sources. On the contrary, recent attempts in the field of data management aim to provide a unified query language or API over various datastores. SparkSQL [14], part of the Apache Spark project [3], and PrestoDB [9], powered by Facebook, are two production systems that provide a query execution engine with connectors to various external systems such as 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.

Polystores have recently been proposed as a means to combine data from heterogeneous sources [21, 38]. They consist of a multitude of data stores, each accommodating different query types. In a polystore environment, data migration among stores is frequent and cumbersome, requiring pairwise, bi-directional connections between the available data stores and creating additional load. Data move can be facilitated by special communication frameworks like PipeGen [24], which use binary buffers to transfer data. Such frameworks can complement IReS, providing efficient move operators whenever data transfer is required.

Recent research works like the Cascading Lingual project [4] and CloudMdsQL [30] 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.

In the field of workflow management, HFMS [34] 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 is a fully open-source platform that targets both low and high level operators.

Musketeer [23] and Rheem [13] 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.

5 Conclusions

Deciding on the exact platforms, configurations and resources to execute long and complex Big Data workflows on, especially when multiple execution criteria are involved, is a daunting task, even for the most knowledgeable and experienced system architect. IReS alleviates this burden, by automatically planning the execution of workflow parts over different platforms, abstracting away their specifics. Based on cost and performance estimations, IReS is able to make the most out of each available platform, matching tasks to the most beneficial runtimes and data to the most suitable stores. IReS proves extremely useful in the case of large workflows with complex structure or of tasks with unknown and hard-to-predict behaviour. Moreover, depending on the workflow and the operators involved, IReS has the potential of yielding significant gains in cost and performance compared to statically scheduled, single-engine executions. 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 as well as synthetic workflows. The experiments showcase (a) a speedup of up to 30% in the execution of the tested workflows, (b) the efficiency of the multi-objective optimizer, which discovers the close-to-optimal pareto plans within a few seconds and (c) the reliability of the system, which manages to recover from failures with minimum impact on the workflow execution time.

References

1. Apache Flink. https://flink.apache.org/
2. Apache Hadoop. http://hadoop.apache.org/
3. Apache Spark. https://spark.apache.org/
4. Cascading Lingual. www.cascading.org/projects/lingual/
5. Cloudera Distribution CDH 5.2.0. http://www.cloudera.com/content/cloudera/en/downloads/cdh/cdh-5-2-0.html
6. Hortonworks Sandbox. http://hortonworks.com/products/hortonworks-sandbox/
7. Kitten. https://github.com/cloudera/kitten
8. monetdb. https://www.monetdb.org/
9. Presto. http://www.teradata.com/Presto
10. Running Databases on AWS. http://aws.amazon.com/running_databases/
11. The Infrastructure Behind Twitter: Scale. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html
12. What is Facebook's architecture? https://www.quora.com/What-is-Facebooks-architecture-6
13. Agrawal, D., et al.: Rheem: Enabling Multi-Platform Task Execution. SIGMOD (2016)
14. Armbrust, M., et al.: SparkSQL: Relational data processing in Spark. In: SIGMOD. pp. 1383–1394. ACM (2015)

15. Bharathi, S., et al.: Characterization of scientific workflows. In: Workshop on Workflows in Support of Large-Scale Science (2008)
16. Bugiotti, F., et al.: Invisible Glue: Scalable Self-Tuning Multi-Stores. In: CIDR (2015)
17. Chawathe, S., et al.: The tsimmis project: Integration of heterogenous information sources. IPSJ pp. 7–18 (1994)
18. Deb, K., et al.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE transactions on evolutionary computation 6(2), 182–197 (2002)
19. Doka, K., Papailiou, N., Tsoumakos, D., Mantas, C., Koziris, N.: IReS: Intelligent, Multi-Engine Resource Scheduler for Big Data Analytics Workflows. In: Proceedings of the 2015 ACM SIGMOD. pp. 1451–1456. ACM (2015)
20. Doka, K., et al.: Mix nmatch multi-engine analytics. In: Big Data. pp. 194–203. IEEE (2016)
21. Duggan, J., et al.: The bigDAWG Polystore System. ACM Sigmod Record 44(2), 11–16 (2015)
22. Giannakopoulos, I., Tsoumakos, D., Koziris, N.: A decision tree based approach towards adaptive profiling of cloud applications. In: IEEE Big Data (2017)
23. Gog, I., et al.: Musketeer: All for One, One for All in Data Processing Systems. In: Eurosys. p. 2. ACM (2015)
24. Haynes, B., Cheung, A., Balazinska, M.: Pipegen: Data pipe generator for hybrid analytics. arXiv:1605.01664 (2016)
25. Henrikson, J.: Completeness and total boundedness of the hausdorff metric. MIT Undergraduate Journal of Mathematics 1, 69–80 (1999)
26. Herodotou, H., et al.: Starfish: A Self-tuning System for Big Data Analytics. In: CIDR (2011)
27. Johnson, N., Near, J.P., Song, D.: Towards Practical Differential Privacy for SQL Queries. Vertica 1, 1000
28. Karpathiotakis, et al.: No Data Left Behind: Real-Time Insights from a Complex Data Ecosystem. In: SoCC. pp. 108–120. ACM (2017)
29. Kohavi, R., et al.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In: IJCAI (1995)
30. Kolev, B., et al.: CloudMdsQL: querying heterogeneous cloud data stores with a common language. Distributed and Parallel Databases pp. 1–41 (2015)
31. Lim, H., Herodotou, H., Babu, S.: Stubby: A Transformation-based Optimizer for Mapreduce Workflows. VLDB (2012)
32. Roth, M.T., Schwarz, P.M.: Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In: VLDB. vol. 97 (1997)
33. Sharma, B., Wood, T., Das, C.R.: HybridMR: A Hierarchical MapReduce Scheduler for Hybrid Data Centers. In: ICDCS (2013)
34. Simitsis, A., et al.: HFMS: Managing the Lifecycle and Complexity of Hybrid Analytic Data Flows. In: ICDE. IEEE (2013)
35. Tomasic, A., Raschid, L., Valduriez, P.: Scaling Access to Heterogeneous Data Sources with DISCO. IEEE TKDE 10(5), 808–823 (1998)
36. Tsoumakos, D., Mantas, C.: The Case for Multi-Engine Data Analytics. In: Euro-Par 2013: Parallel Processing Workshops. Springer (2014)
37. Vavilapalli, V.K., et al.: Apache hadoop yarn: Yet another resource negotiator. In: SoCC. p. 5. ACM (2013)
38. Wang, J., et al.: The Myria Big Data Management and Analytics System and Cloud Services. In: CIDR (2017)
39. Zhang, Z., et al.: Automated profiling and resource management of pig programs for meeting service level objectives. In: ICAC. pp. 53–62. ACM (2012)