ARTICLE IN PRESS

J. Parallel Distrib. Comput. ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets

Sérgio Esteves *, João Nuno Silva, João Paulo Carvalho, Luís Veiga

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

HIGHLIGHTS

- We offer a framework for resource efficient continuous and data intensive workflows.
- We are able to learn correlations between dataflow input and final output.
- We avoid re-executions when input data is predicted not to be impactful to the output.
- We ensure dataflow correctness within a small error constant.
- We achieve controlled performance, task prioritization and high resource efficiency.

ARTICLE INFO

Article history:
Received 19 November 2013
Received in revised form
14 February 2015
Accepted 11 March 2015
Available online xxxx

Keywords:
Workflow
Dataflow
Incremental processing
Continuous processing
Data-intensive
NoSQL
Quality-of-service
Machine learning
Fuzzy logic

ABSTRACT

Currently, there is a strong need for organizations to analyze and process ever-increasing volumes of data in order to answer to real-time processing demands. Such continuous and data-intensive processing is often achieved through the composition of complex data-intensive workflows (i.e., dataflows).

Dataflow management systems typically enforce strict temporal synchronization across the various processing steps. Non-synchronous behavior often has to be explicitly programmed on an ad-hoc basis, which requires additional lines of code in programs and thus the possibility of errors. More so, in a large set of scenarios for continuous and incremental processing, the output of dataflow applications at each execution can suffer almost no difference when comparing to the previous execution, and therefore resources, energy and computational power are unknowingly wasted.

To face such lack of efficiency, transparency, and generality, we introduce the notion of Quality-of-Data (QoD), which describes the level of changes required on a data store that cause the triggering of processing steps. This, so that the dataflow (re-)execution is reduced until its outcome would reach a significant and meaningful variation, which is inside a specified freshness limit.

Based on the QoD notion, we propose a novel dataflow model, with framework (Fluxy), for orchestrating data-intensive processing steps, which communicate data via a NoSQL storage, and whose triggering semantics is driven by dynamic QoD constraints automatically defined for different datasets by means of Fuzzy Boolean Nets. These nets give probabilistic guarantees about the prediction of the cumulative error between consecutive dataflow executions. With Fluxy, we demonstrate how dataflows can be leveraged to respond to quality boundaries (that can be seen as SLAs) to deliver controlled and augmented performance, rationalization of resources, and task prioritization.

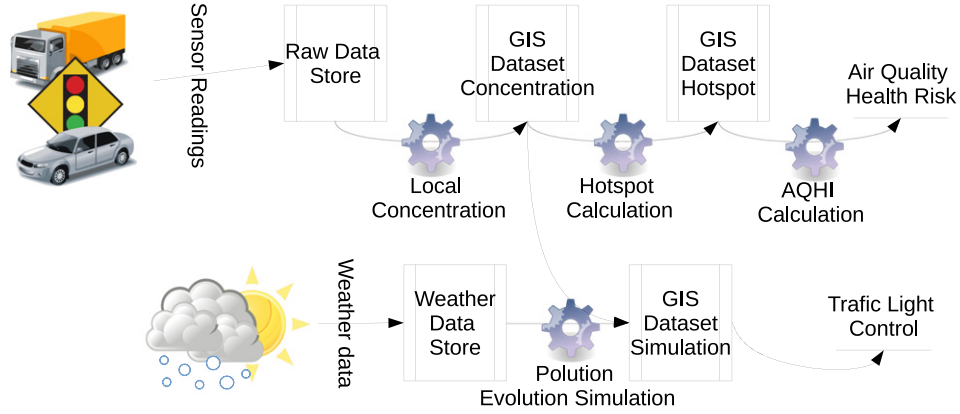
1. Introduction

Whether it is business, engineering or science settings, there is at present a growing need for organizations to analyze, process, organize and store vast quantities of raw data coming on a daily, hourly-basis, or more often, from a number of different sources (e.g., sensors, lab experiments, simulations, individual archives, enterprise and Internet). Essential to innovation, such data-intensive processing is often governed by complex data processing workflows (i.e., dataflows), which allow better expressiveness, maintainability and flexibility in comparison, for instance, with low-level data processing, such as Java map-reduce code.

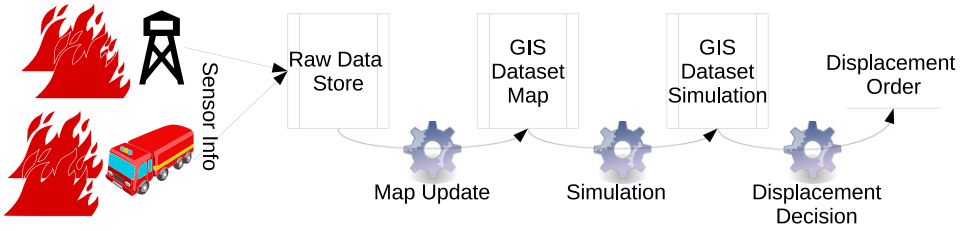
Dataflows can be represented as directed acyclic graphs (DAGs) that express the dependencies between computations and data.

* Corresponding author.
E-mail addresses: sesteves@gsd.inesc-id.pt (S. Esteves), joao.n.silva@inesc-id.pt (J.N. Silva), joao.carvalho@inesc-id.pt (J.P. Carvalho), luis.veiga@inesc-id.pt (L. Veiga).

http://dx.doi.org/10.1016/j.jpdc.2015.03.001
0743-7315/



(a) Air quality monitoring and measurement.



(b) Fire prevention, detection and prediction.

Fig. 1. Use-case scenarios of continuous, incremental, dataflow processing.

These computations, or processing steps, can potentially be decoupled from object location, inter-object communication, synchronization and scheduling. The data is either transferred directly from one processing step to another, or indirectly using intermediate files or via a shared storage system, such as a distributed file system or database.

There are many such data processing workflow applications [5,33]. For simplicity, we exemplify in Fig. 1 two scenarios where dataflows are used to process sensor data gathered from monitoring a large population/region and, through several computing steps, data is transformed and refined in order to produce higher-level output information that supports decisions (e.g., hotspots for pollution and fire risk, government issued indicators of air quality and fire risk for a given city, city district or country region). We detail such an example in Section 2.1.

We further explain some notions and the work setting. We consider asynchrony to take place in dataflow execution when the execution of one of more processing steps is delayed, i.e., it is not carried out as soon as new input is ready, but only when its amount or significance justifies it. Therefore, immediate processing of the new input is avoided. Since results of computations are incrementally incorporated in results of previous computations, allowing outputs to be updated and not recomputed, our work falls in the category of incremental processing.

Typically, we expect such decisions as above to enable savings of computing resources (namely CPU for processing, and network and I/O bandwidth to read/write data) as that is our main aim. These can be made available for other jobs in a shared computing infrastructure, improving its overall efficiency, or simply not hired from a public infrastructure with consequent financial savings. Nonetheless, this comes naturally at a cost, the error (that we intend to limit to preserve usefulness) resulting from maintaining (now) outdated dataflow output results. We address how to bound the impact of this phenomenon later in the paper when we present our model in detail in the next section.

In the use-case scenarios depicted in Fig. 1, this could be exemplified as follows. In case (a) this would amount to, as updated values of the sensors scattered over the city become available (e.g., every 5 min), the entire dataflow would not necessarily be re-executed immediately; only after a number of rounds with updated input data (henceforth referred to as waves) have generated enough modifications, considered relevant for the problem at hand, and for the dataflow results (e.g., high probability of changing the resulting air quality index and/or list of city hotspots regarding poor air quality). The number of input rounds (waves), without corresponding dataflow execution, results in obvious savings in computing resources, but can introduce an error regarding the previous dataflow results and the real air conditions in the city (that we will aim to minimize).

In case (b) the situation is analogous. To a significant extent, variations in updated sensor information (namely temperature) that are produced continuously, may not be significant enough to trigger recalculation, every time, of heat maps, hotspots for fire risk, and issuing alarms for displacement. There is an error associated with this (e.g., inexact knowledge about the average temperature in a given area in the forest) delay of execution, but while it is bounded and manageable, computing resources are saved and the dataflow results are still perfectly useful for decision making.

We do not consider this setting as one of distributed stream processing, for two orders of reasons: technology and concept. Technology wise, distributed stream processing is associated with systems such as Apache Storm[1] and Spark [41] that are not targeted by this work, that deals with systems like Oozie[2] and NoSQL data stores, hence we want to avoid confusion. Concept wise, stream processing deals with continuous input of new (immutable) data items or samples, with no specific container (or data location) associated, and that are processed in windows of fixed size (number of samples). Contrarily, in Map-Reduce based dataflow

[1] https://storm.apache.org/.
[2] http://oozie.apache.org/.

Please cite this article in press as: S. Esteves, et al., Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets, J. Parallel Distrib. Comput. (2015), http://dx.doi.org/10.1016/j.jpdc.2015.03.001

processing, input consists both of new (appended) data items as well as updates to the values of pre-existing (and hence mutable) data containers (rows, columns) referenced by an ID. Execution normally considers all relevant input in the stores, and not just part, hence being batch.

The software infrastructure to setup, execute and monitor dataflows is commonly referred to as Workflow Management System (WMS). Generally, WMSs either enforce strict temporal synchronization (or sequencing) across the various input, intermediate and output datasets (i.e., following the Synchronous Data-Flow (SDF) computing model [26]), or leave the temporal synchronization logic in the programmers' hands, who have often to explicitly program non-synchronous behavior to meet application latency and prioritization requirements. Additionally, these systems do not account for the volume of data arriving at each dataflow step, in the sense that the size of datasets could and should be recalled, and taken into account, to reason more accurately about its actual impact on the overall system performance and dataflow results. Executing a processing step each and every time a small fragment of data is received can have a great impact on performance and load, without actually resulting in a relevant change in the dataflow output (or more importantly, its significance), as opposed to executing it only when a certain substantial, relevant (w.r.t. application semantics) quantity of new data is available.

While not exclusive, this issue gains a special importance on a class of dataflow applications for continuous and incremental processing where the final result of the whole computations does not change significantly in a relatively short-to-medium time window. Examples include: measuring the impact of social business,[3] assessing the level of pollution in a given city [34], detecting gravitational-waves [7], weather forecasting [24], predicting earthquakes [14], among many others. Even for those whose outcome is always changing, such as a web crawler, the impact of the updated results may become only relevant when the differences from the previous crawl accumulate significantly (e.g., relevant change in word counts, page ranking or the number of reverse links).

Dataflow results are only significant if their variation against previous results is above a given, user-specific, threshold (which can be seen as a maximum tolerated error). If a dataflow output is predicted as not significant, we can postpone the execution of tasks for a later time, when more data is available, and save computational resources. In this work we exchange result accuracy, or error, with resource savings by delaying task execution (or enabling asynchrony). We tackle this optimization problem resorting to the notion of Quality-of-Data.

We define Quality-of-Data (QoD), which is akin to Quality-of-Service, as the specification of the overall level of assurance and performance of a system (in this case WMS) in terms of its manipulated data. QoD gives the ability to provide different priority to different datasets, users or dataflows, or to guarantee a certain level of performance and correctness of a dataflow. These guarantees can be enforced, for example, based on data size, number of accesses to a given data store object, or considering delays. With the QoD concept, we are thus able to define and apply temporal synchronization and divergence bounding semantics to dataflows, based on the volume and importance of the data communicated between processing steps; divergence bounding semantics means that the deviation of the output of a dataflow with QoD, in relation to the synchronous model, is expected to not be above a given tolerated error constant. Moreover, relying on QoD, we can augment the throughput of the dataflow and reduce the number of its actual (re-)executions, while keeping the results within acceptable limits, regarding each application's semantics, of staleness and/or divergence. When multiple dataflows continuously executing share an infrastructure (e.g., cloud), resource savings are impactful.

With the current context in mind, we introduce a novel dataflow model, with a framework and library support we call *Flu*χ*y*, for orchestrating data-based and -intensive processing steps which communicate data via a NoSQL storage, and whose triggering semantics is driven by dynamic QoD constraints, automatically defined for different subsets of data by means of Fuzzy Boolean Nets [8]. These nets are used to learn statistical behavior about the execution of dataflows, correlating input variation with output generated error. This error comes from postponing the execution of processing steps when the input variation is not assessed to have sufficient impact to create significant changes in the dataflow output; i.e., error corresponds to the output difference as if we had not delayed the executions (like in the typical synchronous model). Hence, we trade-off result accuracy (with bounded error) with computational resource savings.

Fluχ*y* was developed and deployed using Oozie[4] as WMS, and HBase [17] as the underlying NoSQL storage. Experimental results show that we are able to optimize resource utilization, while giving probabilistic guarantees about the convergence of the dataflow final output and maximum error (i.e., the error that arises from postponing the execution of processing steps against the synchronous model).

Section 2 presents our dataflow model and Section 3 describes our learning approach to bound the cumulative error. Section 4 presents the *Flu*χ*y* framework and Section 5 details its implementation. Section 6 shows our experimental results with *Flu*χ*y*, and Section 7 reviews related work. Section 8 closes the paper.

2. Dataflow model

In this section we describe in detail the QoD dataflow model (from our previous works: [15,16]), employed by *Flu*χ*y*, which was specially designed to address large-scale and data-intensive scenarios that need to continuously and incrementally process large sets of data, while maintaining strong requirements about the quality of service and data provided.

Our model inherits from and extends the traditional workflow model [40] and is to be deployed on cloud scenarios.

Our dataflow model can be expressed as a directed acyclic graph (DAG), where each node (vertex) represents a processing step (designated here by action) that carries out read and write operations in a data store; and the edges between actions represent dependencies and flow of data, meaning that an action needs the output of other precedent action(s) to get executed. More precisely, each action A, in a dataflow D, is executed only after all actions A' preceding A (denoted $A' \prec_D A$) in D have been executed at least once (elaborated hereafter). In addition, actions can be divided into: input actions, which are supplied with data from external sources; intermediate actions, which receive data from other actions; and output actions, whose generated data is read by external consumers.

It implies that the underlying data, shared among processing steps, should be done via tabular NoSQL data stores. Whereas most workflow models rely on files to store and share the data, which cannot achieve the same scalability and flexibility [9], hence we believe this assumption to fit many, if not most, large scale data processing scenarios. In our model, these NoSQL data stores are regarded as collections of object containers (hereafter represented

[3] http://www.socialbusinessindex.com/.

[4] http://oozie.apache.org/.

Please cite this article in press as: S. Esteves, et al., Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets, J. Parallel Distrib. Comput. (2015), http://dx.doi.org/10.1016/j.jpdc.2015.03.001

as o), and may have different granularity and consist of different data storage units; e.g., column, or group of columns in a table. Each of these object containers (present in a previously defined schema) will be associated to a certain level of change necessary in order to trigger subsequent processings step.

In particular, what differentiates and demarcates our model from the other typical DAG dataflows is the triggering semantics: a processing step A, in a dataflow D, is not necessarily triggered for execution immediately, when all its predecessors A' ($A' \prec_D A$) have finished their execution (and inserted new values in the data store). 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 level of changes on the underlying data container, that comply with certain QoD requirements.

This level of data changes is denoted by QoD bound κ and is specified through multi-dimensional vectors. These associate QoD constraints with data object containers, such as a column, or group of columns, in a table of a given column-oriented database. κ bounds the maximum level of changes through vectors of numeric scalars, each defined for one of the following orthogonal dimensions: time (θ), sequence (σ), and value (ν):

Time Specifies the longest span of time a step can be on hold (without being triggered) since its last execution occurred. Considering $\theta(o)$ provides the time (e.g., seconds) passed since the last execution of a certain step, that is dependent on the availability of data in the object container o, this time constraint κ_θ enforces that $\theta(o) < \kappa_\theta$ at any given time.

Sequence Specifies the maximum number of updates that can be applied to an object container o without triggering a step that depends on o. Considering $\sigma(o)$ indicates the number of applied updates over o, this sequence constraint κ_σ enforces that $\sigma(o) < \kappa_\sigma$ at any given time.

Value Specifies the maximum relative divergence between the updated state of an object container o and its previous state, or against a constant (e.g., top value), since the last execution of a step dependent on o. Considering $\nu(o)$ provides that difference (e.g., in percentage), this value constraint κ_ν enforces that $\nu(o) < \kappa_\nu$ at any given time. It captures the impact or importance of updates in the last state.

The QoD bound κ, associated with an object container o, is reached when any of its vectors has been reached, i.e., $\theta(o) \geq \kappa_\theta \vee \sigma(o) \geq \kappa_\sigma \vee \nu(o) \geq \kappa_\nu$. Also, grouped containers (e.g., a column and a row) are treated as single containers, in the sense that modifications performed on any of the grouped objects refer to the same κ.

Moreover, the triggering of a processing step can depend on the updates performed on multiple database object containers, each of which possibly associated with a different κ. Hence, it is necessary to combine all associated constraints to produce a single binary outcome, deciding whether or not the step should be triggered. To address this, we provide a simple QoD specification algebra with the two logical operators *and* and *or* ($\wedge$ and $\vee$) that can be used between any pair of QoD bounds.[5] The *and* operator requires that every associated QoD bound κ should be reached in order to trigger an associated step; while the *or* requires that at least one κ should be reached for the triggering of the processing step. Following the classical semantics, the operator *and* has precedence over operator *or*. For example, a step A can be associated with the expression $\kappa_1 \vee \kappa_2 \wedge \kappa_3$, which causes the triggering of A when κ_1 is reached, or κ_2 and κ_3 have been both reached.

Besides these tridimensional vectors, it is also possible to capture the impact of data through more complex functions with greater expressiveness. For example, through HBase co-processors,[6] or Cassandra triggers,[7] it would be possible to provide user-defined functions, described by Java code snipets, that calculate the significance of updates performed and simply notify the *Fluχy* framework through the same API used by the application libraries that we further describe in Section 4. Nevertheless, providing a higher-level, Domain-Specific Language (DSL), is out of the scope of this particular work, but still in our future plans.

2.1. Prototypical scenario

As a motivational example we describe a dataflow, for continuous and incremental processing, that expresses a simulation of a prototypical scenario inspired by the calculation of the Air Quality Health Index (AQHI),[8] used in Canada. It captures the potential human health risk from air pollution in a certain geographic area, typically a city, while allowing for more localized information. The incoming data fed to this dataflow is obtained through several detectors comprising three sensors to gauge the amount of Ozone (O_3), Particulate Matter ($PM_{2.5}$) and Nitrogen Dioxide (NO_2).

Fig. 2 illustrates the dataflow with the associated QoD vectors and the main columns (some columns were omitted for readability purposes) that comprise the data containers in which the processing steps' triggering depends on. k specifies (i) the maximum time, in hours, the step can be on hold; (ii) the maximum accepted divergence in updates, in units, and (iii) the minimum amount, in percentage, of changes necessary to the triggering (e.g., 20% associated to step C means that this will be triggered when at least 20% of the detectors have been changed by step B).

We describe each processing step in the following.

Step A This step continuously feeds data to the dataflow by reading sensors from detectors that perceive changes in the atmosphere to simulate asynchronous and deferred arrival of update sensory data. The values from each sensor are written in three columns (each row is a different detector) which are grouped as a single data container with one associated k.

Step B Calculates the combined concentration (of pollution) of the three sensors for each detector whose values were changed in the previous step. Every single calculated value is written on column *concentration*.

Step C Processes the concentrations of small areas, called zones, encircled by the previously changed detectors. These zones can be seen as small squares within the overall considered area and comprise the adjacent detectors (until a distance of two in every direction). The concentration of a zone is given by a simple multiplicative model of the concentration of each comprising detector.

Step D Calculates the concentration of points of the city between detectors, thereby averaging the concentration perceived by surrounding detectors; and plots a chart containing a representation of the concentrations throughout the whole probed area, for displaying purposes, and reference of concentration and air quality risk indicator in localized areas of a city. This step can be executed in parallel with Step E.

[5] Currently, *not* is applied explicitly by inverting conditions.

[6] http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/coprocessor/package-summary.html.

[7] http://www.datastax.com/documentation/cql/3.1/cql/cql_reference/trigger_r.html.

[8] www.ec.gc.ca/cas-aqhi/.

Please cite this article in press as: S. Esteves, et al., Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets, J. Parallel Distrib. Comput. (2015), http://dx.doi.org/10.1016/j.jpdc.2015.03.001

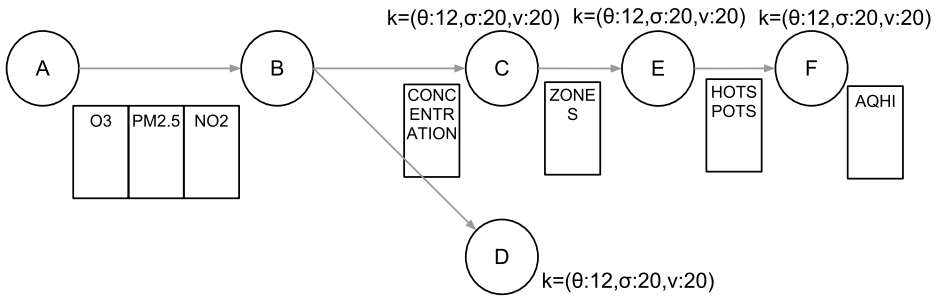


Fig. 2. *Flu*χy dataflow for AQHI calculation.

Step E Analyzes the previous stored zones and respective concentrations in order to detect hotspots; i.e., zones where the overall concentration is above a certain reference. Zones deemed as hotspots are stored in column *hotspots* for further analysis.

Step F Performs final reasoning about the hotspots detected, thereby combining, through a simple additive model, the amount (in percentage) of hotspots identified with the average concentration of pollution on all hotspots. Then, the AQHI index is produced and stored for each wave of incoming data.

2.2. Output error

Procrastinating the execution of processing steps introduces divergence in the dataflow output values, as opposed to the synchronous model. Each time a step is not executed upon a wave of incoming data, due to its QoD requirements, an error may be introduced as in the variation of output that should have been changed and was not. For example, let us assume the output of a dataflow is an integer (output of the dataflow is the output of the last processing steps in the dataflow): in the synchronous model, we get the values 1, 2, and 3 for three sequential waves; whereas in the QoD model we get 1, 1, and 3 for those same waves of data; in the second wave some tasks were not executed in the QoD model (a change of 1 unity was not considered significant) and therefore an error was introduced.

Intuitively, for each step, the QoD bounds serve as a hint both on how to save resources (by postponing re-execution), as well as on how to bound the admissible error (introduced by such savings). Nevertheless, small local errors in steps tend to accumulate over the execution of the dataflow processing chain, which can lead the processing to an undesirable and incorrect state; in practical terms, this error is very difficult to predict, estimate analytically or set bounds on.

The cumulative error, ε, is given in Eq. (1), where x_i is the updated state of the element of order i and x'_i its latest state, m the number of modified elements, and n the total number of elements in the respective data container.

$$\varepsilon = \sum_{i=1}^{m} |x_i - x'_i| \times m \Big/ \left(\sum_{i=1}^{n} x'_i \times n \right). \quad (1)$$

In order to bound the accumulated error and guarantee a certain level of correctness we learn statistical behavior of dataflows, elaborated in the next section.

3. Learning

This section presents our learning approach to bound the cumulative error, intrinsic to our dataflow model, and therefore provide guarantees about the maximum deviation of the outcomes. Our approach falls into the category of supervised learning, since we need a training phase to understand and infer the behavior, in terms of input variation versus generated error, of classes of dataflows over certain periods of time. To accomplish this, we rely on Fuzzy Boolean Nets (FBN) which provide probabilistic guarantees about the prediction of the cumulative error between consecutive dataflow executions. Our model is particularly suitable to dataflows for continuous and incremental processing that exhibit similar input pattern over a cycle of time, i.e., no random and completely uncorrelated input over time. While not limited to, such class of dataflows is commonly found in e-Science [35], especially in monitoring activities related to nature (e.g., weather forecasting, temperature and fire risk assessment), as well as in human activities (e.g., commerce) that follow seasonal changes and patterns.

3.1. Fuzzy Boolean nets overview

FBN [8,38] exhibit the natural or biological neural systems features that lead to a learning capability when exposed to sets of experiments from the real world. In FBN, neurons are grouped into areas. Each area can be associated with a given variable, or concept. Meshes of weightless connections between antecedent neuron outputs and consequent neuron inputs are used to perform if–then inference between areas. Neurons are binary, and the meshes are formed by individual random connections (just like in nature). Each consequent neuron contains m inputs for each antecedent area, that is, a total of $N \times m$ inputs, and up to $(m+1)^N$ internal unitary memories (*ff*), where N is the number of antecedents. This number corresponds to maximum granularity, and can be reduced. In the case of maximum granularity each *ff* is addressed by a unique joint count of activated inputs on the m inputs from each of the N antecedents. As in nature, the model is robust in the sense that it is immune to individual neuron or connection errors (which is not the case of other models, such as the classic artificial neural net) and presents good generalization capabilities. The value of each concept, when stimulated, is given by the activation ratio of its associated area (the relation between active – output '1' – neurons and the total number of neurons). Inference in FBN proceeds as follows: Each consequent neuron randomly samples each of the antecedent areas using its m inputs, with m always much smaller than the number of neurons per area. Each neuron then performs a combinatorial count of the number of activated inputs from every antecedent (those with Boolean value '1'). Neurons have a unitary memory (*ff*) for each possible count combination, and its value will be compared with the corresponding sampled value. If the *ff* associated to the sampled values contains a '1', then the neuron output will be '1'; if the *ff* is '0', then the neuron output will be '0'. As a result of the inference process (which is parallel), each neuron in the consequent area will assume a binary value, and the inference result will be given by the neural activation ratio in the consequent area. As shown in [38], from these neuron micro operations emerges a macro qualitative reasoning capability

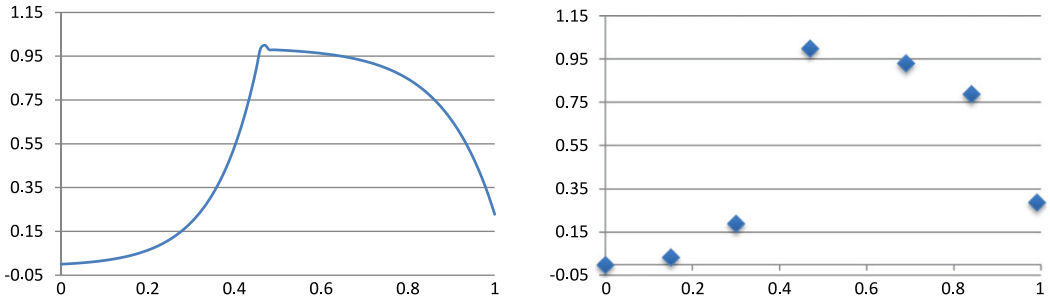


Fig. 3. (Left) Non-linear function used to test interpolation capability in the presence of sparse data; (Right) Training set used to check the interpolation capability in the presence of sparse data.

if *Antecedent*$_1$ *is* A_1 **and** *Antecedent*$_2$ *is* A_2 **and** … **then**
  | *Consequent is* C_i
end

involving the concepts (fuzzy variables), which can be expressed as fuzzy rules of type:
where *Antecedent*$_1$, *Antecedent*$_2$, . . . are fuzzy variables and A_1, A_2, . . . , C_i are linguistic terms with binomial membership functions (such as 'small' and 'high').

Learning is performed by exposing the net to experiments and modifying the internal binary memories of each consequent neuron according to the activation of the m inputs (per antecedent) and the state of that consequent neuron. Each experiment will set or reset the individual neuron's binary memories. Since FBN operation is based on random input samples for each neuron, learning (as well as inference) is a probabilistic process. For each experiment, a different input configuration (defined by the input areas specific samples) is presented to each and every of the consequent neurons, and addresses one and only one of the internal binary memories of each individual neuron. Updating of each binary memory value depends on its selection (or not) and on the logic value of the consequent neuron. As shown in [38], this may be considered a Hebbian (associative) type of learning [19]. Hebbian learning is usually summarized with the sentence "Cells that fire together, wire together": if neural area Y becomes active whenever (or shortly after) neural area X is active, then a relationship should be created (learned) between X and Y. The principles behind Hebbian learning state that simultaneous activation of areas lead to pronounced increases in synaptic strength between those areas, and provide a biological basis for errorless learning methods for education and memory rehabilitation.

A FBN is capable of learning a set of different rules without cross-influence between different rules, and the number of distinct rules that the system can effectively distinguish (in terms of different consequent terms) increases with m. As Universal Approximators, these networks are capable of learning and implementing any possible multi-input single-output function of the type $[0, 1]^n \times [0, 1]$.

3.2. FBN performance under sparse data conditions

When used as universal approximators, FBN compare well with other competing techniques especially when ease of parameterization is an issue. The most favorable results are obtained when the training data is very sparse and with imbalanced datasets; FBN have a rather good performance in such conditions. The following case example compares FBN performance in such conditions with other state of the art approaches.

We compare the behavior of the FBN with other approaches when learning and interpolating sparse data. Basically we show FBN generalization capabilities when compared to other popular alternatives: ANFIS, Artificial Neural Networks (MLP), Cubic spline interpolation, Support Vector Machines (SVM) and Isotonic Regression Model [22]. Since traditional error measures such as the root-mean-square error (RMSE) do not necessarily show how well a method performs, a (non-linear) single input function was chosen in order to allow for a pictorial representation of the behavior of each approach. The used function is depicted in Fig. 3 (Left side). The 7 data points shown in Fig. 3 (Right) were used as the training set.

ANFIS was implemented using the Fuzzy Matlab toolbox,[9] choosing the hybrid optimization method and training error tolerance = 0.005. The resulting optimized ANFIS converged after 100 epochs and uses 3 membership functions. The MLP, Support Vector Machines (SVM) and Isotonic Regression Model were implemented using Weka[10] (other methods present in Weka were also tested, but these were the ones that gave the best results). In order to improve the MLP performance, we used two-hidden layers and 50 000 training epochs instead of the Weka default parameters. Finally the cubic spline interpolation was implemented using an online available tool.[11]

Table 1 shows the obtained results. The FBN has both the best correlation and RMSE, followed by ANFIS and the Cubic Spline Interpolation. The Neural Network also has an acceptable performance, while SVM and the Isotonic Regression model clearly give the worse results.

Even if the FBN outperforms the other tested methods in what concerns the RMSE in the test set, their capabilities to generalize sparse training data are more evident when observing the results presented in Fig. 4, where it is clear that the FBN is the only method able to cope with the sudden inflection in the test function, and the only method to guarantee an acceptable error bounded reply (SVM and IRM are not shown in order to simplify the figure).

FBN can be very useful whenever the cost of obtaining data is high (either in time and/or resources) and a sparse number of sample data is the best one can afford or aspire for. This situation is common in problems where data is obtained from lengthy experiments demanding resources preventing more than a few experiments to be run simultaneously. In what concerns the present work, the used dataset is heavily imbalanced (there are much less training instances over the QoD than under), and it is also sparse. Therefore, FBN appear to be the most adequate technique to be used within *Flu*χy in order to enforce the required QoD guarantees.

[9] http://www.mathworks.com/products/matlab/.
[10] http://www.cs.waikato.ac.nz/ml/weka/.
[11] http://www.akiti.ca/CubicSpline.html.

Table 1
RMSE and correlation results on a very sparse data test.

	FBN	ANFIS	Cubic spline interpolation	NN	SVM	IRM
RMSE (test dataset)	0.052	0.061	0.063	0.071	0.104	0.166
Correlation (test dataset)	0.992	0.990	0.991	0.988	0.965	0.905

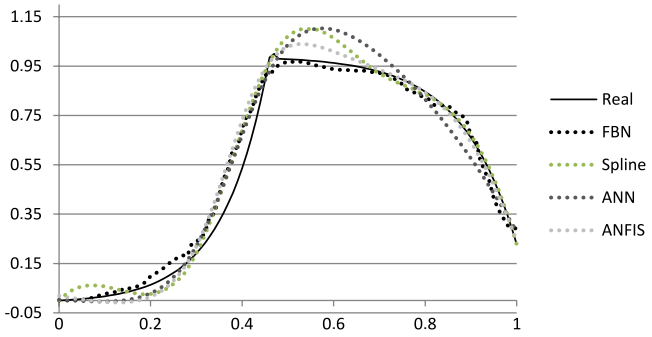


Fig. 4. Correlation results.

3.3. Proposed approach

In the remaining of this section, we only address the more relevant aspects for applying FBN to our problem scenario.

Training phase. We train FBN to estimate the cumulative divergence in dataflow execution output, as a function of the cumulative modifications injected at the input of its first processing step.

During training, at each wave of incoming data (e.g., every 10 min in a real deployment scenario), all processing steps should be executed synchronously without QoD enforcement. For each wave is calculated the *input impact*, ι, and the output generated error (Eq. (1)). ι is calculated through Eq. (2), where x_i is the updated state of the element of order i and x'_i its latest state, and m is the number of modified elements in the respective data container.

$$\iota = \sum_{i=1}^{m} |x_i - x'_i|. \tag{2}$$

When the last processing step of the dataflow completes, a pair with input impact (ι) and cumulative error (ε) is appended to a log. The values of this pair have to be normalized and mapped from 0 to 1 so that they can be processed by the FBN. For that, we calculate the ratio of ι and ε against their corresponding maxima in the entire log.

Once the log reaches a considerable size, it is split in half into two sets: a training set, used to teach the FBN; and a test set, later used to verify the learning discrepancy in the test phase. With the training set, the FBN are then trained using the (ι, ε) pairs. These pairs are presented to the network in sequence for a number of iterations that conforms with the net dimension and training quality (further elaborated in Section 4). Unlike most supervised learning techniques, FBN have excellent generalizing capabilities and can provide a good approximation even for sparse and unbalanced training data (cf. Section 6).

Test phase. Firstly, it is necessary to define an upper bound, $\max_\varepsilon$, specifying the maximum tolerated error for a given dataflow. Then, using the test set, we verify the FBN previously obtained (for each considered QoD) in order to assess the system performance and, foremost, the existence of false negatives, i.e., the percentage of times the FBN estimate an error under the $\max_\varepsilon$ when the real error is above it. Additionally, the percentage and quality of false positives is also assessed, i.e., estimations above the $\max_\varepsilon$ while the real error is under it, and how different was such prediction. False positives are admissible if the real error is close to $\max_\varepsilon$ (e.g. $\max_\varepsilon = 5\%$, real error $= 4.7\%$, estimation is larger than 5%), but indicate a failure if the difference is large (e.g. $\max_\varepsilon = 5\%$, real error $= 1\%$, estimation above 5%). That difference between the estimated and real error is used to assess the quality of the FBN on guaranteeing a certain QoD. The quality is afterwards recalled to guide the amount of necessary training (i.e., poor quality implies more training, and vice-versa).

If the FBN quality is satisfactory, the trained FBN are ready to be used at runtime to estimate $\max_\varepsilon$ for the error given the current amount of change.

Application phase. At this phase the trained FBN receive at each wave of data a tuple with the normalized ι and desired $\max_\varepsilon$. ι is given as input to the FBN and the net is executed a sufficient amount of times (with a certain degree of confidence). FBN operation is probabilistic, therefore, besides the expected error, we also obtain its standard deviation (as a result of the execution). The upper bound ($\max_\varepsilon$) for the error estimation is given by the expected error plus its standard deviation. The system uses the upper bound and the QoD guarantee obtained in the test phase to give the estimation confidence.

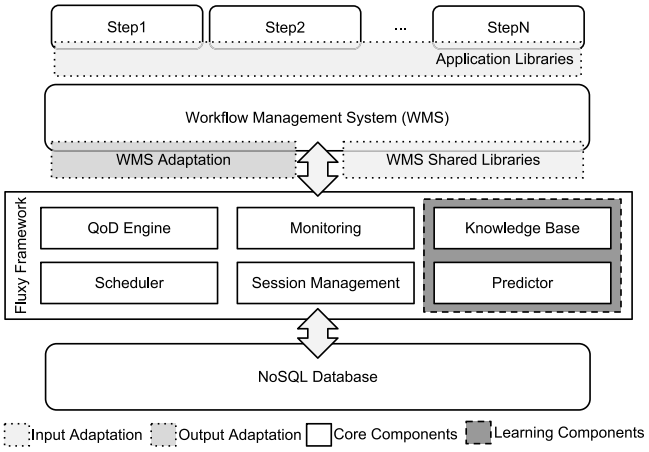


Fig. 5. *Fluχy* middleware framework architecture.

4. *Fluχy* framework

This section presents the architecture and main design choices of the *Fluχy* framework, which implements the model described in the previous section. *Fluχy* enables the construction of quality-driven dataflows in which the triggering of processing steps may be delayed to comply with QoD constraints. These constraints are defined over datasets on which those steps depend and, hence, a tight coupling is assumed between the framework and the underlying storage system.

Fig. 5 depicts the archetypal architecture of the *Fluχy* middleware framework, which operates between a workflow management system and a NoSQL tabular-like database. Processing steps run on top of the workflow manager and they must share the intermediate data through the underlying database. These steps may consist of Java applications, scripts expressed through high-level languages for data analysis (e.g., Apache Pig [31]), map-reduce jobs, as well as other *off-the-shelf* solutions.

Although *Fluχy* can operate with its own provided and simple WMS, we concentrate here on using an existing one (Oozie),

widely deployed, also to assess in what extent it requires changing its implementation and triggering mechanisms. In effect, an adaptation component, WMS Adaptation (colored in medium gray), needs to be provided with a specified API in order to, foremost, receive from *Fluχy*, triggering notifications, and control the execution of the dataflow processing steps. For the input adaptation, we provide database adaptation libraries (colored in light gray) at the client side, so that *Fluχy* may be aware of the updates performed by steps on the underlying database. The Application Libraries component consists of adapting application libraries, that are used by processing steps, to interact directly with the data store via their client APIs; thus, albeit applications need to be slightly modified, we are developing the tools to carry out the required mappings that are straightforward, in order to completely automatize this process.

At a lower level, WMS Shared Libraries are used to adapt processing step applications that need to access the database through some shared library given by the WMS (e.g., pig scripts or any other high-level language that must be compiled by the WMS); it provides transparency to executing steps and therefore applications need not be modified.

Next, we describe the responsibilities and purpose of each of the components that compose the *Fluχy* framework (in white).

Monitoring It analyzes all requests directed to the database. It uses information contained in update requests to maintain the current state of control data regarding the quality-of-data.

Session Management It manages the configurations of the QoD constraints, over data objects, through the meta-data that is provided, along with the dataflow specification, and defined for each different dataflow. A dataflow specification is then derived to the target WMS.

QoD Engine It maintains data structures and control meta-data which are used to evaluate and decide when to trigger next steps, obeying to QoD specifications.

Scheduler It verifies the time constraints over the data. When the time for triggering successor steps expires, the Scheduler notifies the QoD Engine component in order to clear the associated QoD state and notify the WMS to execute the next processing steps.

Knowledge Base It maintains data collected from the Monitoring component, (ι, ε) pairs, to train the FBN and adjust the Predictor component.

Predictor It uses a trained FBN from the Knowledge Base component and predicts the dataflow output error given the dataflow input variation at the current wave. The estimated error is used for adjusting the QoD constraints (i.e., if the error is greater than $\max_\varepsilon$, the QoD is restricted, otherwise is relaxed).

4.1. Session management

Dataflow specification schemas need to be provided to register dataflow applications with *Fluχy*. They should contain the description of the dataflow graph where each processing step must explicitly specify the data containers (e.g., table, column, row, or a group of any of these) it depends on, and the corresponding QoD requirements necessary to its triggering. Precisely, one QoD bound, κ, can be provided either for single database containers associated or for groups of object containers (e.g., several columns covered by the same κ). If no bound is associated with a step A, then it is assumed that A should be triggered right after the execution of its precedent steps (i.e., strict temporal synchronism). Additionally, it is necessary to specify the maximum tolerated error introduced in the dataflow output for QoD adjustment and learning purposes. After dataflow registration, the underlying database schema is extended to incorporate the metadata related with the QoD bound and QoD control state. Specifically, it is necessary to have maps that given a dataflow, a step, and a data container, return the quality bound and current state.

4.2. Evaluation and enforcement of the quality-of-data bounds

The QoD state of a database object container o, for a processing step A, is updated every time an update is perceived by *Fluχy* through the Monitoring component. Upon such event, it is necessary to identify the step A' that made the update ($A' \prec A$) and the affected data object container, o, which is sent by the client libraries; this, in order to retrieve the quality bound and current state associated through the meta-data. Then, given A' and o, we can find all successor steps of A', including A, that are dependent on the updates performed on o, and thus update their QoD state (i.e., the state of each successor step depending on o). Specifically, we need to increment all of the associated vectors σ and re-compute the ratio *modified items*/*total items* ($\sum_{i=1}^{m} |x_i - x'_i| / \sum_{i=1}^{m} x'_i$), hold in all ν vectors. Afterwards, the QoD state of a pair (processing step, object) needs to be compared against its relative QoD reference bound (i.e., the maximum level of changes allowed, κ).

The evaluation of the quality vectors σ and ν, to decide if a step A should be triggered or not, may take place at one of the following times:

(a) every time a write operation is performed by a precedent step of A;
(b) every time a precedent step completes another execution; or
(c) periodically between a given time frame.

These options can be combined together; e.g., it might be of use to combine option (c) with (a) or (b), for the case where precedent steps of A take very long periods of time in performing computations and generating output. Despite option (a) being the most accurate, it is the least efficient, especially when dealing with large bursts of updates.

To evaluate the time constraint, θ, *Fluχy* uses timers to check periodically (e.g., every second) if there is any timestamp in θ about to expire (i.e., a QoD bound that is almost reached). Specifically, references to processing steps are held in a list ordered ascending by time of expiration, which is the time of last execution of a dependent step plus θ. In effect, the Scheduler component starts from the first element of the list checking if timestamps are less than or equal to current time. As the list is ordered, the Scheduler has only to fail one check to ignore the rest of the list.

4.3. Learning

The Monitoring component informs the Knowledge Base about the input data variation and error introduced in the dataflow. These data is stored in a log file and exposed to the FBN, which are constantly being trained and optimized for improved predictions. Since FBN rely on these random samples, learning and inference is a probabilistic process. This process is described as follows.

FBN possess binary neurons which are grouped into areas of concepts (variables). Meshes of weightless connections between antecedent neuron outputs and consequent neuron inputs are used to perform if–then inference between areas. Neurons are binary, and the meshes are formed by individual random connections (like in nature). Each neuron comprises m inputs for each antecedent area, and up to $(m + 1)N$ internal unitary memories, where N is the number of antecedents (corresponding to the maximum granularity). When stimulated, the value of each concept is given by the ratio activated/total neurons.

Please cite this article in press as: S. Esteves, et al., Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets, J. Parallel Distrib. Comput. (2015), http://dx.doi.org/10.1016/j.jpdc.2015.03.001

For rules with N antecedents and a single consequent, each neuron has $N * m$ inputs. The single operation carried out by each neuron is the combinatorial count of activated inputs from every antecedent. For all counting combinations, neurons compare the sampled values with the ones in their unitary memory (FF). If the FF corresponding to the sampled value of all antecedents contains 1, then the neuron output is also 1. Otherwise, the neuron output is 0. As a result of the inference process (which is parallel), each neuron assumes a Boolean value, and the inference result will be given by the neural activation ratio in the consequent area.

Learning is performed by exposing the net to the dataflow input and by modifying the internal binary memories of each consequent neuron according to the activation of the m inputs (per antecedent) and the state of that consequent neuron. Each experiment will set or reset one binary memory of each individual neuron. Due to its probabilistic nature, it must be repeatedly exposed to the same training data for a minimum number of times (r). The optimization of r is not critical since FBN cannot be overtrained. Thus, it is only necessary to guarantee a minimum value that depends on the net parameters (m, N, granularity) and sparsity of the training dataset.

5. Implementation issues

This section details the implementation of a prototype of *Flu*χy we developed, as proof-of-concept, that conforms to the architecture and model aforementioned to demonstrate the feasibility, usefulness and benefits of our dataflow model when deployed as a WMS for high-performance and low latency in large-scale data store-backed dataflows.

5.1. Adopted technology

Starting from the top layer, and to demonstrate that *Flu*χy can be easily and seamlessly integrated with other systems, we have implemented our model using Oozie, which is a Java open-source workflow coordination system to manage Apache Hadoop [39] jobs. In effect, we adapted Oozie by replacing the time-based and data detection triggering mechanisms, with a notification scheme that is interfaced with the *Flu*χy framework process through Java RMI. Generally, Oozie only has to notify when a step finishes its execution, and *Flu*χy only has to signal the triggering of a certain step; naturally, these notifications share the same processing step identifiers.

As for the lower layer, and although the framework can be adapted to work with other non-relational data stores, in the scope of this particular work, our target is HBase [17] (the open-source Java clone of BigTable [11]), which we used as an instance of the underlying storage. This database system is a sparse, multi-dimensional sorted map, indexed by row, column (includes family and qualifier), and timestamp; the mapped values are simply an uninterpreted array of bytes. It is column-oriented, meaning that most queries only involve a few columns in a wide range, thus significantly reducing I/O. Moreover, these databases scale to billions of rows and millions of columns, while ensuring that write and read performance remain constant.

Finally, *Flu*χy was also built in Java, and uses, i.a., the Saxon[12] XPath engine to read and process XML configurations files (e.g., the dataflow description).

5.2. Library support and API

In order to intercept the updates performed by processing steps, we adapted the HBase client libraries by extending the implementation of some of their classes while maintaining their original APIs.[13] Namely, the implementation of the classes *Configuration.java*, *HBaseConfiguration.java*, and *HTable.java*, were modified to intercept every update performed on HBase, especially *put* and *delete* operations, and send the needed parameters (like step, operation, table, and column identifiers) to the *Flu*χy framework.

Applications need therefore only to be slightly modified to use our API. Specifically, only the import declarations of the HBase packages need to be changed to *Flu*χy packages, since our API is practically the same. To ease such process, we provide tools that automatically modify all the necessary import declarations, thereby patching the java bytecode at loading time.

5.3. Dataflow description

The QoD constraints are specified along with standard Oozie XML schemas (version 0.2), and given to *Flu*χy with an associated dataflow description. Specifically, we introduced in the respective XSD the new element *qod*, which can be used inside the element *action*. Inside *qod*, it is necessary to indicate the data containers associated with the elements: *table*, *column*, *row*, or *group*. Each of these elements must specify the three constraints time (a decimal indicating the number of seconds), sequence (an integer), and value (an integer indicating the percentage of modifications), that are combined through the method defined in the qod attribute *combine*. Additionally, the element *group* groups object containers, which are specified through the element *item*, that should be handled at the same QoD degree. These particular dataflow descriptions are then automatically adapted to the regular Oozie schema (i.e., without the QoD elements) and fed to Oozie.

6. Evaluation

This section presents the evaluation of the *Flu*χy framework, building its case on the benefits and advantages against the regular DAG semantics (i.e., SDF with no QoD enforcement). All tests were carried out with the *Flu*χy implementation described previously, using machines with Intel Core 2 Quad CPUs Q6600 at 2.40 GHz, 7825 MB of available RAM memory, and HDD SATA II 7200RPM 16 MB, with Java 6, Oozie, HBase, and the FBN Python module installed.

For the evaluation we relied on the prototypical scenario described in Section 2.1. The number of changed detectors and its positions in the city are randomly chosen in practice, simulating detectors that would perceive changes in certain areas. In effect, each sensor corresponds to a different generating function, following a distribution with smooth variations across space (i.e., realistic in the variations and trends, while full exactness for a given day record is not relevant for our purposes). This sample variations generated provide the necessary updated input data to the dataflow in each (re-)execution, a *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, $PM_{2.5}$ or NO_2. At the end, in the final step of the dataflow, the index is generated, thereby producing 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).

The FBN relevant parameters [8] for training are: $m = 125$ neurons per area; $n = 25$ inputs per neuron; maximum granularity ($n + 1 = 26$). Training was done with 60 iterations, i.e., the

12 http://saxon.sourceforge.net/.

13 http://hbase.apache.org/apidocs/overview-summary.html.

Please cite this article in press as: S. Esteves, et al., Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets, J. Parallel Distrib. Comput. (2015), http://dx.doi.org/10.1016/j.jpdc.2015.03.001

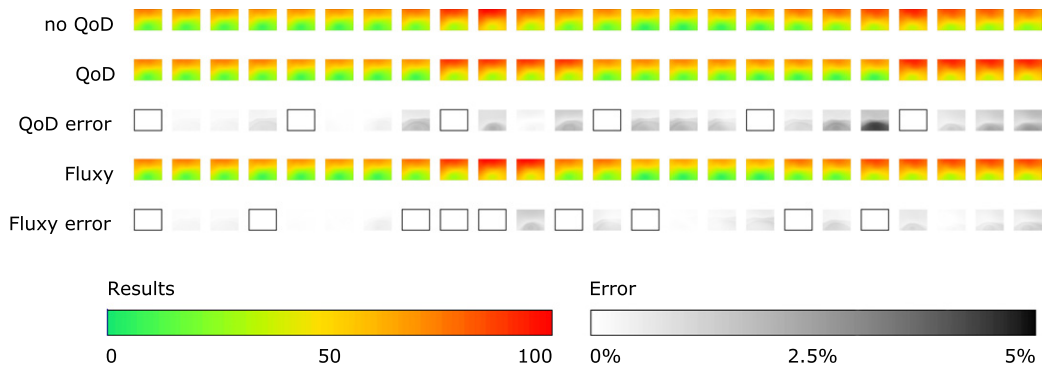


Fig. 6. Samples collected during a day with and without QoD enforcement. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

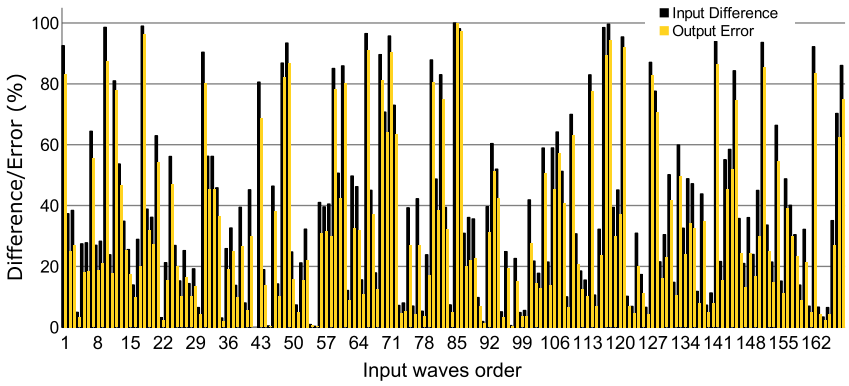


Fig. 7. Evolution of relationship among differences in input tables and in the output tables (seen as error).

FBN was executed 60 times for each input value in order to obtain an error estimation (i.e., the difference to correct output if the dataflow is not re-executed) and its standard deviation for assurance purposes. Training time for 82 waves is less than a minute which means it can be carried out with no impact, for many simultaneous dataflows, in a cloud scheduler/controller machine. Test execution time for 164 waves is also less than a minute. Application execution time for one input is less than 1 s. Training, test and execution times are thus not a relevant factor in system performance, so we will focus the following evaluation on the accuracy of error predictions and resources (i.e., full dataflow re-executions) saved.

To put *Fluχy* into perspective against regular DAG semantics, both with and without FBN-based error prediction, we resort to Fig. 6 that shows the pollution maps generated by step D during a day. As traditionally, red and darker means higher risk, while green and lighter yellow means reduced risk (0–100). The shades of gray in *QoD (error)* and *Fluχy (error)* indicate the per centum difference among the outputs generated by the dataflow execution with QoD (bounding variations in input, up to e.g., 30%) and with full *Fluχy* (i.e., with adaptive QoD driven by FBN prediction). Framed boxes indicate when dataflow re-execution is triggered in each of the approaches, thereby resetting the output error, against the no-QoD output, to zero, increasing (darkening) otherwise. *Fluχy* is aimed at bounding output error at the expense of more (re-)executions necessary than those triggered just by QoD applied to the input differences.

6.1. Results

The data in Fig. 7 shows the evolution, across the 168 waves, of the per centum differences in the input generated by each new sampling (ι), and the corresponding per centum in the output tables, i.e., the error ε that would be introduced by not re-executing the dataflow. It shows that while there is a visible correlation, the relationship is not linear and both input difference and output error are subject to significant variations across waves (e.g., periods of more or less intense rise/fall on pollution during the day). This shows that finding an optimal QoD is not trivial, and arbitrarily fixing a given QoD on input difference is, on its own, unable to assure that the output error is bounded, e.g., to 5%; hence the need for the FBN to learn the input/error relationship and save resources, *while* ensuring the error stays within acceptable limits. Otherwise, the system is not suitable and reliable for decision makers. Fig. 8 highlights the statistical relationship among per centum differences in the input (e.g., QoD to trigger re-execution) and the per centum output error when the re-execution is delayed. The three lines, from top to bottom, depict: (i) the maximum of error occurred for a given input difference, (ii) its mean plus the standard deviation, and (iii) the norm of the output error. It shows correlation, although significant variance and no strict linearity among input difference and output error in many occasions.

Fig. 9 shows the evolution, across waves, of the real (per centum) error perceived in the output due to delaying the triggering of dataflow steps, and the average error predicted by the FBN (and its ceiling/majoration). Albeit the real error stays mostly above the predictions, the important fact is that the trends of the real and predicted error are always in sync. In greater detail, Fig. 10 allows us to confirm that: (i) this deviation is bounded, always less than 5%, averaged at about 3%, and (ii) the deviation decreases as waves go on.

Based on these findings, we use FBN to predict, for each new input (based on its per centum difference against the previous samples), how great (per centum) the modifications in the output are going to be, hence predicting the error of delaying the triggering of steps with a great precision. To assert this assurance

Please cite this article in press as: S. Esteves, et al., Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets, J. Parallel Distrib. Comput. (2015), http://dx.doi.org/10.1016/j.jpdc.2015.03.001

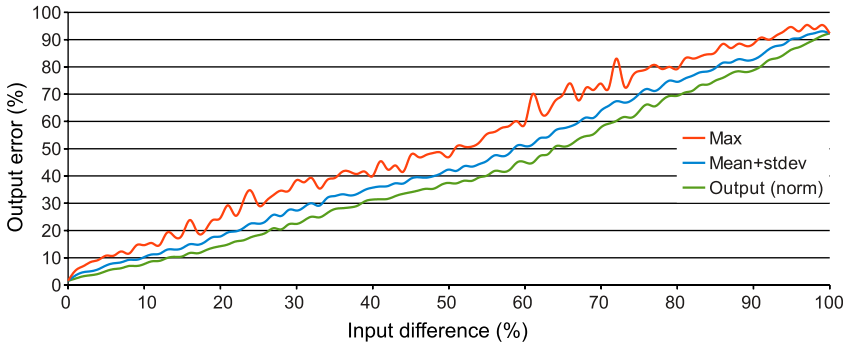


Fig. 8. Input difference and output error correlation.

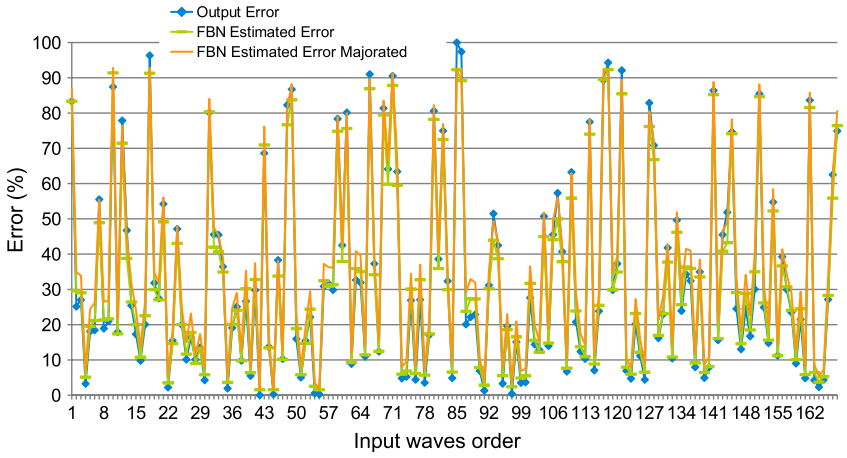


Fig. 9. Output error and FBN error estimation.

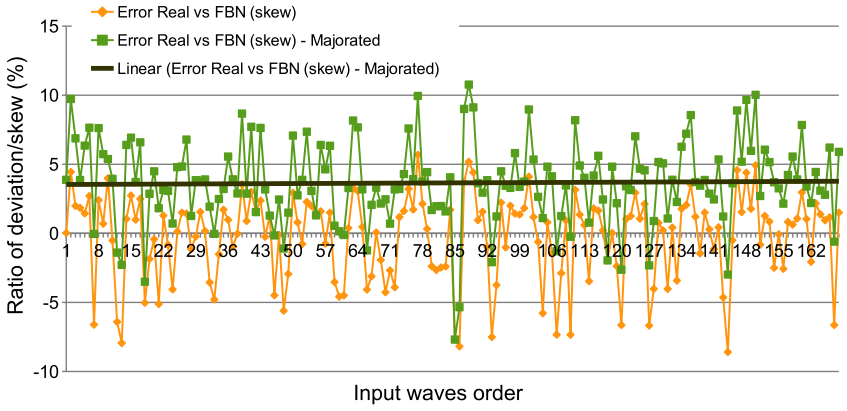


Fig. 10. Detail on the ratio of deviation/skew between observed error and FBN prediction.

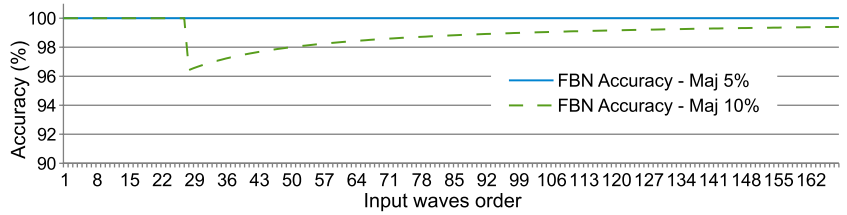


Fig. 11. FBN cumulative prediction accuracy.

given to decision makers, we show in Fig. 11, across the 168 waves, the cumulative percentage of correct predictions; in this case, that the output error would be below 5%, and below 10%. For the first one, it is fully correct and for 10%, where the FBN skew is more amplified, the confidence is always above 96% during the 168 waves (the prediction got an error beyond max_{ε} around wave 28). These results show that *Fluχy* is able to accurately predict, with confidence above 95%, when the output error (due to delaying

Please cite this article in press as: S. Esteves, et al., Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets, J. Parallel Distrib. Comput. (2015), http://dx.doi.org/10.1016/j.jpdc.2015.03.001

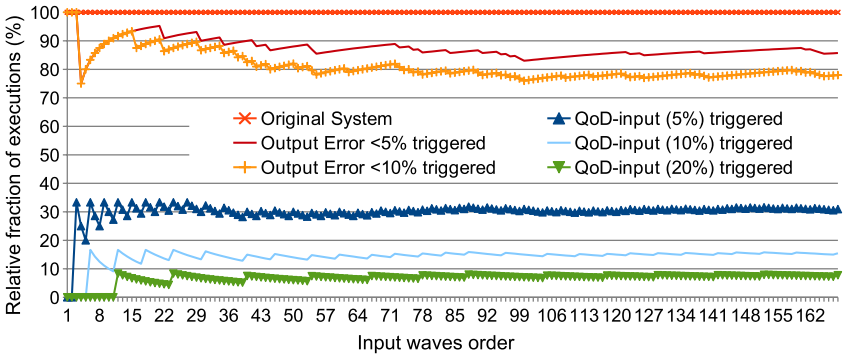


Fig. 12. Resource usage.

triggering of dataflow steps upon new input) is about to surpass a given maximum ($\max_\varepsilon = 5/10\%$).

This leads us to expect significant resource savings, especially in a shared cloud environment where multiple dataflows may be continuously executing, due to avoiding unnecessary dataflow re-execution, *while* being assured that the error introduced is bounded to a small manageable percentage, with good significant confidence. This combines quantifiable enough accuracy in error prediction and confidence (which could be set as an SLA) with significant resource savings, free to execute other dataflows or simply to pay less for infrastructure usage. Fig. 12 shows, normalized to the original system's 168 executions, large executions savings (that are proportional to the resources saved) provided by the sole usage of QoD on the input. The number is progressively lower for QoD of (5%, 10%, 20%) since more input is needed in order to trigger steps, albeit with no guarantees or bounds on the error introduced in the output. *Flu*χy, ensuring output error below 5% and 10%, i.e., almost full (above 95%) confidence is still able to save, at the end, from 12% to 23% of executions, hence almost a quarter of savings in resources, which combined in multiple dataflows is significant.

In summary, *Flu*χy is able to save resources by wisely avoiding (unnecessary) dataflow re-executions, i.e., when the new input is not sufficient to generate errors in the output above a given accuracy target ($\max_\varepsilon$). With a very high statistical confidence, decision makers get probabilistic guarantees on the results. The larger the maximum error allowed, the greater the savings may be; as it is expected but while not sacrificing the statistical confidence. Thus, it is necessary to find optimal QoDs with a good compromise between those factors. Moreover, given the high confidence level with which we can estimate error, we are able to dynamically adjust, during execution, the QoD level applied to the input, in order to enforce the desired SLA (maximum) on the output error, hence enhancing the QoD model.

7. Related work

This section reviews relevant proposed solutions, within the current state-of-the-art, that intersect the main topics approached in this work.

DAGMan [12] is one of the early workflow languages in e-science. It interprets and manages text descriptions of jobs comprising DAGs. DAGMan accounts for job dependencies, allows pre- and post-processing scripts for each vertex and reissues failed jobs. Being a meta-scheduler, it relies on the centralized Condor workload management system for scheduling. Pegasus [23] extends DAGMan in order to allow mapping of workflows jobs on distributed resources and from the description of computation tasks, it performs the necessary data transfers among sites. Pegasus aims at optimizing workflow performance and reliability by scheduling to appropriate resources, but, unlike our system, there are no QoD guarantees on continuous processing or dataflow, and no data sharing.

In [21], it is given a decentralized execution approach to large-scale workflows based on DAGMan. At runtime, adjustments to the workflow execution plan are made to meet QoS objectives; in particular, mapping and migration of tasks to specific resources in multiple domains. This adaptation is performed based on a p2p push algorithm and provides a feedback control mechanism with monitoring of resources. *Flu*χy also provides a feedback control mechanism, but to reason about data significance. We believe that applying our QoD model to large-scale workflows and dynamic scheduling mechanisms, for mapping tasks to resources, could certainly have a significant impact on the overall workflow progress with a great quantity of resources being saved (i.e., the scheduler could adjust the execution plan at runtime to allocate less resources).

Taverna [27] is heavily used in bioinformatics. It is a WMS with interoperability support for a multitude of execution environments and data formats. Data sources and data links are considered as first entities in the dataflow language. Execution can be placed remotely on a large list of resources but without cross-site distribution and no QoD is enforced.

Dryad [20] executes DAGs explicitly created via an imperative API. It includes composition of operators/operations and enabled new ones to be defined, allowing for graph and vertex merger. It allows the construction of computation pipelines spanning across a cluster. It has been integrated with LINQ data query capabilities in.NET languages. It has support for channels of shared mutable data.

Kepler [2] is a solution for managing scientific workflows. It was designed to help scientists and other inexpert computer users to create, execute, and share models and analyses, thereby including a set of features for reducing the inherent complexity of deploying workflows in various computing environments. Other option is Triana [36], a decade proven visual programming environment, focusing on minimum effort, that allows users to compose applications from programming components (drawn from a large library on text, signal and image processing) by drag and drop into a workspace, and connecting them in a workflow graph.

MapReduce (MR) [13] was first created for reverse index creation and page ranking. It forces programmers to obey a strict model that is different from those used for application logic. However, the automatic parallelization and fault-tolerance features made it powerful and led to the development of the open-source solution Hadoop [39]. WMSs for Hadoop, like Oozie, started to arise, e.g., Azkaban,[14] Cascading.[15]

[14] http://sna-projects.com/azkaban/.
[15] http://www.cascading.org.

Please cite this article in press as: S. Esteves, et al., Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets, J. Parallel Distrib. Comput. (2015), http://dx.doi.org/10.1016/j.jpdc.2015.03.001

More modern functionality in MR such as supporting social networks and data analytics are extremely cumbersome to code as a giant set of interdependent MR programs. Reusability is thus very limited. To amend this, the Apache Pig platform [31] eases creation of data analysis programs. The Pig Latin language combines imperative-like script language (foreach, load, store) with SQL-like operators (group, filter). Scripts are compiled into Java programs linked to Map Reduce libraries. An example of productivity and reusability is a word counting script with 6 lines of code. The Hive [37] warehouse reinstates fully declarative SQL-like languages (HiveQL) over data in tables (stored as files in an HDFS directory). Queries are compiled into MR jobs to be executed on Hadoop. SCOPE [10] takes a similar approach to scripting but targeting Dryad [20] for its execution engine.

To avoid recreating web indexes from scratch after each web crawl, Google Percolator [32] does 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. Although it scales better than MR, it has 30-fold resource overhead over traditional RDBMS. Nova [29] 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 datasets. These systems are someway close to *Flu*χ*y* in the sense that tasks are not executed synchronously and incremental/pipeline processing is used extensively, highlighting the importance of the focus of our work on incremental processing.

Yahoo CBP [25] aims at greater expressiveness by expressing 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. Thus, as a result of a partial web crawl, a new input frame is processed. For stateful stages, translator functions combine data from new frame with existing state. CBP provides primitives for explicit control flow and synchronize execution of multiple inputs. It requires an extended MR implementation and some explicit programming for a QoS-enabled dataflow (unlike *Flu*χ*y*, which allows flexible control, without programming, for enforcing QoD constraints).

Nectar [18] for Dryad links data and the computation that generated it as unified hybrid cacheable element. On programs reruns, Nectar replaces results with cached data. Dryad programs need to be enhanced with cache management calls that check and update the cache server. This is transparently done in InCoop [6], which does caching for MR apps. Map, combine and reduce phase results are stored and memoized. A new memorization-aware scheduler is used to repeat tasks where cached output is already stored, reducing data transfers that still cause overhead even if recomputation is avoided. Somehow like *Flu*χ*y*, 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 [28], it is presented a formal programming and scheduling model for defining temporal asynchrony in workflows. The workflow vertices consist of operators, that process data, and data channels, which are pathways through which data flows between operators. These operators have signatures that describe the types and consistency of the blocks (i.e., the atomic units of data) 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 and providing non ad-hoc solutions for introducing asynchronous behavior in workflows, however, it does not account with the volume, relevance or impact of modifications of the data given as input for each workflow step, like in *Flu*χ*y*.

In [30], it is proposed a set of optimization strategies for large-scale parallel dataflow systems. Authors state that adaptive query planning does not resort to a-priori accurate models, but, instead, adopts a trial-and-error and feedback-driven approach (just like *Flu*χ*y*). Key parameters like intermediate data sizes and function costs are hard to estimate a-priori, and thus model-light optimization approaches that gather and react to performance information during runtime are more adequate, which corroborates on the adequacy of *Flu*χ*y* to optimize resource utilization in dataflows. We also take a step further, by trying to determine, with Machine Learning, such key parameters based on observed dataflow patterns in the past. Many other systems rely on feedback control loops to improve workflow execution and performance (e.g., [3,4,1]), however none of them focus on the problematic of efficient usage and saving of resources, like *Flu*χ*y*.

Furthermore, it is important to note that *Flu*χ*y* is not a stream processing system, like Apache Storm[16] or Spark Streaming [41]. Like other typical WMS, OOzie (the engine behind *Flu*χ*y*) is triggered by time (frequency) and data availability, which are discrete events in time, and do not rely on sliding windows (as stream processing does). Besides, data is accumulated and persisted in WMS across processing steps, which does not happen in stream processing systems, where data is mostly kept in volatile memory. Continuous processing in the context of this work means that the same aggregated computation (dataflow) is executed multiple times over (“non-contiguous”) time, and does not mean that dataflows are uninterruptedly receiving new input data (like stream processing engines).

8. Conclusion

We presented *Flu*χ*y*, a novel dataflow model and framework for data-intensive computing that breaks through the SDF model. It enables temporal asynchrony among the various processing steps based on the impact of their input data. This impact relies on a quality-of-service notion that expresses constraints over the divergence of data (QoD). *Flu*χ*y* delivers controlled performance, high resource efficiency, prioritization, flexibility and elasticity, which is essential in cloud-like environments. Hence, also playing an important role in green computing and cloud SLAs (e.g., SaaS services can impose different QoDs for different budgets). Furthermore, *Flu*χ*y* learns statistical behavior from dataflows in order to bound the output deviations and give probabilistic guarantees about the correctness and freshness of the results.

Fluχ*y* was implemented and found both easy to integrate with existing WMS infrastructures, as well as with currently popular NoSQL storage (HBase) for scalability. To demonstrate *Flu*χ*y* feasibility, usefulness, and efficiency, the assessment of *Flu*χ*y* was centered on a realistic prototypical example of intensive data processing, addressing the evaluation of air quality, pollution and health risks, for a city based on sensory data, gathered asynchronously, from thousands of sensors.

Therefore, we find *Flu*χ*y* a compelling effort, within the current state of the art, to improve dataflows execution, in a performance-improved, resource efficient and correct manner and, thus, deliver higher QoS to end-users and drive costs of operation down.

16 https://storm.apache.org/.

Please cite this article in press as: S. Esteves, et al., Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets, J. Parallel Distrib. Comput. (2015), http://dx.doi.org/10.1016/j.jpdc.2015.03.001

Acknowledgments

We would like to thank the anonymous reviewers who greatly contributed to the betterment of this paper.

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

References

[1] K.R. Abbott, S.K. Sarin, Experiences with workflow management: issues for the next generation, in: Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work, CSCW'94, ACM, New York, NY, USA, 1994, pp. 113–120. http://dx.doi.org/10.1145/192844.192886.
URL: http://doi.acm.org/10.1145/192844.192886.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, S. Mock, Kepler: an extensible system for design and execution of scientific workflows, in: International Conference on Scientific and Statistical Database Management, Vol. 0, 2004, p. 423. http://doi.ieeecomputersociety.org/10.1109/SSDM.2004.1311241.

[3] A. Andrzejak, U. Hermann, A. Sahai, FEEDBACKFLOW—an adaptive workflow generator for systems management, in: Second International Conference on Autonomic Computing, 2005. ICAC 2005. Proceedings, 2005, pp. 335–336. http://dx.doi.org/10.1109/ICAC.2005.30.

[4] S. Babu, Towards automatic optimization of mapreduce programs, in: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC'10, ACM, New York, NY, USA, 2010, pp. 137–142. http://dx.doi.org/10.1145/1807128.1807150.
URL: http://doi.acm.org/10.1145/1807128.1807150.

[5] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, K. Vahi, Characterization of scientific workflows, in: Third Workshop on Workflows in Support of Large-Scale Science, 2008. WORKS 2008, 2008, pp. 1–10. http://dx.doi.org/10.1109/WORKS.2008.4723958.

[6] P. Bhatotia, A. Wieder, R. Rodrigues, U.A. Acar, R. Pasquin, Incoop: MapReduce for incremental computations, in: Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC'11, ACM, New York, NY, USA, 2011, pp. 7:1–7:14. http://dx.doi.org/10.1145/2038916.2038923.
URL: http://doi.acm.org/10.1145/2038916.2038923.

[7] D.A. Brown, P.R. Brady, A. Dietz, J. Cao, B. Johnson, J. McNabb, in: I.J. Taylor, E. Deelman, D.B. Gannon, M. Shields (Eds.), A Case Study on the Use of Workflow Technologies for Scientific Analysis: Gravitational Wave Data Analysis, Workflows for e-Science, Springer London, 2007, pp. 39–59.

[8] J.P. Carvalho, J.A. Tomé, Qualitative optimization of fuzzy causal rule bases using fuzzy Boolean nets, Fuzzy Sets and Systems 158 (17) (2007) 1931–1946.

[9] R. Cattell, Scalable SQL and NoSQL data stores, SIGMOD Rec. 39 (4) (2011) 12–27. http://dx.doi.org/10.1145/1978915.1978919.
URL: http://doi.acm.org/10.1145/1978915.1978919.

[10] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou, Scope: easy and efficient parallel processing of massive data sets, Proc. VLDB Endow. 1 (2) (2008) 1265–1276. http://dx.doi.org/10.1145/1454159.1454166.

[11] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, 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, 2006, pp. 15–15.
URL: http://dl.acm.org/citation.cfm?id=1267308.1267323.

[12] P. Couvares, T. Kosar, A. Roy, J. Weber, K. Wenger, Workflow management in condor, in: I.J. Taylor, E. Deelman, D.B. Gannon, M. Shields (Eds.), Workflows for e-Science, Springer, London, 2007, pp. 357–375.
URL: http://dx.doi.org/10.1007/978-1-84628-757-2_22.

[13] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation—Volume 6, OSDI'04, USENIX Association, Berkeley, CA, USA, 2004, pp. 10–23.
URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[14] E. Deelman, et al., 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 Computer Society, Washington, DC, USA, 2006, pp. 14–22. http://dx.doi.org/10.1109/E-SCIENCE.2006.99.

[15] S. Esteves, J.N. Silva, L. Veiga, Fluchi: a quality-driven dataflow model for data intensive computing, J. Internet Serv. Appl. 4 (1) (2013) 12. http://dx.doi.org/10.1186/1869-0238-4-12.
URL: http://www.jisajournal.com/content/4/1/12.

[16] S. Esteves, L. Veiga, WaaS: workflow-as-a-service for the cloud with scheduling of continuous and data-intensive workflows, Comput. J. (2015) http://dx.doi.org/10.1093/comjnl/bxu158. URL: http://comjnl.oxfordjournals.org/content/early/2015/01/08/comjnl.bxu158.abstract.

[17] L. George, HBase: The Definitive Guide, first ed., O'Reilly Media, 2011, URL: http://www.amazon.de/HBase-Definitive-Guide-Lars-George/dp/1449396100/ref=sr_1_1?ie=UTF8&qid=1317281653&sr=8-1.

[18] P.K. Gunda, L. Ravindranath, C.A. Thekkath, Y. Yu, L. Zhuang, 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 Association, Berkeley, CA, USA, 2010, pp. 1–8.
URL: http://dl.acm.org/citation.cfm?id=1924943.1924949.

[19] D.O. Hebb, The Organization of Behavior: A Neuropsychological Theory, John Wiley and Sons, 1949.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys'07, ACM, New York, NY, USA, 2007, pp. 59–72. http://dx.doi.org/10.1145/1272996.1273005.
URL: http://doi.acm.org/10.1145/1272996.1273005.

[21] S. Kalayci, G. Dasgupta, L. Fong, O. Ezenwoye, S.M. Sadjadi, Distributed and adaptive execution of condor dagman workflows, in: SEKE, 2010, pp. 587–590.

[22] V. Kecman, Learning and Soft Computing: Support Vector Machines, Neural Networks, and Fuzzy Logic Models, MIT Press, Cambridge, MA, USA, 2001.

[23] K. Lee, N.W. Paton, R. Sakellariou, E. Deelman, A.A.A. Fernandes, G. Mehta, Adaptive workflow processing and execution in pegasus, Concurr. Comput.: Pract. Exper. 21 (16) (2009) 1965–1981. http://dx.doi.org/10.1002/cpe.v21:16.

[24] X. Li, B. Plale, N. Vijayakumar, R. Ramachandran, S. Graves, H. Conover, Real-time storm detection and weather forecast activation through data mining and events processing, Earth Sci. Inf. 1 (2008) 49–57. http://dx.doi.org/10.1007/s12145-008-0010-7.

[25] D. Logothetis, C. Olston, B. Reed, K.C. Webb, K. Yocum, Stateful bulk processing for incremental analytics, in: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC'10, ACM, New York, NY, USA, 2010, pp. 51–62. http://dx.doi.org/10.1145/1807128.1807138.
URL: http://doi.acm.org/10.1145/1807128.1807138.

[26] B. Ludäscher, et al., Scientific process automation and workflow management, in: A. Shoshani, D. Rotem (Eds.), Scientific Data Management, in: Computational Science Series, Chapman & Hall, 2009, (Chapter 13) URL: http://daks.ucdavis.edu/~ludaesch/Paper/ch13-preprint.pdf.

[27] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, C.A. Goble, Taverna, reloaded, in: SSDBM, 2010, pp. 471–481.

[28] C. Olston, Modeling and scheduling asynchronous incremental workflows, Tech. Rep., Yahoo! Research, 2011.

[29] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V.B. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, X. Wang, Nova: continuous Pig/Hadoop workflows, in: Proceedings of the 2011 International Conference on Management of Data, SIGMOD'11, ACM, New York, NY, USA, 2011, pp. 1081–1090. http://dx.doi.org/10.1145/1989323.1989439. URL: http://doi.acm.org/10.1145/1989323.1989439.

[30] C. Olston, B. Reed, A. Silberstein, U. Srivastava, Automatic optimization of parallel dataflow programs, in: USENIX 2008 Annual Technical Conference on Annual Technical Conference, ATC'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 267–273. URL: http://dl.acm.org/citation.cfm?id=1404014.1404035.

[31] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, 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, 2008, pp. 1099–1110. http://dx.doi.org/10.1145/1376616.1376726.
URL: http://doi.acm.org/10.1145/1376616.1376726.

[32] D. Peng, F. Dabek, 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, 2010, pp. 1–15.
URL: http://dl.acm.org/citation.cfm?id=1924943.1924961.

[33] L. Ramakrishnan, D. Gannon, A survey of distributed workflow characteristics and resource requirements, Tech. Rep., Indiana University, Bloomington, IN, USA, 2008.

[34] M. Richards, M. Ghanem, M. Osmond, Y. Guo, J. Hassard, Grid-based analysis of air pollution data, Ecol. Modell. 194 (1–3) (2006) 274–286. http://dx.doi.org/10.1016/j.ecolmodel.2005.10.042.
URL: http://www.sciencedirect.com/science/article/pii/S0304380005005259.

[35] I.J. Taylor, E. Deelman, D.B. Gannon, Workflows for e-Science: Scientific Workflows for Grids, Springer, 2006,
URL: http://www.worldcat.org/isbn/1846285194.

[36] I. Taylor, M. Shields, I. Wang, A. Harrison, The triana workflow environment: architecture and applications, in: I. Taylor, E. Deelman, D. Gannon, M. Shields (Eds.), Workflows for e-Science, Springer, New York, Secaucus, NJ, USA, 2007, pp. 320–339.

[37] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive—a warehousing solution over a map-reduce framework, in: IN VLDB'09: Proceedings of the VLDB Endowment, 2009, pp. 1626–1629.

[38] J.A. Tomé, J. Carvalho, Fuzzy Boolean nets—a nature inspired model for learning and reasoning, Fuzzy Sets and Systems 253 (2014) 1–27. http://dx.doi.org/10.1016/j.fss.2014.04.020.

[39] T. White, Hadoop: The Definitive Guide, first ed., O'Reilly Media, Inc., 2009.

[40] J. Yu, R. Buyya, A taxonomy of workflow management systems for grid computing, J. Grid Comput. 3 (2005) 171–200. http://dx.doi.org/10.1007/s10723-005-9010-8.

[41] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 10–17.
URL: http://dl.acm.org/citation.cfm?id=1863103.1863113.

Please cite this article in press as: S. Esteves, et al., Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets, J. Parallel Distrib. Comput. (2015), http://dx.doi.org/10.1016/j.jpdc.2015.03.001

Sérgio Esteves received his B.S. and M.S. degrees in Computer Science and Engineering in 2009, from Instituto Superior Técnico (IST), Universidade de Lisboa, Portugal. He is a Ph.D. student at IST and a researcher in the Distributed Systems Group at INESC-ID since 2009. His research interests include parallel and distributed computing; massive scale software systems for big data; data/workflow management; stream processing systems; and machine learning.

João Nuno Silva has a Ph.D. in Computer and Systems Engineering (2011) by Instituto Superior Técnico, Lisbon University. He is an Assistant Professor at Instituto Superior Técnico (Electrical and Computer Engineering Department) and a researcher at INESC-ID, in the Distributed Systems Group. He is the head instructor of Operating Systems and Distributed Systems courses in the Electrical and Computer Engineering Master Degree, at Instituto Superior Técnico. His research interests include middleware for mobile and cloud computing. He is the Technical Manager of the PCAS FP7 European Project.

João Paulo Carvalho has a Ph.D. (2002) and M.Sc. (1996) degree from Instituto Superior Técnico, Universidade de Lisboa, Portugal, where he is currently a Professor at the Department of Electrical Engineering and Computation. He has taught courses on Computational Intelligence, Distributed Systems, Computer Architectures and Digital Circuits since 1998. He is also a senior researcher at L2F—Spoken Language Systems Laboratory, INESC-ID Lisboa, where he has been working since 1991. His current main research interest involves applying Computational Intelligence techniques in Engineering and Soft Sciences.

He has authored over 90 papers in international scientific Journals, book chapters and peer-reviewed conferences. He was program co-chair and organizer of IFSA-EUSFLAT 2009, webchair for IEEE-WCCI 2010, publicity chair of FUZZ-IEEE2015 and program committee member of several conferences in the area of computational intelligence.

Luís Veiga is a tenured Assistant Professor at Instituto Superior Técnico (IST), ULisboa, Senior Researcher at INESC-ID, and Group Manager of GSD for 2014–2015. He coordinates locally the FP7 CloudForEurope project, participates in FP7 Timbus project on digital preservation and virtualization. He has lead 2 National funded research projects on P2P cycle-sharing and virtualization, and is locally coordinating 2 on distributed virtual machines and multicore programming, and evaluated FP7 and third-country project proposals (Belgium). He has over 75 peer-reviewed scientific publications in journals, conferences, book chapters, workshops (Best Paper Award at Middleware 2007, and Best-Paper Award Runner Up at IEEE CloudCom 2013, Best-Paper Award Candidate at IEEE CloudCom 2014). He was General Chair for Middleware 2011, and belongs to Middleware Steering and Program Committee. He was Virtualization track co-Chair for IEEE CloudCOM 2013, and is Local Chair for Euro-Par 2014 track on Distributed Systems and Algorithms. He was an "Excellence in Teaching in IST" mention recipient (2008, 2012), and awarded Best Young Researcher at INESC-ID Prize (2012), and Best Researcher Overall at INESC-ID Prize (2014).

He has previously served in international conferences as member of program committee, proceedings editor (ACM Middleware 2011, EuroSys 2007, ACM PPPJ 2007 and 2008, and MobMid/M-MPAC Workshop at ACM Middleware 2008, 2009, and 2010) and as reviewer.

He is a member of the Scientific Board of Erasmus Mundus European Master and Joint Doctorate in Distributed Computing. He is Chair of IEEE Computer Society Chapter, IEEE Section Portugal for 2014–2015.

Please cite this article in press as: S. Esteves, et al., Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets, J. Parallel Distrib. Comput. (2015), http://dx.doi.org/10.1016/j.jpdc.2015.03.001