Elastic Stream Processing with Latency Guarantees

Björn Lohrmann
Technische Universität Berlin
Email: bjoern.lohrmann@tu-berlin.de

Peter Janacik
Technische Universität Berlin
Email: peter.janacik@tu-berlin.de

Odej Kao
Technische Universität Berlin
Email: odej.kao@tu-berlin.de

***Abstract*—Many Big Data applications in science and industry have arisen, that require large amounts of streamed or event data to be analyzed with low latency. This paper presents a reactive strategy to enforce latency guarantees in data flows running on scalable Stream Processing Engines (SPEs), while minimizing resource consumption. We introduce a model for estimating the latency of a data flow, when the degrees of parallelism of the tasks within are changed. We describe how to continuously measure the necessary performance metrics for the model, and how it can be used to enforce latency guarantees, by determining appropriate scaling actions at runtime. Therefore, it leverages the elasticity inherent to common cloud technology and cluster resource management systems. We have implemented our strategy as part of the Nephele SPE. To showcase the effectiveness of our approach, we provide an experimental evaluation on a large commodity cluster, using both a synthetic workload as well as an application performing real-time sentiment analysis on real-world social media data.**

I. Introduction

Many Big Data applications from the areas of science and industry require large amounts of streamed data to be analyzed in (near) real-time. In these so-called *stream processing applications*, data usually arrives from various outside sources and must be immediately processed to yield quick results. Examples can be found in the context of market research, real-time monitoring, financial trading, live media streaming and many others. Often, stream processing applications are 'under-the-hood' components of large websites, providing personalized content- and ad-serving [1], [2] or real-time recommendations [3]. The definition of *real-time* depends very much on the application itself. For an application performing computation for dashboard-style monitoring, a delay of up to several seconds may be acceptable. An application that is part of a user-interactive system, 100 ms (round-trip) is the upper time limit for the application to be perceived as fluid [4].

Due to the real-time requirements of these applications, the traditional "process-after-store" model [5] of the established batch-style Big Data analytics platforms [6]–[8] is not a good fit. This has led to the creation of highly scalable general-purpose Stream Processing Engines (SPEs) [1], [9]–[18]. These engines efficiently execute data flows with arbitrary user code in a highly distributed and parallel fashion and generally are optimized towards low latency. Some of them are elastic [9]–[14], while some are not (yet) [1], [16]–[18]. The unelastic ones require permanent peak-load resource provisioning to remain low latency in the face of varying and bursty load. In general, elastic systems require a *scaling policy*, that determines when and how much to scale. The elastic engines mentioned above, either do not specify a policy [9], [11], or employ a policy that is rate-based [13] or based on CPU load thresholds [10], [14]. However, which particular stream rates or CPU load thresholds lead to a particular latency for a complex UDF-heavy data flow is not in the scope of these policies.

This paper proposes a strategy to provide latency guarantees in SPEs that execute UDF-heavy data flows. The goal is to provide latency guarantees *while minimizing resource consumption*. We are building on results from one of our previous publications [16] that introduced (i) semantics for *latency constraints* as a formalization of an application's latency requirements and (ii) techniques to optimize the execution of a data flow towards constraint fulfillment under the assumption of a *fixed but sufficient resource provisioning*. In this paper we drop this assumption and explore how resource elasticity can be employed towards the same goal. The contributions of this paper are as follows:

1) We experimentally motivate the use of elasticity to provide latency guarantees and the design choices for our solution.
2) We introduce a model to estimate the latency of a UDF-heavy data flow, when the data parallelism of the data flow changes.
3) To minimize resource consumption while maintaining latency guarantees, we introduce an automated strategy to adjust data parallelism at runtime.
4) We experimentally evaluate the effectiveness of the scaling strategy on a large commodity cluster with 130 worker nodes, using an application performing real-time sentiment analysis on real-world social media data.

The paper is organized as follows. Section II highlights common design principles of today's scalable stream processing engines and shows how latency constraints for this class of systems can be expressed. Section III experimentally motivates the use of elasticity to guarantee latency constraint fulfillment, while Section IV describes the details for our reactive scaling strategy. Section V experimentally evaluates our strategy with a synthetic as well as a real-world workload from the social media domain. Section VI summarizes related work and Section VII finalizes the paper with a brief conclusion.

II. Background

Today's scalable SPEs [1], [9], [15], [16] are geared at scalable and fault-tolerant execution of data flows in commodity cluster environments and follow similar design principles. They differ from the more traditional Data Stream

This paper has been accepted for publication at ICDCS'15. This version is a preprint identical in content to the version to be published.

Management Systems (DSMS) [19]–[21] and Complex Event Processing (CEP) systems in the following key aspects:

a) UDF-heavy Data flows: They focus on the highly parallel execution of data flows with embedded User-Defined Functions (UDFs) and generally do not (yet) provide a more specialized declarative language to formulate queries. Being inspired by MapReduce and its descendants, the data flows are usually directed graphs, as described in Subsection II-A1.

b) Master-Worker: The system architecture usually follows a master-worker pattern.

c) Producer-Consumer: The engine runtime on the worker nodes executes the tasks and transfers data between them, using a producer-consumer pattern, where individual data items are put into one or more send and receive queues.

A. Stream Processing Jobs with Latency Constraints

For the remainder of the paper, we assume a stream processing job to be a data flow that is modeled as a Directed Acyclic Graph (DAG), which is representative for many stream processing engines. In the following we provide a summary of the of the relevant concepts and formalisms already used in our previous work [16].

First, we formally introduce two representations of a stream processing job, the *job graph* and the *runtime graph*. Then we introduce the formal semantics for *latency* and *latency constraints*.

1) The Job Graph: The job graph is provided by the user and indicates to the master node which user code to run and with which degree of parallelism this should be done. It shall be defined as a DAG $JG = (JV, JE)$ that consists of *job vertices* $jv \in JV$ connected by directed *job edges* $je \in JE$. The user attaches a User-Defined Function (UDF) to each job vertex and can specify a current, minimum and maximum degree of parallelism $p_{jv}, p_{jv}^{min}, p_{jv}^{max} \in \mathbb{N}^+$. Usually one also has to specify a wiring pattern (also called "stream grouping" [9]) that specifies the communication pattern to use (e.g. key-partitioned, broadcast etc.) when connecting the tasks of two adjacent job vertices.

2) The Runtime Graph: The runtime graph is a parallelized version of the job graph to be used during the job's execution. It shall be defined as a DAG $G = (V, E)$ where V is a set of *tasks* and E is a set of *channels*. Each *task* $v \in V$ has a corresponding job vertex and will execute an instance of its UDF. A *channel* $e = (v_1, v_2) \in E$ is a communication channel along which a task v_1 can send *data items* of arbitrary size to a task v_2.

For simplicity of notation we shall sometimes treat a job vertex jv as a set of tasks $jv \subseteq V$ with $|jv| = p_{jv}$. Analogous, each job edge je can be treated as a set of channels $je \subseteq E$.

3) Task and Channel Latency: Let us assume a task $v \in V$ with an inbound channel e_{in} and an outbound channel e_{out}. We define the *task latency* $l_v(d, e_{in}, e_{out})$ incurred by data item d that is consumed from channel e_{in} by task v as either *Read-Ready* or *Read-Write task latency*.

Read-Ready (RR) task latency is the time difference between (1) data item d being consumed from channel e_{in} and (2) task v becoming ready to read the next data item from *any* of its input channels. This definition is a good match for tasks with UDFs that perform computation strictly per data item e.g. map- and filter-like UDFs.

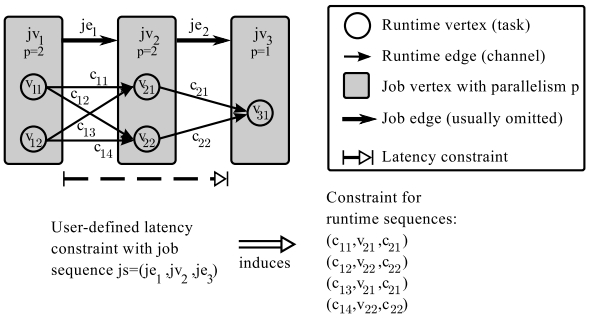

Fig. 1. Stream processing job example with and constraint.

Read-Write (RW) task latency is the time difference between (1) data item d being consumed from channel e_{in} and (2) the next time task v writes any data item to e_{out}. This definition is a good match for tasks with UDFs that perform computation based on more than one previously consumed data item, e.g. UDFs performing aggregation over windows.

We expect the UDF to specify, which one of the two options shall be used to measure its task latency, because UDFs and the computation they perform are opaque to the engine.

For a channel $e = (v_i, v_j) \in E$, we define the *channel latency* $l_e(d)$ as the time difference between (1) the data item d being emitted into the channel by v_i and (2) being consumed from the channel by v_j.

4) Sequences and Sequence Latency: A *job sequence* js shall be defined as an n-tuple of connected job vertices and job edges, where both the first and last element can be a job vertex or a job edge. Each job sequence induces a whole set of *runtime sequences* $\{s_1, \ldots, s_n\}$ within the runtime graph. Each data item d that enters the runtime sequence s_i incurs a latency before being fully processed. We refer to this latency as the data item's *sequence latency* $sl(d, s_i)$, measured in an arbitrary but fixed time unit. Sequence latency is defined as the sum of task and channel latencies of the sequence's tasks and channels. Figure 1 shows an example of the graphical notation we use for jobs and constraints.

5) Latency Constraints: Consider a stream processing job, where incoming data has to be processed as fast as possible, e.g. within a few milliseconds. If this job is to be deployed at scale in an unelastic SPE, one has to permanently provision the SPE with enough resources to withstand peak load (which may be unknown or unknowable). As shown later in Section III-C, given a fixed amount of computational resources, an increase in input load also increases latency due to queueing effects. Thus, provisioning for peak load (if known) already requires implicit knowledge about how much latency a given job can tolerate. A *latency constraint* explicitly declares this otherwise implicit requirement to the SPE. The SPE now has the opportunity to take automated measures to *guarantee* the constraint under varying load without permanent peak-load resource provisioning.

Formally, each job can be annotated with multiple *latency*

constraints. A constraint is a tuple (js, ℓ, t), that expresses a desired upper bound of ℓ time units on the mean sequence latency of all data items passing through all the runtime sequences $\{s_1, \ldots, s_n\}$ of js during any given time span of t time units. More formally, if D_t is the set of data items entering s_i during a time span of t time units, then

$$\frac{1}{|D_t|} \sum_{d \in D_t} sl(d, s_i) \leq \ell \quad (1)$$

must hold true, in order not to violate the constraint. Note that a constraint does not make a statement about the latency incurred by individual data items, which may well be lower or larger than ℓ. Instead it defines a desired a "statistical" upper bound for data items within a finite time interval t (e.g. 10 s). Given the lack of hard real-time capabilities in the commodity servers, networks and operating systems that we target as the execution platform, and the complexity of most real-world setups, we doubt that meaningful hard upper bounds can be enforced.

III. Motivation: Why use elasticity in latency constrained stream processing?

Today's SPEs are designed to run on large clusters of commodity hardware, which commonly also assumes Ethernet and TCP/IP (or UDP/IP) as networking protocols. It is well known that in such networks transmission latency and maximum throughput are linked via the amount of bytes that are sent per packet [22]. In [16], we have used this effect to our advantage, by *adaptively batching* data items, so that latency constraints are guaranteed without "choking" throughput. This approach works well, as long as compute resources are *sufficiently* provisioned. However, so far we have only considered a static resource provisioning. In this section, our goal is to motivate the *additional* exploitation of resource elasticity in order to also support workloads, where such an a-priori static resource provisioning is unknown or prohibitively expensive. For illustration, we will analyze the dynamic behavior of a very simple stream-processing job with *static resource provisioning* but varying computational load. We use this simple job to showcase the inherent trade-offs and difficulties in scalable real-time stream processing.

A. PrimeTester Job Description

1) Job Structure: The *PrimeTester* job (see Figure 2) is designed to produce a step-wise varying computational load, so that a steady-state can be observed at each step. The *Source* tasks produce random numbers at a rate that varies over time, and send them round-robin to *Prime Tester* tasks, that test them for probable primeness. The tested numbers are then again round-robin forwarded to the *Sink* tasks to collect the results. Testing for probable primeness is a compute intensive operation if done many times, thus by controlling the *Source* tasks' sending rate, we control computational load at the *Prime Tester* tasks. The job's runtime is split up into several *phases* and *phase steps*. During each step, all *Source* tasks send at the same constant rate. Each step lasts 60s and each phase has at least one step. The phases are as follows:

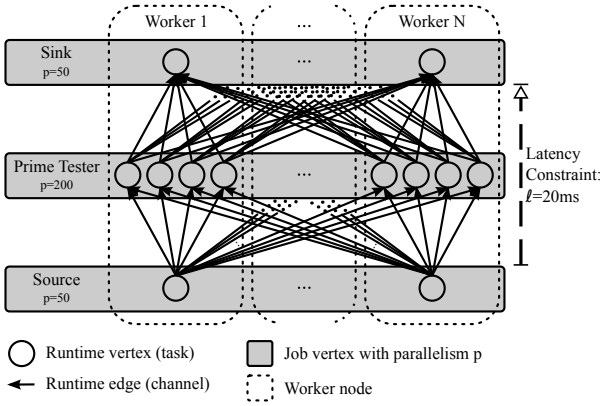

Fig. 2. DAG of *PrimeTester* Job

1) *Warm-Up:* Numbers are produced at a low rate. This phase has one step and serves as a baseline reference point.
2) *Increment*: Numbers are produced at step-wise increasing rates.
3) *Plateau*: Numbers are produced at the peak rate of the previous phase for the duration of one step.
4) *Decrement*: Numbers a produced at step-wise decreasing rates, until the warm-up rate is reached again.

B. Setup and Metrics

The job was implemented for Apache Storm [9] and our prototype of the Nephele engine [16], that can enforce latency constraints. Nephele is the streaming-capable execution engine of the Stratosphere[1] project (now Apache Flink[2]). Each run of the job was done on the same commodity cluster (see Appendix A) and a static resource provisioning of 50 worker nodes (200 CPU cores). The job consisted of 50 *Source*, 200 *Prime Tester* and 50 *Sink* tasks. It was run four times in total, with different *configurations*. We describe each configuration and classify its *optimization goals* (latency, throughput, or trade-off):

- *Storm* is Apache Storm `v0.9.2-incubating` with default settings and without guaranteed message processing. We chose Storm as a representative of systems optimized towards *low latency*, as it generally ships data items immediately and individually, i.e. unbatched.
- *Nephele-IF* is Nephele configured with instant flushing, thus it employs the same strategy as Storm and is meant to ensure comparability between Storm and Nephele, as these have different codebases.
- *Nephele-16KiB* is Nephele configured to use a fixed output buffer size of 16KiB on each channel. Each data item is serialized into the buffer, and the buffer is only sent when full. Thus, it is optimized towards *maximum throughput* by sending data items in large

[1] http://www.stratosphere.eu/
[2] http://flink.incubator.apache.org/

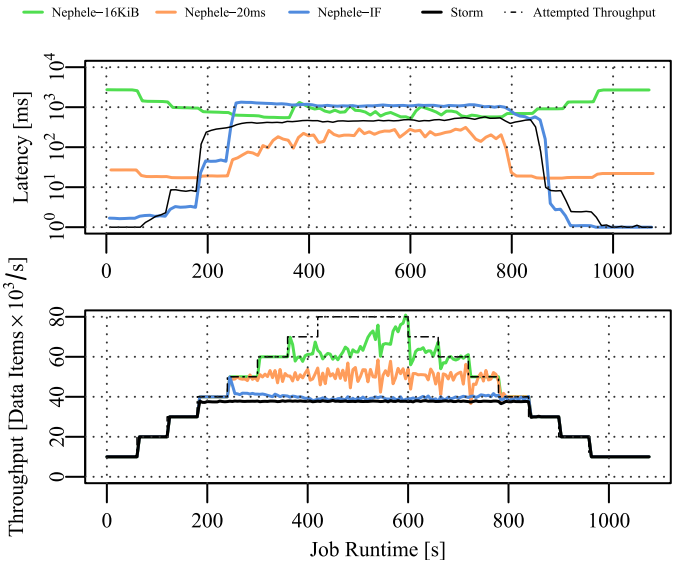

Fig. 3. Latency and Throughput of the Prime Tester Job on 50 worker nodes and 4 *PrimeTester* tasks per worker.

batches. Larger output buffer sizes than 16KiB had no measurable impact on throughput.

- *Nephele-20ms* is Nephele configured to enforce a 20 ms latency constraint between data items leaving the *Source* tasks and data items entering the *Sink* tasks. This configuration applies *adaptive output batching* as described in our previous work [23]. It *trades off* latency and maximum throughput, by batching as much as possible while still guaranteeing the 20 ms constraint.

For each configuration we were interested in the performance metrics *mean latency* and *mean throughput*. Mean latency measures the mean latency of all data items being processed within a 10 s period. A particular data item's latency is defined as the elapsed time between it being emitted at a *Source* task and it being consumed at a *Sink* task. To reduce the measurement overhead, we take a random sample of the data item latencies within each 10 s period and compute the mean over the sample. Mean throughput measures the mean rate at which the source tasks have emitted data items during a 10 s period. We further differentiate between *attempted* and *effective* mean throughput. In case of bottlenecks, a source task's *attempted* throughput, as dictated by the job's current phase step, will eventually be throttled down to the *effective* throughput due to backpressure. Backpressure is an effect that starts at bottleneck tasks and propagates backwards through the DAG via queues and TCP connections.

C. Prime Tester Job Results

Figure 3 shows the results in terms of the previously defined performance metrics.

In the *Warm-Up* phase, all configurations can keep up with the attempted throughput. The configurations with instant data shipping (Storm and Nephele-IF) provide the lowest latency of 1-2 ms or less on average. Nephele-20ms batches data items as necessary to guarantee the 20 ms latency constraint. Nephele-16KiB has a latency of almost 3 s, because the 16 KiB buffers on each channel take a long time to be filled. During the first steps of the *Increment* phase, each configuration has a step-wise increasing but otherwise steady queue waiting time between *Source* and *Prime Tester* tasks. With increasing attempted throughput, *Prime Testers* tasks eventually turn into bottlenecks and their input queues loose steady-state and grow until full. At this point backpressure throttles the *Source* tasks and the each configuration reaches its individual *effective throughput*, that is significantly lower than the maximum *attempted throughput*.

Looking at the latency measurements, one can see that the configurations with instant flushing (Storm and Nephele-IF) are the first to loose steady-state queue waiting time at roughly 180 s into the job. They are followed by Nephele-20ms two increment steps later at 300 s and by Nephele-16KiB at 360 s. Once bottlenecks have manifested, the latency of a configuration is mainly affected by how long the input queues can get. As resources such as RAM are finite, both Nephele and Storm limit the maximum input queue length. Also, it can be observed that increasing throughput leads to a *super-linear* increase in waiting time of data items at the input queues of *Prime Tester* (and *Sink*) tasks. This effect is present in every configuration, but *dominates* latency in configurations geared towards low latency. It is important to note for Nephele-20ms, that at the 240 s threshold, Nephele fails to guarantee the 20 ms constraint because adaptive batching cannot compensate for the increase in steady-state queue waiting time anymore.

When looking at the effective throughput measurements, one can see that the configurations with instant flushing peak ca. 40×10^3 data items per second. The configurations using batching show increased *effective throughput*, as shown by Nephele-20ms peaking at 52×10^3 data items per second (a 30% improvement), and Nephele-16KiB peaking at 63×10^3 data items per second (a 58% improvement). This is because batching reduces data shipping overhead in many places of the architecture (for example lower overhead for ISO/OSI transport layer headers, reduction of the number of system calls and hardware interrupts, etc.) This allows the task threads to spend more time on computation.

In conclusion, output batching has a measurable positive impact on maximum effective throughput, hence *adaptive batching* is an effective tool to enforce latency constraints up to a certain point. This point is reached when either the steady-state input queues have become so long, that the resulting waiting times make the constraint impossible to enforce, or when queues have lost steady-state and are growing due to bottlenecks. Thus, a strategy that guarantees latency constraints has to also address the issue of *queue waiting time*. Preventing bottlenecks is equally important, but rather mandatory than sufficient. Exploitation of elasticity offers an opportunity to control queue waiting time without resorting to load-shedding.

IV. A Strategy for Elastic Stream Processing with Latency Guarantees

This section presents a reactive strategy to enforce latency constraints under a varying load scenario without permanent resource provisioning for peak loads. Of the following subsections, Section IV-A presents the assumptions required for our

strategy to work. Section IV-B gives an overview of the system architecture, that enforces latency constraints and collects *measurement data* about the job for this purpose. Then we describe our strategy bottom-up, starting with Section IV-C, that uses the measurement data to build a *latency model*, that allows to reason about latency under varying degrees of parallelism. The two techniques *Rebalance* and *ResolveBottlenecks* presented in Section IV-D and Section IV-E use the latency model to find appropriate scaling actions. Section IV-F finalizes the description of the strategy, by showing when and how to employ *Rebalance* and *ResolveBottlenecks*.

A. Assumptions

For the remainder of the paper, we assume the following conditions to be true, in order to be able to effectively enforce latency constraints under varying load:

a) Homogeneous Worker Nodes: Worker nodes executing the same UDF must be sufficiently similar in their main performance properties, i.e. number of CPU cores, CPU core speed and NIC bandwidth. Worker nodes that are significantly slower than their peers will introduce *hot spot tasks* that lag behind other tasks executing the same UDF.

b) Effective Load-Balancing: Exploitation of *data parallelism* is common in today's scalable SPEs and requires a way of balancing the load of the data parallel tasks executing the same UDF. We consider a load-balancing *effective*, if it avoids *hot spot tasks* that incur a significantly higher computational load than other tasks executing the same UDF. In our setting, load-balancing is achieved via partitioning the data stream, so that each data item is assigned to one or more partitions. Simple partitioning strategies are round-robin and key-based partitioning. The latter may suffer from data skew, therefore special care needs to be taken to manage skew. Load balancing strategies are not in the scope of this work. For an example see [21], that presents a dynamic load-balancing approach for data parallel relational operators such as windowed joins and aggregates.

c) Elastically Scalable UDFs: We consider a UDF *elastically scalable*, when changing the number of data parallel tasks executing it does not impact the correctness of the result values. In particular, during a scaling action stream partitions need to be ad-hoc remapped to consumer tasks. For UDFs with round-robin stream partitioning, this is not an issue. However, UDFs expecting a key partitioning, e.g. grouped aggregations, usually depend on it to maintain application semantics. Again, we consider this out of the scope of this paper. Solutions for the domain of continuous SQL-style queries can be found in [21]. For stateful UDFs, [10], [12], [14] demonstrate solutions to the management of key-partitioned task state in the face of elastic scaling.

B. System Architecture

The first objective of the architecture is to continuously measure latency and other aspects of running tasks and their channels. This data is used to build a predictive latency model in Section IV-C. The second objective is to determine appropriate scaling actions to enforce latency constraints (using the latency model) and to and to execute these actions.

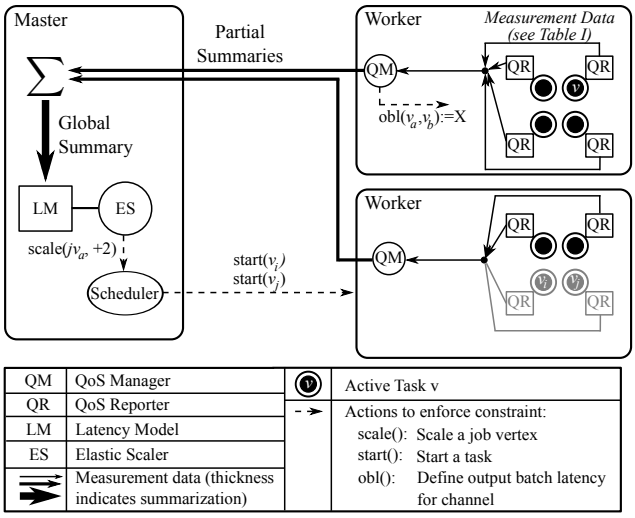

Fig. 4. System architecture for elastic scaling with latency guarantees.

Figure 4 provides an overview of the main architectural elements. The collection of measurement data is carried out by *QoS Reporters* and *QoS Managers* (see Section IV-C for details). QoS Reporters measure the basic task and channel performance metrics listed in Table I and report these to *QoS Managers*[3] once per *measurement interval* (e.g. once per second). Once per *adjustment interval* (e.g. once very 10 seconds), each QoS Manager scans the measurement data it has received for constraint violations and configures the *adaptive output batching* of channels as necessary to enforce the constraint. Adaptive output batching enforces an upper limit on the time data items wait before being shipped and has been discussed in our previous work [16]. Additionally, each QoS Manager computes a so-called *partial summary* of its collected measurement data and sends it to the master node. The master node aggregates the partial summaries into a *global summary*, and uses the global summary to initialize the *latency model* described in Section IV-C. Then the *Elastic Scaler* on the master node uses the initialized latency model to optimize the job's data parallelism so that latency constraints are guaranteed while minimizing the resource footprint. As a consequence, it may issue *scaling actions* to the *Scheduler*, that decides which tasks to start and stop on which worker nodes.

Accordingly, we have different degrees of summarization of the measurement data in Table I. For scalability reasons, each QoS Manager only collects measurement data for a subset of all latency constrained tasks and channels. Hence a QoS Manager's summary is *partial* and by itself not useful for controlling parallelism. The master node merges the partial summaries into a global one, that can be used to initialize the latency model.

Hence, latency constraints are enforced on two levels. First, by QoS Managers that control adaptive output batching, which is still the primary mean to enforce constraints, while resources are sufficiently provisioned. Second, the Elastic Scaler adjusts job vertex parallelism (i.e. the resource provisioning), to both

[3]Where to run QoS Managers and how to assign QoS Reporters to them has been discussed in our previous work [16].

TABLE I. PERFORMANCE MEASUREMENT DATA OF THE RUNTIME GRAPH.

Symbol	Description
Measured by random sampling	
l_v	Mean *task latency* (see Section II-A3) of the data items consumed by task v. Whether this is read-ready or read-write latency depends on what is requested by the task's UDF.
$\overline{S_v}$, $Var(S_v)$	Mean and variance of task v's *service time*. Service time is the queueing theoretic notion of how long a task is busy with a data item. Coincidentally, service time is equivalent to the notion of read-ready task latency. S_v is a random variable with an unknown probability distribution.
$\overline{A_v}$, $Var(A_v)$	Mean and variance of task v's data item *interarrival time*. A_v is a random variable with an unknown probability distribution.
l_e	Channel e's mean *channel latency* (see Section II-A3), i.e. the mean time between a data item being emitted producer-side and it being processed consumer-side.
obl_e	Channel e's mean *output batch latency*, i.e. the mean time data items wait due to batching before being actually shipped. Therefore $obl(e) < l(e)$ always holds.
Derived using above measurements.	
$c_X = \frac{\sqrt{Var(X)}}{\overline{X}}$	Coefficient of variation of $X \in \{S_v, A_v\}$.
$\lambda_v = \frac{1}{\overline{A_v}}$	Data item *arrival rate* at task v.
$\mu_v = \frac{1}{\overline{S_v}}$	*Service rate* of task v, i.e. its maximum processing rate.
$\rho_v = \lambda \overline{S_v}$	*Utilization* of task v.

enable adaptive output batching and exert control over queue waiting times, the necessity of which has already been motivated in Section III-C.

C. Latency Model

In this section we will explain the structure of the master node's latency model and how it relates to the raw measurement data obtained from QoS Reporters.

1) Measurement Data and Global/Partial Summaries: Let us assume a constrained job sequence js. Each summary, whether global or partial, contains tuples

$$\begin{aligned}(l_{jv}, \overline{S_{jv}}, c_{S_{jv}}, \overline{A_{jv}}, c_{A_{jv}}, \lambda_{jv}) \quad & \forall jv \in V(js) \\ (l_{je}, obl_{je}) \quad & \forall je \in E(js).\end{aligned}$$

where $V(js)$ and $E(js)$ are the job vertices and job edges within the job sequence js. These job vertex (job edge) values are averages over the measurement values of their respective tasks and channels. See Table I for an overview of what we measure for tasks and channels. For example, $\overline{S_{jv}}$ is the *average* service time of the currently running tasks in job vertex jv and defined as

$$\overline{S_{jv}} = \frac{1}{p_{jv}} \sum_{v \in jv} \left[\frac{1}{m} \sum_{t=1,\ldots,m} \overline{S_v^{(t)}} \right], \tag{2}$$

where p_{jv} is the parallelism of jv and $\overline{S_v^{(i)}}$ with $i = 1, \ldots, m$ are the past m service time measurements for the tasks $v \in jv$. The other values in the global summary are also averages over their task/channel counterparts. For reasons of brevity we omit their formulae as they are structurally identical to Equation 2.

All partial summaries are structurally identical to the global summary. However each QoS Manager has measurement data only for a subset of all constrained tasks and channels and computes its partial summary from that data. The master node merges the partial summaries to obtain the global summary defined above.

2) The Latency Model: We consider each task within a job vertex to be a single-server queueing system. Our assumptions of homogeneous worker nodes and effective load balancing minimize the differences between tasks of the same job vertex. Hence we can apply the available formulae from queueing theory to the values in the global summary. This enables us to estimate the queue waiting time for different degrees of parallelism of the job vertex. Specifically, we model each task to be a GI/G/1 queueing system, i.e. the probability distributions of data item interarrival and service times are generally unknown. For this case, Kingman's formula [24] approximates the actual queue waiting time of the *average* task in job vertex jv as

$$W_{jv}^K = \left(\frac{\rho_{jv}/\mu_{jv}}{1 - \rho_{jv}} \right) \left(\frac{c_{A_{jv}}^2 + c_{S_{jv}}^2}{2} \right), \tag{3}$$

where each input value is either from the global summary or derived using values from the global summary with the respective formulae from Table I. The *actual queue waiting time* of the tasks within job vertex jv is

$$\begin{aligned} W_{jv} &= l_{je} - obl_{je} \\ &= e_{jv} W_{jv}^K, \end{aligned}$$

where l_e and obl_e are the average channel and output buffer latency of the *ingoing* job edge je of jv in the job sequence js. The newly introduced coefficient e_{jv} is therefore

$$e_{jv} = \frac{l_{je} - obl_{je}}{W_{jv}^K}. \tag{4}$$

Less formally, we use the coefficient e_{jv} to "fit" Kingman's approximation to the last measurement data obtained for job vertex jv and job edge je. The core idea of the latency model is to predict W_{jv} when jv's degree of parallelism changes. In this context, e_{jv} ensures that we at least obtain the currently measured queue waiting time for the current degree of parallelism. Without e_{jv} the model might recommend a scale-down, when a scale-up would actually be necessary.

From our assumptions follows that changing the data parallelism of a job vertex, anti-proportionally changes the average data item arrival rate λ_{jv}. Hence the utilization ρ_{jv} becomes a function

$$\rho_{jv}(p^*_{jv}) = \lambda_{jv}\overline{S_{jv}}\frac{p_{jv}}{p^*_{jv}} \tag{5}$$

where $p^{min}_{jv} \leq p^*_{jv} \leq p^{max}_{jv}$ is a valid degree of parallelism for jv, and p_{jv} is the current degree of parallelism for which we have measurements in the global summary. Since the utilization ρ_{jv} is part of the Kingman formula, W_{jv} can be expressed as a function of p^*_{jv}, too.

Without loss of generality we assume all constrained job vertices $jv_1, \ldots, jv_n$ in js to be elastically scalable and by $p^*_1, \ldots, p^*_n$ we shall denote valid degrees of parallelism for them. The total queue waiting time in js can therefore be modeled as

$$\begin{aligned} W_{js}(p^*_1, \ldots, p^*_n) &= \sum_{i=1}^{n} W_i(p^*_i) \\ &= \sum_{i=1}^{n} e_i \left(\frac{\lambda_i \overline{S_i}^2 p_i}{p^*_i - \lambda_i \overline{S_i} p_i} \right) \left(\frac{c^2_{A_i} + c^2_{S_i}}{2} \right). \end{aligned}$$

We can now use W_{js} as a rough predictor of queue waiting time in job sequence js, when varying the degrees of parallelism of the job vertices within.

Note, that we assume the average service time $\overline{S_i}$ and its coefficient of variation c_{S_i} to be unaffected by the change in parallelism for each job vertex. We make the same assumption for c_{A_i}. One might argue that since by changing the data item interarrival time $\overline{A_i}$, its coefficient of variation c_{A_i} will also change. We will for now neglect this effect and reserve it for future work.

D. The Rebalance Technique

Using the latency model, the *Rebalance* technique chooses new degrees of parallelism for the constrained job vertices of a running job. The goal is to modify parallelism in such a way that resource consumption is minimized, while latency constraints are satisfied. It is applicable at any point in time where we have a fresh global summary and no bottlenecks exist, i.e. each constrained job vertex's utilization is sufficiently smaller than 1. See Section IV-E for a discussion of bottlenecks and their removal.

For a *single* latency constraint (js, ℓ, t) with job vertices $jv_1, \ldots, jv_n \in V(js)$, this problem can be formulated as an optimization problem, with a linear objective function and several side conditions. We must choose new degrees of parallelism $p^*_1, \ldots, p^*_n \in \mathbb{N}$ so that the *total parallelism*

$$F(p^*_1, \ldots, p^*_n) = \sum_{i=1}^{n} p^*_i \tag{6}$$

is minimized and the side conditions

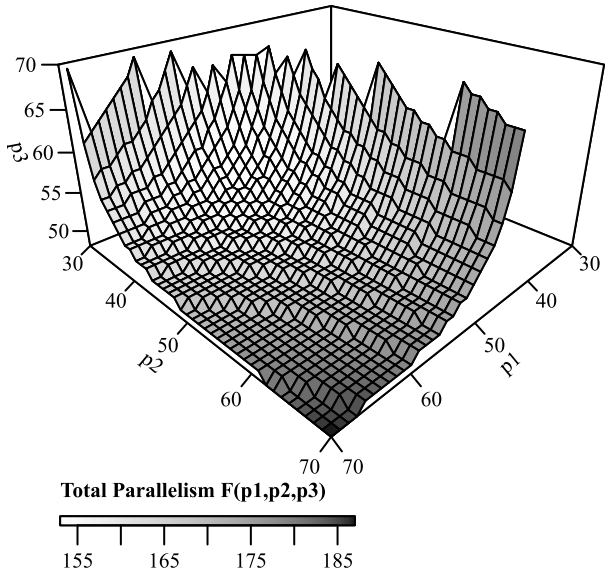

Fig. 5. Degrees of parallelism p_1, p_2, p_3 of three exemplary job vertices, so that p_3 is minimal for given p_1, p_2 while $W(p_1, p_2, p_3) \leq \hat{W}$.

$$W_{js}(p^*_1, \ldots, p^*_n) \leq \hat{W}_{js} \tag{7}$$

$$p^*_i \leq p^{max}_i, \forall i = 1, \ldots, n \tag{8}$$

$$p^*_i \geq p^{min}_i, \forall i = 1, \ldots, n \tag{9}$$

hold, where $\hat{W}_{js}$ is a chosen queue wait limit for js (see Section IV-F). Since each $W_i(p^*_i)$ is a monotonically decreasing function, the same holds true for $W_{js}(p^*_1, \ldots, p^*_n)$. The optimal solution is among a set of *solution candidates*, where decreasing any degree of parallelism p^*_i would violate one of the side conditions. For $n = 3$, an exemplary set of solution candidates is plotted as a 3D surface in Figure 5. The lighter the coloration of the surface, the lower the total parallelism $F(p^*_1, \ldots, p^*_n)$. As we can see from the example, *multiple optima* may exist. We may also have the case that the optimization problem cannot be solved due to the side conditions.

To solve the optimization problem for a *single* latency constraint, we propose to use the *Rebalance* technique described in Algorithm 1. It is given a constrained job sequence js and a maximum allowed queue waiting time $\hat{W}_{js}$ for vertices inside the sequence. Additionally, it is handed a set P_{min} that can contain a minimum required parallelism for vertices of the sequence. If we have to deal with *multiple* latency constraints with overlapping sequences on the same job graph, individual *Rebalance* invocations should be done for them. In this case, P_{min} ensures that the chosen degrees of parallelism of a $Rebalance(js_1, ...)$ invocation, are not overwritten with lower degrees of parallelism by another $Rebalance(js_2, ...)$ invocation (see Section IV-F).

Algorithm 1 implements a gradient descent with variable step size. It first determines whether it is actually possible to solve the optimization problem by testing if (7) can be fulfilled at maximum scale out. If so, it starts searching for a

Algorithm 1 Rebalance($js, \hat{W}_{js}, P_{min}$)

Require: A job sequence js, a total queue wait limit $\hat{W}_{js}$ and minimum degrees of parallelisms P_{min}.

$p_i \leftarrow p_{jv_i}^{max}$ for $\forall jv_i \in V(js)$
if $W_{js}(p_1, \ldots, p_n) \leq \hat{W}_{js}$ **then**
  $p_i \leftarrow p$ for $\forall (jv_i, p) \in P_{min}$
  while $W_{js}(p_1, \ldots, p_n) > \hat{W}_{js}$ **do**
    $C = \{i | i = 1, \ldots, n \text{ where } p_i < p_i^{max}\}$
    $\Delta_i \leftarrow W_i(p_i + 1) - W_i(p_i)$ for $\forall i \in C$
    $c_1 \leftarrow \min\{i \in C | \Delta_i = \min\{\Delta_j\}\}$
    if $|C| > 1$ **then**
      $c_2 \leftarrow \min\{i \in C | i \neq c_1, \Delta_i = \min\{\Delta_j | j \neq c_1\}\}$
      $p_{c_1} \leftarrow min(p_{c_1}^{max}, P_\Delta(c_1, \Delta_{c_2}))$
    else
      $w \leftarrow \hat{W}_{js} - W_{js}(p_1, \ldots, p_n) + W_{c_1}(p_{c_1})$
      $p_{c_1} \leftarrow P_W(c_1, w)$
    end if
  end while
end if
return $\{(jv_i, p_i) | jv_i \in V(js)\}$

lower cost solution, starting with the minimum scale out. For each iteration of the while loop, the general idea is to increase parallelism for the job vertex that yields the highest decrease in queue waiting time. More formally, a specific job vertex jv_{c_1} is chosen, so that increasing jv_{c_1}'s parallelism yields the highest decrease in queue waiting time Δ_{c_1}. If there is a jv_{c_2} yielding the second highest decrease in queue waiting time, then $P_\Delta(c_1, \Delta_{c_2})$ picks a new degree of parallelism for jv_{c_1} that is high enough so that jv_{c_2} will become the jv_{c_1} in the next iteration of the while loop. This is the "step-size", which is computable as

$$P_\Delta(i, \delta) = \left\lceil \frac{2b_i - 1}{2} + \sqrt{\left(\frac{1 - 2b_i}{2}\right)^2 - \frac{a_i + \delta(b_i^2 - b_i)}{\delta}} \right\rceil,$$

where $a_i = \lambda_i \overline{S_i}^2 p_i \left(\frac{c_{A_i}^2 + c_{S_i}^2}{2}\right)$ and $b_i = \lambda_i \overline{S_i} p_i$. Otherwise, if there is no jv_{c_2}, this means we are at the last iteration. In this case, $P_W(c_1, w)$ picks a new degree of parallelism for jv_{c_1} that is just high enough so that $W_{c_1}(p_{c_1}) \leq w$. This can be computed as

$$P_W(i, w) = \left\lceil \frac{a_i}{w} + b_i \right\rceil.$$

Implementations of $Rebalance$ can exploit the fact that all but one value of C and Δ_i can be reused between iterations of the while loop. This enables implementations to use standard data structures such as sorted sets, that offer add/remove/peek operations with $log(n)$ complexity. Hence, $Rebalance$ can be implemented with $O(n \log(n) m)$ complexity, where n is the number of job vertices in js and m is their highest degree of parallelism. In practice, due to the variable step size of the gradient descent method, we will require significantly less than m steps for each job vertex.

E. The ResolveBottlenecks Technique

The previously presented *Rebalance* technique is only applicable, if no bottlenecks exist. The goal of the *ResolveBottlenecks* technique is to resolve the bottleneck by scaling out, so that *Rebalance* becomes applicable again at a later point in time. It is applicable at any point in time, where we have a fresh global summary, at least one bottleneck exists and the bottleneck is resolvable by scaling out. The latter may not be the case for fully scaled out or non-elastic job vertices, or if no more resources are available for scale out. In either of those cases, the user needs be informed and take appropiate actions, e.g. make more cluster resources available.

Given a constrained job sequence js with vertices $jv_1, \ldots, jv_n$, the condition for jv_i to be a bottleneck is to have a utilization $\rho_i \geq \rho_{max}$, where ρ_{max} is a value close to 1. For each such vertex jv_i we will choose a new degree of parallelism

$$p_i^* = \min\{p_i^{max}, \max\{2p_i, 2\lambda_i p_i \overline{S_i}\}\}. \quad (10)$$

The general idea of *ResolveBottlenecks* is to be a last resort. A consumer-side bottleneck first causes input queue growth, that eventually leads to backpressure throttling producer side tasks. During input queue growth, a producer task emits data items at a rate higher than the consumer task's service rate. This leads to an artificially high consumer side utilization of ≥ 1, rendering the queueing formulae of the latency model unusable (see Equation 3). Once intermediate queues are full, backpressure sets in and forces the producer task to wait before sending. This manifests itself in an artificial increase of producer side service time. In both cases, *Rebalance* is either not applicable or will exhibit erratic scaling behavior, as accurate values in the global summary are not available. Hence, *ResolveBottlenecks* will at least double the bottlenecks' degree of parallelism, hopefully resolving them.

F. Putting it all together: ScaleReactively

Based on *Rebalance* and *ResolveBottlenecks*, we can now define a coherent strategy, that reacts to latency constraint violations with appropriate scaling actions, while keeping the total resource consumption low.

Algorithm 2 provides an overview of the strategy. For each constrained sequence js, $hasBottleneck(js)$ tests the bottleneck preconditions and applies the *ResolveBottlenecks* technique described in Section IV-E. Otherwise, the $Rebalance$ technique is used. P_{min} for $Rebalance$ is initialized in such a way, that later invocations of $Rebalance$ can only increase the degrees of parallelism chosen by earlier $Rebalance$ invocations. For simplicity, $\hat{W}_{js}$ is chosen as 20% of time available for data item shipping. The remaining 80% are reserved for adaptive batching, employing the techniques presented in our previous work [16]. After all constraints have been considered, the scaling actions resulting from the new degrees of parallelism are triggered with $doScale()$.

V. Experimental Evaluation

This section provides an experimental evaluation of our reactive scaling strategy based on our prototypical implementation inside the Nephele SPE. The engine was configured with

Algorithm 2 ScaleReactively(LC)

Require: A set of latency constraints LC

$P \leftarrow EmptyMap()$
for all $(js, \ell, \_) \in LC$ **do**
  if $hasBottleneck(js)$ **then**
    $P^* \leftarrow ResolveBottlenecks(js)$
  else
    $P_{min} \leftarrow \{(jv_i, \max(p_{jv}^{min}, P.jv)) | jv \in V(js)\}$
    $\hat{W}_{js} \leftarrow 0.2 \left(\ell - \sum_{jv \in V(js)} l_{jv}\right)$
    $P^* \leftarrow Rebalance(js, \hat{W}_{js}, P_{min})$
  end if
  $P.jv \leftarrow max(P.jv, p^*)$ for $\forall (jv, p^*) \in P^*$
end for
$doScale(P)$

a measurement interval of 1 s and an adjustment interval of 5 s, i.e. every 5 s a global summary was available on the master node. Nephele's scheduler interfaces with Nephele's own resource manager that leases and releases worker nodes as required. Using cluster resource managers such as YARN[4] or Mesos[5] is possible but left to future work. The same holds for IaaS clouds such as Amazon EC2, however rapid scale-ups may be hindered by VM startup times in the minute range, thus requiring a pool of preallocated VMs. Spinning up new tasks via Nephele's scheduler is a fast operation (1-2 s), and can be done in large batches. However, scale-ups need some time to have the desired effect on the measurement data. Furthermore, starting new tasks may initially worsen measured channel latency, because new TCP/IP connections need to be established. For these reasons, we added an inactivity phase of 2 adjustment intervals after scale-ups, e.g. the Elastic Scaler remains inactive for 10 s after starting new tasks. Scale-down operations take longer to be completed, because intermediate queues need to be drained and each task shutdown needs to signaled to and confirmed by upstream/downstream tasks. However, no inactivity phase is required afterwards.

We evaluated the scaling strategy using two jobs. First, the previously introduced *PrimeTester* job, and second, the *TwitterSentiment* job, that performs a sentiment analysis on social media data. Both jobs were run on the cluster described in Appendix A with the full pool of 130 workers available for elastic scaling. Section V-A describes the results of running the *PrimeTester* job with elastic scaling. Section V-B describes the *TwitterSentiment* job and its results.

A. PrimeTester Job Setup and Results

We ran the Nephele-20ms *PrimeTester* job (see Section III-B) with 32 Source tasks and an elastic number of *Prime Tester* tasks ranging from $p^{min} = 1$ to $p^{max} = 520$. As a baseline for comparison, we also ran the unelastic Nephele-16KiB *PrimeTester* job with 32 Source tasks and 175 *Prime Tester* tasks. The fixed parallelism of 175 tasks was manually determined to be as low as possible, while at the same time not leading to overload with backpressure at peak rates. Each configuration was run multiple times with similar results.

[4]http://hadoop.apache.org/
[5]http://mesos.apache.org/

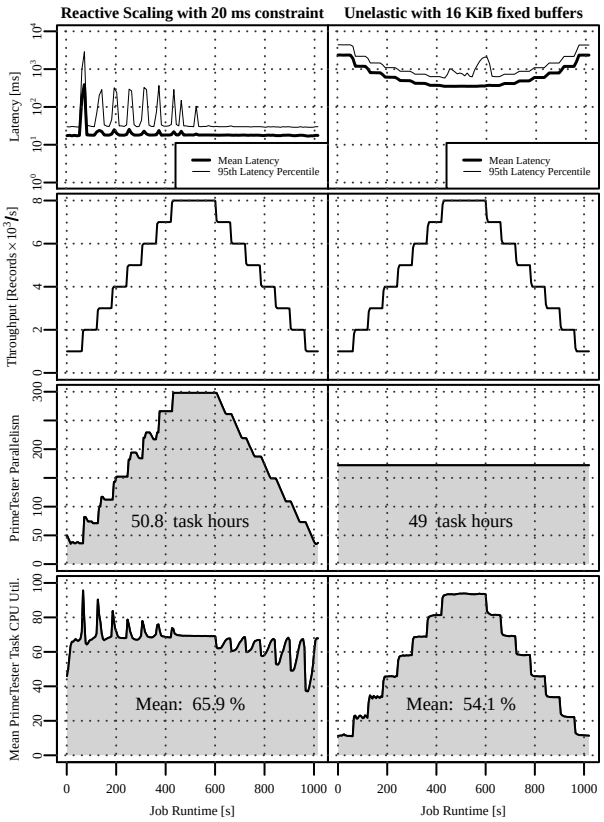

Fig. 6. Result of running the *PrimeTester* job with and without reactive scaling.

Figure 6 shows the results of both configurations. For the elastic Nephele-20ms, the constraint could be enforced ca. 91% of all adjustment intervals. There is one significant constraint violation when the Source task emission rate doubles from 10^4 to 20^4 data items per second, while transitioning from the *Warm-Up* to the *Increment* phase. The reason for this is that during the *Warmup* phase, the *PrimeTester* parallelism dropped to 36 tasks, which is a desired effect of the $Rebalance()$ operation, as it is designed to minimize resource consumption while still fulfilling the constraint. All other *Increase* phase steps resulted in less severe and much shorter constraint violations that were quickly addressed by scale-ups, because the relative increase in throughput decreases with each step. Since we have designed a *reactive* scaling strategy the constraint violations resulting from large changes in emission rate cannot be avoided. Most such scale-ups are slightly larger than necessary, as witnessed by subsequent scale-downs correcting the degree of parallelism. The overscaling behavior is caused by the facts that (1) the queueing formula we use is an approximation and most importantly (2) the *error coefficient* (see Equation 4) can become overly large when bursts of data items increase measured queue latency. However, we would argue that this behavior is useful, because it helps to resolve constraint violations quickly, albeit at the cost of temporary over-provisioning.

The 95th latency percentile is ca. 30 ms once scale-ups have resolved temporary queue build-up, and is naturally much more sensitive to changes in emission rate than the mean latency. Since most of the latency is caused by (deliberate)

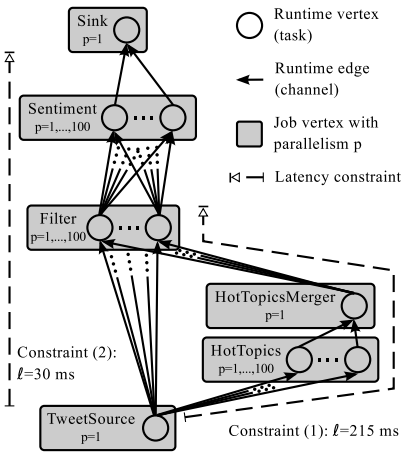

Fig. 7. Job and runtime graph structure of the *TwitterSentiment* job with elastic vertices.

output batching, 30 ms is within our expected range. Rerunning the job with reduced output batching improved the 95th latency percentile, but also resulted in configurations that were not able to reach the peak emission rate due to increased overhead in data shipping.

The unelastic but manually provisioned baseline system Nephele-16KiB is optimized towards maximum throughput and tuned to withstand peak load with minimal resource consumption. In consequence, both mean latency and the 95th latency percentile are much higher and do not go lower than 348 ms and 564 ms respectively. We measure resource consumption in "task hours", i.e. the amount of running tasks over time. Despite manual tuning, the amount of consumed task hours is almost equal to the elastic Nephele-20ms configuration. While the two configurations are at par in this respect, choosing a higher latency constraint of 30/40/50/100 ms for elastic scaling, yielded improved task hour consumption of 46.4/44.3/41.8/37.6, while still providing much lower latency.

B. TwitterSentiment Job and Results

The *TwitterSentiment* job processes tweets arriving from outside the data flow. The job identifies popular topics that people are tweeting about, and computes a sentiment[6] (positive, neutral, negative) for each tweet concerning a popular topic. As input we replay a 69 GB dataset of JSON-encoded, chronologically ordered English tweets from North America logged via the public Twitter Streaming API during a two-week period in August 2014. The rate of tweets is variant with significant daily highs and lows (see Throughput plot in Figure 8).

1) TwitterSentiment Job Description: Figure 7 gives an overview of the job's structure. It consists of six distinct job vertices, half of which are elastically scalable. The *TweetSource (TS)* task replays JSON-encoded tweets at the correct historic rates or a multiple thereof. We use it to replay two weeks worth of tweets within the 100 minute time frame of the experiment. Each tweet is forwarded twice by the *TS*. The first copy is sent round-robin to a *HotTopics (HT)* task, that extracts popular topics such as hashtags from the tweet. Each

[6]We use the LingPipe library: http://alias-i.com/lingpipe/

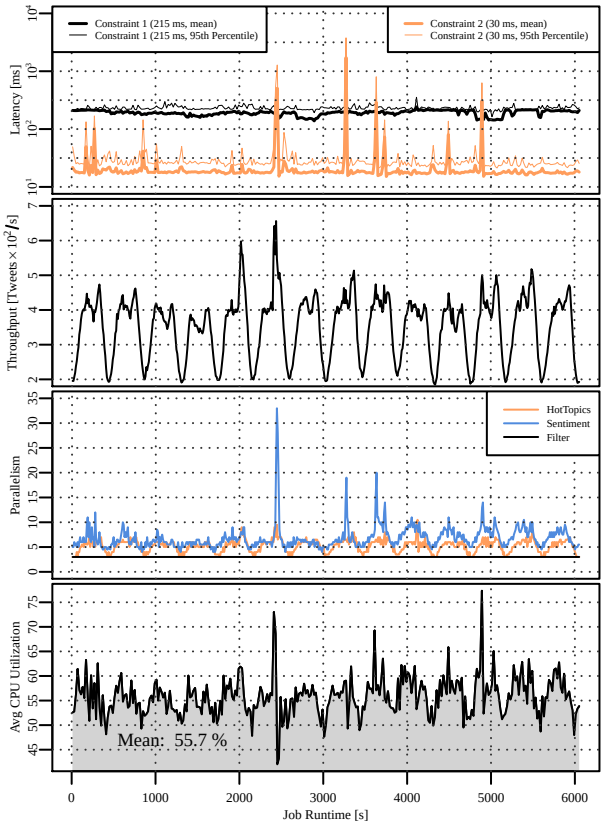

Fig. 8. Result of running the *TwitterSentiment* job with reactive scaling.

HT task maintains its own partial list of currently popular topics, sorted by popularity, and periodically forwards this list to the *HotTopicsMerger (HTM)* task. HT tasks perform time-based window aggregation with 200 ms windows. The HTM task merges all partial lists into a global one and broadcasts it to all *Filter (F)* tasks. We therefore define Constraint (1) with $\ell = 215\ ms$ for the job sequence $(e_4, HT, e_5, HTM, e_6, F)$. The TS sends the second copy of each tweet round-robin to a *Filter* task that matches it with its current list of popular topics. Only if the tweet concerns a currently popular topic, it is forwarded to a *Sentiment (S)* task, that attempts to determine the tweet's sentiment on the given topic. The result of the sentiment analysis is then forwarded to the *Sink (SI)* task, that tracks the overall sentiment on each popular topic. We define Constraint (2) with $\ell = 30\ ms$ for the job sequence (e_1, F, e_2, S, e_3). Three vertices (*F*, *S* and *HT*) are elastically scalable, each one with $p^{min} = 1$ and $p^{max} = 100$.

2) TwitterSentiment Job Results: Figure 6 shows the result of running the *TwitterSentiment* job. Constraint (1) with $\ell = 215\ ms$ was fulfilled in 93% of all adjustment intervals. The 95th latency percentile stays close to the constraint throughout the job, because the constraint leaves little room for output batching. The parallelism of the HT task is frequently adjusted to variations in the tweet rate. Due to two reasons both mean and 95th percentile latency on the constraint's sequence are relatively insensitive to tweet rate variations. First, the fixed task latency caused by the window aggregation dominates this sequence. Second, bursts in tweet rates are fully absorbed by HT tasks due to their time-based windowing behavior.

Constraint (2) with $\ell = 30\ ms$ was fulfilled 96% of all adjustment intervals, however there are significant spikes caused by tweet bursts. Outside of such bursts, the 95th latency percentile stays close to 25 ms. The tendency to slightly over-provision with respect to what would actually be necessary to fulfill the constraint can be observed in this benchmark as well. Here, the reason is the highly variant tweet rate, that often changes faster than the scaling strategy can adapt. Because scale-ups are fast and scale-downs take time, the system stays slightly over-provisioned. This is further evidenced by a mean task CPU utilization of 55.7%. Again, choosing a higher latency constraint would trade off utilization and resource consumption against latency.

The tweet rate varies heavily between day and night, peaking with 6734 tweets per second at around 2400 s into the job. Most notable about the peak is that its tweets seemed to affect one or very few topics, resulting in a significant load spike for the *Sentiment* vertex. The resulting violation of Constraint (2) was mitigated by a significant *Sentiment* scale-up with ca. 28 new tasks. Other vertices were not scaled up to the same extent, because their relative change in load was not as high.

VI. Related Work

Distributed stream processing has been the subject of vivid research over the past decade. We roughly categorize the existing systems along the criterion of streaming language.

A. Systems for Declarative Continuous Queries

Continuous processing of declarative queries was the focus of early distributed SPEs like Aurora* [19]. Aurora* exploits pipeline and to some extent data parallelism, but its scalability is limited by its load-balancing approach (box splitting), where a whole data stream has to pass through a single node. It selectively drops tuples ("load-shedding") if the system load gets too high. The use of load-shedding, is governed by user-specified QoS functions, that model the effect of dropped tuples, latency and other factors on QoS. Borealis [20] extends Aurora* and refines its strategies for load-shedding, where the effects on QoS can be computed at every point in the data flow. Finally, StreamCloud [21] extends Borealis by improving its query parallelization and load-balancing strategies, avoiding the limitations of box-splitting. StreamCloud can also exploit cloud elasticity and dynamically change the parallelism of query operators at runtime. Its elastic reconfiguration protocol takes special care of stateful operators with sliding-window semantics (e.g. a windowed aggregate). The goal of StreamCloud's scaling policy is to avoid bottlenecks by evenly balancing CPU load. The authors of FUGU [25] optimize operator placement towards a target *host* CPU utilization, by using reinforcement learning to control the operator migration at runtime. In [26] FUGU is extended to reduce the latency spikes caused by migration of stateful operators, towards fulfilling an end-to-end latency constraint. In this work, we do not look at the problem of operator/task placement, hence we consider it orthogonal to our approach.

B. Systems for UDF-heavy data flows

Scalable general-purpose SPEs focus on the execution of UDF-heavy data flows on large clusters or clouds and generally exploit pipeline, task and data parallelism. SEEP [14], Millwheel [10] and ChronoStream [12] are highlighted by their explicit management of key-partitioned UDF state for elastic and fault tolerant execution of stateful dataflows. SEEP checkpoints UDF state to upstream nodes, while Millwheel checkpoints it to a replicated highly-available datastore. ChronoStream replicates the state of each UDF to a peer node, i.e. a node that executes another instance of the same UDF. The scaling policies of SEEP and Millwheel prevent overload by scaling out when tasks cross a CPU utilization threshold. ChronoStream is highlighted by its support for vertical scaling (adjusts worker process threading level) and its lightweight UDF state reconstruction scheme, that exploits state replica locality and does not require state repartitioning. Timestream [11] is highlighted by tracking output and state dependencies to restore the state of LINQ operators during faults and scaling, but delegates the scaling policy to applications. In System S [27], which has been commercialized as IBM Infosphere Streams, applications are specified with the declarative SPL language that enables the heavy use of user-defined operators. In [28] a scaling policy and state management scheme for SPL applications is proposed. The scaling policy is driven by a control algorithm that adjusts data parallelism to prevent overload, by adapting to congestion and throughput measurements performed at runtime. Operator state is local to each stream partition the operator receives and a incremental migration protocol is proposed, that minimizes the amount of state transfer between hosts. Storm [9] is an elastic open-source SPE that provides at-least-once processing. Storm supports elastic scaling, but triggering scaling actions is up to users[7]. Recently, Storm has added the Trident API that checkpoints the state of aggregation-type operators for fault tolerance. The reactive scaling strategy of our prototype is orthogonal to the fault tolerance and state management protocols of these systems. Their scaling policies are designed to prevent overload/bottlenecks, conversely our policy is designed to minimize the violation of user-defined latency constraints, which includes bottleneck prevention.

Hadoop Online [29] and the Muppet system [18] (now mupd8) provide a MapReduce-like programming abstraction for streamed data. mupd8 is highlighted by persisting task state in a key-value store to mitigate the effects of its lossy failover strategy. The S4 [1] engine has been influential in the development of the current SPEs and is highlighted by its decentralized architecture. Spark Streaming [15] proposes the D-Streams programming model that unifies batch and stream computation by processing incoming data in MapReduce-style mini-batches. D-Streams is further highlighted by its fault tolerance based memory-resident *resilient distributed datasets*, that can efficiently recover data lost due to failures by recomputation. In [30] fixed-point iteration is used to adaptively minimize the end-to-end latency of the mini-batch model. Elf [31] proposes a novel design for a fault-tolerant stream processing engines with a DHT-based many-masters architecture to support running large numbers of data flows that perform batch or stream computation, can share data and whose functionality can be changed at runtime.

[7] http://storm.apache.org/documentation/Command-line-client.html

VII. Conclusion

We have presented an elastic strategy to enforce constraints over average latencies in scalable general-purpose stream processing engines, while minimizing the resource footprint. For this purpose we have introduced a queueing theoretic latency model, that suits itself well to optimization. Using the model, our strategy *minimizes* data parallelism, so that a given set of latency constraints is fulfilled. Frequent repetition of this process yields a system that enforces latency constraints by quickly adapting to changes in load. We have prototypically implemented our strategy in the Nephele engine. An experimental evaluation with a microbenchmark and a more complex job processing real-world social media data, showed good results, as our strategy enforced latency constraints as low as 20 ms over 90% of the time. In comparison to unelastic stream processing engines, no permanent peak load provisioning is required to consistently obtain low latency. Further, the ability to declare a application-specific latency constraint to an elastic stream processing engine, addresses the often tricky problems of resource provisioning and parallelization in stream processing. For future work we intend to focus on improving the prediction quality of our latency model and to improve efficiency by reducing the number of scaling actions.

Appendix

A. Description of commodity cluster used in experiments

The cluster consists of a master node and 130 worker nodes. Each worker node is equipped with a 2012 Intel Xeon E3-1230 V2 3.3 GHz processor (four physical CPU cores) and 16 GB RAM. All nodes worker nodes are connected via 1 GBit Ethernet in a single-switch star topology. Each node ran Gentoo Linux (kernel version 3.6.11) and Java 1.7.0.13.

References

[1] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *2010 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 2010.

[2] "Yahoo Talks Apache Storm: Real-Time Appeal," http://www.informationweek.com/big-data/big-data-analytics/yahoo-talks-apache-storm-real-time-appeal/d/d-id/1316840, Oct. 2014.

[3] "How Spotify Scales Apache Storm," https://labs.spotify.com/2015/01/05/how-spotify-scales-apache-storm/, Jan. 2015.

[4] J. Nielsen, *Usability engineering*. Elsevier, 1994.

[5] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, 2005.

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, 2008.

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proc. of the 2nd USENIX conference on Hot topics in cloud computing*. USENIX, 2010.

[8] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl *et al.*, "The stratosphere platform for big data analytics," *The VLDB Journal*, 2014.

[9] "Storm, distributed and fault-tolerant realtime computation," http://storm.apache.org/, Jan. 2015.

[10] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proc. of the VLDB Endowment*, vol. 6, no. 11, 2013.

[11] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in *Proc. of the 8th ACM European Conference on Computer Systems*. ACM, 2013.

[12] Y. Wu and K.-L. Tan, "Chronostream: Elastic stateful stream computation in the cloud," in *2015 IEEE 31st International Conference on Data Engineering (forthcoming)*, 2015.

[13] K.-U. Sattler and F. Beier, "Towards elastic stream processing: Patterns and infrastructure." in *BD3@ VLDB*, 2013.

[14] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013.

[15] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. of the 24th ACM Symposium on Operating Systems Principles*. ACM, 2013.

[16] B. Lohrmann, D. Warneke, and O. Kao, "Nephele streaming: stream processing under QoS constraints at scale," *Cluster computing*, vol. 17, no. 1, 2014.

[17] "Apache Flink: Home," http://flink.apache.org/, Jan. 2015.

[18] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan, "Muppet: Mapreduce-style processing of fast data," *Proc. of the VLDB Endowment*, vol. 5, no. 12, 2012.

[19] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, "Scalable distributed stream processing," in *Proc. of the 1st Biennial Conference on Innovative Data Systems Research*, 2003.

[20] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The design of the borealis stream processing engine," in *Second Biennial Conference on Innovative Data Systems Research*, 2005.

[21] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, 2012.

[22] J. Nagle, "Congestion control in ip/tcp internetworks," http://tools.ietf.org/html/rfc896, 1984.

[23] D. Warneke and O. Kao, "Exploiting dynamic resource allocation for efficient parallel data processing in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, Jun. 2011.

[24] J. F. C. Kingman, "The single server queue in heavy traffic," in *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 57, no. 04. Cambridge University Press, 1961.

[25] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE, 2014.

[26] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proc. of the 8th ACM International Conference on Distributed Event-Based Systems*. ACM, 2014.

[27] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, implementation, and evaluation of the linear road bnchmark on the stream processing core," in *Proc. of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006.

[28] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, 2014.

[29] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proc. of the 7th USENIX symposium on Networked systems design and implementation*. USENIX, 2010.

[30] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proc. of the ACM Symposium on Cloud Computing*. ACM, 2014.

[31] L. Hu, K. Schwan, H. Amur, and X. Chen, "Elf: efficient lightweight fast stream processing at scale," in *Proc. of the 2014 USENIX Annual Technical Conference*. USENIX, 2014.