Journal of Internet Services and Applications
a SpringerOpen Journal

RESEARCH Open Access

# *Fluχ*: a quality-driven dataflow model for data intensive computing

Sérgio Esteves*, João Nuno Silva and Luís Veiga*

**Abstract**

Today, there is a growing need for organizations to continuously analyze and process large waves of incoming data from the Internet. Such data processing schemes are often governed by complex dataflow systems, which are deployed atop highly-scalable infrastructures that need to manage data efficiently in order to enhance performance and alleviate costs.

Current workflow management systems enforce strict temporal synchronization among the various processing steps; however, this is not the most desirable functioning in a large number of scenarios. For example, considering dataflows that continuously analyze data upon the insertion/update of new entries in a data store, it would be wise to assess the level of modifications in data, before the trigger of the dataflow, that would minimize the number of executions (processing steps), reducing overhead and augmenting performance, while maintaining the dataflow processing results within certain coverage and freshness limit.

Towards this end, we introduce the notion of Quality-of-Data (QoD), which describes the level of modifications necessary on a data store to trigger processing steps, and thus conveying in the level of performance specified through data requirements. Also, this notion can be specially beneficial in cloud computing, where a dataflow computing service (SaaS) may provide certain QoD levels for different budgets.

In this article we propose *Fluχ*, a novel dataflow model, with framework and programming library support, for orchestrating data-based processing steps, over a NoSQL data store, whose triggering is based on the evaluation and dynamic enforcement of QoD constraints that are defined (and possibly adjusted automatically) for different sets of data. With *Fluχ* we demonstrate how dataflows can be leveraged to respond to quality boundaries that bring controlled and augmented performance, rationalization of resources, and task prioritization.

**Keywords:** Dataflow, Workflow, Quality-of-Data, Data store, NoSQL

## 1 Introduction

Current times have been witnessing an increase of massively scale web applications capable of handling extremely large data sets throughout the Internet. These data-intensive applications are owned by organizations, with cutting edge performance and scalability requirements, whose success lies in the capability of analyzing and processing terabytes of incoming data feeds on a daily-basis. Such data processing computations are often governed by complex dataflows, since they allow better expressiveness and maintainability than low-level data processing (e.g., java map-reduce code).

Dataflows (or data processing workflows) can be represented as directed acyclic graphs (DAGs) that express the dependencies between computations and data. These computations, or processing steps, can potentially be decoupled from object location, inter-object communication, synchronization and scheduling; hence, being highly flexible on supporting parallel scalable and distributed computation. The data is either transferred directly from one processing step to another using intermediate files or via a shared storage system, such as a distributed file system or a database (which is our target in this particular work).

Another extensive use of dataflows has been for continuous and incremental processing. Here, vast amounts of raw data are continuously fed, as input, to cross an incremental processing pipeline in order to be transformed

*Correspondence: sesteves@gsd.inesc-id.pt; luis.veiga@inesc-id.pt
Instituto Superior Técnico - UTL / INESC-ID Lisboa Distributed Systems Group
Rua Alves Redol, 9, 1000-029 Lisboa, Portugal

© 2013 Esteves et al.; licensee Springer. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

into final structured and refined data. Examples include data aggregation in databases, web crawlers, data mining, and others from many different scientific domains, like sky surveys, forecasting, RNA-sequencing, or seismology [1-5].

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 across the various input, intermediate and output data sets (i.e., following the SDF computing model [6]), or leave the temporal logic in the programmer hands, who have often to explicitly program non-synchronous behavior to meet application latency and prioritization requirements. For example, processing news documents faster than others in a web indexing system; or, in the astronomy domain, processing images, collected from ground-based telescopes, of objects that are closer to Earth first, and only then images that do not require immediate attention. Moreover, these systems do not account with the volume of data that arrives on each dataflow step, which could and should be used to reason about their performance impact on the system. Precisely, executing a processing step each time a small fragment of data arrives can have a great impact on performance, as opposed to executing only when a certain substantial quantity of new data is available. Such issues are addressed in this work with a data quality-driven model based on the notion of what we call Quality-of-Data.

Informally, we define Quality-of-Data (QoD) as the ability to provide different priority to different data sets, users or dataflows, or to guarantee a certain level of performance of a dataflow. These guarantees can be enforced, for example, based on data size, number of hits in a certain data store object, or delay inclusion. High QoD should not be confused with high level of performance, but instead it conveys in the capability of strictly complying with QoD constraints defined over data sets.

With the QoD concept,[a] we are thus able to define and apply temporal semantics to dataflows based on the volume and importance of the data communicated between processing steps. Moreover, relying on QoD we can augment the throughput of the dataflow and reduce the number of its executions while keeping the results within acceptable limits. Also, this concept is particularly interesting in (public) cloud computing, where a dataflow service (SaaS) may provide different QoD levels for different budgets. Therefore, this work can also give a contribute to the new studies addressing the cost and performance of deploying dataflows in the cloud (e.g., [7]).

Given the current envisagement, we propose a novel dataflow model, with framework and programming library support, for orchestrating data-based computation stages (actions), over a NoSQL data store, whose triggering is based on the evaluation and dynamic enforcement of QoD constraints that are defined, and possibly adjusted automatically, for different sets of data. With this framework, named *Fluχ*, we enable the setup of dataflows whose execution is guided and controlled to comply with certain QoD requirements, delivering thus: controlled performance (i.e., improved or degraded); rationalization of resource usage; execution prioritization based on relative importance of data; and augmented throughput between processing steps.

We implemented *Fluχ* using an existing WMS, that was adapted to enforce our model and triggering semantics, and adopted, as the underlying data store, the HBase tabular storage [8]. Our results show that *Fluχ* is able to: i) ensure result convergence, hence showing that the QoD model does not introduce significant errors, ii) save significant computational resources by avoiding wasteful repetitive execution of dataflow steps, and iii) consequently, reduce machine load and improve resource efficiency, in cluster and cloud infrastructures, for equivalent levels of data *value* provided to, and as perceived by, decision makers.

Shortcomings of state-of-the-art solutions include (to the best of our knowledge): i) lack of tools to enable transparent asynchronous behavior in workflow systems; ii) no support for dataflows to share data through highly-scalable cloud-like databases; iii) lack of integration, in mostly loosely-coupled environments, between the workflow management and the underlying intermediate data (which is seen as opaque); and iv) no quality of service, and of data, is enforced (at least in a flexible manner).

The remainder of this article is organized as follows. Section 2 presents the *Fluχ* dataflow model based on the QoD notion. Section 3 describes an archetypal meta-architecture of the *Fluχ* middleware framework, and Section 4 offers its relevant implementation aspects. Then, Section 5 presents a performance evaluation, and Section 6 reviews related work. Finally, we draw all appropriate conclusions in Section 7.

## 2 Abstract dataflow model

In this section we describe the *Fluχ* dataflow model which was specially designed to address large-scale and data-intensive scenarios that need to continuously and incrementally process very large sets of data while maintaining strong requirements about the quality of service and data provided. Moreover, our model implies that the underlying data, shared among processing steps, should be done via tabular data stores; whereas most workflow models rely on files to store and share the data, which cannot achieve the same scalability and flexibility.

Our dataflow model can be expressed as a directed acyclic graph (DAG), where each node represents a processing step (designated here by *action*) that must perform changes in a data store; and the edges between actions represent dependencies, meaning that an action needs the output of another action to get executed (naturally, these dependencies need to be decoupled from WMS implementation so that the same actions can be combined in different ways). 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 in: 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.

Unlike the other typical models, our approach takes a step further: the end of execution of a node $A$ does not mean that the successor nodes $A$' (denoted $A' \succ_D A$), that depend on $A$, will be immediately triggered (like it usually happens). Instead, successor nodes should be triggered as soon as $A$ has finished its execution and has also performed a sufficient level of changes in the data store that comply with certain QoD requirements (which can cause a node being executed multiple times with the successor nodes being triggered only once). If such changes do not occur in a given time frame, successor nodes would eventually be triggered. Hence, the QoD requirements evaluate the volume of data input fed to an action that is worth its execution. This is the key difference and novelty of our approach that breaks through the SDF (synchronous data-flow) computing model.

The amount of data changes (QoD) necessary to trigger an action, denoted by $\kappa$, is specified using multi-dimensional vectors that associate QoD constraints with data object containers, such as a column or group of columns in a table of a given column-oriented database. $\kappa$ bounds the maximum level of changes through numeric scalar vectors defined for each of the following orthogonal dimensions: time ($\theta$), sequence ($\sigma$), and value ($\nu$).

***Time*** Specifies the maximum time an action 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 action that is dependent on the availability of data in the object container $o$, this time constraint $\kappa_\theta$ enforces that $\theta(o) < \kappa_\theta$ at any given time.

***Sequence*** Specifies the number of still unapplied updates to an object container object container $o$ upon which, the action that depends on $o$ is triggered. Considering $\sigma(o)$ indicates the number of applied updates over $o$, this sequence constraint $\kappa_\sigma$ 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 initial state, or against a constant (e.g., top value), since the last execution of an action dependent on $o$. Considering $\nu(o)$ provides that difference (e.g., in percentage), this value constraint $\kappa_\nu$ enforces that $\nu(o) < \kappa_\nu$ at any given time. It captures the impact or importance of updates in the last state.

These constraints are used to trigger the execution of actions. When they are reached, the action is executed (or scheduled to be executed). Access to the object containers is not blocked but update counters are still maintained in synch. Only if specified (and it is not required for the intended applications in this paper), will the constraints, when reached, block access to the data containers, preventing further updates until the action is re-executed.

The QoD bound $\kappa$, 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 change the same $\kappa$.

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

Furthermore, we also allow a unique definition for the combination of all $\kappa$ bounds, instead of individually specifying operators for every pair of bounds. The pre-built available definitions are:

**all ($\forall$)** An action is triggered *iff* all associated $\kappa$ bounds are reached.

**at-least-one ($\exists_1$)** An action is triggered *iff* at least on associated $\kappa$ is reached.

**majority ($\lceil(n+1)/2\rceil$)** An action is triggered *iff* the majority of associated $\kappa$ bounds are reached (e.g., 2 of 3 bounds, or 3 of 4 bounds, are reached).

These definitions are, afterwards, automatically unfolded in regular expressions containing *and* and *or* operators.

### 2.1 Prototypical example

Figure 1 depicts a simple and partial example dataflow of a typical web crawler, which serves as motivation and familiar prototypical example to introduce dataflows. Step *A* crawls documents over the web and stores their text on stable storage, either in a file system or as an opaque object in a database (e.g., no-SQL), along with some metadata extracted from the document contents, HTTP response headers, or derived from some preprocessing based on title words, URL, or tags. Depending on the class of the accessed pages, their content is stored in different tables: one for the news items, other for the remaining static pages.

Steps *B* and *C* are similar in function, in the sense that they process existing documents, generating word counts of the words present in the document, along with the URL containing them. The difference being that step *B* processes specifically only those documents identified or marked as containing news-related content in the previous step.

Since news pages change more frequency and are more relevant, *B* has stricter QoD requirements, and therefore is processed faster (i.e., activated more often). Its divergence bounds are lower, meaning that it will take less time (200 vs 300 seconds), fewer new documents crawled (100 vs 500), and/or fewer modifications on new versions of crawled documents (10% vs 20% of contents), in order to activate it.

Finally, in step *D*, all the information generated by the previous executions of steps *B* and *C* is joined and the inverted index (*word* ↦ {*list of URLs*}, for each word) is generated.

This whole process could be performed resorting to Map-Reduce programs but, as we describe in Section 6, since Map-Reduce programs are becoming increasingly larger and more complex, their reuse can be leveraged chaining them into workflows, reducing development effort. In Section 5 we will address a more elaborate example.

## 3 Architecture

In this section we present the architecture and design choices of the *Fluχ* middleware framework that is capable of managing dataflows following the model described in the previous section (Section 2). The *Fluχ* framework, is designed to be tightly coupled with a large-scale (NoSQL) data store, enabling the construction of quality-driven dataflows in which the triggering of processing steps (actions) may be delayed, but still complying with QoD requirements defined over the stored data.

This framework may be particularly useful in public cloud platforms where it can be offered as a Software-as-a-Service (SaaS) in which the QoD requirements are defined according to certain budgets; i.e., small budgets would have stricter QoD constraints, and large budgets looser QoD constraints.

Figure 2 shows a distributed network architecture in the cloud whereby a dataflow is set up to be executed upon a cluster of machines connected through a local network. More precisely, a coordinator machine, running a WMS with *Fluχ*, allocates the dataflow actions to available worker nodes and the input/output data is communicated between actions via a shared cloud tabular data store. In this particular work we abstract from the details of scheduling and running actions in parallel; our focus here is that actions share input, intermediate, and output data through a distributed cloud database (instead of intermediate files, like it usually happens).

Figure 3 depicts an archetypal meta-architecture of the *Fluχ* middleware framework, which operates in the middle of a dataflow manager and an underlying non-relational tabular storage. Actions of a dataflow run on top of the dataflow manager and they must share data through the underlying storage. These actions may consist

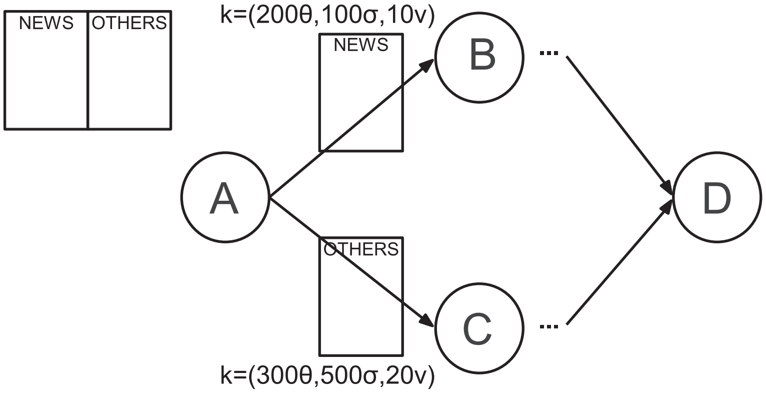

**Figure 1 Dataflow example with different priorities.**

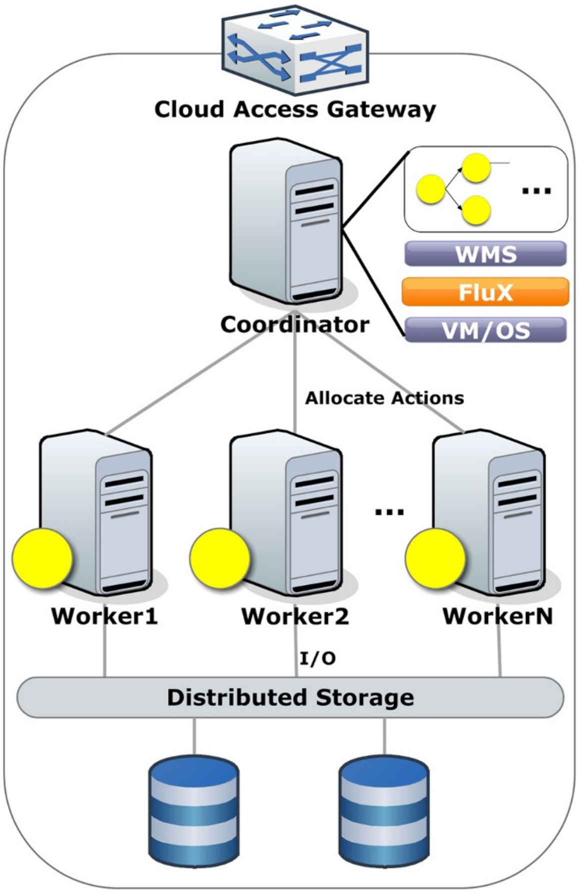

**Figure 2** Network architecture.

of Java applications, scripts expressed through high-level languages for data analysis (e.g., Apache Pig [9]), map-reduce jobs, as well as other out-of-the-box solutions. The components outlined with no solid line dash are optional meta-components for the adaptation of *Fluχ*.

The framework can operate either with its own provided simple WMS, or with an existing dataflow manager by means of the WMS Adaptation Component (colored in red). This inherent dependency of our framework with a WMS concerns mainly to the triggering notifications. With our WMS, we simply use a provided API through which *Fluχ* signals the triggering of actions. While using an existing WMS, we need to change its source and provide an adaptation component that controls the triggering of actions upon request.

Since *Fluχ* needs to be aware of the data modifications performed by actions in the underlying database, we contemplate three different solutions, regarding the adaptation of database libraries, that can be derived from the meta-architecture. The components colored in gray within the middleware are the core components and should be included by every derived solution; and the components colored in blue represent the three different alternatives for the adaptation, which are described as follows.

***Application Libraries*** This solution consists of adapting application libraries, referenced in actions, that are used to interact directly with the data store via its client API. It is a bit intrusive in the sense that applications need to be modified, albeit we intend to provide tools so that this process may be completely automatized.

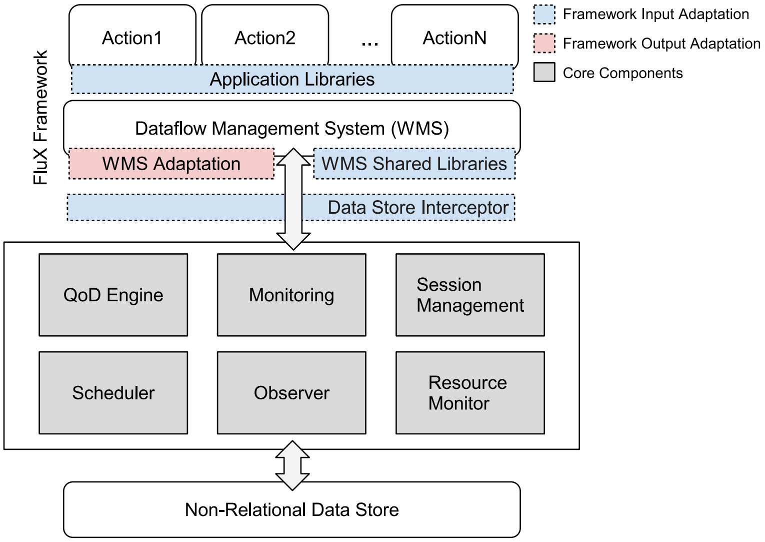

**Figure 3** *Fluχ* middleware framework meta-architecture.

***WMS Shared Libraries*** This alternative is on a lower layer and works for actions that need to access the database through WMS shared libraries (e.g., pig scripts or any other high-level language that must be compiled by the WMS). It provides transparency to actions, that do not need to be modified to work with *Fluχ*.

***Data Store Interceptor*** This solution functions as a proxy that implements the underlying database communication protocol and intercepts the calls from the applications or WMS directed to the database; hence, achieving full transparency regarding action code. Applications may only interact with the data store via this proxy, and therefore they should define as the database entry-point the address of the proxy (probably in the form of URL).

Next, we describe the responsibilities and purpose of each of the components present in the *Fluχ* framework.

***Monitoring*** This component analyzes all requests directed to the database. It uses information about update requests to maintain the current state of control data regarding the quality-of-data; and also collects statistics regarding access patterns to stored data (mainly read operations) in order to automatically adjust the QoD levels, in the view of the improvement of the overall system performance.

***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 actions, obeying to QoD specifications.

***Scheduler*** This component verifies the time constraints over the data. When the time for triggering of successor actions 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.

***Observer*** It provides mechanisms to scan the data store for modifications in case the updates performed do not go through the Monitoring component.

***Resource Monitor*** This component is responsible for monitoring the resource utilization and load of the machines allocated to execute dataflows. It informs the QoD Engine about the computation loads at runtime in order to automatically tune the QoD constraints.

### 3.1 Session management, metadata and dataflow isolation

Dataflow specification schemas need to be provided to register dataflows with the *Fluχ* framework. They should contain the description of the dataflow graph where each action must explicitly specify the underlying database object containers (e.g., table, column, or row) it depends and the relative QoD requirements necessary to the action triggering. Precisely, one QoD bound, $\kappa$, can be provided either for single database containers associated or for groups of object containers (e.g., several columns covered by the same $\kappa$); these two ways of associate $\kappa$ imply different QoD evaluation and enforcement.

QoD constraints (time, sequence, and value) can be specified as either single values or intervals of values. The former guarantees always the same quality degree, while the latter is used for dynamic adjustment at runtime: each interval relies on two numerical scalars that are used for specifying the minimum and maximum QoD bounds respectively, and the QoD Engine component adjusts $\kappa$ within the interval as needed. If no bound is associated with an action $A$, then it is assumed that $A$ should be triggered right after the execution of its precedent actions (i.e., strict temporal synchronism). 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, an action node, and a database object container, return the quality bound and current state.

It may happen that database object containers, associated with actions of a certain dataflow, can be being written by other dataflows or external applications (and thus changing the triggering semantics). To disentangle such conflicts, we consider three isolation modes through which our framework can be configured:

***Normal Mode*** It relies on an optimistic approach in which it is assumed that nothing changes the database containers besides the dataflows. In this case, different dataflows will share the same QoD state; i.e., whenever data is changed on a DBMS object container, the QoD state of all actions associated with that object are changed irrespective of which active dataflow has caused the modifications.

***Observer Mode*** This (pessimistic) mode assumes that dataflows are not the only entities performing changes on database objects. Therefore, it resorts to observers to scan the objects to detect modifications, since it is not guaranteed that every update passes through the Monitoring component.

***Isolation Mode*** In this mode each dataflow should only work with its own inputted data and have its own QoD state irrespective of how many dataflows or external processes are also writing to the same DBMS objects. This mode implies the creation of a notify column (described hereafter) per each dataflow.

Since database object containers are likely to receive a vast volume of data items (e.g., a column with millions of keys being written), it could be very inefficient for observers to scan the whole columns and find those that have been changed. Therefore, we resort to a notification mechanism where each updated item in a container needs to write an entry in an auxiliary data structure. For example, every key written in a certain column would have to also write a timestamp in a special column (notify column); and, thus, the scans will only cover that notify column, which is much more efficient in a column-oriented NoSQL data store.

### 3.2 Evaluation and enforcement of quality-of-data bounds

The QoD state of a database object container $o$, for an action $A$, is updated every time an update is perceived by *Flu*$\chi$ through the Monitoring and Observer components. Upon such event, it is necessary to identify the action $A'$ that made the update ($A' \prec A$) and the affected object container, $o$, which is sent by the client libraries; this, in order to retrieve the quality bound and current state associated through the metadata. Then, given $A'$ and $o$, we can find all successor actions 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 action depending on $o$). Specifically, we need to increment all of the associated vectors $\sigma$ and re-compute the ratio *modified keys*/*total keys*, hold in all $\nu$ vectors. Afterwards, the QoD state of a pair (action, object) needs to be compared against its relative QoD reference bound (i.e., the maximum level of changes allowed, $\kappa$).

The evaluation of the quality vectors $\sigma$ and $\nu$, to decide if an action $A$ 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 action of $A$; b) every time a precedent action finishes completely its 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 actions 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, $\theta$, *Flu*$\chi$ uses timers to check periodically (e.g., every second) if there is any timestamp in $\theta$ about to expire (i.e., a QoD bound that is almost reached). Specifically, references to actions are held in a list ordered ascending by time of expiration, which is the time of last execution of a dependent action plus $\theta$. In effect, the Scheduler component starts from the first element of the list checking if timestamps are older or equal than current time. As the list is ordered, the Scheduler has only to fail one check to ignore the rest of the list; e.g., if the check on the first element fails (its timestamp has not expired yet), the Scheduler does not need to check the remaining elements of the list.

As described in Section 2, the possible various QoD states, associated with an action, can be combined using provided operators. If no operators or mode are provided, the mode *all* is used, enforcing that every single associated QoD bound should be reached in order to trigger the relative action. If any limit is reached and an action is initiated, all QoD state vectors associated with that action are reset: $\theta$ receives a new timestamp, $\sigma$ and $\nu$ go to zero.

### 3.3 Dynamic adjustment of quality-of-data constraints

As previously mentioned, users may also specify intervals of values on the QoD vectors (instead of single values), and let the framework automatically adjust the quality constraints (within the intervals), hence varying the level of data modifications necessary to trigger successor actions, while preventing excessive load and error accumulation. This adjustment, performed by the QoD Engine component, is driven by two factors: i) the frequency of recent write operations to data items, during a given time frame; and ii) the current availability of computer resources and relative capabilities.

As for the former factor, we relax the QoD bound upon many consecutive updates, in an attempt to reduce the inherent overhead of triggering a given action an excessive number of times; i.e., we try to feed an action with as much data as possible within the upper boundary, as we anticipate further new input, instead of triggering that action with smaller subsets of that same data; hence, increasing throughput and resource efficiency. Conversely, we restrict the bound when updates are becoming less frequent and more spaced in time to increase the speed and reduce latency of the pipeline and dataflow processing steps.

The other factor, adjustment based on resource availability, consists of monitoring (based on a library abstracting system calls from different operating systems) at runtime the computing resources such as CPU, memory and disk usage, and determine, based on reference values, if each machine (or weighted for all in a set of allocated machines) is, or is not, fully utilized in order to decrease, or increase, the dataflow processing speed; i.e., if a machine (or a the set of machines) is underutilized the QoD bound is restricted to augment the overall dataflow performance; otherwise, if a machine is overloaded, the QoD bound is relaxed. This adjustment is performed in a progressive manner to avoid jitters.

These two factors can be entwined in the following way. Assuming the outcome of the assessment of each factor is either: *restrict*, *relax*, or *none*; if one factor decides *relax* and the other decides *restrict*, then no action is taken (i.e., factors disagree). If one factor decides *relax* or *restrict* and the other decides *none*, then the resulting bound is *relaxed* or *restricted* respectively. Otherwise, the factors agree and the adjustment is made in accordance with the outcome (*relax*, *restrict*, or *none*).

For not-so-expert users, we also provide a mechanism to automatically and dynamically adjust the QoD constraints. Users have only to specify the *significance factor*, i.e., the percentage of changes in the dataflow output (or against a reference) that would be meaningful and *significant* to decision-makers. For example, an air-sampling smoke detector should only issue a signal to a fire alarm system if the concentration of micro particles of combustion found is high enough (or *significant*); e.g., the fire alarm should not be triggered by the smoke of a simple cigar.

In this mode, vector element $\theta$ is simply set to a default constant. Figure 4 shows how the sequence constraint ($\sigma$) is adjusted, by successive approximation and assessment, ensuring that the target *significance* is met at the output of the final step. This, by inferring, backwards along the dataflow, the maximum QoD at each step that still achieves it.

```
 1 private void init(List<Step> steps) {
 2   for(s : steps) {
 3     s.setSeq(1);
 4     s.setQoDComplete(false);
 5     s.setState(0);
 6   }
 7 }
 8
 9 private void qodSeqUpdate(Step step) {
10   double currentDelta = step
11     .getCurrentOutDelta();
12   double targetDelta = step.getTargetDelta();
13   if(step.getState() == 0) {
14     if(currentDelta <  targetDelta)
15       step.setSeq(step.getSeq() * 2);
16     else
17       step.setState(1);
18   }
19   if(step.getState() == 1) {
20     int mid = Math.abs(step.getSeq() - step.
           getPreviousSeq()) / 2;
21     if(isEqualWithinEpsilon(currentDelta,
           targetDelta)) {
22       step.setQoDComplete(true);
23       if(step.previousStep() != null) {
24         step.previousStep().setTargetDelta(
25           step.getCurrentInDelta());
26         qodSeqUpdate(step.previousStep());
27       }
28     } else if(currentDelta < targetDelta)
29       step.setSeq(step.getSeq() + mid);
30     else if(currentDelta > targetDelta)
31       step.setSeq(step.getSeq() - mid);
32   }
33 }
```

**Figure 4 QoD dynamic adjustment.**

First, the $\sigma$ constraint in all steps, besides the first, is initialized to 1, meaning that every time a step completes its execution, performing at least 1 update, all its successors are triggered (like in the SDF model). The *qodSeqUpdate* method is called upon a wave of incoming data over the steps that have not been adjusted yet (checked through the *qodComplete* boolean). First (lines 13-18), $\sigma$ is doubled until the amount of variation in the output (*currentDelta*) goes above the significance factor (*targetDelta*). This variation is calculated by summing the differences (in absolute value) between current and previous row's values and dividing by the sum of all previous values. The goal is to make *currentDelta* and *targetDelta* to match or be within a given small $\varepsilon$ (method *isEqualWithinEpsilon*).

After this first stage to find the maximum, $\sigma$ starts to converge to the optimal value, thereby decreasing its value when *currentDelta* is greater than *targetDelta*, and increasing when it is lower (lines 28-31). If they match (lines 21-27) - in reality within a given $\varepsilon$, the optimal value of $\sigma$ was found, and the *qodSeqUpdate* is called recursively for the predecessor step (if any), thereby setting its *targetDelta* to the *currentInDelta* (i.e., the current amount of variation in the input of the current step).

Applying this mechanism to the $\sigma$ constraint is sufficient for dataflows where output variation across waves is mostly stable (not necessarily linearly dependent), given the number of updates to the input. When this relationship does not hold, the dynamic adjustment mechanism targets the $\nu$ constraint instead, using an analogous approach to Figure 4. This way, it attempts to determine the maximum magnitude of the modifications made at the input of each step, regardless of the actual number of updates, that would still not produce any relevant change in the significance of the dataflow output results.

## 4 Implementation

In this section we present the relevant implementation details of a developed prototype, as a proof of concept, with the architecture aforementioned to demonstrate the advantages of our dataflow model when deployed as a WMS for high-performance and large-scale data stores.

### 4.1 Adopted technology

Starting from the top layer, and to avoid reimplementing basic workflow capabilities, we have implemented our model using Oozie, [10] which is a Java open-source workflow coordination system to manage Apache Hadoop [11] jobs. Hence, we adapted the Oozie triggering semantics, by replacing the time-based and data detection triggering mechanisms, with a notification scheme that is interfaced with the *Flu*$\chi$ framework process through Java RMI. In general, Oozie only has to notify when an action finishes its execution, and *Flu*$\chi$ only has to signal the triggering of a certain action; naturally, these notifications share the same action 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 BigTable [12] open-source Java clone, HBase [8], 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, *Fluχ* was also built in Java, and uses, i.a., the Saxon http://saxon.sourceforge.net/ XPath engine to read and process XML configurations files (e.g., the dataflow description); and the SIGAR http://support.hyperic.com/display/SIGAR/Home library for monitoring resource usage and machine loads. For efficiency, we followed the solution of adapting the HBase client libraries used by Java classes, representing the type of actions we tried at evaluation stage.

### 4.2 Library support and API

In order to intercept the updates performed by actions, we adapted the HBase client libraries by extending the implementation of some of their classes while maintaining their original APIs. http://hbase.apache.org/apidocs/overview-summary.html 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 action, operation, table, and column identifiers) to the *Fluχ* 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 *Fluχ* 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.

### 4.3 Definition of dataflows with QoD bounds

The QoD constraints, referring to the maximum degree of data modifications, are specified along with standard Oozie XML schemas (version 0.2), and given to the *Fluχ* middleware 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 object containers associated, i.e., using 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. Next, we present an example, in Figure 5, omitting some details for readability purposes.

These particular dataflow descriptions are then automatically adapted to the regular Oozie schema (i.e., without the QoD elements) and fed to the Oozie manager.

```
<workflow-app name=”sample-wf”
          xmlns=”uri:oozie:workflow:0.1”>
      ...
    <action name=”myfirstjavajob”>
        <java>
            <job-tracker>foo:9001</job-tracker>
            <name-node>bar:9000</name-node>
            <prepare>
                <delete path=”${jobOutput}”/>
            </prepare>
            <configuration>
                <property>
                    <name>mapred.queue.name</name>
                    <value>default</value>
                </property>
            </configuration>
            <main-class>org.apache.oozie.
                  MyFirstMainClass</main-class>
            <java-opts> -Dblah </java-opts>
            <arg>argument1</arg>
            <arg>argument2</arg>
        </java>
        <qod>
            <column id=”column0”>
                <time>600</time>
                <sequence>2000</sequence>
                <value>40</value>
            </column>
        </qod>
        <ok to=”mysecondjavajob”/>
        <error to=”errorcleanup”/>
    </action>
    <action name=”mysecondjavajob”>
        <java>
                ...
        </java>
        <qod combine=”all”>
            <column id=”column1”>
                <time>300</time>
                <sequence>1000</sequence>
                <value>20</value>
            </column>
            <group>
                <item type=”column” id=”column2” />
                <item type=”row” id=”123” />
                <time>600</time>
                <sequence>5000</sequence>
                <value>60</value>
            </group>
        </qod>
        <ok to=”end”/>
        <error to=”fail”/>
    </action>
    <kill name=”fail”>
        <message>Java failed, error message
             [${wf:errorMessage(wf:lastErrorNode())}]
        </message>
    </kill>
    <end name=”end”/>
</workflow-app>
```

**Figure 5 *Fluχ* dataflow description.**

Hence, our framework controls the upper workflow management system and it is not necessary to perform additional configurations on such external systems (i.e., all configurations must go through $Flu\chi$). Nevertheless, we envision in the future for a more general dataflow description, where it can be, afterwards, automatically adapted to a range of popular WMSs.

## 5 Evaluation

This section presents the evaluation of the $Flu\chi$ framework and its benefits when compared with the regular DAG semantics (i.e., SDF with no QoD enforcement). More precisely, and attending to our objectives, we analyze the gains of $Flu\chi$ with dataflows for continuous and incremental processing in terms of: i) result convergence, as the dataflow execution pipeline advances; ii) error coverage; and iii) machine loads and resource usage through the amount of executions performed/saved. All tests were conducted using 6 machines with an Intel Core i7-2600K CPU at 3.40GHz, 11926MB of available RAM memory, and HDD 7200RPM SATA 6Gb/s 32MB cache, connected by 1 Gigabit LAN.

### 5.1 Prototypical scenario

For evaluating our model and framework we relied on a dataflow, for continuous and incremental processing, that expresses a simulation of a prototypical scenario inspired by the calculation the Air Quality Health Index (AQHI), www.ec.gc.ca/cas-aqhi/ 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. More specifically, the incoming data fed to this dataflow is obtained through several detectors equally distributed over an area of 10000 square units. Each detector comprises three sensors to gauge the amount of Ozone ($O_3$), Particulate Matter ($PM_{2.5}$) and Nitrogen Dioxide ($NO_2$). In effect, each sensor corresponds to a different generating function, following a distribution with smooth variations across space (i.e., realistic while exactness not relevant for our purposes), which will provide the necessary data to the dataflow. These 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).

Figure 6 illustrates the dataflow with the associated QoD vectors and the main HBase columns (some columns were omitted for readability purposes) that comprise the object containers in which the processing steps' triggering depends on. $k$ specifies i) the maximum time, in seconds, the action can be on hold; ii) the minimum amount, in percentage, of changes necessary to the triggering (e.g., 20% associated to step $C$ means that this action will be triggered when at least 20% of the detectors have been changed by step $B$; and iii) the maximum accepted divergence, in units.

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 (i.e., randomly chosen in practice) 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 object 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 (a number from 0 to 100 also) 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

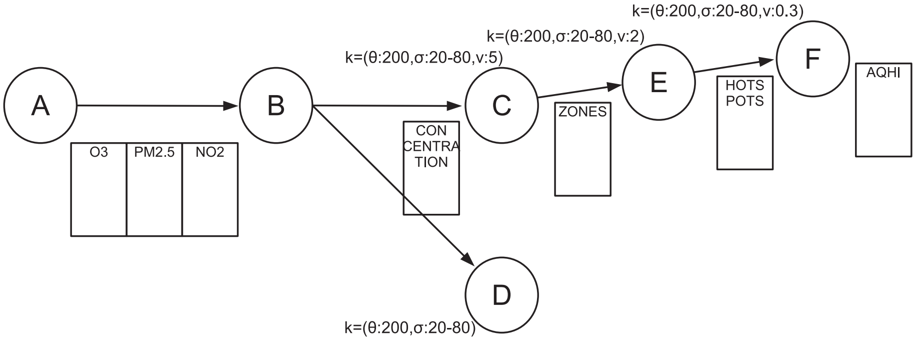

**Figure 6** $Flu\chi$ **dataflow for AQHI calculation.**

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 (as traditionally, red and darker means higher risk, while green and lighter yellow means reduced risk). This step can be executed in parallel with Step E.

***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 analyzation.

***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 ($O_3$, $PM_{2.5}$ and $NO_2$) on all hotspots. Then, the AQHI index is produced and stored for each wave of incoming data.

We conducted the evaluation for 2500 ($50 \times 50$) to 40000 ($200 \times 200$) detectors with 1 to 6 nodes and averaged the results over several runs to reduce noise. We simulated this experiment as though we were analyzing the pollution of a city for a week, with a wave of incoming data (from changed detectors) fed to the dataflow at each hour, which performs 168 waves in total (24 hours per 7 days). Also, we used distributions of pollution with 3 different tendencies in the generating functions (mimicking the sensors): increasing over time, decreasing over time, and globally uniform over time. Following, we analyze the most important aspects of correctness and performance, for all the steps with QoD enforcement in the AQHI dataflow.

### 5.2 Step C analysis

Through Figure 7 we may see the pollution concentration, on average of all zones, per each wave, while varying the QoD sequence vector, $\sigma$, in 20, 40, 60 and 80% of changed detectors (new data), and comparing against the concentration without using QoD. As depicted, the zone concentration on average with QoD converges to the concentration without QoD. It takes more time (or waves) to converge as we increase the minimum percentage of detectors detecting changes ($\sigma$). In this particular trial, the tendency configured on the generating functions was to increase the pollution as the number of waves increase. Our trials allow us to show that the differences between values calculated with and without QoD are always representatively small and bounded. Moreover, our other trials also show that the values of concentration with and without QoD mostly converge, i.e. differences are diminishing. This confirms the initial motivation that it would waste resources, for most purposes, to execute the dataflow completely for each wave, as the increase in output accuracy may be deemed as not significant, or relevant.

Figure 8 shows the maximum deviation (or error) of the concentration calculated, in relation to the pollution observed with no QoD, when varying $\sigma$ from 10 to 100%, meaning triggering execution only when there is new input for all sensors. The maximum error always stays below our defined threshold (vector $\nu$) and the error increases with a linearly tendency as the waves or number of changed detectors increase. Despite, some noise and jitters (introduced by the variation of hundreds or thousands of sensors), the linear trend is clearly observable.

Through Figure 9 we can see that the number of executions decreases in an almost linear fashion as the

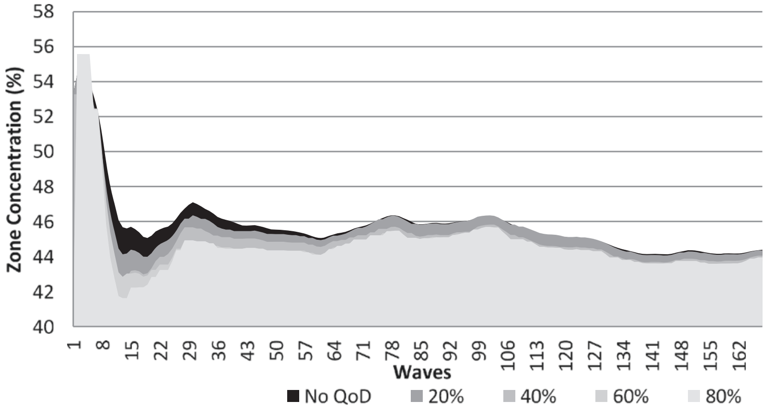

**Figure 7 Average pollution concentration in zones for number of updates up to 20, 40, 60 and 80% of detector count as $\sigma$.**

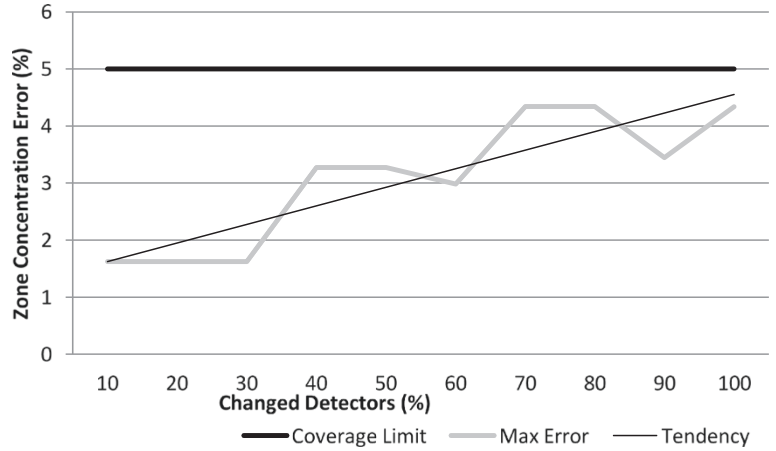

**Figure 8** Zone concentration maximum error with increasing QoD bounds $\sigma$.

allowed percentage of changed detectors ($\sigma$) increases. The number of step executions performed without QoD is naturally equal to the number of waves, 168, corresponding to the 100%. When $\sigma$ was 25% we saved about 20% of 168 executions (i.e., fewer 33 executions than using regular DAG semantics); and for 80% of detected changes we only performed 80 executions (48%). The machine loads and resource utilization were naturally proportional to the savings presented here.

### 5.3 Step D analysis

We present the graphs generated during a day (24 samples) using regular DAG semantics and contrast them against the *Fluχ* model with QoD, for 20, 40, 60 and 80% of variation in vector $\sigma$.

Without QoD, Figure 10 illustrates the evolution of the concentration of pollution in the city during a day. Areas colored in shades of green represent safer zones with lower pollution concentrations (low health risk); yellow areas represent medium pollution concentration (moderate health risk); and colors ranging from orange to red indicate hotspots (high and very high health risk).

Figure 11 presents a similar matrix on the left and a difference matrix on the right. The former illustrates the evolution of pollution, but enforcing QoD, which means that not all 24 samples are generated, and thus there are repeated samples (i.e., during 2 or more hours the samples can be equal). The latter shows the differences between the repeated samples and the original ones (generated for each hour without QoD) with a maximum error of 5%, representing the darkest areas. Hence, brighter areas mean that the differences were minimal.

Figure 11a depicts a matrix with tiles generated when 20% of detectors have perceived changes. The divergence was minimal: only the 5th not-updated tile was darker (above 2.5% of difference) as at that hour the pollution had already decreased.

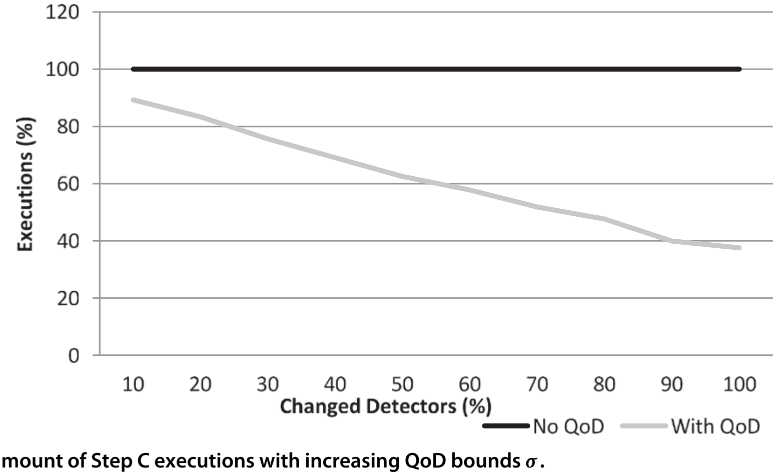

**Figure 9** Comparison of amount of Step C executions with increasing QoD bounds $\sigma$.

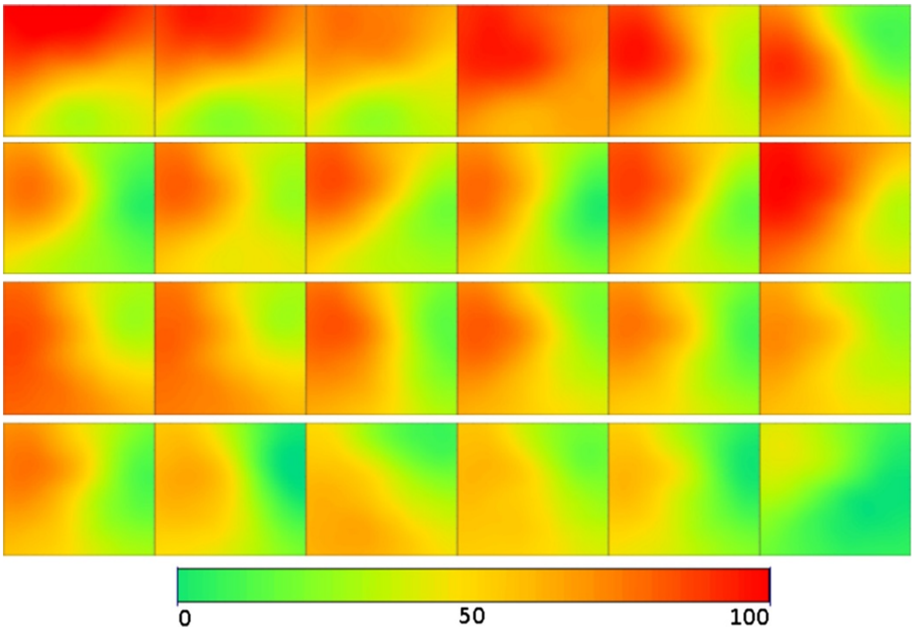

**Figure 10 Samples collected during 24 hours with no QoD enforcement.**

In Figure 11b, 40% of changed detectors are needed in order to generate new and updated tiles. The black and white matrix shows slightly darker tiles than the previous trial (Figure 11a) for the first hours of the day. Again, the levels of pollution at that hour were decreasing. When dealing with the opposite situation, levels increasing, the vector $\nu$ component comes to place and guarantees that a strong variation on pollution (above 5%) concentrations will cause the graph to be re-generated.

Through Figure 11c we can see that the generated tiles follow the same tendency of becoming darker as $\sigma$ augments. The difference matrix shows moderate variations in the tiles per hour, however notice that more than a half of the detectors have perceived changes (this hints that it might not be the most appropriate value of $\sigma$ for a real environment).

Finally, with a $\sigma$ of 80% (Figure 11d) more error was introduced, but still within the acceptable limit of 5%. The contiguous black and white tiles do not show much difference in their color, but, instead, on the location of the pollution concentrations; meaning that there is not much variation in the overall concentration levels of pollution and that the pollution is flowing from area to area.

To conclude, we can see that for higher levels of changed detectors (60 and 80%) the differences and errors are higher, but this higher divergence on some tiles happened due to the levels of pollution being greater with QoD than with the original tiles calculated without QoD (and not the opposite, which would be more dangerous). Notwithstanding, black and white graphs in general were brighter and thus acceptable (especially for realistic and lower levels of $\sigma$), supporting the intuitive notion and our arguments that the dataflow does not need to be recalculated every time a single, or a few, changes occur.

### 5.4 Step E analysis

Now using uniform distributions to generate the pollution concentrations, we may observe, through the charts of Figure 12, that the most divergence of concentration in hotspots, between using QoD and no QoD, occurs when $\sigma$ is 40 and 60% (i.e., the percentage of minimum changes necessary in the concentration of zones to trigger step $E$). The concentrations are very close with and without QoD for 20% of $\sigma$ due to the small oscillations and peaks of the generated values. As for the 80%, the error is also smaller, since there are even less oscillations; i.e., the average is more stabilized as step $E$ is executed fewer times.

Figure 13 shows in percentage the number of hotspots for each wave when varying $\sigma$ for 20, 40, 60 and 80% of sensors. As the previous figures show, the most divergence happens in the waves leading to the middle of the sequence in the graph (waves 35-85) for the same reasons explained.

Figure 14 shows that the maximum deviation error follows an order 2 polynomial tendency, and therefore we will have, for an uniform distribution of pollution, higher errors when the percentage of changed zones are set in the middle of the range (unlike when pollution is increasing or decreasing, as afore demonstrated). Furthermore, when step $E$ was triggered, it was never due to the error

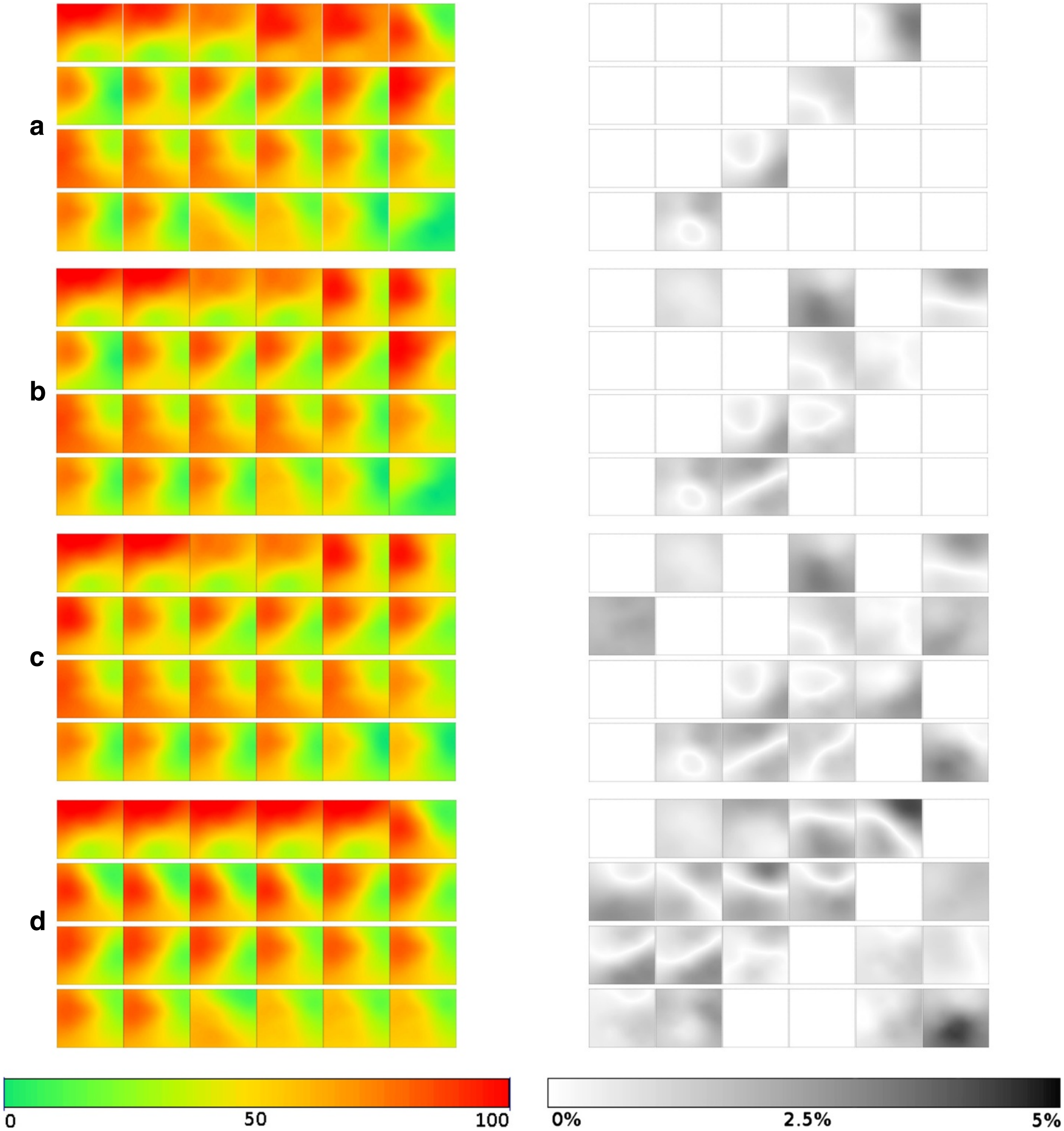

**Figure 11 Samples collected, and differences against the regular SDF model, during 24 hours. a** requiring 20% of changed detectors. **b** requiring 40% of changed detectors. **c** requiring 60% of changed detectors. **d** requiring 80% of changed detectors.

being greater than $\nu$, 2, which happened due to the regular tendency in the concentration distribution.

In Figure 15 we may see the impact in the percentage of executions when combining the QoD of steps $C$ and $E$ (i.e., minimum percentage of changes in zones and detectors). For this particular trial, step $E$: i) presents an improvement, almost linear, in the number of executions when no QoD is enforced on step $C$; and ii) only improves starting from 75% when QoD is enforced for the detectors. In a dataflow with pipeline processing, like the one considered, it is natural that the QoD of previous or upstream steps influence the executions of current and downstream steps in the pipeline, since the inputted data is derived from upstream, i.e., from the beginning of the processing.

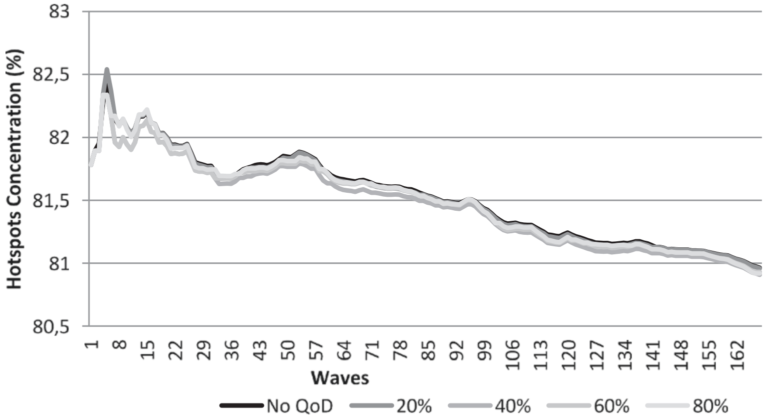

**Figure 12** **Average concentration of pollution in hotspots for number of updates up to 20, 40, 60 and 80% of zones count as $\sigma$.**

### 5.5 Step F analysis

Since the Air Quality Health Index is a single discrete scalar value, we observe a step plot represented by the lines of the chart depicted in Figure 16, where we compared the accumulated average of the index with and without QoD for levels of changes in the number of hotspots ($\sigma$) of 20, 40, 60, and 80%. Due to the uniform distribution of pollution used, the lines are roughly parallel starting on the 18th wave, and, as the $\sigma$ increases, the QoD lines become further distant from the *No QoD* line, meaning an increasing on the deviation of the index. Nevertheless, this deviation reaches our coverage limit of 0.3 ($\nu$) roughly from 60% of $\sigma$ and therefore the divergence of the lines corresponding to 60 and 80% is much smaller. Moreover, the step effect is higher for greater values of $\sigma$, so the index is steady until $\sigma$ or $\nu$ are reached.

Through Figure 17 we may see that the error increases with the percentage of changed hotspots and roughly follows a linear tendency. This increase is more abrupt from 20 to 60%, also showing the impact that $\nu$ had on the index values; i.e., the increase was smaller from 60%.

We fixed the QoD of the previous steps in the dataflow and analyzed the gains in terms of executions of step $F$ (Figure 18). A great quantity of executions were saved, even for 20% of changed hotspots where about 70% of the total executions without any QoD (i.e., 168 executions) were spared. At 80% of changed hotspots, only about 5% of the total executions were performed with an error not greater than 0.3. It is natural that, as we go through the actions of the pipeline, the number of executions with QoD is reduced, since the noise from the raw data injected in the dataflow is funnelled through the processing chain into more refined and structured data.

### 5.6 Overall analysis

Figure 19 shows the running time of a complete cycle of 168 waves with different loads (2500, 10000, 22500, and 40000 detectors) for 1 to 6 nodes. As the number of nodes increases we may see that the time remains roughly constant, showing that our model with HBase can achieve scalability, and almost practically constant access times. We stress that these are only exemplificative: real

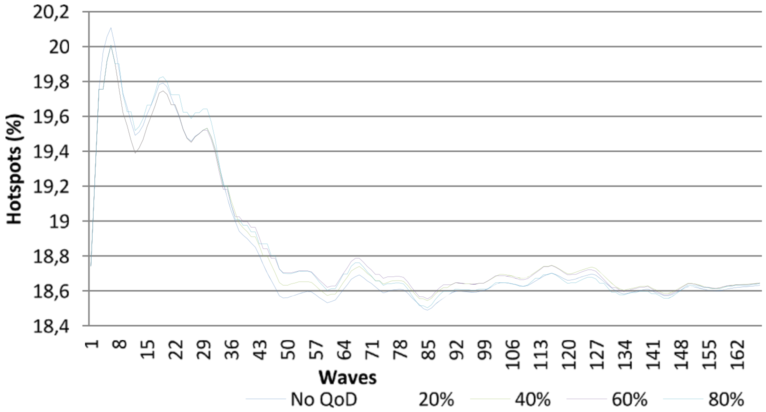

**Figure 13** **Amount of hotspots for different QoD levels.**

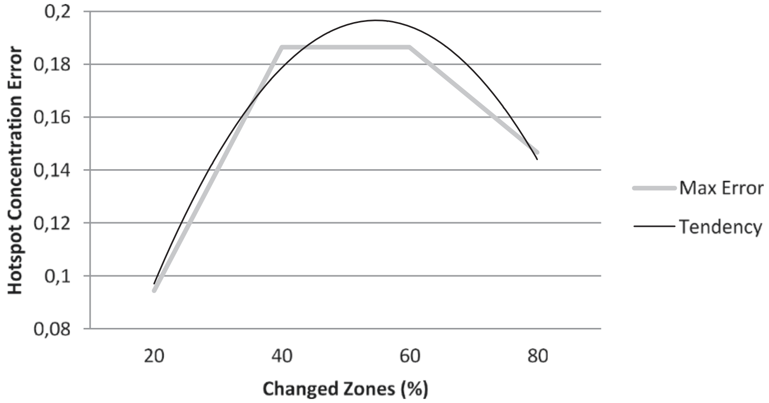

**Figure 14** Hotspot concentration error.

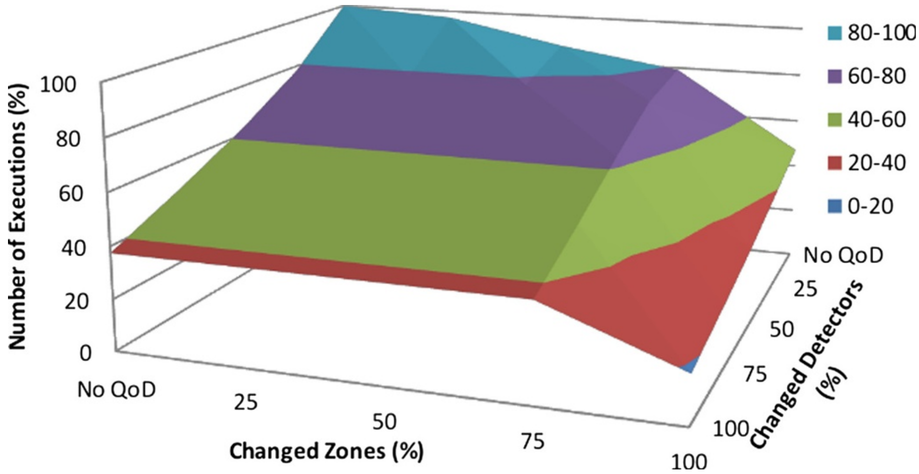

**Figure 15** Hotspot executions.

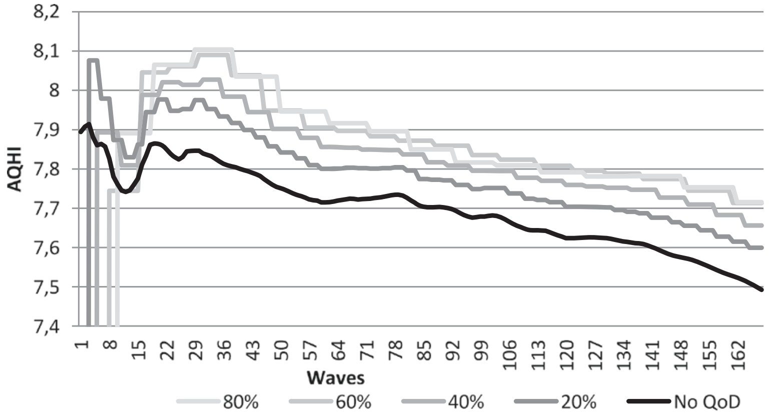

**Figure 16** AQHI for number of updates up to 20, 40, 60 and 80% of hotspots count as $\sigma$.

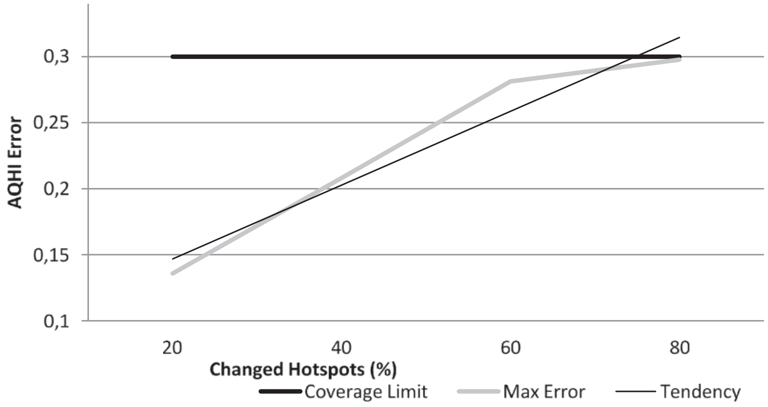

**Figure 17** AQHI maximum error.

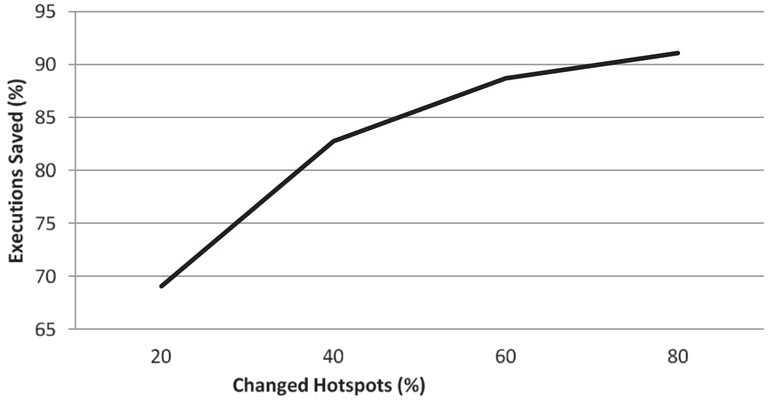

**Figure 18** AQHI saved executions.

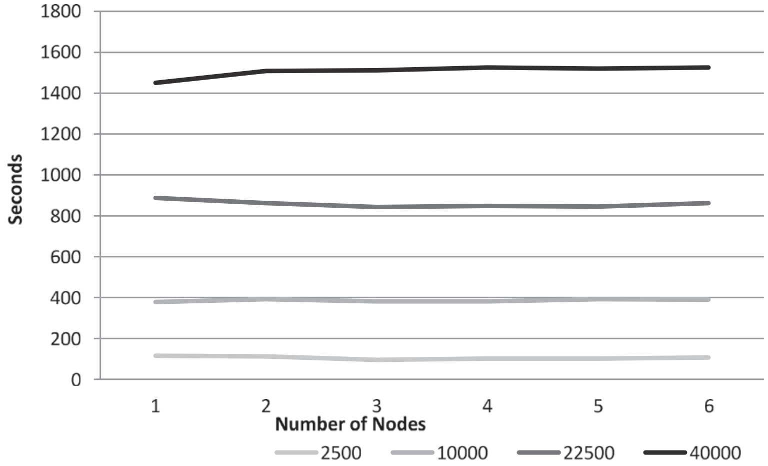

**Figure 19** Execution times for 168 waves with different loads (number of sensors) and increased distribution (number of storage nodes).

life calculations for each wave may involve greater computational effort both due to complexity and to higher sampling rates; possibly, many other dataflows may be also being executed in a shared infrastructure. Thus, gains in real life settings may be more significant.

In Figure 20, the average load of tasks during a cycle of 168 waves is shown when $\sigma$ is 25, 50, 75, and 100%, as the cluster increases in size from 1 to 6 nodes. The total tasks are calculated by multiplying the executed dataflow's tasks (6) by the total number of waves (168). Tasks executions are scheduled across the cluster worker nodes by following a round-robin scheduling, hence saved executions will tend to adhere to this distribution as well. In fact, the average load observed is naturally in line with what would result from dividing the total number of tasks by the number of nodes in the cluster. We can see that i) the gains with QoD are higher for higher ratios of *tasks / number_of_nodes*, and ii) the loads converge, in absolute values, as the number of nodes increases. More importantly, we assessed the load balancing across the cluster, and observed, as depicted in Figure 21 that, for all QoD levels, the load across the 6 nodes in the cluster is very evenly distributed around the average values. Achieving resource savings by avoiding dataflow executions and ensuring load balance across the cluster, combined, allow the system to scale effectively.

Through Figure 22 we may see the variation of the output error as waves go by. This error, which comes from postponing the triggering of actions, corresponds to the deviation of the output that should have been modified having the dataflow been completely executed; i.e., this error is calculated by summing the differences (in absolute value) between current and previous row's values and dividing by the sum of all previous values. Also due to the restrictions on $\nu$, the steps are triggered when greater variations in magnitude occur and, therefore, the maximum error observed never goes above 25%, for the QoD range of values that we used in $\sigma$. Decision-makers should settle for a percentage of error that they can tolerate, i.e, up to a value that carries enough *significance* for the given activity, and depending on how critical it is, and their systems are. Notwithstanding, we consider an error up to 15% as quite acceptable for most monitoring activities, given the extensive gains in saved resources. Note that on average the error stayed under that mark.

### 5.7 Discussion

The results and patterns observed, for the executions of the AQHI dataflow with different QoD divergence bounds, corroborate the intuitive notion that most of the times, just because there is new data available, it would be neither necessary nor useful to re-execute the dataflow as the final results would suffer little or no difference, thus wasting resources and computational power. This also happens with other tests we performed with fire risk analysis in forests, and social impact of companies in blog references.

The problem with ad-hoc approaches is that the user is left with an all-or-nothing approach, or to simply define periodical (guessing) execution. With QoD, dataflow users and developers can define, with a sound model and approach, the precise conditions when they consider each individual step of a dataflow should be re-executed due to changes in its input being considered as relevant. Furthermore, we can improve resource efficiency in a predictable way as savings are proportional to the percentage of avoided re-executions.

## 6 Related work

In this section we review relevant solutions, within the current state of the art, that intersect the main topics approached in this work. First, we describe general and

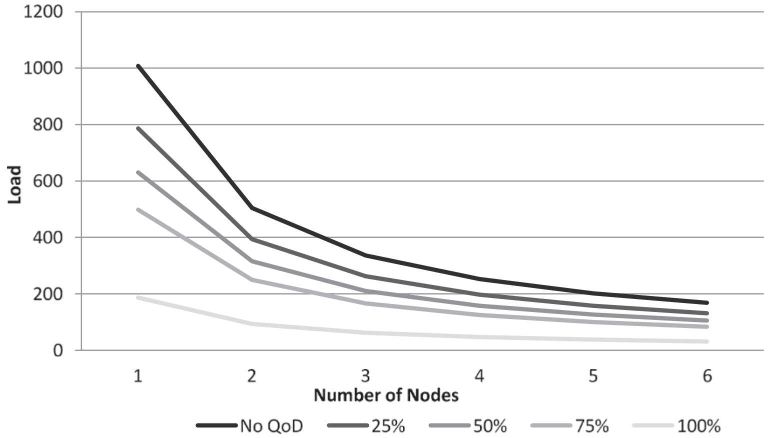

**Figure 20 Worker's load average from 1 to 6 nodes.**

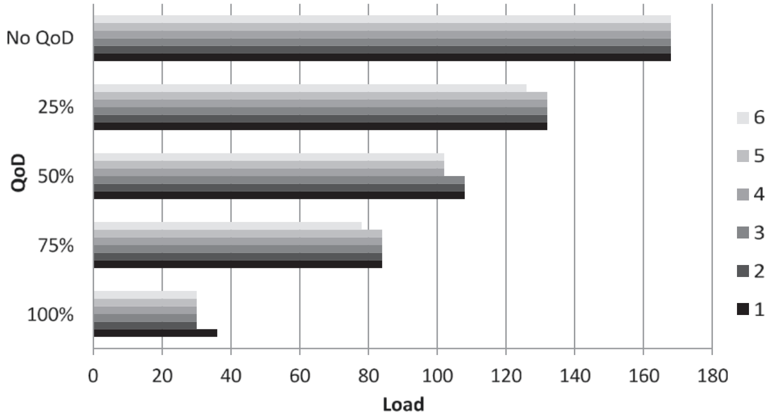

**Figure 21** Tasks' load balancing and saved executions across nodes.

e-science data/workflow systems. Next, we focus on solutions for incremental processing.

### 6.1 Workflow systems

DAGMan [13] is one of the early workflow languages in e-science. It interprets and manages text descriptions of jobs comprising directed acyclic graphs (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 Condor workload management system (which is centralized) for scheduling and does not represent data as a first-class entity. Still, DAGMan is very popular due to its integration with Condor.

Taverna [14], part of the myGrid project, is heavily used in bioinformatics. It is a workflow management system 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.

Triana [15] is 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.

Pegasus [16] is a long running project that extends DAGMan in order to allow mapping of workflows of jobs to remote clusters, and cloud computing infrastructures. It maps jobs on distributed resources and from the description of computation tasks, it performs necessary data transfers (required files) among sites. Pegasus aims at optimizing workflow performance and reliability by scheduling to appropriate resources but there are no

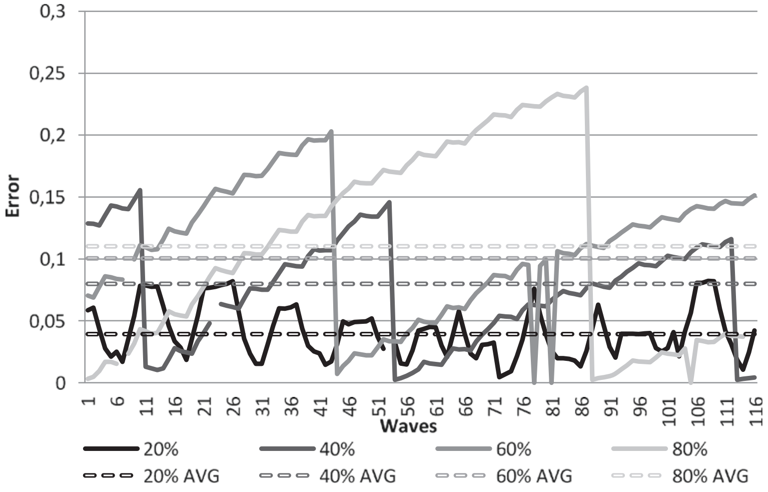

**Figure 22** Output error evolution across waves for different QoDs.

QoD guarantees on continuous processing or data flow, and no data sharing.

Dryad [17] 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 as C#, SQL and others. It has support for channels of shared mutable data.

Kepler [18] is a solution for managing scientific workflows. It was designed to help scientists and other non-expert 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 (e.g., in the Grid).

Our work is akin, and can be regarded as an advance, to the support for conditional workflows [19], supported by Triana and Kepler, but absent in dominant approaches such as Pegasus and DAGMan. First, they target mainly grid computing and not dataflows manipulating cloud storage. Second, in the approaches supporting conditional workflows, the conditions to be evaluated need to be expressed explicitly in the workflow, i.e. almost programmatically, and are usually actual functional decisions required at execution time. These are inserted in order to take independent paths of execution in a workflow depending on some shared state. We do not require workflow designers to *pollute* workflow descriptions with numerous conditional nodes assessing QoS or QoD criteria, they need only be expressed declaratively, outside the dataflow. Thus, the same dataflow description can be instantiated multiple times, and by different users, with different QoD criteria. Still, our approach does not forbid the usage of conditional nodes, it simply does not mandate it. Moreover, the enforcement of quality criteria is automated, based on information gathered from the cloud storage when data objects are updated. In essence, the conditional behavior of executing dataflow steps only when relevant new input is available, is completely declarative, automated and driven by goal-like criteria, instead of explicit, replicated across every node describing steps, and evaluated by manually developed, and opaque, code.

### 6.2 Incremental processing

MapReduce [20] is inspired by the map and reduce primitives in functional programming. Computation is divided into two sequential phases. The first is a mapping phase, which operates over each element in the input and produces a set of intermediate key/value pairs. A reduce phase follows where all values sharing the same key are processed and aggregated based on some application level logic. This allows for automatic parallelization. MapReduce is used in large clusters to analyze in parallel huge data sets in domains such as web log and graph analysis. It automatically partitions input data, schedules execution across the cluster, and handles nodes failures. It is batch-oriented so changes in input require full execution from scratch. While allowing custom functions for input partitioning, comparisons, and preliminary key/ value reduce, executed locally by combiners, MapReduce still forces programmers to obey a strict model different of those used for application logic. Though, the automatic parallelization and fault-tolerance features have drawn an enthusiastic community that has developed a complete open-source port of the original proprietary system in Hadoop [11]. Like Oozie, a few other workflow managers have arisen for Hadoop, such as Azkaban, http://sna-projects.com/azkaban/ Cascading, http://www.cascading.org and Fluxua. https://github.com/pranab/fluxua.

MapReduce is a powerful abstraction for simple tasks, e.g. word counting, that have to be applied to colossal amounts of data. This was its initial purpose: reverse index creation and page rankings, essentially weighted sums. More modern functionality such as supporting online social networks and data analytics are extremely cumbersome to code as a giant set of interdependent MapReduce programs. Reusability is thus very limited. To amend this, the Apache Pig platform [9] 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 [21] warehouse reinstates fully declarative SQL-like languages (HiveQL) over data in tables (stored as files in an HDFS directory). Queries are compiled into MapReduce jobs to be executed on Hadoop. SCOPE [22] takes a similar approach to scripting but targeting Dryad [17] for its execution engine.

HyMR [23] is a hybrid MapReduce workflow system that combines Hadoop and Twister [24] to enable efficient processing of iterative data analysis applications. It points out the inability of Hadoop to directly support iterative parallel applications, thereby requiring a driver program to orchestrate application iterations (each piped as a separate MapReduce job). This, however, has drawbacks, such as forcing the user to manually set the number of iterations (making it impossible for a program to ensure convergence to a given condition), and the re-scheduling overhead of mapping and reduce tasks on every application iteration. Twister, by its turn, allows iterative applications to run without any of those problems. However, it requires intermediate output files to be transferred from one node to another, instead of using and benefiting from a shared distributed file system, such as HDFS from Hadoop, with fault tolerance mechanisms. HyMR, therefore, combines Twister and Hadoop to take the best of each and support iterative programs. We also share data ultimately through Hadoop, albeit at a higher semantic level with HBase noSQL storage; however there is no performance reasoning about the data semantics and output impact in HyMR.

To avoid recreating web indexes from scratch after each web crawl, as most sites change slowly, Google Percolator [25] does incremental processing on top of BigTable, replacing batch processing of MapReduce. It provides row and table-wide transactions, snapshot isolation, with locks stored in special Bigtable columns. Observers allow programmers to monitor columns. Notify columns are set when rows are updated, with several threads scanning them. Applications are sets of custom-coded observers. At most one transaction is run when a column is modified, but several updates may be fed to the same transaction. Timestamps allow identifying new rows since last execution. Although it scales better than MapReduce, it has 30-fold resource overhead over traditional RDBMS. Nova [26] is similar but has no latency goals, accumulating many new inputs and processing them lazily for throughput. Moreover, Nova provides data processing abstraction through Pig Latin; and supports stateful continuous processing of evolving data sets.

Yahoo CBP [27] 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 (e.g., 1 hour epoch), 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 MapReduce implementation and some explicit programming when a QoD-enabled dataflow.

InCoop [28] aims at transparently detecting the repeated execution of the same task (code and input data) and retrieve from cache the results of previous executions. It allows simply restarting jobs from scratch when new data is available. Most re-computation is prevented and cached results used instead. 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 re-computation is avoided. Content-based splitting minimizes number of reprocessed partitions. Somehow like *Fluχ*, this project attempts to reduce the number of executions of processing steps; however, it implies that the input/output datasets are repeated or intersected among each other.

Nectar [29] for Dryad links data and the computation that generated it as unified hybrid cacheable element. When data is unused for long, it is removed and replaced by the computation that produced it to be rerun later if needed. On Dryad programs reruns, Nectar replaces results partially, or totally, with cached data. Dryad programs need to be enhanced with cache management calls that check and update the cache server. Cached results and modified programs are managed in a central store. Cacheable elements include sub-expressions, and DAGs shared by different processes operating on the same data. Like InCoop, Nectar is advantageous only for scenarios where input/output is repeated, whereas the QoD model fits a broader range of scenarios.

In [30], it is presented a formal programming and scheduling model for defining temporal asynchrony in workflows (motivated by the need of low-latency processing of critical data). 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 (which are 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.

## 7 Conclusion

In this article we presented *Fluχ*, a novel dataflow model with framework and library support, for data-intensive computing, capable of orchestrating different data-based computation steps, while enforcing quality constraints over the data shared among those steps. With *Fluχ*, we aim at enhancing the workflow and dataflow paradigms with quality-of-service notions, expressed by constrains over the divergence of data and the bounds on input data, that should trigger re-execution of a computational step, and update of its output. We call this enforcement quality-of-data (QoD).

Such quality-of-data enforcement is thus used to guide, and to some extent, autonomously schedule the execution and triggering semantics of dataflows. This allows achieving controlled performance and high resource efficiency, flexibility and elasticity, which is essential in today's cloud-like environments. Such properties are increasingly more relevant nowadays, where data is digitally flowing all over the world, throughout the Internet: ranging from smartphones to desktops, and where a single click or tap on an application may generate large streams of information, that need to be properly, and resource efficiently, processed in support of keeping up the pace in the innovation space.

The *Fluχ* model and supporting framework and library were implemented and found both easy to integrate with existing WMS infrastructures, as well as with currently popular cloud tabular storage (HBase) for scalability. To demonstrate *Fluχ* feasibility, usefulness, and efficiency, the assessment of *Fluχ* 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. The evaluation of *Fluχ* revolved around three fundamental criteria: i) result convergence, showing that using QoD divergence bounding criteria does not introduce significant errors in results; ii) execution overhead, showing that we are able to avoid large numbers of multiple repetitive executions of dataflow steps; and iii) that due to the aforementioned, we reduce machine load, e.g., in cluster, grid or cloud infrastructures, as well as improving resource usage efficiency for the same level of data *value* generated by the dataflows.

Therefore, we find *Fluχ* 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.

## Endnote

[a]*Quality-of-Data* is a novel concept, akin to SLA, different from *data quality*, that traditionally refers to other issues such as internal data correctness, semantic coherence, data adherence to real-life sources, or data appropriateness for managerial and business decisions.

### Competing interests

The authors declare that they have no competing interests.

### Authors' contributions

All authors read and approved the final manuscript.

**Acknowledgements**

This work was partially funded by FCT under projects PTDC/EIA-EIA/102250/2008, PTDC/EIA-EIA/108963/2008, and PEst-OE/EEI/LA0021/2011, and PhD grant SFRH/BD/80099/2011. We also would like to thank the anonymous reviewers who greatly contributed to the betterment of this work.

Received: 6 February 2013 Accepted: 6 February 2013
Published: 4 April 2013

### References

1. Ahrens J, Hendrickson B, Long G, Miller S, Ross R, Williams D (2011) Data-intensive science in the us doe: Case studies and future challenges. Comput Sci Eng 13(6): 14–24. doi:10.1109/MCSE.2011.77
2. Deelman E, Callaghan S, Field E, Francoeur H, Graves R, Gupta N, Gupta V, Jordan TH, Kesselman C, Maechling P, Mehringer J, Mehta G, Okaya D, Vahi K, Zhao L (2006) Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing, E-SCIENCE '06. IEEE Computer Society, Washington, p 14. doi:10.1109/E-SCIENCE.2006.99
3. Falgout J (2011) Dataflow programming: Handling huge loads without adding complexity the basic concepts of dataflow programming. Dr. Dobb's. http://www.drdobbs.com/database/dataflow-programming-handling-huge-data/231400148
4. Livny J, Teonadi H, Livny M, Waldor MK (2008) High-Throughput, kingdom-wide prediction and annotation of bacterial non-coding RNAs. PLoS ONE 3(9): e3197+. doi:10.1371/journal.pone.0003197
5. York DG, et al. (2000) The sloan digital sky survey: Technical summary. Astronomical J 120(3): 1579
6. Ludäscher B, Altintas I, Bowers S, Cummings J, Critchlow T, Deelman E, Roure DD, Freire J, Goble C, Jones M, Klasky S, McPhillips T, Podhorszki N, Silva C, Taylor I, Vouk M (2009) Scientific process automation and workflow management. In: Shoshani A Rotem D (eds). Scientific Data Management, Computational Science Series, chap. 13. CRC press, Boca raton. http://www.crcpress.com/product/isbn/9781420069808
7. Juve G, Deelman E, Berriman GB, Berman BP, Maechling P (2012) An evaluation of the cost and performance of scientific workflows on amazon ec2. J Grid Comput 10(1): 5–21
8. George L (2011) HBase: The Definitive Guide, 1edn. O'Reilly Media, Sebastopol. http://shop.oreilly.com/product/0636920014348.do#
9. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A (2008) Pig latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. SIGMOD '08. ACM, New York, pp. 1099–1110. doi:10.1145/1376616.1376726
10. The Apache Software Foundation (2013) Apache Oozie Workflow Scheduler for Hadoop. http://oozie.apache.org/
11. White T (2009) Hadoop: The Definitive Guide, 1st edn. O'Reilly Media, Inc., Sebastopol. http://shop.oreilly.com/product/0636920021773.do
12. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2006) Bigtable: a distributed storage system for structured data. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7. OSDI '06. USENIX Association, Berkeley, pp. 15–15
13. Couvares P, Kosar T, Roy A, Weber J, Wenger K (2007) Workflow management in condor. In: Taylor IJ, Deelman E, Gannon DB, Shields M (eds). Workflows for e-Science. Springer, London, pp. 357–375
14. Missier P, Soiland-Reyes S, Owen S, Tan W, Nenadic A, Dunlop I, Williams A, Oinn T, Goble CA (2010) Taverna, reloaded. In: SSDBM. Springer-Verlag Berlin, Heidelberg, pp. 471–481
15. Taylor I, Shields M, Wang I, Harrison A (2007) The Triana workflow environment: architecture and applications. In: Taylor I, Deelman E, Gannon D, Shields M (eds). Workflows for e-Science. Springer, New York, Secaucus, pp. 320–339
16. Lee K, Paton NW, Sakellariou R, Deelman E, Fernandes AAA, Mehta G (2009) Adaptive workflow processing and execution in pegasus. Concurr Comput: Pract Exper 21(16): 1965–1981. doi:10.1002/cpe.v21:16
17. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D (2007) 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, pp. 59–72. doi:10.1145/1272996.1273005
18. Altintas I, Berkley C, Jaeger E, Jones M, Ludäscher B, Mock S (2004) Kepler: An extensible system for design and execution of scientific workflows. Sci Stat Database Manag Int Conf 0(423). http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1311241
19. Bahsi EM, Ceyhan E, Kosar T (2007) Conditional workflow management: A survey and analysis. Sci Program 15(4): 283–297
20. Dean J, Ghemawat S (2004) 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, pp. 10–10
21. Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R (2009) Hive- a warehousing solution over a map-reduce framework. In: IN VLDB '09: Proceedings of the vldb endowment. Very Large Data Base Endowment Inc., USA, pp. 1626–1629
22. Chaiken R, Jenkins B, Larson PA, Ramsey B, Shakib D, Weaver S, Zhou J (2008) Scope: easy and efficient parallel processing of massive data sets. Proc VLDB Endow 1(2): 1265–1276. http://dl.acm.org/citation.cfm?id=1454166

23. Ruan Y, Guo Z, Zhou Y, Qiu J, Fox G (2012) Hymr: a hybrid mapreduce workflow system. Tech. rep., Indiana University, Bloomington, IN
24. Ekanayake J, Li H, Zhang B, Gunarathne T, Bae SH, Qiu J, Fox G (2010) Twister: a runtime for iterative mapreduce. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10. ACM, New York, pp. 810–818. doi:10.1145/1851476.1851593
25. Peng D, Dabek F (2010) Large-scale incremental processing using distributed transactions and notifications. In: Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10. USENIX Association, Berkeley, pp. 1–15
26. Olston C, Chiou G, Chitnis L, Liu F, Han Y, Larsson M, Neumann A, Rao VB, Sankarasubramanian V, Seth S, Tian C, ZiCornell T, Wang X (2011) Nova: continuous pig/hadoop workflows. In: Proceedings of the 2011 international conference on Management of data, SIGMOD '11. ACM, New York, pp. 1081–1090. doi:10.1145/1989323.1989439
27. Logothetis D, Olston C, Reed B, Webb KC, Yocum K (2010) Stateful bulk processing for incremental analytics. In: Proceedings of the 1st ACM symposium on Cloud computing, SoCC '10. ACM, New York, pp. 51–62. doi:10.1145/1807128.1807138
28. Bhatotia P, Wieder A, Rodrigues R, Acar UA, Pasquin R (2011) Incoop: Mapreduce for incremental computations. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC '11. ACM, New York, pp. 7:1–7:14. doi:10.1145/2038916.2038923
29. Gunda PK, Ravindranath L, Thekkath CA, Yu Y, Zhuang L (2010) Nectar: automatic management of data and computation in datacenters. In: Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10. USENIX Association, Berkeley, pp. 1–8
30. Olston C (2011) Modeling and scheduling asynchronous incremental workflows. Tech. rep., Yahoo! Research

doi:10.1186/1869-0238-4-12
**Cite this article as:** Esteves *et al.*: $\boldsymbol{Flu\chi}$**: a quality-driven dataflow model for data intensive computing.** *Journal of Internet Services and Applications* 2013 **4**:12.

**Submit your manuscript to a SpringerOpen journal and benefit from:**

► Convenient online submission
► Rigorous peer review
► Immediate publication on acceptance
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

**Submit your next manuscript at ► springeropen.com**