Adaptive Execution of Continuous and Data-intensive Workflows with Machine Learning

Sérgio Esteves, Helena Galhardas, and Luís Veiga
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
{sergio.esteves,helena.galhardas,luis.veiga}@tecnico.ulisboa.pt

ABSTRACT

To extract value from evergrowing volumes of data and to drive decision making, organizations frequently resort to the composition of data processing workflows. The typical workflow model enforces strict temporal synchronization across processing steps without accounting the actual effect of intermediate computations on the final workflow output. However, this is not the most desirable in a multitude of scenarios. We identify a class of applications for continuous data processing where the workflow output changes slowly and without great significance in a short time window, thus squandering compute resources with current approaches.

To overcome such inefficiency, we introduce a novel workflow model, for continuous and data-intensive processing, capable of relaxing triggering semantics according to the impact that input data is assessed to have on changing the workflow output. To estimate this impact, learn the correlation between input and output variation, and guarantee correctness within a given tolerated error constant, we rely on Machine Learning. The functionality of this model is implemented in SmartFlux, a middleware framework which can be integrated with existing workflow managers. Experimental results indicate substantial savings in resource usage, while not deviating the workflow output beyond a small error constant with a high confidence level.

CCS CONCEPTS

• **Information systems → Data management systems**;

KEYWORDS

Workflow Management, Machine Learning, Resource Efficiency

ACM Reference Format:
Sérgio Esteves, Helena Galhardas, and Luís Veiga. 2018. Adaptive Execution of Continuous and Data-intensive Workflows with Machine Learning . In *19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3274808.3274827

1 INTRODUCTION

Current trends are being characterized by ever-growing volumes of data flowing over the globe throughout wide-scale networks. In this context, new distributed and high-scalable infrastructures are required to manage and process data efficiently. This includes infrastructures enabling workflow composition, denominated Workflow Management Systems (WMSs), for being highly expressive, flexible and scalable.

A workflow is usually modeled as a Directed Acyclic Graph (DAG) to express the dependencies and relations between computation and data. The workflow paradigm has been extensively used in a number of different settings (e.g., eScience, engineering, industrial), encompassing activities as diverse as web crawling, data mining, protein folding, sky surveys, forecasting, RNA-sequencing, or seismology [10, 19, 30, 44].

Generally, a WMS triggers the execution of an entire workflow graph based on either time frequency (e.g., every 20 minutes) or data availability (e.g., when new files show up in a given folder). We refer to continuous data processing when the same workflow compute graph is executed repeatedly, and in sequence, for a possibly unbounded number of times.

Traditionally, WMSs enforce strict temporal synchronization throughout the various dependencies of processing steps (i.e., following the Synchronous Data-Flow (SDF) computing model [32]). That is, a step is immediately triggered for execution as soon as all its predecessor steps have finished their execution. Should the temporal logic be relaxed, for example, to respond to application requirements of latency or prioritization, programmers have no other choice than to explicitly program non-synchronous behavior. This ad-hoc programming increases the complexity of the application and the chance of error occurrence.

In addition, typical WMSs do not take into account the volume of data arriving at each processing step and its actual impact on changing the final workflow output (i.e., the output produced by processing steps that do not have any successor steps). We argue that such an assessment should be used to control the workflow execution and drive the triggering of steps towards meaningful results. This issue is even more important in workflows for data-intensive and continuous processing where many resources can be purposelessly wasted if new input and intermediate datasets do not cause significant changes on the workflow output.

In fact, fully executing a processing step every time a small fragment of data is received can have a great impact on performance and machine load, without actually changing substantially the workflow output and its significance to the problem being addressed; as opposed to executing it only when a certain substantial, relevant (w.r.t. application semantics) quantity of new updated data is available.

Further, there is a class of workflow applications for continuous data processing where the output of final processing steps does not change significantly in a short time window. A natural fit to

Middleware '18, December 10–14, 2018, Rennes, France

ACM ISBN 978-1-4503-5702-9/18/12…$15.00
https://doi.org/10.1145/3274808.3274827

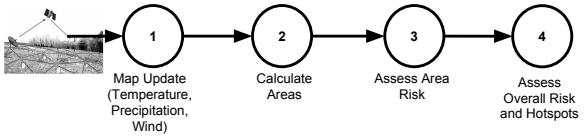


Figure 1: Motivational example: fire risk assessment

this class are monitoring applications, which are commonplace [17]. For example, a workflow that is constantly processing data coming from a network of temperature sensors, to detect fires in forests, would not need to be always computing tasks (e.g. calculating hotspots, updating the risk level) whose output would not change significantly in the presence of small jitters in temperature. The workflow would only issue a displacement order to a fire department if a significant proportion of the sensors detect a steep increase in temperature. This way, tasks should have a specification of the minimum impact that their input data is required to have in order to make their executions worthy.

As a motivational example, consider the case of assessing the fire risk in a given forest through a sensor network that captures temperature, precipitation and wind. Figure 1 depicts a workflow that, periodically (e.g., every 5 seconds, every half an hour), receives data from the sensors and executes the following processing steps: 1) updates an internal representation of the forest map; 2) divides the map into small areas; 3) assesses the fire risk in each area; and 4) assesses the overall risk and contiguous risky areas (hotspots).

The temperature, precipitation and wind measures will probably not change every half an hour, or at least not significantly to pose a risk. Changes in the sensor readings will cause increasingly smaller changes in the data as we go through the steps of the workflow; e.g., the temperature of an area, one piece of data generated by step 2, which results from the average temperature of its composing sensors, will only change if a large fraction of sensor readings change (i.e., more than one piece of data in step 1 should change). Likewise, the risk of an area, one piece of data generated by step 3, which consists of the classification of different temperature ranges into different risk levels, will probably only change after an area has been updated several times (many more than one piece of data in the output of step 2). As a result, the output of the workflow, generated by step 4, will remain almost unchanged during most of the time. Only output variations higher than a certain threshold will be deemed as significant. Therefore, we consider that a substantial amount of resources is wasted in re-executing the entire workflow.

Other examples, that fall in this same application class, include: measuring the impact of social business [7], detecting gravitational-waves [16], weather forecasting [29], predicting earthquakes [19], among others. Even for those applications where the outcome changes more frequently, such as a web crawler, the impact of the updated results may become only relevant when the differences from the previous crawls accumulate significantly (e.g., relevant change in word counts, page ranking or the number of reverse links).

In this paper, we address the problem of providing asynchrony in workflows resorting to the notion of Quality-of-Data. We define Quality-of-Data (QoD), [1] in this context, as the entirety of features or characteristics that data must have towards its ability to satisfy the purpose of changing the workflow output significantly w.r.t. the application specific semantics, following the principle described in [27]. By specifying such data requirements that govern the workflow execution, we are thus able to enforce a certain level of performance in terms of the number of resources engaged per time unit and average latency to obtain a result. These requirements can be enforced, for example, based on the size, frequency and magnitude of new updates.

To this end, we introduce a novel workflow model, for continuous and data-intensive processing, that is capable of intelligently guiding the triggering of processing steps according to patterns observed in the flow of data, towards a meaningful and significant output, while respecting QoD constraints. Hence, we enable adaptive execution in workflows that is driven by the data characterized impact. To assess how different input data patterns affect the workflow output, we resort to Machine Learning with Random Forests [15], which is the classification algorithm that yielded better performance in general comparing to others (cf. § 3). Specifically, we learn statistical behaviors of workflows by correlating input variation with output generated deviation, arising from deferring the execution of processing steps.

As a proof of concept, we developed SmartFlux, a middleware framework that enforces our asynchronous model and can be integrated with existing WMSs. In this work, we integrate it with a widely-deployed WMS, Apache Oozie [26]. Our experimental results show that, with SmartFlux, we are able to deliver high resource efficiency. Specifically, we are able to save a substantial amount of resources w.r.t. the synchronous model, while not deviating the workflow output beyond a small error constant with a high confidence level: up to 30% less executions while enforcing a QoD (an error bound) as low as 5% with a confidence over 95%.

The main contributions of this paper are:

(1) a novel workflow model that enables asynchronous triggering of processing steps;
(2) a solution modeled with Machine Learning techniques to enable the adaptive execution of workflows towards meaningful results in a resource-efficient manner;
(3) a framework, that can be coupled with existing WMSs, and its extensive experimental evaluation with two realistic benchmarks as a proof-of-concept.

The remainder of this paper is structured as follows. § 2 details our workflow model. § 3 describes our learning approach to bound the output error. § 4 presents the design and architecture of SmartFlux and § 5 its experimental evaluation. Related work follows in § 6 and § 7 concludes the paper.

2 ABSTRACT WORKFLOW MODEL

In this section, we describe our abstract workflow model that enables temporal asynchrony across processing steps, based on the predicted impact that observed data patterns in the input will have on the workflow output.

[1] Quality-of-Data is akin to Quality-of-Service, and should not be confused with issues such as internal data correctness, semantic coherence, data adherence to real-life sources, or data appropriateness for managerial and business decisions.

Our model is specifically designed for continuous processing and data-intensive scenarios where long-lasting workflow applications are regularly fed with new or updated raw data from a given source (e.g., network of sensors, Internet, social network, radio telescopes). We refer to each time a workflow is fed with new data as a *wave*. We describe how the task model, the data model (QoD), and the input and output monitoring work together to achieve temporal asynchrony.

Our workflow model inherits from and extends the traditional workflow model [45] where strict temporal synchronization is enforced. Processing steps communicate data through an underlying storage system, a *data container*. It can be a set of directories (filesystems); or keyspaces, tables, columns, rows, or any combination of these (databases).

The main feature that differentiates our model from the other typical DAG workflows is its triggering semantics: a processing step A, in a workflow D, is not necessarily triggered for execution immediately, when all its predecessors A' ($A' \prec_D A$) have finished their execution. Instead, A should only be triggered as soon as all predecessor steps A' have completed at least one execution and have, also, carried out a sufficient (or significant) level of changes on the underlying data containers that comply with certain QoD requirements. This way, a processing step can be re-executed several times without necessarily triggering the execution of successor nodes; i.e., step triggering is guided by the rate of data changes, and not exclusively by the end of a single execution of predecessor nodes, as it usually happens in the regular workflow model.

Intuitively, the QoD defines how sensitive a step is to the impact that new input will have on changing the output after execution. This establishes how much the input needs to change in order for it to update the latest output of a step in a significant manner to the application. If input changes are insufficient, we can simply avoid the execution and save resources while providing (approximate) results faster. Further, in many cases (e.g., updating a map grid of temperatures), a deferred execution of a processing step will process fresh data that outdates, by overriding, previous input. Hence, the load of a deferred execution is not increased.

In practice, the QoD defined for a step needs to correspond to the impact on its input (which comes from the generated output of predecessor steps) that makes its output reach a maximum defined tolerated error (with application meaning for decision makers; e.g., maximum 10% deviation). Hence, the target impact on input corresponds to the input necessary, in terms of quantity and quality (or significance), for reaching a threshold that specifies the maximum deviation of the output tolerated for that step. This output deviation in a step is thus seen as an error introduced by deferring its re-execution, as opposed to the synchronous model. In return, skipping executions saves resources from being wastefully engaged. We now describe the metrics to calculate the input impact and the output error.

2.1 Input Impact

The input impact ι of a processing step is a metric that captures the amount and magnitude of changes applied to its associated data container in relation to a previous state. Every time new data updates are performed on a data container, that holds the input of a step, the input impact is calculated based on the new updated data and its previous versions. The previous versions correspond either to the state of the data on the previous wave, or the state of the data on the wave where the latest execution of the step occurred. The former implies that the input impact is accumulated with the impact measured for previous waves that occurred after the execution of the associated step. The latter allows computations to cancel each other out: if we get the value x_i on wave w equal to x'_i on wave y, regardless the number of waves occurred between y and w without triggering the associated step, the error comes as zero. Further, since steps are potentially not all executed in the same wave, the input impact of a step is only calculated when its predecessors have generated output, which will possibly not happen in every wave.

Users can define their own functions to capture the impact of changes in a data container (technical details elaborated in § 4.2). Nonetheless, we provide, by default, two generic functions that can serve well a wide set of scenarios according to our experiments. They are described in the following equations.

$$\iota = \sum_{i=1}^{m} |x_i - x'_i| \times m \quad (1) \qquad \iota = \frac{\sum_{i=1}^{m} |x_i - x'_i| \times m}{\sum_{i=1}^{m} max(x_i, x'_i) \times n} \quad (2)$$

In Equations 1 and 2, x_i is the updated state of the ith element and x'_i its latest state, m and n are the number of modified elements and the total number of elements in the associated data container, respectively, and *max* is the function that returns the maximum between two numbers. If a new element is inserted, its latest state x'_i is zero (which increases the impact).

Equation 1 captures the differences in magnitude between the updated and latest saved state of elements, multiplied by the number of modified elements. Equation 2 divides the result of Equation 1 by the maximum between the updated and latest state of the elements, multiplied by the total number of elements in the data container. Hence, it captures the relative impact over a previous state, returning a value between 0 (no changes) and 1 (difference introduced by new data with higher or equal magnitude to that of the previous state).

Further, if a processing step receives input from more than one predecessor step, then we calculate the input impact produced by each predecessor step and combine them through the geometric mean by default.

2.2 Output Error

The output error ε of a processing step is a metric that attempts to measure the error penalty of postponing its execution. Each time a step is not executed at a given wave of data, it incurs a certain error that can be seen as the cost of the changes that were missed in the corresponding data container. Hence, if a step is always executed at each wave of data the error is zero. Like the input impact, the output error can be cumulative or not depending on whether error cancellation is allowed for an application.

Users also have the flexibility of providing their own functions to compute the output error (cf. § 4.2). Nonetheless, we offer the following generic functions to calculate the output error, denoted by ε.

$$\varepsilon = \frac{\sum_{i=1}^{m} |x_i - x'_i| \times m}{\sum_{i=1}^{n} x'_i \times n} \quad (3) \qquad \varepsilon = \sqrt{\frac{\sum_{i=1}^{m} (x_i - x'_i)^2}{m}} \quad (4)$$

In Equations 3 and 4, x_i is the updated state of the i^{th} element and x'_i its latest state, m and n are the number of modified elements and the total number of elements in the associated data container respectively.

Equation 3 captures the relative impact of new updates on the latest state. It returns a value between 0 (no error) and 1 (new data has higher or equal magnitude to the previous state). Equation 4 corresponds to the frequently used Root-Mean-Square Error (RMSE), which captures the deviation between the updated and previous states of elements, thereby attenuating the impact of small differences and further penalizing larger differences. It is up to the user to decide which function works better for a particular problem.

Having the input impact (ι), output error (ε), and the maximum tolerated error (max_ε), which can be seen as the decision making boundary, for a processing step (s), we trigger its execution when we predict through ι that $\varepsilon > max_\varepsilon$. For such prediction, we learn the statistical correlation between ι and ε during a period of synchronous execution (see § 3).

2.3 Limitations and Generality of the Model

Our model is suitable for applications that exhibit regular input patterns over a period of time (i.e., no random or uncorrelated input/output over time). We show examples of how this class of applications is actually commonplace in continuous workflow processing. As long as there is a correlation between input and output, our system is able to capture it and predict accurately, with a high confidence interval, when and which steps should be skipped or executed. This is a central premise for our system to work.

Following, we briefly describe that there is an intuitive relation between input and output for three pipeline/workflow applications (due to space constraints we abstract from the details of processing steps that perform the computations).

PageRank: Processes the content of crawled documents and builds an histogram with the differences against previous states of links. It is only worthy to process the new crawled documents if the differences in the link counts is sufficient to significantly change the page rank of documents, according to decision makers.

LIGO [16]: Detects gravitational waves that are linked to the occurrence of events in the universe. The output, regarding the detection of events (like exploding stars), is strongly associated with the input which corresponds to laser data waveforms. It is only worthy to explore the input data if the simple characterization of waveforms can lead to a true inspiral event.

CyberShake [19]: Performs seismic hazard estimation for a given site. The input corresponds to rupture descriptions and the output is an hazard map. It is only worthy to recompute parts of the map if the new probability variations of ruptures are impactful against a previous state.

In these applications, as well as in a great part of applications for continuous and incremental processing, there is generally an association between input and output. The correct characterization of the data input can allow us to predict the significance on the output.

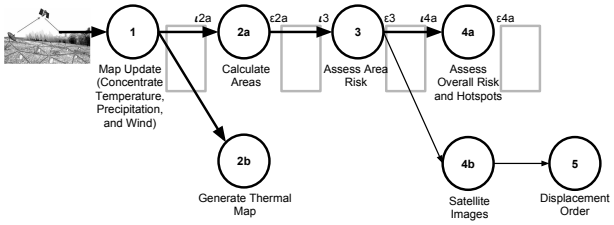


Figure 2: Use-case scenario of continuous and incremental workflow processing: fire risk assessment. The rectangles in grey represent the data containers.

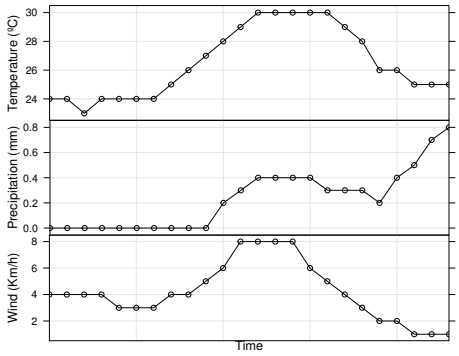


Figure 3: Temperature, precipitation and wind evolution hour by hour for a day in the Amazon rainforest

2.4 Prototypical Scenario

Figure 2 illustrates a workflow, extension of the workflow in Fig. 1, that assesses the fire risk for a given forest region based on a network of sensors equally distributed. For a normal day in the Amazon rainforest, for instance, we can see in Figure 3 that temperature, precipitation, and wind, vary progressively over 24 hours without major steep slopes (this also holds, even more so, when we assume higher frequency of sensor readings, e.g., every second). Such characteristics make this scenario propitious for resource reasoning and savings.

The first processing step (in Figure 2) receives data (temperature, precipitation, wind) from sensors every time interval, aggregates it through some function, and stores the result for each sensor in the corresponding data container. Since this is the first step that updates a data container, it must always be executed at every wave; i.e., it is not possible to maintain sensory data across waves without the execution of this step. **Step 2a** divides the forest into smaller areas and combines the measures of all sensors in each area. This step is only executed when the input impact ι_{2a} is sufficient to cause the error ε_{2a} to reach $max(\varepsilon_{2a})$, which is the user-defined maximum tolerated error. **Step 2b** generates a thermal graphical map for some monitoring station.

Step 3 assesses the fire risk of each area by comparing the values calculated in **step 2a** with some threshold. This step is only triggered when significant measurement differences in some areas are perceived or when a sufficient number of areas is updated by **step 2a**.

Step 4a, which is the workflow output, assesses the overall fire risk and identifies groups of areas with highest risk in the forest. This step is expected to have its output changed slowly over time and $max(\varepsilon_{4a})$ should be set to a value such that the difference in the overall fire risk across waves is significant to decision makers. **Step 4b** gathers satellite images in case areas identified in **step 3** exhibit very high temperature levels (on fire). **Step 5** issues a displacement order to a fire department in case the fire is confirmed through the analysis of satellite images. These two last steps are critical for fire detection and therefore they do not tolerate error.

To estimate and correlate error with input impact, we learn the statistical behavior of the workflow with Machine Learning by executing the workflow synchronously for a restricted period of time, as we elaborate next.

3 LEARNING APPROACH

This section introduces our learning approach to bound the output error, arising from the delayed execution of processing steps, and to provide guarantees about the maximum deviation of workflow outputs. Specifically, we make use of Machine Learning classification techniques to predict how input data affects the output of processing steps.

Our learning approach is based on predictions that are naturally not perfect, and therefore the guarantees we refer in this paper are *probabilistic guarantees*; i.e., we are able to ensure that error bounds are respected *within* a confidence interval. These are the same kind of guarantees offered by other systems such as the ones proposed in [9, 40]. This confidence interval is expected to be high ($> 90\%$) as long as our central premise holds; i.e., that there is a correlation between input and output (cf., § 2.3). This premise is verified during a test phase (elaborated later on in this section).

3.1 Classification

Generally, classification algorithms try to estimate a function $h(x)$ that, given an N-dimensional input data set, predicts which of two possible classes form the output ($h : \mathbb{R}^N \rightarrow \{\pm 1\}$). The estimation of this function, which corresponds to the construction of a model, is based on a supplied set of training examples encompassing tuples with known correct values of input and corresponding output; i.e., supervised learning. The obtained classifier is then able to assign new unseen examples to one class or another.

In our particular problem, we need to predict which steps generate an error exceeding their corresponding maximum bounds (max_ε) for a given input. Hence, the output of the classifier is no longer a single binary value, but a set of values representing the configuration of steps that should be executed or not for each wave of data; i.e., multi-label classification [41]. As an example, the matrices below represent, for 5 waves (rows), a pipeline with 3 steps (columns) querying the classifier by sending the input impact ι calculated for each step (X); and receiving in return the sequence of steps that should be executed or not (Y).

$$h(\underbrace{\begin{bmatrix} 694.86 & 601.6 & 498.3 \\ 191.24 & 886.1 & 498.3 \\ 278.13 & 1071.4 & 498.3 \\ 433.78 & 233.78 & 664.24 \\ 551.53 & 523.8 & 956.52 \end{bmatrix}}_{X}) = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{Y}$$

At the first wave, with the input impact of the 3 steps, $\{694.86, 601.6, 498.3\}$, the classifier estimates that only the first step should be executed ($\{1, 0, 0\}$), so that its error does not exceed the maximum defined bound max_ε. Then, the second step is executed at the third wave ($\{0, 1, 0\}$); that is, it needs to accumulate input for 3 waves so that its output changes significantly (i.e., its error reaches max_ε). Finally, the third step is executed only at the fifth wave.

To learn the correlations between input impact and respective incurred error, it is necessary to train the classifier and construct a model. Then, in a test phase, the quality of the trained classifier is assessed and, if the accuracy is not satisfactory, more training may be required. These two sequential phases, training and test, can be performed either regularly from time to time or on-demand (useful if data patterns start to change suddenly). Further, these phases take place while the workflow is running and producing results with real datasets, hence making this an online process.

3.2 Classification Algorithm Selection

To select a good Machine Learning classification algorithm for our problem, we performed several experiments using the applications described in § 5. Using the ROC area, a metric to assess the performance of a classifier, we compared the following widely-deployed algorithms: Bayes Network, J48 tree, Logistic, Neuronal Network, Random Forest, and Support Vector Machine. Random Forest (RF) [15] and Support Vector Machine (SVM) [25] yielded better ROC areas on average for all the experiments: 0.86 and 0.82 respectively (values approaching 1 mean optimal classifier and 0.5 is comparable to random guessing). However, since SVM requires more parameterization (e.g., selecting a proper kernel to capture linear or non-linear data correlations, or using cost matrices to weight unbalanced datasets) [38], and default parameterization in RF often performs well [15], we adopted RF as our default learning approach, although they can be switched.

Training Phase. Unless a training set is given beforehand, a training phase starts taking place when the workflow is executed for the first time. During this phase, all processing steps of the workflow are executed synchronously (without any QoD enforcement). The duration of this phase is configured by users with a specified number of waves. At each wave, the input impact ι and corresponding (simulated) output error ε are calculated for each step, and a tuple containing ι and a binary value, indicating whether the $\max_\varepsilon$ of that step is reached, is appended to a log (i.e., training-set).

Test Phase. We assess the quality of the trained model measuring: (*i*) accuracy, the proportion of instances correctly classified; (*ii*) precision, the number of classified instances that are truly of a class divided by the total number of instances classified as belonging to

that class; and (*iii*) recall, the number of instances classified as a given class divided by the number of instances that are truly of that class. We perform a 10-fold cross-validation on the training-set.

High values of recall mean that we are avoiding the existence of false negatives; i.e., the percentage of times the model estimated incorrectly that the error was below max_ε. Hence, a good recall is necessary to ensure that the error stays within max_ε. As for precision, high values mean that we are avoiding to estimate incorrectly the error as being above max_ε, which is necessary to mitigate resource waste.

Typically, classification algorithms can be adjusted to favor results on a given metric (e.g., recall), and to specify whether it is more important to comply with error bounds or save resources. In RF, this adjustment is performed through two parameters: the maximum number of trees to be generated, and the maximum depth of the trees. If results are not satisfactory, w.r.t. defined thresholds, a training phase takes place again and more instances are collected. Otherwise, it means that we are able to provide probabilistic guarantees regarding error compliance. As there is a correlation between input and output, it is always possible to get a satisfactory result (e.g., over 90% accuracy) with more training.

Application Phase. After a sufficiently accurate model is built, the application phase takes place and the workflow starts running asynchronously in an adaptive way. At each wave, the input impact ι is calculated for each step and fed to the classifier, which in return indicates which steps should be executed.

4 SMARTFLUX DESIGN & IMPLEMENTATION

SmartFlux is a middleware framework that provides functionality conforming to the workflow model described previously. It couples a WMS with a data store by monitoring data transfers and controlling the triggering of processing steps. With this coupling, SmartFlux enables the deployment of quality-driven workflow applications, where processing steps are triggered based on the impact that their computations are predicted to have in the workflow output.

As we are targeting data-intensive applications, our focus in this work regarding data communication is put on distributed (columnar) Key-Value stores (e.g., Cassandra [28], HBase [21]), since they can achieve high performance.

Figure 4 shows the architecture of the SmartFlux middleware framework, operating between a WMS and a data store. Processing steps run atop the workflow manager and they share data through the underlying storage system. These steps may consist of Java applications, scripts expressed through high-level languages for data analysis (e.g., Apache Pig [35]), Map-Reduce jobs, or other off-the-shelf solutions.

SmartFlux can work with either its own provided basic WMS or existing open-source WMS. Here, we focus on using existing WMS, to assess to what extent it requires changing its implementation and triggering mechanisms. To connect our framework with a WMS, an adaptation component, *WMS Adaptation* (colored in grey), needs to be provided with a specific API so that SmartFlux can issue triggering notifications and receive state information, thereby orchestrating the execution of the processing steps of a workflow.

Since SmartFlux needs to be aware of the updates that the processing steps apply to the data store, we provide three options

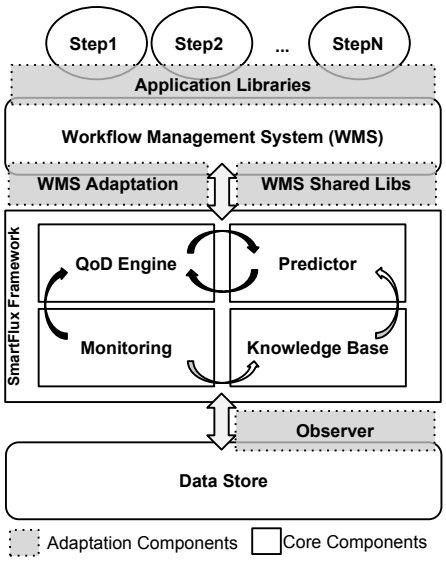


Figure 4: SmartFlux Framework Architecture

(components colored in grey): i) Application Libraries, ii) WMS shared libraries, and iii) Observer. The *Application Libraries* component corresponds to adapted driver libraries, used by processing steps to interact directly with the data store via their client APIs. Although applications may need to be slightly modified (e.g., changing package names in the imports of Java classes), we provide tools to completely automate this process.

At the WMS level, *WMS Shared Libraries* represent adapted shared libraries that are used by processing steps to interact with the data store through the WMS (e.g., pig scripts or any other high-level language that must be interpreted/compiled by the WMS). Finally, at a lower level, the *Observer* component corresponds to custom code that is triggered and executed at the data store level upon client requests (e.g., co-processors in HBase or triggers in Cassandra). These two last options provide transparency to executing steps and avoid changes in the application code.

Next, we describe the responsibilities and purpose of each core component in the SmartFlux framework (in white).

Monitoring: It analyzes, through the adaptation components, all requests directed to the data store. This involves identifying all affected data containers and calculating the corresponding input impact and, during the training phase, also the error. Note that the simplicity of get-put interfaces works in our favor in this process. Afterwards, the calculated values are sent to the QoD Engine.

QoD Engine: It maintains the current state of control data (input impact, error) along with workflow specification and meta-data defined by the user, such as the error bounds for each step. Based on this data, and after querying the Predictor, it evaluates and decides when and which steps should be triggered for execution during the application phase.

Knowledge Base: It maintains data collected through the Monitoring component during the training phase: input impact and a binary value indicating whether $\varepsilon > max_\varepsilon$ for each considered step. This

data forms the training-set that is used by the Predictor to build a classification model.

Predictor: It informs the QoD Engine of which steps should be triggered for execution, thus predicting which error bounds are exceeded given the input impact of considered steps. For that, it uses a classifier (RF by default) with a trained model.

4.1 General Operation Flow

We consider two different operating modes: i) training mode; and ii) execution mode. In the training mode, a workflow is executed synchronously and we collect metrics about the input impact and output deviation for each processing step that tolerates error. After a predetermined number of waves, a classification model is built with the previous collected data.

The training mode is represented by the white curved arrows in Figure 4: the Monitoring component, that gets data from the adaptation components, feeds the Knowledge Base with statistical information about the data updated in the data store; then, the Predictor component builds a classification model based on the data sets with the metrics contained in the Knowledge Base (input impact, error).

The execution mode is represented by the dark curved arrows: the Monitoring component collects statistical information from data store R/W requests, and sends to the QoD Engine computed input impact metrics at each wave of data; after, the QoD Engine queries the Predictor with input impact data and gets in return the configuration of processing steps that should be executed (i.e., where $\epsilon > max_\epsilon$).

4.2 Adopted Technology and Integration

We integrated our framework with a widely deployed WMS, Oozie [26]. In effect, we extended Oozie with a notification scheme that is interfaced with the SmartFlux framework process through Java RMI. Generally, Oozie only has to notify when a step finishes its execution, and SmartFlux only has to signal the triggering of a certain step; naturally, these notifications share the same processing step identifiers. The QoD error bounds are specified along with standard Oozie XML version 0.2 schemas, and given to SmartFlux with an associated workflow description. Specifically, we changed the XSD to accept a new element inside the element *action* (i.e., processing step) which specifies the data containers associated with steps (table, column, row, or group of any of these) and their corresponding error bounds, which are values from 0 to 1.

As our underlying distributed Key-Value storage, we adopt HBase [21], the open-source Java clone of BigTable [18]. This column-oriented data store is a sparse, multi-dimensional sorted map, indexed by row, column, and timestamp; the mapped values are simply an uninterpreted array of bytes. Due to its complexity, we decided to intercept data store updates by adapting the HBase client libraries. To this end, we extended the implementation of some library classes while maintaining their original API; namely, sending the data containers and respective data to SmartFlux inside writing methods (e.g., put, delete). Since our API is the same as the original one, only import declarations need to be modified to SmartFlux packages in the application code.

Regarding our Machine Learning implementation, we adopted MEKA [37], a multi-label classification library in Java based on the well known WEKA [24] Toolkit.

Input Impact and Output Error API. We provide an API through which users can implement custom functions to capture the input impact and corresponding output error. This API comprises 2 main Java method signatures that need to be implemented: *update* and *compute*. *update* is called for every element in the corresponding data container, receiving as arguments the current and previous values of the element. It relates both values (e.g., calculating their difference) and updates the metric state, such as summations.

As for *compute*, it is called to calculate the overall input impact or output error of a processing step, when no more elements are expected to be received on the data container. We plan in the future to provide a high-level DSL language for non-expert users.

Finally, we made available the source code of the prototypes we developed in the context of this paper [5, 6].

5 EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation of our workflow model with the SmartFlux framework. We evaluate the following aspects:

(1) We show why Machine Learning is needed in our problem. Specifically, we show that the correlation patterns between input impact and error are linear and non-linear, making them difficult to predict with rigid non-statistical methods.
(2) We analyze how accurate is SmartFlux in making predictions. Specifically, we analyze SmartFlux ability to use resources efficiently while complying with error bounds. When error bounds are violated, we quantify the number of violations and respective deviations, and, with that, we obtain confidence intervals for error compliance.
(3) We assess the benefits of SmartFlux in terms of saved executions and resources for different error bounds.
(4) We assess the overhead of SmartFlux.

All tests were conducted using 6 machines with an Intel Core i7-2600K CPU at 3.40GHz, 11926MB of RAM memory, and HDD 7200RPM SATA 6Gb/s 32MB cache, connected by 1 Gbps LAN.

5.1 Test Scenarios

We conducted an in-depth analysis by selecting two interesting applications, which represent two different and realistic scenarios for continuous and incremental processing: (*i*) LRB, a variable tolling system for an urban expressway structure based on the Linear Road Benchmark [11]; and (*ii*) AQHI, a system based on a network of sensors to classify the quality of the air in a geographic location, inspired by the Air Quality Health Index (AQHI) used in Canada [2].

LRB. In the first scenario, we have a variable tolling system for a fictional expressway system where different toll rates are charged. The data inputted to the workflow is generated by the MIT-SIMLab (a simulation-based laboratory) [11] and consists of vehicle position reports and historical query requests. Position reports are emitted every 30 seconds and they identify a vehicle's exact location in the expressway system. Through these reports, we generate statistics comprising average vehicle speed, number of vehicles and existence

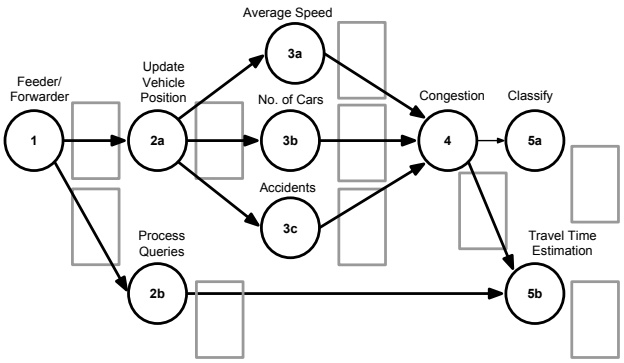


Figure 5: Workflow of the Linear Road. Rectangles in grey represent data containers

of accidents, for every segment of every expressway for every minute. Then, these statistics are used to determine toll rates for the segments where the vehicles are in.

Historical query requests, by turn, are issued by vehicles to estimate travel time and cost for a journey on an expressway. Figure 5 depicts the workflow we designed for this tolling system. Processing steps are described as follows.

Step 1 receives, separates, and stores position reports and queries from vehicle transponders into different data containers to be processed by step 2a and 2b respectively. **Step 2a** updates vehicle positions in the urban expressway, which includes updating every segment of every expressway with new vehicle data. This step is only staged to execution when there is a sufficient number of position reports (complying with the QoD of step 2a). **Steps 3a**, **3b**, and **3c** (forking from **2b**) assess the average speed for all cars in a segment in the last 5 minutes, the number of cars, and the existence of accidents on every segment of every expressway, respectively. Each of these steps is only triggered when significant differences (according to predefined error bounds) in vehicle positions are perceived against a previous state. Also, the input impact is maintained separately for each of these three steps. **Step 4** (joining **3a,b,c**) computes the level of congestion for every segment of every expressway based on the average speed and the number of vehicles, as well as the presence of accidents nearby. This represents the calculation of the toll in the original benchmark. **Step 5a** identifies and classifies areas in the expressway system where the traffic congestion is low, medium, or high. **Step 2b** processes and prioritizes queries; and **step 5b** estimates travel time and cost for a journey. These two last steps are executed synchronously since they generate replies to real time queries.

AQHI. Figure 6 depicts the workflow that calculates the air quality index (AQHI) on a given geographic region. This index represents a classification of the potential health risk that comes from air pollution. The workflow input is injected from detectors with three sensors to gauge the amount of Ozone (O_3), Particulate Matter ($PM_{2.5}$) and Nitrogen Dioxide (NO_2) in the atmosphere. In practice, each sensor corresponds to a different generating function, following a distribution with smooth variations across space These sample variations provide the necessary input data to the workflow

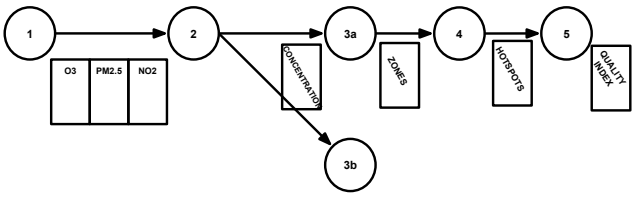


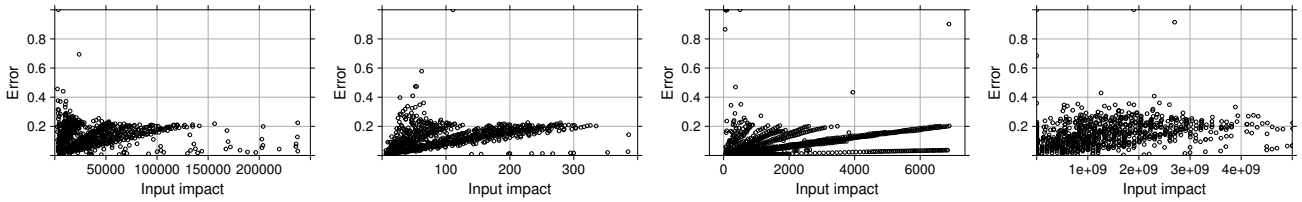
Figure 6: Workflow of the Air Quality Health Index

in each wave, corresponding to an hour of the day, for a total of 168 waves for a full week simulated. The generating functions return a value from 0 to 100, where 0 and 100 are, respectively, the minimum and maximum known values of O_3, $PM_{2.5}$ and NO_2. The workflow output corresponds to the generation of an index, a number, that is mapped into a class of health risk: low (1-3), moderate (4-6), high (7-10), and very high (above 10).

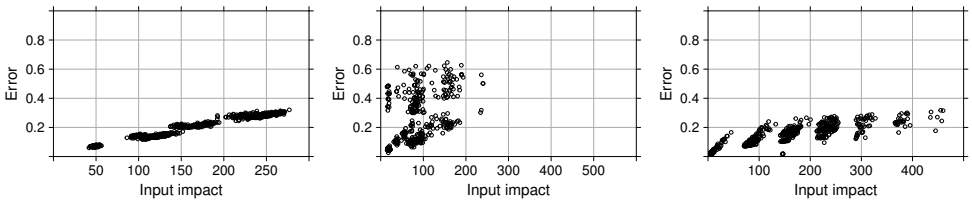
Step 1 simulates asynchronous and deferred arrival of sensory data. It continuously receives data from the atmospheric sensors and feeds the workflow by updating the first data container composed of 3 columns. **Step 2** calculates a single value, through a multiplicative model, representing the combined concentration of the 3 sensors for each detector. Every single calculated value is written on the column concentration of the data store. **Step 3a** divides the considered region in smaller areas and computes the aggregated concentration of pollution from detectors in each area. **Step 3b** processes the concentration of the area between detectors, thereby averaging the concentration perceived by surrounding detectors. It also plots a chart containing a representation of the pollution concentrations throughout the whole probed area for displaying purposes. **Step 4** assesses which of the previous stored zones have a concentration above a specified reference, which represents a point from which a zone is considered an hotspot (i.e., zone exhibiting an high level of pollution). **Step 5** reasons about the hotspots previously detected and, through a simple additive model that combines the number of hotspots with the average concentration of pollution on hotspots, it calculates an index that classifies the overall level of pollution in the given geographic region.

5.2 Learning Effectiveness

Correlation between Input Impact and Error. Fig. 7 shows the correlation between input impact and error for the main processing steps of LRB (figures 7(a)-7(d)) and AQHI (7(e)-7(g)), using a maximum tolerated error of 20%. We measure the sample Pearson correlation coefficient [13], *r*, which has a value between +1 and -1, where 1 is total positive linear correlation, 0 is no linear correlation, and -1 is total negative linear correlation. We observe that the correlations vary across steps and workloads, and that they are mostly neither linear nor trivial to be simply deduced; i.e., the coefficient *r* is closer to zero than to 1 in most cases (and especially for LRB), indicating that there is not a strong linear correlation. If they were obvious, other simpler techniques like linear regression would suffice. Hence, we justify the use of Machine Learning to



(a) LRB(3a) Avg Speed (r=0.065) (b) LRB(3b) Number of Cars (r=0.15) (c) LRB(4) Congestion (r=0.065) (d) LRB(5a) Classify (r=0.11)



(e) AQHI(3a) Zones (r=0.68) (f) AQHI(4) Hotspots (r=0.31) (g) AQHI(5) AQHI (r=0.87)

Figure 7: Correlation between input impact and error for the main processing steps of LRB and AQHI

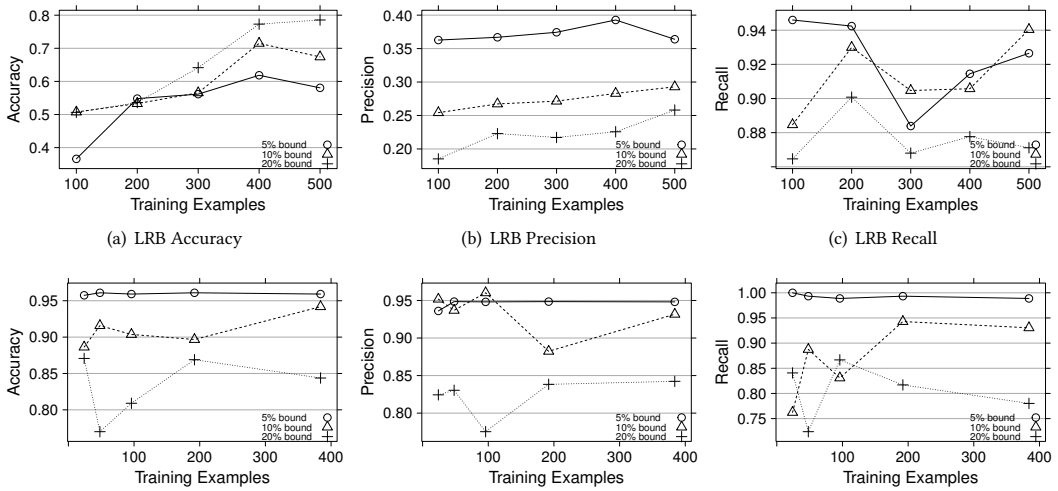
learn these complex patterns, that vary according to the computations being performed, and ensure the error is bounded, in a generic and seamless way.

Prediction Accuracy. Figure 8 shows the accuracy, precision, and recall of our learning model while varying the number of examples in the training-set, for LRB and AQHI using error bounds of 5, 10, and 20%. The examples contained in the test-sets were taken in subsequent waves as those of training-sets. 500 test examples for LRB and 384 for AQHI (respectively corresponding to a cycle of a pattern that repeats across time). Since LRB exhibited in general more variance in the (ι, ε) correlation patterns, we decided to optimize its classifier for recall, hence minimizing max_ε violations.

For figures 8(a)-8(c), we observe that the accuracy of the classifier for LRB improves as the number of training examples and error bound increase, up to 80% when $max_\varepsilon = 20\%$. This indicates that, with 500 examples, our learning model was able to predict the execution of steps in 60 upto 80% of the times in an optimal manner. Optimal means that max_ε was never exceeded and step re-execution was postponed as much as possible. However, not having a fully accurate model does not mean that max_ε is exceeded; e.g., re-execution can happen one wave before the ideal one, preventing max_ε from being reached, but also leaving space for one execution that could have been saved. We can also notice that the recall is always above 86% for more than 300 examples in the training-set, meaning that false negatives were reduced and true negatives augmented (i.e., maximizing max_ε compliance). As a consequence of optimizing for recall, we also get more false positives (less saved executions), which is represented by the precision metric.

As for AQHI, figures 8(d)-8(f), we observe that, with a bound of 5%, all metrics yield values equal or higher than 95%, which constitutes an excellent result (i.e., almost optimal resource savings and error compliance). The main reason for this is that the error variation, from wave to wave, was most of the time above 5% for the first 2 steps, which caused their re-execution in almost every wave. For an error bound of 10%, accuracy was roughly stable across different training-sets, and above 90% for more than 100 examples in training-set. For the same error bound of 10%, recall increased with the number of training examples upto roughly 100%, showing that max_ϵ was almost never violated. Conversely, precision slightly decreased with the number of examples, showing that the steps were re-executed more than the ideal necessary to stay within error limits. Finally, for $max_\varepsilon = 20\%$, there is an initial accentuated decline for accuracy and recall until roughly 100 training examples, probably corresponding to less than a complete pattern cycle. After that, accuracy goes from roughly 80 to 90%, and recall from 80 to 100%. AQHI is more stable than LRB, as expected since there is more bias and less variability in the input data, changing overall more smoothly across time. Therefore, the classifier requires less training examples to perform accurate predictions on new unseen examples. Intuitively, the higher the bound (i.e., the *slack* we allow for data modification over time), the higher potential for saving resources, but the less ability to avoid large deviations in the outcome of the execution.

Measured versus Predicted Errors. Across waves, Figure 9 shows the difference between predicted and measured errors for the last processing steps (that determine the workflow output) of LRB and AQHI using error bounds of 5,10, and 20%. The predicted errors were calculated by accumulating the simulated errors (when compared against the output of synchronous executions), according to the binary values returned by the classifier across waves. Figures 9(a)-9(c), 9(g)-9(i), show the predicted and measured errors in absolute value (Error). Figures 9(d)-9(f) and 9(j)-9(l) show the difference between predicted and measured errors (Prediction Deviation) for LRB and AQHI, respectively. A negative difference on a wave means that we were predicting the error below its actual (measured) value, and thus error bound violation did not happen for that wave. A positive difference on a wave means that the step was not executed and the predicted error stayed above max_ε. Globally, to maximize the ratio number-of-savings/number-of-violations, the predicted and measured errors should be as close as possible, so



(a) LRB Accuracy (b) LRB Precision (c) LRB Recall

(d) AQHI Accuracy (e) AQHI Precision (f) AQHI Recall

Figure 8: Accuracy, Precision, and Recall for LRB and AQHI with error bounds of 5, 10, and 20%

that the prediction deviation is around zero most of the time. Note that the figures show only markers for the outliers to the global trend.

For LRB with an error bound of 5 and 10%, we can see that the predicted error stayed below the measured error for most of the time, with a deviation down to -0.1 (figures 9(d), 9(e)). When max_ε was violated, 3 and 4 times with a bound of 5 and 10% respectively, the difference between predicted ε and max_ε was never above 0.3, and only 1 time above 0.15 (figures 9(d), 9(e)). For a bound of 20%, figures 9(c)-9(f), the quality of the prediction was degraded: the predicted error exceeded max_ε for a higher number of waves, albeit the prediction error was below 0.15 for most of the failed waves and below 0.45 for all waves. Nevertheless, max_ε violation occurred in less than 10% of the 500 waves, 82% of which with minor violation (< 0.15). Therefore, the potential for resource savings can be leveraged just at the expense of limited and mostly predictable additional error.

Regarding AQHI, figures 9(g)-9(l), we can see that, with an error bound of 5%, the deviation between predicted and measured error was minimal and max_ε violation happened in only 4 waves (< 0.012). With a bound of 10%, more prediction errors arose after 200 waves, albeit never exceeding 0.32 overall and 0.10 for the majority. Finally, for a bound of 20%, the number of prediction errors increases with errors staying below 0.6 overall and 0.25 for the majority. In a separate test, we optimized the classifier for recall, which produced no prediction errors, but degraded resource efficiency.

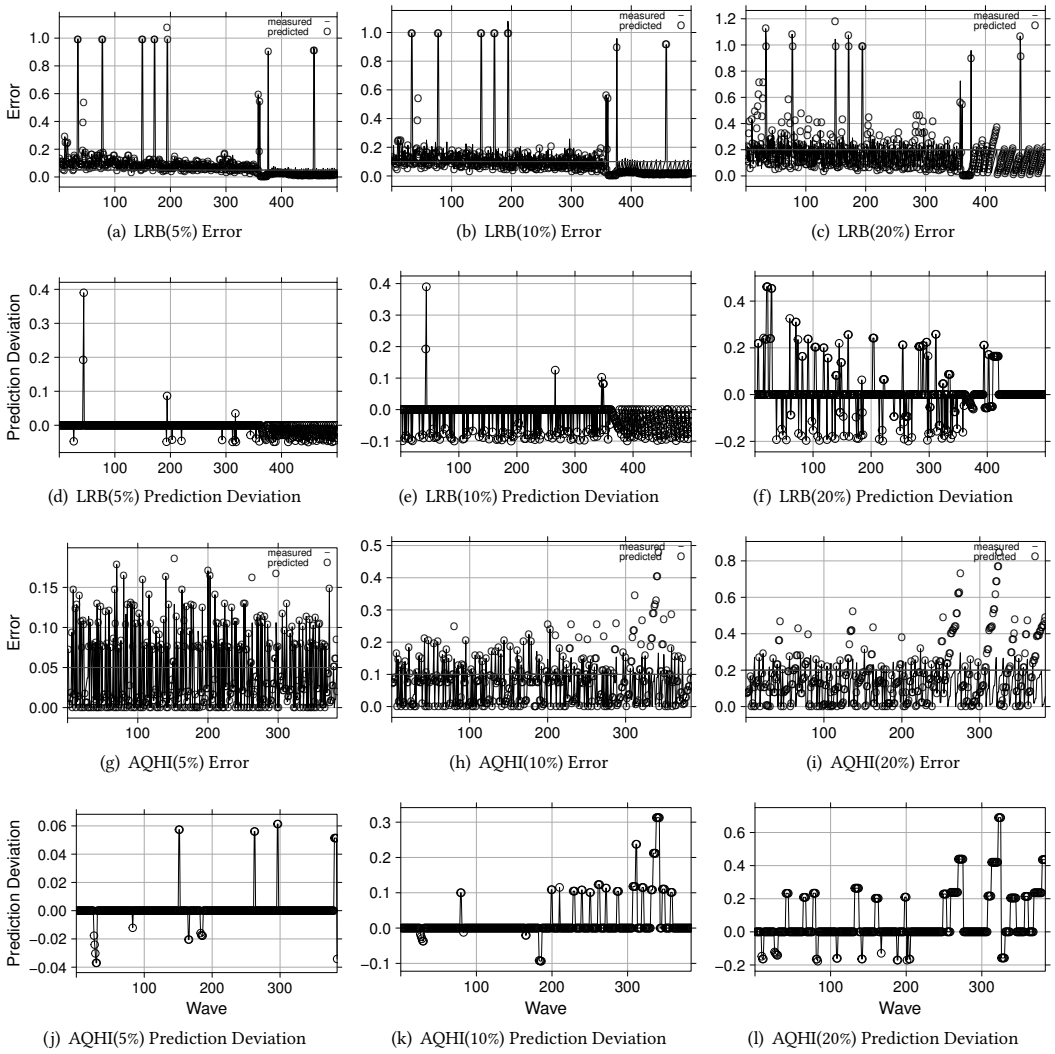
To conclude, the larger the error bound is, the higher is the number and magnitude of the errors obtained for prediction. This is expected as larger bounds on output difference allow for more (cumulative) deviation over time.

Confidence Levels. Figure 10 shows, for LRB and AQHI, the confidence of SmartFlux in complying with defined error bounds, which corresponds to the normalized cumulative sum of correct waves where max_ε was respected. We can see that, apart from the first 100 waves, the level of confidence was always above 95% for error bounds of 5 and 10% (i.e., for more than 95% of the times we are able to comply with error bounds of 5 and 10%). Nevertheless, with a bound of 20%, the confidence level raised quickly to more than 95 and 90% in LRB and AQHI respectively. This indicates that our system is reliable for decision makers. It can provide SLA-like guarantees stated as a confidence level (in %, that can be regarded as a probability) of being (consistently) under a given error limit provided by the user. This is akin to current cloud SLAs that promise to honor availability (or limits to latency - a limit on time) for a given percentage of the time, that can also be regarded as a probability.

To show how well SmartFlux makes *intelligent* decisions, we compare it with some naive approaches for an error bound of 5%. This bound was selected in order to get the best possible confidence from these approaches. The results of this comparison is given in Figure 11, where *random* consists of randomly skipping step execution (executing or not executing a step on a given wave has equal probability), and *seqX* consists of executing steps at every X waves. We observe that, either for LRB or AQHI, none of the approaches was better than SmartFlux, which offers more than 95% of confidence on error bound compliance. However, the other approaches revealed higher confidence in LRB than in AQHI, albeit never above 90% for most part of the waves. Note that a difference of 1% in confidence is statistically significant. The reason for such difference in these workloads lies in the fact that LRB can be better approximated by a linear function than AQHI (*seq2* has a pure linear behavior). However, only a Machine Learning approach can cover all cases, since polynomials can fit any type of correlation.

5.3 Resource Efficiency and Performance

Figure 12 shows the trade-off between the output error and the amount of executions (resources engaged), when compared with the synchronous model (SDF), (thus also illustrating resource savings due to avoided executions). For the cumulative sum of executions



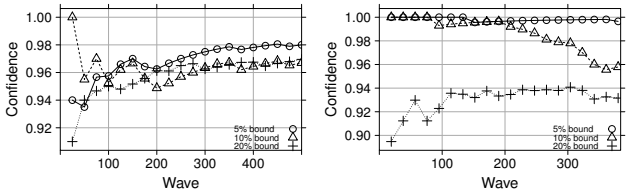
(a) LRB(5%) Error (b) LRB(10%) Error (c) LRB(20%) Error

(d) LRB(5%) Prediction Deviation (e) LRB(10%) Prediction Deviation (f) LRB(20%) Prediction Deviation

(g) AQHI(5%) Error (h) AQHI(10%) Error (i) AQHI(20%) Error

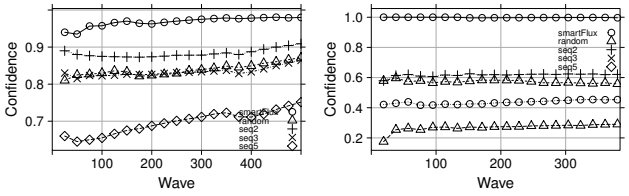
(j) AQHI(5%) Prediction Deviation (k) AQHI(10%) Prediction Deviation (l) AQHI(20%) Prediction Deviation

Figure 9: Difference between measured and predicted error for the last processing steps of LRB and AQHI with error bounds of 5, 10, and 20%



(a) LRB (b) AQHI

Figure 10: Confidence in respecting error bounds

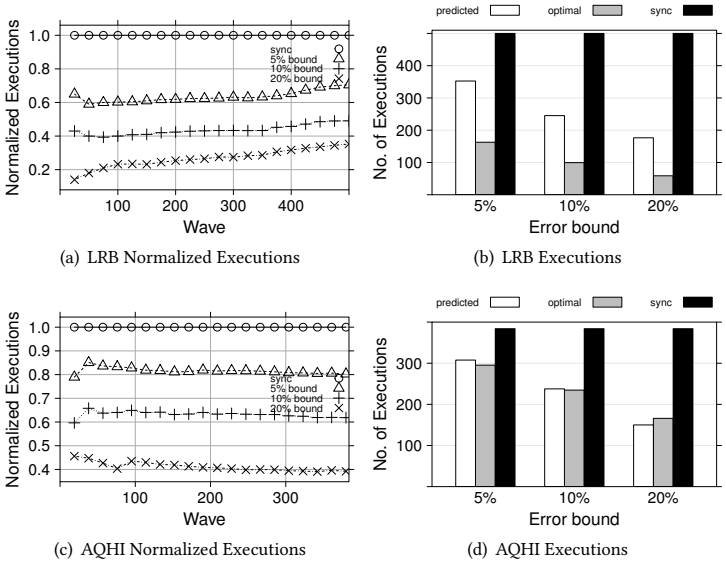


(a) LRB (b) AQHI

Figure 11: Comparison of confidence levels for different triggering approaches with an error bound of 5%

normalized over waves in LRB (Figure 12(a)), we can see that, with a bound of 5%, the workflow steps were executed on average less than 70% of the times in relation to the SDF model; i.e., more than 30% of the executions were saved, even for such a strict error bound.

With a bound of 10 and 20%, SmartFlux performed roughly 42 and 25% of the executions respectively, leading to resource savings up to 75%.



(a) LRB Normalized Executions (b) LRB Executions

(c) AQHI Normalized Executions (d) AQHI Executions

Figure 12: Executions performed with QoD versus synchronous model for LRB and AQHI

Every time an execution is avoided, the latest emitted results are made available immediately, resulting in very low execution time. This assessment is almost free by virtue of being several orders of magnitude faster than the actual execution, e.g., under 1 second against several minutes or even hours of execution. Therefore, to applications, it is transparently perceived as faster execution and entails a reduction in workload average execution times, up to 75% reduction, which can be considered as a 4x speedup on average.

Nonetheless, in Fig. 12(b) we can observe that we are not optimally efficient w.r.t. saving executions (i.e., delaying step triggering as much as possible without incurring in error violations, as it would be performed by a perfect fully-accurate predictor). This happened due to the optimization performed in the classifier to favor recall, leading to fewer saved executions yet to higher error compliance (which is more important for confidence in decision-making). Without favoring recall, and having a more close to optimal accuracy, more than 50% of the total predicted executions are saved.

For normalized executions in AQHI (Figure 12(c)), we see that the workflow is more stable, since the amount of saved executions is roughly the same across waves for each of the considered bounds. With a max_ε of 5, 10 and 20%, SmartFlux executes roughly 80, 60 and 40% of the times respectively on average against the SDF model; hence, corresponding to 20, 40 and 60% of saved executions as max_ε increases. As the correlation between input impact and error was more uniform over time, the patterns of this workflow were better predicted, as shown in Figure 12(d): the total number of predicted executions was very close to the optimal number for each of the considered bounds.

Once again, since assessment is almost free when compared to executing the workloads, this entails that 20, 40 and 60% of executions are perceived as near-zero execution time. On average, this can be considered as speedups of 1.25x, 1.66x and 2.5x, respectively. The overhead for each executed task is always close to 0%. Building the classification model is the only relevant source of overhead, albeit less than a second. Retrieving additional metadata from HBase has negligible overhead, piggybacked in previous requests.

With SmartFlux, we are thus able to save resources in exchange of allowing the occurrence of small yet bounded errors. As shown, roughly 20-30% of unnecessary executions are saved for a bound of 5%, and roughly 20-60% are saved for bounds of 10% and 20%, which is substantial in a cloud environment, where resources are paid for or shared among a multitude of users and applications.

Overhead. There are the following sources of overhead: i) monitoring accesses to the data store ii) computing the input impact; iii) computing the output error; iv) writing the training set to disk; v) building the classification model; vi) persisting previous computation state; and vii) classifying instances with input impact values. We relied on Application Libraries to intercept read/write calls to the data store (cf. § 4) and on the equations presented in § 2 to compute the input impact and output error. For each wave of data, we measured the running time of tasks that were executed by SmartFlux and compared with the time they take using the clean WMS version (without SmartFlux). The overhead for each task was always close to 0%. Note that the overall overhead of the system, for a large bounded period, is negative, since we are skipping executions with SmartFlux. Building the classification model took the longest time (among all sources of overhead), albeit less than a second. Also, persisting previous state took roughly 0% of overhead, since: i) we set writings to HBase to be non-blocking; and ii) reads were part of requests to read the actual state; i.e., when retrieving column families from HBase, we get the column qualifiers corresponding to the actual and previous state for the same time as we were requesting only one column qualifier.

6 RELATED WORK

This section reviews relevant proposed solutions, within the current state-of-the-art, that intersect the main topics approached in this paper. In the workflow domain, popular WMSs include: DAGMan, Pegasus, Taverna, Dryad, Kepler, Triana, Galaxy [12]. Since more complex analytics are extremely cumbersome to code as a giant set of interdependent MapReduce (MR) applications (i.e., limited reusability), WMSs running atop MR Hadoop [43], like Oozie [26], also started to arise, e.g., Azkaban [3], Cascading [4], Tez [1], and Pig [35]. Asynchronism has been explored in seminal work [42], but limited to a join operator and with the sole purpose of avoiding waiting for data from multiple concurrent predecessor tasks.

To avoid recreating web indexes from scratch after each web crawl, Google Percolator [36] performs incremental processing on top of BigTable, replacing batch processing of MR. It provides row and table-wide transactions, snapshot isolation, with locks stored in special Bigtable columns. Notify columns are set when rows are updated, with several threads scanning them. Applications are sets of custom-coded observers. It scales better than MR, but with 30-fold resource overhead over traditional RDBMS. Nova [34] is similar but has no latency goals, accumulating many new inputs and processing them lazily for throughput. Moreover, Nova provides data processing abstraction through Pig Latin; and supports stateful continuous processing of evolving data sets.

Yahoo CBP [31] aims at greater expressiveness by specifying incremental processing as dataflows with explicit mention when computation stages are stateless or stateful. Input is split by determining membership in frames of new records, allowing grouping input to reduce messaging. CBP offers primitives for explicit control flow and synchronize execution of multiple inputs, but requires extended MR implementation and explicit programming to be QoD-enabled.

Nectar [23] for Dryad links data and the computation that generated it as unified hybrid cacheable element. On programs reruns, Nectar replaces results with cached data, which requires cache management calls that update the cache server. This is transparently done in InCoop [14], which does caching for MR applications. Map, combine and reduce phase results are stored and memoized. Somehow like SmartFlux, this project attempts to reduce the number of executions; however, it implies that the input/output datasets are repeated or intersected among each other, whereas the QoD model fits a broader range of scenarios.

In [33], the authors present a formal model for defining temporal asynchrony in workflows. The operators have signatures that describe the types and consistency of the blocks accepted as input and returned as output. Data channels have a representation of time to a relation snapshot, with an interval of validity, which are used to enforce consistency invariants. These constraints, types of blocks permitted on output, freshness and consistency bounds, are then used by the scheduler which produces minimal-cost execution plans. This project shares our goals of exploring non ad-hoc solutions to enable asynchronous behavior in workflows, however, it does not account with the volume, semantically relevance or impact of modifications of the inputted data.

In our previous work [20], we propose a workflow model where task triggering is based on 3 user-defined constraints: i) the time to trigger a task; ii) the number of updates on the data; and iii) the magnitude of the updates. This allows flexible data-based execution, however it is difficult to manually set a combination of constraints in a workflow in order to keep the error within manageable levels; and no reasoning is performed about the impact computations have on varying the output. Thus, such model is not usable for scenarios where the freshness of the results needs to be guaranteed (like it is achievable with SmartFlux).

It is important to note that we do not discard any data, like it happens in load shedding [39]. In load shedding, a fraction of the input data is shed to alleviate overloaded servers and preserve low latency for query results. Contrarily to discarding data, we accumulate it up to the point where it causes significant changes on the output of the workflow. The observed errors happen not because we are making computations with incomplete data, but because we are not performing the computations and generating new output (i.e., errors come from stale data in the output).

Further, there has also been a recent effort to enable approximate processing in data processing systems (e.g., MapReduce, Stream Processing) in order to reduce latency (and possibly resource usage). However, these systems usually only target specific aggregation operators (e.g., sum, count) in structured languages [8, 9, 22]. In our work, we provide approximate results for general-purpose computations; i.e., we are agnostic to the code that is running on each processing step and solely observe the data that is inputted and its effect on modifying the output. Effectively learning the correlation between input and output allows us to bound the error and give (probabilistic) guarantees about the correctness of the results. This makes SmartFlux unique.

7 CONCLUSION

We presented a novel workflow model, for continuous and data-intensive processing, capable of dynamically controlling the triggering of processing steps based on the predicted impact that input data might have on changing the workflow output. This impact, and level of triggering control provided, represents the QoD that governs the system to attain resource efficiency, and improved performance, while maintaining results meaningful. To ensure correctness and freshness of these results, we bound the output deviation by making use of Machine Learning with Random Forests.

We also proposed SmartFlux, a middleware framework implementing our workflow model that can be effortlessly integrated with existing WMSs. Experimental results indicate that we are able to save a significant amount of resources in exchange of allowing small bounded errors to exist (up to 30% savings, which results in up to 1.4x speedups, while adhering to a strict bound of 5%). We provide compliance with error bounds with a high confidence level ($> 95\%$).

ACKNOWLEDGMENTS

We sincerely thank Andreas Wichert and Catarina Moreira for their insight on Machine Learning aspects, and the anonymous reviewers and our shepherd Fabio Kon for their overall feedback. This work was supported by national funds through Fundação para a Ciência e a Tecnologia with reference UID/CEC/50021/2013, PTDC/EEI-SCR/6945/2014.

REFERENCES

[1] [n. d.]. Apache Tez. https://tez.apache.org/.
[2] [n. d.]. AQHI. http://www.ec.gc.ca/cas-aqhi/.
[3] [n. d.]. Azkaban. https://azkaban.github.io/.
[4] [n. d.]. Cascading. http://www.cascading.org.
[5] [n. d.]. Oozie with SmartFlux. https://bitbucket.org/sergioesteves/oozie.
[6] [n. d.]. SmartFlux. https://bitbucket.org/sergioesteves/smartflux.
[7] [n. d.]. Social Business Index. http://www.socialbusinessindex.com/.
[8] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. 2014. Knowing when You'Re Wrong: Building Fast and Reliable Approximate Query Processing Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 481–492. https://doi.org/10.1145/2588555.2593667
[9] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 29–42. https://doi.org/10.1145/2465351.2465355
[10] James Ahrens, Bruce Hendrickson, Gabrielle Long, Steve Miller, Rob Ross, and Dean Williams. 2011. Data-Intensive Science in the US DOE: Case Studies and Future Challenges. *Computing in Science and Engg.* 13, 6 (Nov. 2011), 14–24. https://doi.org/10.1109/MCSE.2011.77
[11] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the 30th International Conference on Very Large Data Bases - Volume 30 (VLDB '04)*. VLDB Endowment, 480–491. http://dl.acm.org/citation.cfm?id=1316689.1316732
[12] Adam Barker and Jano van Hemert. 2008. Scientific Workflow: A Survey and Research Directions. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski (Eds.). LNCS, Vol. 4967. Springer Berlin Heidelberg, 746–753. https://doi.org/10.1007/978-3-540-68111-3_78
[13] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. In *Noise reduction in speech processing*. Springer, 1–4.
[14] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. 2011. Incoop: MapReduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11)*. ACM, New York, NY, USA, Article 7, 14 pages. https://doi.org/10.1145/2038916.2038923
[15] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (2001), 5–32. https://doi.org/10.1023/A:1010933404324
[16] Duncan A. Brown, Patrick R. Brady, Alexander Dietz, Junwei Cao, Ben Johnson, and John McNabb. 2007. A Case Study on the Use of Workflow Technologies for Scientific Analysis: Gravitational Wave Data Analysis. In *Workflows for e-Science*, Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields (Eds.). Springer London, 39–59. http://dx.doi.org/10.1007/978-1-84628-757-2_4
[17] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2002. Monitoring Streams: A New Class of Data Management Applications. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*. VLDB Endowment, 215–226. http://dl.acm.org/citation.cfm?id=1287369.1287389
[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=1267308.1267323
[19] Ewa Deelman et al. 2006. Managing Large-Scale Workflow Execution from Resource Provisioning to Provenance Tracking: The CyberShake Example. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing (E-SCIENCE '06)*. IEEE, Washington, DC, USA, 14–. https://doi.org/10.1109/E-SCIENCE.2006.99
[20] Sérgio Esteves, João Nuno Silva, and Luís Veiga. 2013. Flux: a quality-driven dataflow model for data intensive computing. *J. Internet Services and Applications* 4, 1 (2013), 12:1–12:23. https://doi.org/10.1186/1869-0238-4-12
[21] Lars George. 2011. *HBase: The Definitive Guide* (1 ed.). O'Reilly Media. http://www.amazon.de/HBase-Definitive-Guide-Lars-George/dp/1449396100/ref=sr_1_1?ie=UTF8&qid=1317281653&sr=8-1
[22] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. 2015. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 383–397. https://doi.org/10.1145/2694344.2694351
[23] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10)*. USENIX, Berkeley, CA, USA, 1–8. http://dl.acm.org/citation.cfm?id=1924943.1924949
[24] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* 11, 1 (Nov. 2009), 10–18. https://doi.org/10.1145/1656274.1656278
[25] Marti A. Hearst, ST Dumais, E Osman, John Platt, and Bernhard Scholkopf. 1998. Support vector machines. *Intelligent Systems and their Applications, IEEE* 13, 4 (1998), 18–28.
[26] Mohammad Islam, Angelo K. Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. 2012. Oozie: towards a scalable workflow management system for Hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies (SWEET '12)*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/2443416.2443420
[27] J.M. Juran and A.B. Godfrey. 1999. *Juran's quality handbook*. McGraw Hill. http://books.google.ca/books?id=beVTAAAAMAAJ
[28] Avinash Lakshman and Prashant Malik. 2009. Cassandra: structured storage system on a P2P network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC '09)*. ACM, New York, NY, USA, 5–5. https://doi.org/10.1145/1582716.1582722
[29] Xiang Li, Beth Plale, Nithya Vijayakumar, Rahul Ramachandran, Sara Graves, and Helen Conover. 2008. Real-time storm detection and weather forecast activation through data mining and events processing. *Earth Science Informatics* 1 (2008), 49–57. Issue 2. http://dx.doi.org/10.1007/s12145-008-0010-7 10.1007/s12145-008-0010-7.
[30] Jonathan Livny, Hidayat Teonadi, Miron Livny, and Matthew K. Waldor. 2008. High-Throughput, Kingdom-Wide Prediction and Annotation of Bacterial Non-Coding RNAs. *PLoS ONE* 3, 9 (2008), e3197+. https://doi.org/10.1371/journal.pone.0003197
[31] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. 2010. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1807128.1807138
[32] Bertram Ludäscher et al. 2009. Scientific Process Automation and Workflow Management. In *Scientific Data Management*, Arie Shoshani and Doron Rotem (Eds.). Chapman & Hall, Chapter 13. http://daks.ucdavis.edu/~ludaesch/Paper/ch13-preprint.pdf
[33] Christopher Olston. 2011. *Modeling and scheduling asynchronous incremental workflows*. Technical Report. Yahoo! Research.
[34] Christopher Olston et al. 2011. Nova: continuous Pig/Hadoop workflows. In *Proceedings of the 2011 international conference on Management of data (SIGMOD '11)*. ACM, New York, NY, USA, 1081–1090. https://doi.org/10.1145/1989323.1989439
[35] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*. ACM, New York, NY, USA, 1099–1110. https://doi.org/10.1145/1376616.1376726
[36] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 1–15. http://dl.acm.org/citation.cfm?id=1924943.1924961
[37] Jesse Read, Albert Bifet, Geoff Holmes, and Bernhard Pfahringer. 2011. Streaming Multi-label Classification. In *Proceedings of the Second Workshop on Applications of Pattern Analysis, WAPA 2011, Castro Urdiales, Spain, Oct, 2011*. 19–25. http://jmlr.csail.mit.edu/proceedings/papers/v17/read11a/read11a.pdf
[38] Ingo Steinwart and Andreas Christmann. 2008. *Support vector machines*. Springerverlag New York.
[39] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. 2007. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 159–170. http://dl.acm.org/citation.cfm?id=1325851.1325873
[40] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. 2004. Top-k Query Evaluation with Probabilistic Guarantees. In *Proceedings of the 30th International Conference on Very Large Data Bases - Volume 30 (VLDB '04)*. VLDB Endowment, 648–659. http://dl.acm.org/citation.cfm?id=1316689.1316746
[41] Grigorios Tsoumakas and Ioannis Katakis. 2007. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)* 3, 3 (2007), 1–13.
[42] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. 2003. Workflow Patterns. *Distrib. Parallel Databases* 14, 1 (July 2003), 5–51. https://doi.org/10.1023/A:1022883727209
[43] Tom White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.
[44] Donald G. York et al. 2000. The Sloan Digital Sky Survey: Technical Summary. *The Astronomical Journal* 120, 3 (2000), 1579.
[45] Jia Yu and Rajkumar Buyya. 2005. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing* 3 (2005), 171–200. Issue 3. http://dx.doi.org/10.1007/s10723-005-9010-8