Imperial College London
Department of Computing

Framework for Dynamic Scaling in Master-Slave Stream Processing Systems

by

Martin Rouaux

Supervisors:
Dr. Peter Pietzuch
Large-Scale Distributed Systems Group

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing for Industry of Imperial College London

November 2014

Abstract

Stream Processing Systems provide the ability to deal with today's large data volumes by modelling data as streams, processing and querying them on the fly without any previous storage required. In this context, workloads are typically unknown in advance and queries running continuously need to adapt in order to cope with constantly changing data arrival rates. The solution to these challenges is given by the system's ability to dynamically scale when needed.

We propose and describe the implementation of a generic framework to support dynamic auto-scaling in Stream Processing Systems architected following a master-slave paradigm. We define an abstract model for scaling policies and an embedded domain-specific language to facilitate the specification of scaling rules within policies. We also integrate our proposed framework with the SEEP system. Finally, we evaluate the framework by verifying its behaviour against a set of predefined simulated workloads and by running analytic queries against a real-world workload derived from a set of Google cluster traces.

Acknowledgements

To Dr. Peter Pietzuch and Raul Castro Fernández from the Large-Scale Distributed Systems Group for their support, expert advise and encouragement.

To my family for their support and constant encouragement, especially to my wife and daughter for their patience.

Table of Contents

List of Figures

List of Source Code and Algorithms

1 Introduction

1.1 Motivation

In recent years, the volume of data that needs to be processed has increased significantly. This is in part explained by the growth in the number of data sources (e.g.: mobile devices, web feeds, social networking websites, sensors and scientific instruments, etc). The challenges that this brings about are not only related to the storage of large data volumes but also querying them [1].

In this context, data might not be modelled best as persistent relations but rather as transient data streams. Traditional database systems are designed to run one-time queries over finite data sets that have been previously stored, producing precise results with stable query plans. Given that these systems are not designed for rapid and continuous loading of individual data items, it is simply not feasible to store the arriving data first to then run queries on it [2].

Stream Processing Systems address these challenges by processing and querying data streams on the fly without storage [1]. These systems run continuous queries on arriving data streams, possibly producing approximate results and adapting query plans to a workload that can change over time [2].

Stream Processing Systems that run in a distributed manner can partition continuous queries into smaller parts, executing them on different servers to exploit intra-query parallelism. Typically, in a scalable system, increasing the number of servers that run the various parts of the query will allow to system to cope with data arriving at higher rates. Conversely, workloads where data items arrive at a lower pace will require fewer servers. In such an environment, an important question is how many servers to provision and allocate for each part of the query.

One simple and common approach to answer this question consists in estimating peak load and provision for it, even if it is infrequent. There are some disadvantages to this approach, firstly that many or most of the provisioned resources might lie idle for long periods of time and secondly, it is difficult or outright impossible to predict peak rates for unknown workloads. Alternatively, we could provision for average use, which potentially wastes fewer resources than the previous approach. However, there are also some disadvantages in this case. Again, it is impossible to predict average rates given an unknown workload. Additionally, any above-average load will usually result in degraded system performance [3]. The ability to answer this question accurately becomes even more relevant in cloud-computing environments, where typically resources are provisioned on demand and paid for on a “pay-as-you-go” basis.

Dynamic auto-scaling can solve most of these problems, even in the presence of workloads that are initially unknown. The servers running the various query parts in a distributed environment can be instrumented to report variables that act as proxies for the traits that characterise the input workload. Hence, the Stream Processing System can make scaling

decisions based on these variables and provision more resources for those query parts that represent a bottleneck for the system throughput. Similarly, when idle resources are detected for a particular part of the query, these can be de-allocated in order to reduce the cost of running and operating the system.

Current commercial cloud-computing platforms offer general-purpose auto-scaling solutions. We believe that a similar solution can be implemented but tailored and customised to the particular use cases of Stream Processing Systems.

1.2 Project Objectives

Our main objectives for this project are:

- Create a generic framework that enables dynamic reactive scaling of continuously running queries on Stream Processing Systems that follow a master-slave architectural paradigm.
- Define a language for the specification of scaling rules for queries running on Stream Processing Systems, enabling query developers to easily test their queries with various scaling policies.
- Integrate the proposed scaling framework with an existing Stream Processing System, such as the SEEP system, and evaluate the framework's performance with simulated and real-world workloads.

1.3 Report Structure

This report is organised in the following chapters:

- In **Chapter 2**, we start with a brief description of the general characteristics that define Stream Processing Systems. Then, in particular, we cover in more detail the architecture of the SEEP System.
- In **Chapter 3**, we review some published literature where attempts are made to introduce dynamic scaling into Stream Processing Systems, focusing in particular on architectural choices and testing strategies. Additionally, we explore existing solutions for dynamic scaling of applications in cloud-computing environments, specifically those offered by major Infrastructure-as-a-Service and Platform-as-a-Service providers, such as Amazon Web Services and Google's Cloud Platform.
- We present the proposed framework in detail in **Chapter 4**. Following a top-down approach, we start by presenting the models assumed by our framework, both for Stream Processing Systems (SPS) and also scaling policies. In doing so, we provide a concrete definition of what we consider a scaling policy and its effect on stream queries. Then, we present the architecture of our proposed framework and its extension and integration points with Stream Processing Systems and, in particular,

with SEEP. Finally, and without going into too much detail, we discuss some technical implementation aspects of the solution and its effects for users writing stream queries.

- **Chapter 5** evaluates and discusses the different tests performed to evaluate our scaling framework, running on top of the SEEP System. On one hand, the first part of the chapter concentrates on well-defined tests using simulated streams of events to verify, in a controlled and predictable manner, whether the proposed solution fulfils its stated goal. In other words, the intention is to confirm that the framework can accurately decide when and how to scale in and out. On the other hand, the second part of the chapter takes a different approach and aims at testing the system with real-world data. In this part of the chapter, we focus on verifying that the scaling framework does not hamper the scalability of the SEEP system. Furthermore, we verify the ability of our framework to auto-scale SEEP dynamically in response to a variable real-world workload.
- Lastly, **Chapter 6** presents some conclusions of our work and hints some possible directions for further work on this project.

2 Background

In this chapter, we will provide some background information on Stream Processing Systems (SPS) or Data Stream Processing Systems (DSPS), clearly defining them and outlining the key traits and characteristics that set them apart from other types of systems and modes of data processing. Then, we will describe the architecture of the SEEP System in detail. This is a reliable fault-tolerant SPS capable of supporting stateless and stateful processing of data streams. SEEP is particularly relevant to our work because we will integrate our generic scaling framework with it, in order to run various scaling tests.

2.1 Stream Processing Systems

A data stream can be defined as a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety [4]. Data might come from a variety of sources (e.g.: sensors, computer programs, etc) that generate values at regular or irregular intervals. These are known as data sources and the collection of values that they produce conform data streams [5].

Traditional Database Management Systems (DBMS), which execute complex queries over data stored persistently, typically struggle to process data streams and are unable to easily support applications that require on-line analysis of rapidly changing streams of data. Time-series information can be stored in normal tables, albeit with explicitly defined timestamp attributes, or encoded as binary large objects to exploit data locality. The first approach is expensive because the stream spreads across a very large number of tuples, negatively affecting performance. Likewise, the second approach is inefficient when we attempt to query individual items in the data stream [5].

Data Stream Processing Systems (DSPS) address the limitations of traditional DBMS and enable the efficient processing of data streams by implementing a different processing query model for streams. In particular, DSPS do not assume that data streams are stored entirely in persistent storage nor do they guarantee the correctness of results, only providing approximate responses to queries. Furthermore, in a DSPS, queries on a data stream run continuously over a period of time and incrementally return new results as new data arrive. These are known as long-running, continuous, standing and persistent queries [4].

A DSPS or continuous query system breaks the traditional paradigm of a request-response style system in which a finite answer set is computed for each query. Instead, users register queries that specify their interests over unbounded data streams. Based on these queries, an engine continuously filters and synthesizes incoming data from the streams and delivers unbounded results back to the users [6].

Thus, the characteristics of data streams and continuous queries determine the requirements that DSPS must meet [4]. These are:

- The data model and query semantics must allow order-based and time-based operations. This essentially means support for sliding windows, where only a subset of the data items that arrived recently are taken into consideration by queries (e.g. queries over a five-minute sliding window).
- The inability to store a complete stream to persistent storage suggests the use of approximate summary structures, referred to in the literature as synopses or digests. As a result, queries over the summaries may not return exact answers.
- Streaming query plans may not use blocking operators that must consume the entire input before any results are produced. As stated before, new query results must be produced incrementally as new data arrives from data sources. For example, if the system is analysing alarms from sensors over a period of a month, only providing conclusions after a month might be too late to take any corrective actions.
- Due to performance and storage constraints, back-tracking over a data stream is not feasible. On-line stream algorithms are restricted to making only one pass over the data and it is not possible to look into past data beyond the scope of the current sliding window.
- Continuously running queries may encounter changes in system conditions throughout their execution lifetimes (e.g.: variable rates at which data items are streamed from sources). Shared execution of many continuous queries is needed to ensure scalability.

From an architectural point of view, it is possible to define an abstract architecture for a DSPS [1] [4]. Below we show a diagram of such an architecture and provide a brief description of the function fulfilled by each of its components:

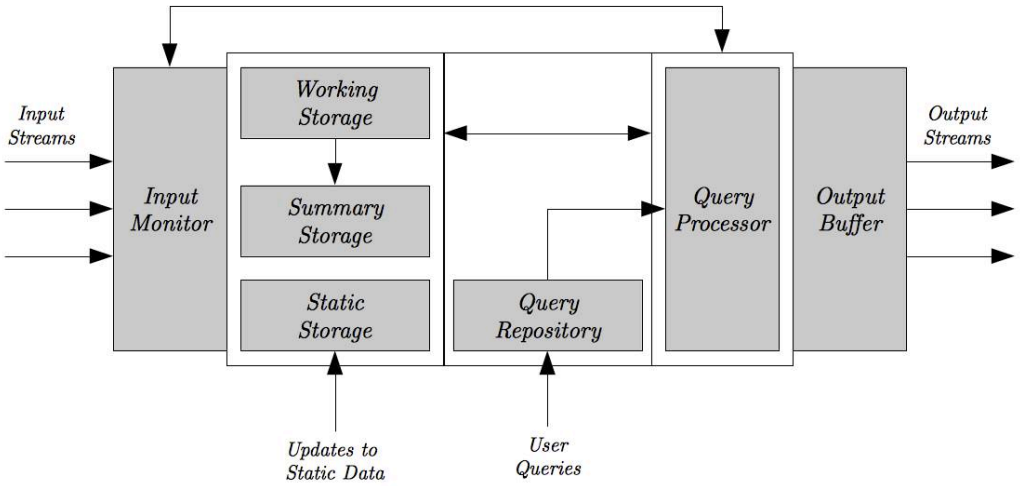

Figure 2.1: Abstract architecture for a Data Stream Processing System [4]

The components of a DSPS shown above are:

- ***Input Monitor***: regulates the input rates, possibly by dropping packets if the system is unable to keep up. The challenge in this case resides in deciding which items from the stream to drop. These can be selected at random or potentially, the system can apply semantic load-shedding, where packets to drop are chosen based on their meaning and impact on the query result [1].
- ***Working Storage, Summary Storage and Static Storage***: data partitions that provide different type of storage. In particular, these partitions provide temporary working storage (e.g.: for queries requiring a sliding window), summary storage for stream synopses and static storage for meta-data (e.g.: physical location of each source), respectively.
- ***Query Repository***: long-running queries are registered in this component and placed into groups for shared processing. Queries can be expressed using different types of stream query languages. Relation-based languages, such as CQL (Continuous Query Language), see the items in the data stream as relational tuples. The specification of a sliding window allows converting streams into relations, use relational algebra for queries on these relations and eventually convert results back to streams using special ad-hoc operators [7]. Other languages take a non-declarative more procedural approach, where queries are constructed by defining a workflow that specifies how data flows through various query operators [5].
- ***Query Processor***: communicates with the input monitor and may re-optimise the query plans in response to changing input rates. Query plans are given by operators, queues between operators, state of each operator (synopses or digests, plus the current sliding window) and base input streams. Storing the state of each operator is challenging as this might become quite large [1]. Additionally, this component is responsible for scheduling the execution of operators, minimising queue sizes and delays between operators, but maintaining quality-of-service guarantees [4].
- ***Output Buffer***: results from queries might be temporarily buffered or directly streamed to users.

So far we have provided an introduction to DSPS running locally on a single machine. In fact, systems such as Aurora [5] and STREAM [7] were initially designed to run on a single host. A distributed DSPS can be constructed by interconnecting multiple DSPS instances over a network, allowing for intra-query parallelism to be exploited. Such an architecture provides better scalability and can handle geographically distributed stream sources. Query plans in a distributed environment become more complex and these also need to consider operator placement in the network (e.g.: which operators will run where), resolve stream connections between hosts and allocate resources such as CPU and network bandwidth in each host [1]. Query optimisation also becomes significantly more complicated in a distributed environment. Some distributed DSPS prototypes rely on continuous monitoring

of processing sites to decide which sections of a query workflow to allocate to each site. Thus, statistics reflecting the site performance are passed to local, neighbourhood and global optimisers to make such allocation decisions [8]. Furthermore, a distributed DSPS also needs to cope with distributed and partial failures.

Some examples of distributed DSPS include the Borealis system [8], Apache S4 (a Java framework, inspired by the MapReduce model, that allows developing distributed, scalable and partially fault-tolerant applications for processing unbounded data streams while exposing a simple high-level simple programming model, relying on some well-known tools such as the Spring Framework) [9], Apache Storm (provides a set of general primitives to build real-time fault-tolerant stream processing systems, connecting spouts that produce streams with bolts that process those streams) [10] and the SEEP system, which we will describe in the detail in the next section.

2.2 The SEEP System

Particularly within the domain of distributed SPS systems, we are interested in scalable stream processing in the context of cloud-computing platforms. Such environments offer a virtually infinite pool of resources, where new hosts to run query operators can be procured quickly and at a low cost [1]. Thus, scalable SPS systems face two new challenges. Firstly, to benefit from the pricing model of cloud-computing, a SPS needs to be able to dynamically scale-out on demand, acquiring additional virtual machines and parallelising operators when the stream rate increases to take advantage of intra-query parallelism. Secondly, failures typically become more likely as deployments become larger, especially for cloud-hosted SPS. Therefore, SPS systems must be fault-tolerant with fast recovery times and low per-machine overheads [11].

The SEEP system addresses these two challenges for queries that contain stateful operators. These are operators that have large amounts of state which potentially depend on the complete history of previously processed tuples. Furthermore, the state of any operator in the query can be affected by both dynamic scale-out and failure recovery. SEEP adopts an integrated approach that tackles both problems simultaneously, by externalising internal operator state so that the SPS can perform explicit operator state management [11]. Thus, the SPS:

- Obtains access to operator state through a well-defined interface. This interface defines a set of primitives for state management, including checkpoint, backup, restore and partition operator state. These primitives allow SEEP to treat state generically and combine them to support more complex operations, such as scale-out and failure recovery.
- Maintains information about the precise set of tuples that have been processed by an operator and are thus reflected in its state
- Assumes a stream data model with a key attribute in order to be able to partition

state with correct semantics. Essentially, this means that the schema for tuples must always include a partitioning key which SEEP can use to partition the domain and automatically decide which tuple belong to each partition when dynamically scaling out stateful operators.

The state of an operator in SEEP is divided into ***processing state***, ***buffer state*** and ***routing state***. The processing state of an operator depends on input tuples and the history of past tuples seen by the operator. Typically, operators maintain a summary value representing past tuples, which constitutes its actual processing state. The buffer state of an operator is given by its output buffers, which are usually interposed between upstream and downstream operators. Finally, the routing state tells an upstream operator which downstream operator to send a tuple to based on the tuple's partitioning key. As SEEP is capable of dynamically scaling out, the downstream operators of a given upstream operator might change frequently at run-time. Therefore, this information needs to be part of the operator's state.

The operator state, as defined above, can be manipulated by SEEP using these primitive operations, which are described in [11]:

- ***Checkpoint***: the SPS obtains the processing state and the buffer state of an operator, along with the timestamp of the most recent tuples that affected both states. This primitive is executed asynchronously every checkpointing interval. More frequent checkpoints minimise the number of tuples that need to be replayed in order to bring the processing state in the event of a failure.
- ***Backup***: the operator state as returned by the checkpoint primitive is backed up to an upstream operator. Operators should balance the backup load across all of their partitioned upstream operators to avoid triggering their scale-out due simply to backup overhead. Additionally, the upstream operator can clean its buffer state of tuples as these have already been processed by the downstream operator.
- ***Restore***: backed up state is restored from an upstream operator to a downstream operator to either recover from a failure or to redistribute state across partitioned operators. Once the checkpoint state has been restored, the upstream operator needs to replay the tuples in its output buffer in order to ensure that the downstream state is up-to-date.
- ***Partition***: when a stateful operator scales out, its processing state must be split across the new partitioned operators. This is done by repartitioning the key space of the tuples processed by the operator. In addition, the routing state of its upstream operators must be updated to account for the new partitioned operators. Finally, the buffer state of the upstream operators is also partitioned to ensure that unprocessed tuples are dispatched to the correct partition.

The above primitives allow SEEP to perform dynamic scale-out and to provide failure recovery in an integrated manner. Moreover, periodically backed up state on upstream

nodes can be used by SEEP to recover from failures faster, thus offering a hybrid approach with a lower resource overhead than active fault-tolerance and faster recovery times than passive fault-tolerance [1].

SEEP can combine the state management primitives to offer fault-tolerant scale-out capabilities. Essentially, this consists in: (i) partitioning the last available backup upstream, (ii) creating new backups of the partitioned original backup to survive failure, (iii) create new downstream operator instances to replace the original downstream operator being scaled out, (iv) restore the partitioned backups and finally, (v) replay tuples in the output buffers of the upstream operator. Obviously, the routing state of the upstream operator would also need to be updated to reflect the new execution graph before resuming normal processing of the input data streams [11].

Lastly, we want to describe the architecture of SEEP deployed to a cloud environment, Amazon Web Services in particular. The diagram below shows how a query graph is submitted to a ***Query Manager***, which performs a mapping of query operators to virtual machines, to obtain an execution graph. The execution graph is used by a ***Deployment Manager*** to initialise virtual machines, deploy operators, set up stream communication and start stream processing. The ***VM Pool*** consists simply in a pool of pre-allocated cloud virtual machines. This pool masks typical delays observed when provisioning new virtual machines in cloud environments. The ***Scale-out Coordinator*** and ***Recovery Coordinator*** deal with dynamic on-demand scale-out of operators and recovery in the event of a failure, respectively. Bottleneck detection in SEEP is based on a simple scaling policy that accounts exclusively for CPU utilisation on virtual machines. When k consecutive reports from an operator are above a user-defined threshold δ for CPU utilisation, the ***Bottleneck Detector*** notifies the Scale-out Coordinator to parallelise the operator. SEEP focus is on compute bottlenecks as these are the most common type observed in practice [11].

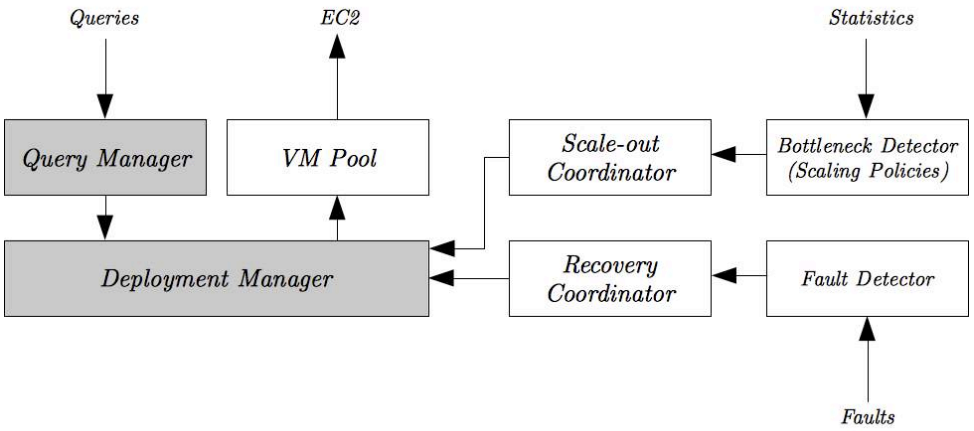

Figure 2.2: Architecture of the SEEP system in a cloud environment [11]

2.3 Summary

We have presented a definition of Data Stream Processing Systems (DSPS), introducing an abstract architecture for this type of systems and briefly describing each of its abstract components. We also mentioned some examples of SPS such as Aurora, STREAM, Borealis, Apache S4, Apache Storm and SEEP.

Particularly for SEEP, we described its architecture in more detail and its ability to provide fault-tolerant scale-out of stateful operators by means of upstream backups and checkpoints. SEEP is relevant as we have chosen to use this SPS to evaluate our solution.

3 Related Work

This chapter presents an overview of previous related work in the area of dynamic scaling in the context of SPS systems and cloud environments. In general, we can say the works presented here either fall into two categories in terms of bottleneck detection within queries:

i. systems where fixed pre-defined monitored variables act as triggers for scaling actions within the SPS or

ii. systems where users have the ability to specify which variables to monitor and when these should trigger scaling actions.

Furthermore, depending on the environment, provisioning additional resources for scaling can be time consuming. Thus, when additional resources do become available to be used within the SPS, the situation that caused the scaling in the first place might no longer exist (e.g.: a sudden peak in workload might disappear by the time extra servers are allocated). The works presented in this section cope with this lead time in mainly two manners:

(a) by pre-allocating resources and placing them in a pool from where these are provisioned upon scaling, or

(b) by attempting to predict the evolution of monitored variables and triggering the provisioning of additional resource sufficiently early (e.g.: if there is a lead time of 2 minutes, the system needs to be able to make predictions on events that might happen 2 minutes in the future).

All in all, the goal is to minimise the lead time for additional resources. The approach described in (a) usually implies reactive scaling (e.g.: additional resources are provisioned after a peak in workload occurs). Differently, the approach associated to (b) can be described a predictive scaling (e.g.: additional resources are provisioned before a peak in workload occurs).

Lastly, the final part of this chapter will be devoted to exploring existing solutions for dynamic scaling of applications in popular cloud-based computing environments. We are interested in these platforms as the scaling solutions they offer have been tested extensively in the field already and used on a daily basis by real-world applications with dynamic and constantly changing workloads.

3.1 Overview of Previous Works

In SEEP, it is assumed that bottlenecks within queries are typically compute-bound. Hence, the bottleneck detection mechanism is based on a simple scaling policy that accounts exclusively for CPU utilisation on servers. When a certain number of consecutive reports from a query operator are above a user-defined threshold for CPU utilisation, a

scale-out action is triggered. Additional servers come from a pool of pre-allocated cloud virtual machines, which masks lead times observed when provisioning new virtual machines in cloud environments [11]. According to the categorisation we defined above, SEEP would correspond to type (i) and (a).

The Borealis system supports a hierarchical model for bottleneck detection and resolution, based on various types of monitoring and optimization components. Local monitors run on each Borealis node and collect local statistics (e.g.: utilisation and queuing delays for various resources including CPU, disk, bandwidth, etc). These statistics are periodically forwarded to the site's end-point monitor. Monitors run continuously and trigger optimizers whenever they detect problems. There are three levels of collaborating optimizers that can act to resolve a bottleneck: (i) local optimizers, which temporarily shed data tuples on a node to cope with a bottleneck, (ii) neighbourhood optimizers, which can rebalance load across neighbouring sites, and finally (iii) the global optimizer, which can rebalance load and re-route tuples to any site with slack resources or, alternatively, shed data if no such sites are found [8]. Borealis is a reactive system in terms of bottleneck detection. However, resources are not allocated dynamically on-demand. In contrast, the system can dynamically re-organise the flow of streams through nodes and sites optimising resource utilisation and resolving any existing bottlenecks.

We will now describe three systems that correspond to the category (b) that we presented before: AGILE, PRESS and CloudScale. These systems attempt to predict resource utilisation patterns and scale-out early, thus masking any lead times and ensuring that additional resources will be already available when needed. The techniques proposed in each case are similar and aimed primarily at cloud environments, although still relevant in the context of SPS systems.

The AGILE system attempts to generate medium-term resource demand predictions. These, coupled with a resource pressure model, determine whether servers should be added or removed from the pool supporting a given service. The system follows a master-slave architecture, with slaves collecting monitor data. The master receives the data from slaves and maintains a dynamic resource pressure model (a model that determines the amount of resources required to meet an application's committed obligations in terms of performance).

Differently from other prediction schemes, AGILE doesn't focus on short-term predictions nor does it need to assume cyclic workload patterns. Arbitrary workload patterns are allowed, however different techniques are used by the master to predict the resource demand when facing cyclic and acyclic workloads. AGILE produces predictions for the former by means of Fourier transforms, whereas the later requires a prediction model that combines wavelets and Markovian models [12].

The PRESS system uses different approaches for resource utilisation predictions, depending on the nature of the input workload. Periodic and non-periodic workloads are treated using a hybrid approach that employs signature-driven and state-driven prediction algorithms to

achieve both high accuracy on near future predictions and low overhead. The model first employs a fast Fourier transform (FFT) to identify repeating patterns called signatures. Then, whenever a signature is discovered, the prediction model uses it to estimate future resource demands. Otherwise, the prediction model employs a discrete-time Markov chain to predict the resource demand in the near future. Underestimations from the model are particularly harmful, not only because they can violate service-level agreements but also because they effectively set a cap on resources available to the system, making it impossible to know the actual resource demand for the input workload. This makes future predictions less accurate. PRESS deals with this risk and attempts to reduce the chance of under-provisioning by padding (e.g.: increasing) the predicted value by a small amount, always allocating slightly more resources than what the model predicts [13].

Finally, CloudScale uses a prediction model similar to that described for PRESS, based on the Fast Fourier Transform for cyclic workloads and discrete-time Markov chains otherwise. However, differently from PRESS, CloudScale offers a richer set of techniques to correct underestimations in resource utilisation predictions. There are two types of error correction: (i) online adaptive padding, which dynamically adds a determined cushion value to the predicted resource demand, or (ii) reactive error correction, which detects and corrects underestimation errors that are not prevented by the padding scheme. The former constitutes a proactive approach whereas the later is more of a reactive nature [14].

3.2 Dynamic Scaling in the Cloud

We now present some existing solutions for dynamic scaling of applications in popular cloud-based computing environments. In particular, we intend to provide an overview of these solutions for dynamic scaling:

- Amazon Web Services
- Microsoft Windows Azure
- Google Cloud Platform

These are general purpose solutions in the sense that they are generic and intended to be used with a wide variety of systems, rather than an SPS only. However, we do believe that some aspects and architectural concepts present in these solutions are applicable to SPS type of systems and could be utilised in our scaling framework.

Additionally, these solutions are mostly reactive rather than predictive. None of the cloud-based computing environments described here offers a scaling solution that attempts to predict the evolution of monitored variables and trigger the early provisioning of additional resources. In contrast, these solutions simply let changes in workload reflect on monitored variables to then react accordingly, typically based on scaling rules defined al least partially by users.

3.2.1 Amazon Web Services

Amazon Web Services (AWS) is the cloud-computing platform offered by Amazon, which enables external users to requisition and consume scalable and elastic computational capabilities. The AWS platform offers various services, including compute power, storage, content delivery, caching, managed relational and NoSQL distributed databases, managed Hadoop clusters, load balancing, auto scaling, security, resource and application monitoring, amongst others [15]. In particular, we are interested in describing the Amazon Elastic Compute Cloud (EC2) service, the CloudWatch monitoring service and the Auto Scaling service, all of which integrate and cooperate to offer dynamic scaling for user applications.

The Amazon EC2 service provides scalable computing capacity, allowing users to launch as many virtual servers as needed. These virtual servers, typically referred to as instances, support various configurations of CPU, memory, storage, and networking capacity. Additionally, EC2 supports preconfigured templates to boot instances, known as Amazon Machine Images (AMI), which determine the operating system and the additional software to run on each virtual machine [16].

Amazon CloudWatch monitors AWS resources and user applications in real-time, in order to provide system-wide visibility into resource utilization, application performance and operational health. The service collects and tracks metrics at regular intervals, with metrics being the relevant system variables that measure utilisation and performance. Most AWS services expose predefined metrics, such as CPU utilisation and rate of I/O operations for EC2 instances, or HTTP status code response rate and latency for load balancing resources. Furthermore, user applications are allowed to define their own metrics and push them to the AWS CloudWatch service [17].

Lastly, the CloudWatch service can send notifications or automatically make changes to the resources being monitoring based on user defined rules. These rules constitute alarms, which are typically formed by the following elements [17]:

- ***Alarm Threshold***
    - ***Metric***: an alarm watches a single metric over a specified time period. A metric represents a time-ordered set of data points, which can either come directly from the user application or from predefined AWS system variables. Metrics are uniquely defined by a name (e.g.: "CPU Utilisation"), a namespace (e.g.: "Amazon EC2", an AWS service name) and one or more dimensions (e.g.: "i-12612e51", an EC2 instance name). Similarly, each data point has a time stamp and optionally, a unit of measure.
    - ***Value Threshold***: an alarm defines a threshold for its watched metric. The threshold is given by a value and a comparison operator. Usually, the resulting expression must evaluate to true for several data points in order for the state of the alarm to change.

- ***Number of Periods***: actions are invoked for sustained state changes only. The state of a CloudWatch alarm must have changed and been maintained for a specified number of periods (e.g.: "CPU Utilisation is above or equal to 50% for at least 3 consecutive periods").

- ***Alarm Actions***
  - ***State***: an alarm has three possible states: *OK*, when the metric is within the defined threshold; *ALARM*, the metric is outside of the defined threshold; and finally, *INSUFFICIENT_DATA*, when not enough data is available to determine the state of the alarm.
  - ***Action***: an alarm can send notifications to users or other AWS services when its state changes. Particularly, notifications can be sent to scaling policies defined in the Auto Scaling service. We will provide further detail later on but, in general, it is possible to say scaling policies define whether to requisition or release EC2 instances when a notification is received. Thus, the CloudWatch action (a notification) triggers a scaling action within the Auto Scaling service (use more or fewer EC2 instances). Alarm actions ensure that the CloudWatch and Auto Scaling services are loosely coupled.

The Auto Scaling service allows increasing or decreasing the number of EC2 instances allocated to an application. Collections of EC2 instances are grouped together in auto-scaling groups. The number of instances in a group can be set manually or via a scaling schedule (e.g.: when application load is predictable and known in advance, it is possible to tell the Auto Scaling service to scale up or down a group at certain times of the day). When the size of a scaling group changes, the service will either launch or terminate EC2 instances to match the desired size [18].

Additionally, the service allows specifying scaling policies and associate them to groups. A policy can be defined as set of instructions that tells the Auto Scaling service how to respond to messages from other Amazon services. These instructions will typically imply a change in the desired size of the associated scaling group and, as defined in [18], are typically given by the following parameters:

- ***Scaling Adjustment***: this is the number of instances by which to scale. A positive adjustment increases the current size of the group (i.e.: launch more EC2 instances) and a negative adjustment decreases the current size (i.e.: terminate existing EC2 instances).
- ***Adjustment Type:*** indicates how the adjustment should be interpreted by the service. There are three different types of adjustments supported by the service:
  - ***Change in Capacity***: the adjustment is interpreted as an absolute value (e.g.: if there are 5 instances currently in the group and the adjustment is 1, there will be 6 instances in the group after a scaling action).

- ***Exact Capacity***: the adjustment is interpreted as the actual desired size of the scaling group (e.g.: if the adjustment is 3, there will be exactly 3 instances in the group after a scaling action, no matter what the initial size of the group was).
- ***Percent Change in Capacity***: the adjustment is interpreted as a relative percentage value (e.g.: if there are 5 instances currently in the group and the adjustment is 40, there will be 7 instances in the group after a scaling action).

Lastly, as we mentioned before, the EC2 service, the CloudWatch service and the Auto Scaling service cooperate in order to offer dynamic scaling for user applications. In Cloud Watch, it is possible to define actions when a metric transitions to the *ALARM* state. An action can consist in sending messages to a scaling policy in the Auto Scaling service. Thus, a metric exceeding or being below a threshold for a sufficiently long period of time will trigger an alarm in Cloud Watch. The alarm will send a message to a policy that will eventually result in new EC2 instances starting or existing ones being terminated, depending on whether the policy is intended to scale up or down. In conclusion, this collaboration between services enables applications hosted on AWS to scale dynamically in response to a changing workload, reflected either directly or indirectly by a set of monitored system metrics. The diagram below illustrates how these services work together:

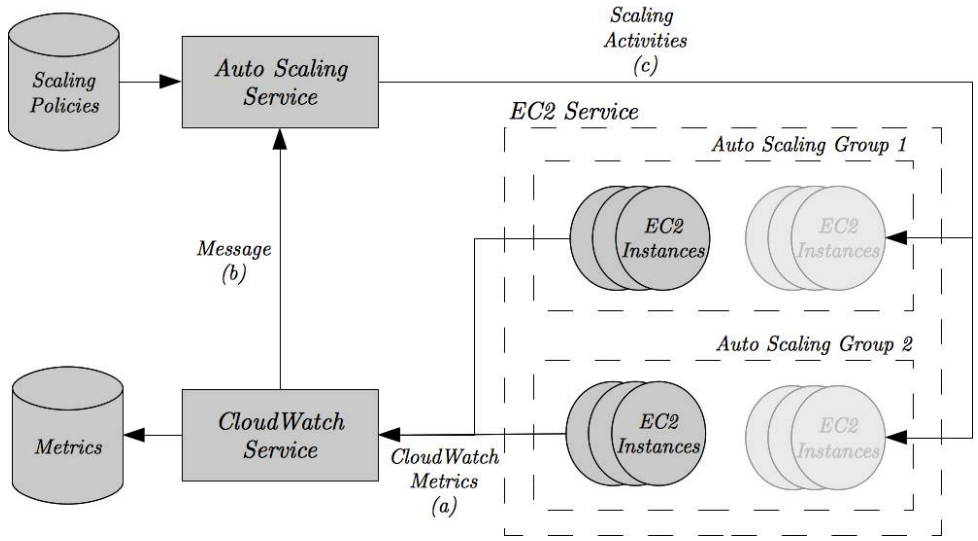

Figure 3.1: Dynamic scaling in Amazon Web Services

3.2.2 Microsoft Windows Azure

Windows Azure is the cloud-computing service offered by Microsoft which enables customers to run applications and store data on Internet-accessible machines owned by Microsoft. The service is formed by the following components: Compute, Storage, Fabric Controller, Content Delivery Network (CDN) and Connect [19]. In particular, we are interested in the Compute and Fabric Controller components, which are described in more detail below:

- ***Compute***: runs applications in the cloud. The applications largely see a Windows

environment not significantly different from a regular on-premises Windows installation, except for the programming model. Azure applications need to be implemented as one of these roles:

  - ***Web roles***: machines primarily intended to run web servers and facilitate the creation of web-based applications.
  - ***Worker roles***: designed to run a variety of Windows-based code. Instances fulfilling this role do not run a web server but can take work off-loaded by web role instances.
  - ***VM roles***: runs a user provided Windows Server image.

  One application can consist of multiple instances of each role. The cloud service provides built-in load balancing to spread requests evenly across them. Whenever an application is submitted to Azure for execution, it must specify how many instances of each role it requires. The service will the spin-off instances with the correct version of the operating system and additional software, depending on the role [19]. It is worth mentioning that none of the roles is restricted to the. .NET environment and that non-Microsoft languages, such as PHP or Java, are also supported.

- ***Fabric Controller***: the component is responsible for deploying, managing and monitoring all customer applications by means of distributed agents running on each machine. It handles all resources associated to an application, including load balancers. Additionally, it ensures that enough instances of each role are provisioned for a given application (e.g.: if two worker instances are required and one crashes, the Fabric Controller will ensure that a new one is started) [19].

We can already observe that Windows Azure imposes more restrictions on user applications than AWS. Where as the later doesn't impose any restrictions on applications running on EC2 instances, Azure makes the distinction between worker and web roles. Applications need to be architected and implemented with this in mind. A further difference can be seen in terms of support for dynamic scaling of applications. AWS supports dynamic scaling of applications by means of a set of collaborating built-in services. On the other hand, Windows Azure requires the use of an add-on library which, even if developed and maintained by Microsoft, is not part of the default Azure offering and must be configured and deployed by user. We will now describe this library in detail.

The Microsoft Enterprise Library consists of reusable software components that are designed to assist developers with common enterprise development challenges. It includes a collection of functional application blocks addressing specific cross-cutting concerns such as data access, caching, logging and validation. Thus, these application blocks help address the common problems that developers face from one project to the next [20].

In particular, the Autoscaling Application Block (WASABi) lets developers add automatic scaling behaviour to Windows Azure applications. The block can be hosted on Windows

Azure in its own worker role instance or in an on-premises application. WASABi can be used without modification; it provides all of the functionality needed to define and monitor autoscaling behaviour in a Windows Azure application [21].

WASABi can automatically scale a Windows Azure application based on rules. The scaling operations triggered by these rules will typically alter the number of role instances for the application. Increasing the number of role instances will help the application maintain its desired throughput in response to changes in its workload. WASABi also incorporates a stabilizer component, which is designed to minimize the number of scaling operations that the block performs by means of cool down periods (e.g.: time after a scaling operation during which no further scaling operations are permitted). This helps to prevent oscillations, giving application a chance to settle down with the new number of role instances.

As described in [21], there are two types of rules:

- ***Constraint Rules***: set upper and lower bounds on the number of role instances. Such rules can apply at any time or during specific time periods (e.g.: allow for a higher upper bound at certain times of the day where load is known to be higher).
- ***Reactive Rules***: change the number of instances in response to unpredictable changes in the input workload. The rules can monitor the application's workload using standard and custom Windows performance counters, in addition to Windows Azure queue lengths. Reactive rules are forbidden from changing the number of role instances unless there is a constraint rule that applies at the same time.

Reactive rules are more interesting in the context of dynamic scaling. We will describe its composing elements in detail and show an example below. Each reactive rule is formed by:

- ***When Element***: defines the boolean expression that is evaluated to determine whether the rule's action should be executed or not. It is possible to nest elements recursively to construct more complex expressions, combining them with logical AND or OR operators. Each element has two attributes: an operand, which is usually an alias for a performance counter, and a value to compare against.
- ***Actions Element***: defines the scaling actions that are performed when the expression in the rule's when element evaluates to true. Each element has two attributes: the target, which is usually the name of a role, and a percentage or absolute scaling value.

WASABi allows defining rules in XML and store them either locally (i.e.: on the same server where the application block is running) or using the Windows Azure storage service for unstructured data. Furthermore, it is also possible to define rules programatically and store them directly in source code [21]. An example of a rule expressed in XML is shown in source code example 4.1 below.

Finally, the scaling application block provides for application throttling rules. Essentially,

this consists in a reactive rule that specifies an action that changes the application configuration dynamically. The idea is to turn on or off application features depending on the workload. If the application is struggling to cope with the current workload, certain resource-intensive features can be disabled (i.e.: enable load-shedding for an SPS or reduce the bit rate of transcoded media stream, etc).

```
<rules xmlns="http://schemas.microsoft.com/practices/2011/entlib/autoscaling/rules"
    <constraintRules>
        <rule name="Default" description="Always active" enabled="true" rank="1">
            <actions>
                <range min="1" max="10" target="WebRole"/>
            </actions>
        </rule>
    </constraintRules>

    <reactiveRules>
        <rule name="ScaleUp" rank="10">
            <when>
                <greater operand="CPU_RoleWeb" than="60"/>
            </when>
            <actions>
                <scale target="WebRole" by="25%"/>
            </actions>
        </rule>

        <operands>
            <performanceCounter alias="CPU_RoleWeb" source="WebRole"
                performanceCounterName="\Processor(_Total)\% Processor Time"
                timespan="00:60:00" aggregate="Average"/>
        </operands>
    </reactiveRules>
</rules>
```

Source Code 3.1: WASABi scaling rule for Windows Azure application (in XML)

3.2.3 Google Cloud Platform

Google Cloud Platform is a collection of cloud-computing services offered by Google. These services are hosted on the same infrastructure that the company uses to run its most popular sites, such as its own search engine, Gmail and Youtube. Amongst some of the services offered by Google, we can mention App Engine, Compute Engine, Cloud SQL, Cloud Storage, Cloud Datastore, BigQuery, Hadoop and Cloud DNS [3].

With respect to support for dynamic scaling, the App Engine and Compute Engine services are of interest to us. App Engine is Google's Platform-as-a-Service (PaaS) offering. App Engine applications are easy to build, easy to maintain and easy to scale as traffic and data storage needs change [3]. Since the service operates a fully managed environment, there are no servers or base software for application developers to maintain. A developer simple needs to load his application to the service to get it running.

On the other hand, Compute Engine is Google's Infrastructure-as-a-Service (IaaS) offering. The service provides virtual machines that run on Google's common computational infrastructure. The service offers scale, performance and value that allow launching large compute clusters easily [3]. Similarly to the EC2 service in AWS, this service offers a non-managed environment. Hence, application developers are responsible for configuring and maintaining virtual machines. This obviously gives more flexibility at the expense of

additional operational effort.

The Google Cloud Platform does not offer a built-in service to handle dynamic scaling of compute resources. In other words, there is no built-in support for scaling App Engine or Compute Engine virtual machines based on the application workload. However, similarly to Windows Azure with WASABi, there are add-on tools that can be used to achieve dynamic scaling. In particular, Google's Orchestrator and Status Publisher tools [22] can be used to dynamic scale in and out any App Engine application.

The Orchestrator tool is implemented as a regular App Engine application. It periodically queries all running Compute Engine instances for their statuses. Depending on the type of application, the status may be CPU load and memory usage or the number of pending tasks. Then, based on user-defined criteria and the current load, the Orchestrator starts up new instances or terminates existing ones. In addition, each Compute Engine instance runs a local agent that exposes an endpoint for querying current CPU and memory load, which the Orchestrator periodically queries to find out the state of all instances. This local agent is the Status Publisher tool we mentioned before [22]. The user-defined criteria that determines when to scale is usually a set of heuristics stored in an XML file that the Orchestrator reads when starting.

The following diagram illustrates the architecture of the solution:

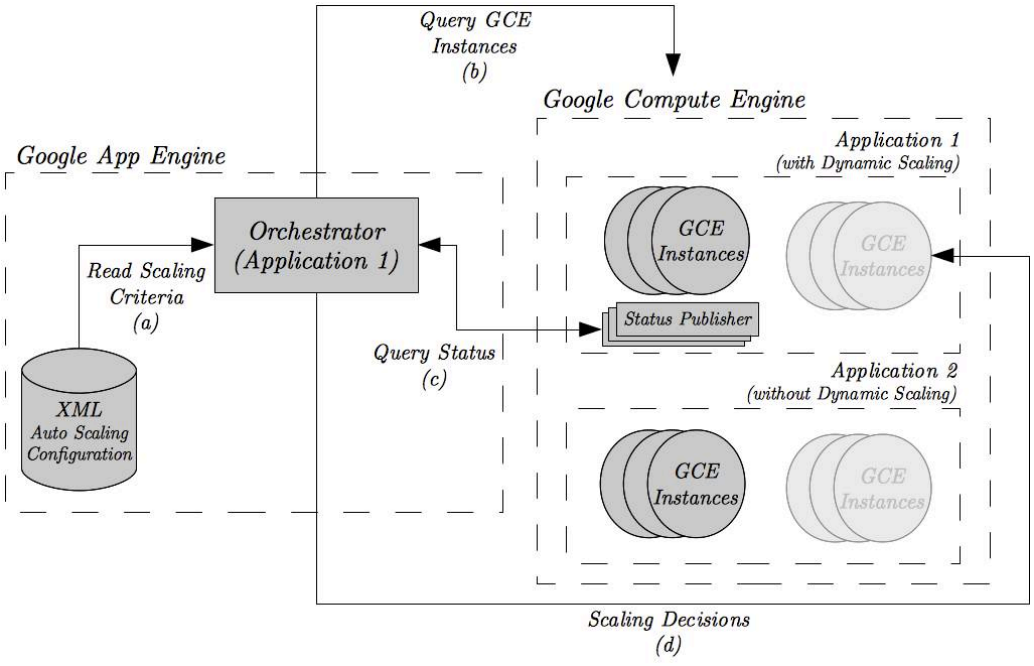

Figure 3.2: Orchestration of a Google Compute Engine application for auto-scaling

The default implementation of the Orchestrator tool supports, amongst others, the following parameters to determine when to scale in or put Compute Engine instances:

- ***maximum-ave-cpu-load***: an integer between 0 and 100 representing a percentage. When the average CPU utilisation of all running instances exceeds this threshold,

one or more new instances are created.

- ***minimum-ave-cpu-load***: an integer between 0 and 100 representing a percentage. If the minimum average CPU utilisation for all instances is lower than this value, the Orchestrator determines one or more instances to be shut down.
- ***maximum-ave-memory-usage***: similar to *maximum-ave-cpu-load*, but for memory usage.
- ***minimum-ave-memory-usage***: similar to *minimum-ave-cpu-load*, but for memory usage.
- ***num-instances-to-create***: an integer greater than 1. By setting this value to a number greater than 1, more than one instance can be spun up when a scaling-out decision is made.
- ***num-instances-to-shut-down***: similar to *num-instances-to-create*, but for shutting down instances.

As can be seen, the default implementation of the Orchestrator is quite simple and it only implements a basic set of CPU and memory based rules for scaling. The expectation is for application developers to extend the implementation of the Orchestrator and customise it with the heuristics that are deemed appropriate for each application [22]. In this sense, the solution offered by Google Cloud Platform can be better described as a pattern for dynamic scaling rather than a generic solution applicable to a vast variety of user applications, as it the case for AWS and Windows Azure. In this case, Compute Engine developers are responsible for providing a concrete implementation of the pattern that addresses their dynamic scaling requirements.

Furthermore, the default implementation does not provide any cool down or stabilising mechanism. Potentially, this is dangerous as the Orchestrator can generate oscillations where instances are spun up and terminated immediately after. This behaviour can have an impact in terms of end-user experience and cost. Finally, the default implementation only offers absolute scaling (i.e.: heuristics can only specify an absolute number of instances to start or stop). There is no support for relative scaling based on current number of instances or support for bounds on the size of an application, similar to what constraint rules achieve in Windows Azure WASABi.

3.3 Summary

In this chapter, we presented the reactive scaling solutions implemented by the SEEP and Borealis SPS systems. We also presented previous works describing systems that address the issue of predictive scaling, such as AGILE, PRESS and CloudScale. Even though these are mainly targeted at generic applications running on cloud environments, some concepts are applicable in the context of DSPS.

We also described existing commercial solutions for dynamic scaling of applications in popular cloud-based computing environments. We presented the architecture that supports dynamic scaling in Amazon Web Services, Microsoft Windows Azure and the Google Cloud Platform. We highlighted the aspects of each solution that we consider more relevant to our work and our proposed solution.

4 Framework for Dynamic Scaling

In this chapter we present our framework for dynamic scaling on Stream Processing Systems. First, we present an abstract model for such systems that clearly states the assumptions made by our framework in terms of system architecture. Secondly, we present a model for describing scaling policies and how these apply to queries executed by Stream Processing Systems. Subsequently, we present the framework architecture and its integration points, covering in detail the integration with SEEP. Finally, we cover some implementation details, focusing primarily on the API that enables query designers to quickly and effortlessly define scaling policies for their queries.

Our framework builds on the dynamic scaling facilities available in Amazon Web Services and Microsoft Windows Azure cloud platforms. Our model of dynamic scaling policies is the result of combining aspects of AWS policies and WASABi reactive rules, but in both cases adapting them to work in the context of an DSPS.

4.1 Model of Stream Processing Systems

Our intention is to design and develop a generic framework for the definition of scaling policies for queries running on SPS systems. Hence, given our intent of achieving a generic solution, it is necessary to make certain assumptions in terms of the architecture and the processing model for queries of the underlying system. As a consequence, the framework is designed to operate against an abstract SPS system. In this section, we present the model we adopted for the underlying abstract SPS while developing the proposed scaling framework.

4.1.1 Master-Slave architecture

Our model assumes that the SPS follows a master-slave architecture, the functions executed by these two types of processes differ significantly:

- The **Master** process acts as a coordinator, periodically sending and receiving control messages from the slaves. Additionally, it orchestrates the execution of the different data transformation operations that need to be applied to the stream of events, deciding which operations are executed where. Lastly, there is only one instance of the master process for the entire SPS at a logical level. This does not mean that multiple physical instances can exist to provide fault tolerance, however these instances need to appear as one to the rest of the system.
- There are multiple **Slave** processes that execute data transformation operations, exchanging data with other peers. The stream of events flows through the slaves, but never through the master. Depending on the nature of the query and the data transformation operations, the stream of events can be partitioned by the SPS and

routed to multiple slaves running different instances of the same operation. Furthermore, slaves periodically send control messages to the master, reporting their current state. Hence, the master is always aware of the state of all the slaves.

The following diagram depicts the architecture described above, assuming the SPS is executing a query formed by the operations $\{OP_1, OP_2, OP_3,\}$:

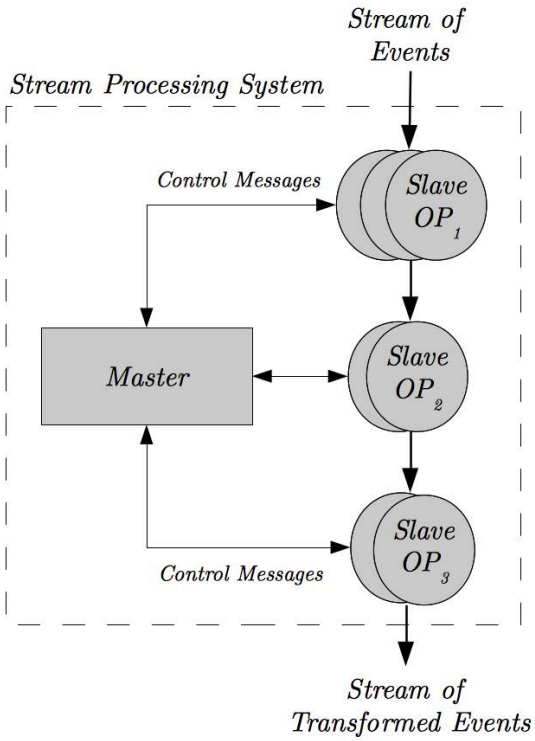

Figure 4.1: Architecture of an abstract SPS

Our abstract model does not make any assumptions in terms of how slave processes are implemented. However, in practice, is quite likely that slave processes will run on different physical hosts, with the connecting arrows above becoming real network paths (note we use the term path here instead of connection, mainly because we don't make any assumptions either as to whether slaves communicate via TCP segments or UDP datagrams).

4.1.2 Logical Queries vs. Physical Queries

Our model of an abstract SPS assumes that the data transformations to apply to the incoming stream of events form a **Logical Query**. Each data transformation operation applied to the stream of events is referred to as an **Operator**. Equivalently, an operator is a user defined function (UDF) that transforms an input n-tuple $(i_0, i_1,\ldots i_{n-1})$ into an output m-tuple $(o_0, o_1,\ldots o_{m-1})$. Thus, a logical query is then a sequence of operators connected to each other, where the output of an upstream operator is the input of the next downstream operator, with matching cardinalities for inputs and outputs. Logical queries are static and cannot change over time.

Similarly, we define a **Physical Query** as a one-to-many mapping of a logical query into a set of slave processes. Multiple instances of a logical operator can exist in the physical query, each running on a different slave process. The underlying SPS is responsible for connecting and routing tuples correctly from one slave to the next, preserving the connections that exist in the logical query. In contrast with logical queries, physical ones are dynamic. So, the one-to-N mapping of operators to slaves can change at any time to one-to-M, with N different from M and $M \geq 1$ (i.e.: all logical operators are always mapped to at least one slave).

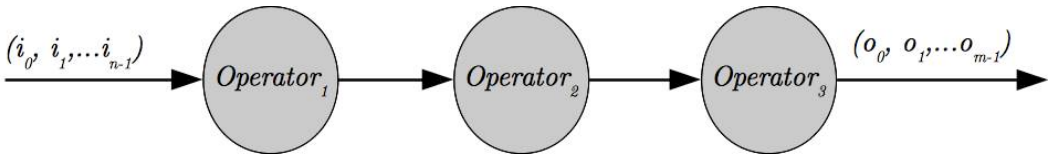

Figure 4.2: Logical query with three operators

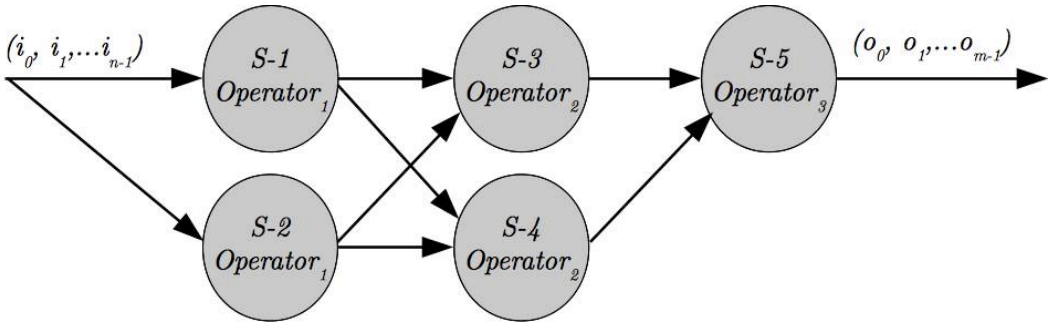

Figure 4.3: Physical query with five slaves and three operators

We make the distinction between a logical and a physical query because, as will be explained in detail in the following sections, we want query developers to define scaling rules based on the logical definition of the query. Determining the physical implementation of the query (i.e.: number of instances per operator), which is dynamic and can change over time, is the responsibility of the scaling framework in conjunction with the master process, with the later orchestrating the decisions made by the former.

Our assumption of logical and physical queries is similar to the distinction between query graphs and execution graphs made in [11].

4.1.3 Uniquely Identifiable Operators

As we stated previously, in our abstract model of an SPS we assume that logical queries are composed of operators, each of which applies a particular data transformation to the incoming stream of events. We further this assumption by requiring that operators in the logical query can be identified. In other words, each operator should have a unique name or

identifier that allows the SPS to determine the location of a particular operator. The SPS should be able to tell which slave processes are running instances of a given logical operator.

Our abstract SPS model does not make any assumptions in terms of the nature of the operator identifiers or names. We do not require these to be of a particular data type or follow a specific format.

4.1.4 Summary

We have so far presented an abstract SPS model on top of which our scaling framework will be designed and implemented. The most relevant assumptions are:

- In the SPS, there are master and slave processes
- The master coordinates and the slaves process the stream.
- Data transformations are expressed as logical queries and mapped to physical queries by the SPS in conjunction with the scaling framework.
- The SPS provides a mechanism to uniquely identify operators.

In the following sections, these topics will be visited again when we present the design and architecture of the proposed scaling framework. When any of these assumptions has a significant impact on a design decision, this will be stated explicitly.

4.2 Model of Dynamic Scaling Policies

We aim to design and develop a generic framework for the definition of scaling policies that provide for the dynamic and automatic scaling of queries running on top of SPS and similar systems. Therefore, it is necessary to first define what we consider a scaling policy, what are its composing elements and how this elements interact with each other and eventually dictate what operators to scale and by how much. In this section, we describe our model for scaling policies and scaling rules, following a top-down approach ranging from policies themselves down to its basic components such as triggers, actions and other applicable constraints.

4.2.1 Scaling Policies

In our model, **Scaling policies** consist of one or more **Scaling Rules**. Using Extended Backus-Naur Form (EBNF) notation, a scaling policy is defined as:

```
scalingPolicy     = scalingRules;
scalingRules      = { scalingRule };
```

Figure 4.4: Definition of a scaling policy in EBNF notation

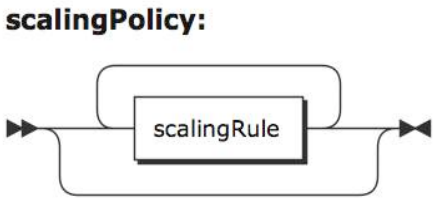

Figure 4.5: Syntax diagram for a scaling policy

Scaling policies are expressed in terms of the logical query and not the physical query. In fact, at runtime, the policies will dynamically determine the form of the one-to-many function that transforms the logical query into a physical query, mapping logical operators to slave processes. In general, and without going into any specific details, a rule will determine how many slave processes to allocate for a type of logical operator given a set of external conditions that describe the current state of the SPS. The combined effect of all the rules that form the scaling policy is what determines the actual shape of the mapping function.

An empty policy that defines no scaling rules should not have any effect on the execution of the logical query. In other words, the SPS will statically determine the mapping of operators in the logical query to slave processes and this mapping is not altered at runtime (i.e.: while the query is running and the stream of events is being processed).

In the following section, we provide a detailed definition of scaling rules in order to complete the model present this far.

4.2.2 Scaling Rules

As in the previous section, we will provide a definition of **Scaling Rules** in the context of our framework. In our model for dynamic scaling policies, rules are meant to unambiguously specify what scaling **Actions** a system should take (e.g.: allocate more or less slave processes to a certain operator) when faced with certain conditions. As we will explain below, conditions are expressed as action **Triggers** and are typically bi-dimensional, in the sense that, in order to be fired, not only certain system quantity needs to be exceeded but also this has to be so for a certain period of time. Finally, rules also include **Constraints** that either limit the effect of scaling actions or dynamically restrict the set of conditions that could potentially trigger scaling.

Definition

A scaling rule consists of the following elements:

- ***Rule Name***: this is a human readable name for the rule. It is meant to be used by the actual implementation of our scaling framework to provide more meaningful

information in status messages and logging details. It serves no other purpose in terms of the model described here.

- ***Operator***: unique identifier for the operator to which the scaling rule applies. We assume that the underlying SPS is capable of uniquely identify operators in a logical query. Furthermore, our model also allows a rule to apply to a single operator or all operators in a logical query.
- ***Action***: action the rule should execute when all of its action triggers are fired. For those rules that have more than one trigger, it is required that all evaluate to true in order for the scaling action to be carried out.

  There are two basic types of actions: **scale-out** and **scale-in**. The former means that one or more instances of the operator need to be created and new slave processes need to be allocated in order for these instances to run on. In contrast, the later indicates that one or more instances of the operator referenced by the rule must be stopped and the associated slaves de-allocated and released.
- ***Action Triggers***: a trigger is an entity that defines which conditions enable the execution of the rule's action. As mentioned before, a trigger has two dimensions: a value dimension (certain system quantity needs to exceed or remain below a given threshold) and a time dimension (value threshold needs to be met for at least a certain period of time). We will provide a more detailed definition below.

  Rules can have more than one trigger and it is necessary that all triggers fire in order for the scaling action to be executed. The state of all rule triggers is combined by a logical AND operator rather than a logical OR.
- ***Scale Constraint***: if a rule executes scale-out actions infinitely, it is likely that we will end up with an infinite number of operator instances in the physical query. Scale constraints allow rules to set a limit on how much scaling a rule can cause and help prevent the above situation.

  Rules must explicitly define a limit for scale-out actions. However, in our model, there is an implicit constraint for scale-in actions as operators are not allowed to scale-in below 1. In other words, at any given time, there must be at least one slave process running each operator in the logical query. Scaling below 1 would be equivalent to removing an operator from the logical query and this is forbidden.

  The following sections provide a more detailed explanation of scale constraints and the different types of limits supported by the model described here.
- ***Scale Guard Time***: a scaling rule might execute its associated action in order to alleviate a particular operator that is struggling to deliver its expected performance, usually manifested as a system quantity that falls below or above a defined threshold. The change in the physical query might not have an immediate impact on the system and the affected operator might continue to struggle for some time. This

means that conditions for the rule that triggered the action in the first place might still evaluate to true in spite of an action having been executed already.

The guard time is intended to prevent this situation by defining a grace period after a scale action during which the same action will not be executed again. Similarly, it can also help prevent opposing scaling actions to be executed one immediately after the other. We refer to this aspect of our model as **hysteresis,** which helps in avoiding incurring into unnecessary costs due to allocating and de-allocating slave processes in a very short period of time (this is a situation typically penalised by public cloud providers).

Again, using EBNF notation, a scaling rule would be defined as:

```
action            = “scale-out” | “scale-in”;

ruleName          = (* Human readable name for the scaling rule *)

operatorId        = (* Any valid operator identifier, depends on SPS *)
operatorAll       = (* Wildcard for all the operators in the query *)
operator          = operatorId | operatorAll;

relativeScaleFactor     = (* An integer value *)
absoluteScaleFactor     = (* An integer value *)
scaleFactor             = relativeScaleFactor | absoluteScaleFactor;

scalingRule       = ruleName ,
                    operator ,
                    action ,
                    scaleFactor ,
                    { actionTrigger },
                    [ scaleConstraint ] ,
                    [ scaleGuardTime ];
```

Figure 4.6: Definition of a scaling rule in EBNF notation

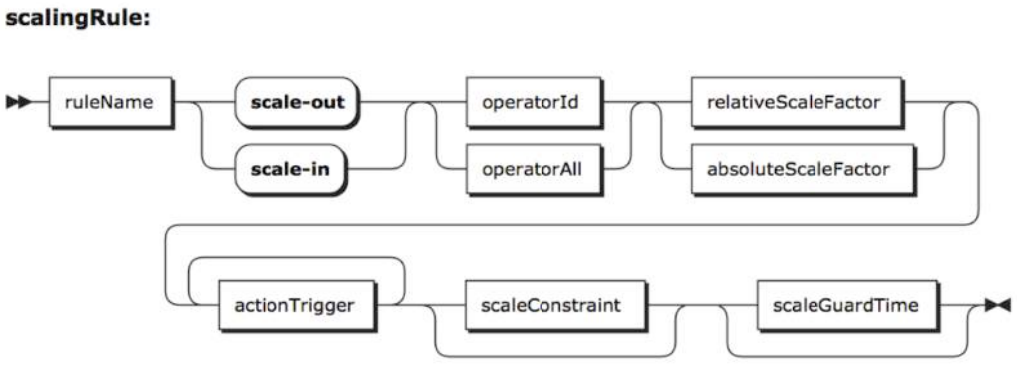

Figure 4.7: Syntax diagram for a scaling rule

Action Triggers

An action trigger is composed of a metric name and a condition that determines when the

trigger should be fired. The condition can be further decomposed into one value and one time sub-condition, both of which need to be satisfied in order for the trigger to be fired.

So, we can define these elements in detail as:

- ***Metric Name***: this is the name or identifier of the metric to which the trigger applies. A metric is typically a system quantity that reflects the state of the totality or a smaller part of the system and, particularly applied to our context, the state of a slave process executing a logical operator. Our model is abstract and does not make assumptions in terms of what metrics the SPS is capable of exposing to scaling rules. However, some sample metrics could be CPU utilisation, memory utilisation, input queue length, average latency for the operator instance, etc.
- ***Value Threshold***: this is the first dimension or sub-condition of the trigger. The threshold specifies a value that has to be either exceeded or not exceeded at all by the trigger metric. For instance, a rule could state that "CPU utilisation has to be above 30%". In this case, the value threshold is 30%.
- ***Time Threshold***: this is the second dimension or sub-condition of the trigger. The threshold specifies a period of time during which the value threshold has to either be exceeded or not exceeded (our model contemplates both cases) by metric values sampled from the SPS.

  Having a temporal dimension in triggers is relevant as scaling rules should be resilient to short-lived variations in metrics. Ideally, rules should only respond to longer term trends or at least, trends that last sufficiently long in order to amortise the cost of provisioning and allocating new slave processes.

Finally, the model aims at the value threshold always being evaluated in conjunction with the time threshold, as the former needs to either be exceeded or not exceeded by the metric value during the time period specified by the later. So, furthering the previous example, a rule would state that "CPU utilisation has to be above 30% for at least 45 seconds". In this case, the value threshold is 30% and the time threshold is 45 seconds.

Thus, in EBNF notation, a scaling rule would be defined as:

```
metricValue       = (* A floating point value *)
timeValue         = (* A floating point value *)

metricName        = (* The name of the metric exposed by the SPS *)

timeUnit          = "minutes" | "seconds";
valueComparator   = "above" | "below";

valueThreshold    = valueComparator , metricValue;
timeThreshold     = timeUnit , timeValue;

actionTrigger     = metricName ,
                    valueThreshold ,
                    timeThreshold;
```

Figure 4.8: Definition of an action trigger in EBNF notation

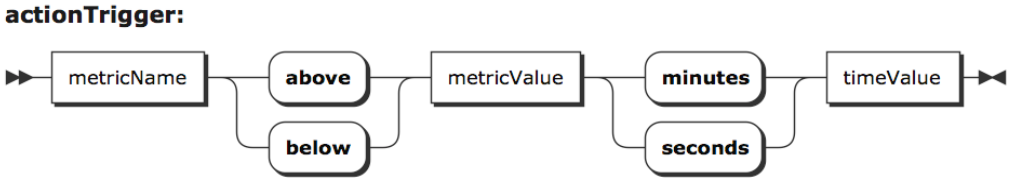

Figure 4.9: Syntax diagram for an action trigger

Constraints

The model for scaling rules provides the ability to set constraints that can prevent the execution of a scaling actions under certain conditions. There are two types of constraints:

- ***Scale Constraint***: a scale constraint specifies an absolute or a relative limit on the number of slave processes that can be allocated to a certain operator in the query. Both are integer numbers although their meaning is somewhat different. An absolute constraint defines the maximum number of slaves in the SPS that can execute a logical operator at any given time. In contrast, a relative constraint defines the maximum ratio between the initial and the current number of slave processes running an operator. In both cases, a scaling action would be prevented from running if the constraint is violated. For instance, a relative constraint of 2 effectively sets a maximum size for the operator of 2 if se start with a single instance in the logical query. However, if the logical query contains 2 instances of the same operator, then the maximum size is 4 given the same relative constraint.
- ***Scale Guard Time***: this element defines a grace period when the rule is forbidden from triggering any additional scaling. The grace period is measured since the last scaling action of a particular type and is expressed as a time threshold. This enables the model to capture restrictions such as "no further scale-out at least after 5 minutes since the last scale-out" or "don't scale-in if this operator was scaled-out during the last 15 minutes". This supports the concept of hysteresis we mentioned before.

```
relativeScaleConstraint       = (* An integer value *)
absoluteScaleConstraint       = (* An integer value *)

scaleConstraint   = relativeScaleConstraint | absoluteScaleConstraint;


lastAction        = "scale-out" | "scale-in";
scaleGuardTime    = lastAction ,
                    timeThreshold;
```

Figure 4.10: Definition of constraints for a scaling rule in EBNF notation

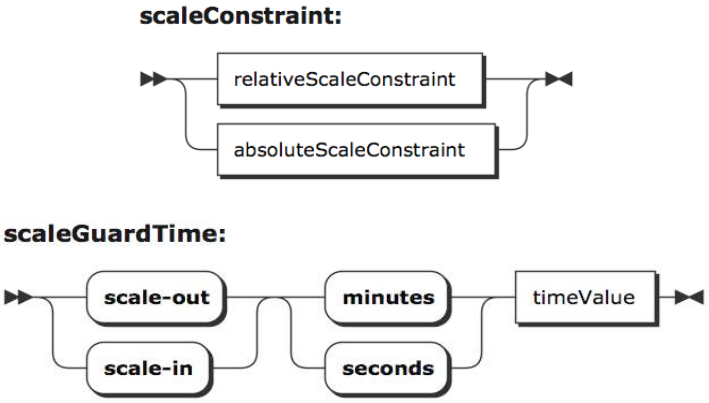

Figure 4.11: Syntax diagram for constraints in a scaling rule

4.2.3 Summary

We have presented a model for the definition of scaling policies that can to used to control the dynamic scaling of queries running on top of systems that conform to our abstract SPS model. Following a top-down approach, we have described in detail the components of scaling rules and given a precise syntax-like definition of each one, covering the elements in the model that determine whether to scale or not, when to do it and by how much.

4.3 Architecture

We will now present the architecture for our generic scaling framework, based on the abstract SPS model and the model for dynamic scaling policies described in previous sections. We will follow a top-down approach, where we first present a high-level overview of an abstract SPS working in conjunction with our proposed scaling framework. Secondly, we will describe each of the components of the scaling framework, including the facilities to define scaling rules. Finally, we will describe the integration mechanisms exposed by the framework to allow its integration with concrete SPS systems that conform to our abstract SPS model.

4.3.1 Overview

This diagram shows an augmented view of the abstract SPS model defined previously (see Figure 4.1). Here, we incorporate the components of our scaling framework, showing how these integrate with the master and slave processes of the abstract SPS.

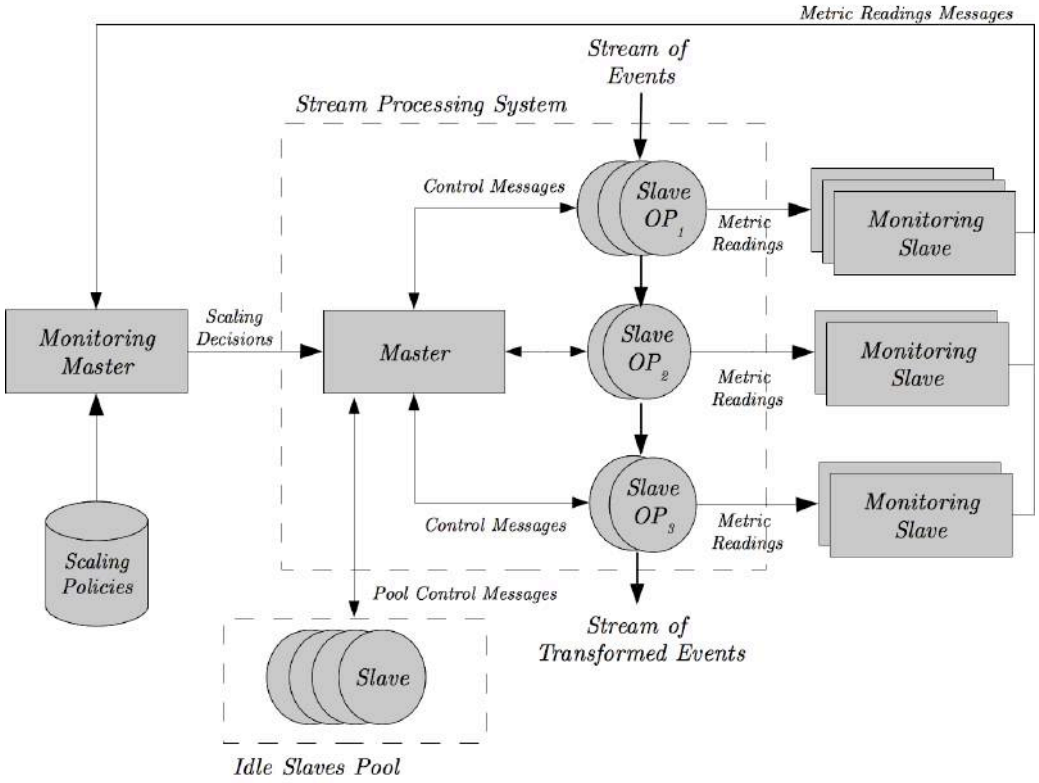

Figure 4.12: Architecture of an abstract SPS with the scaling framework

Thus, the scaling framework is formed by the following components, each performing a specific function:

- ***Monitoring Slave***: this component collects metrics from the SPS slave processes, with a metric being a system quantity that reflects the state of the system (e.g.: CPU utilisation, latency for the operator, memory utilisation, etc). Then, the component packages the metric readings into a message that is sent to the *Monitoring Master*. This procedure is repeated at regular intervals, in order to provide the *Monitoring Master with* an updated and accurate picture of the SPS state at any given time.

  There is one instance of this component per SPS slave process. Whenever a new slave process is spawned, a new instance of the *Monitoring Slave* component will be created and initialised, ensuring that the *Monitoring Master* is also eventually aware of the new slave's state.

- ***Monitoring Master***: this component receives the metric readings transmitted by all the *Monitoring Slave* instances. For each metric, the component maintains a sliding history of past readings depending on the duration of the time thresholds of the defined scaling policies for the running query. Using these past readings, it evaluates all active policies and watches for any triggered scaling actions. Effectively, if any triggers are fired during the evaluation phase, the *Monitoring Master* will pass on the corresponding scaling decision to the master process of the abstract

SPS, assuming that this process is capable of adjusting the physical query accordingly, changing the number of instances of a particular logical operator.

There is one instance of this component for the entire SPS.

- ***Scaling Policies***: these are the scaling policies, defined according to our model for dynamic scaling policies presented previously. These policies are only known by the *Monitoring Master* and its existence is transparent to both the SPS and *Monitoring Slave* instances. Additionally, the architecture does not make any assumptions on how policies are defined and stored, as this is considered an implementation aspect of the framework.

In the next sections, we will offer more detailed descriptions of the design and architecture of the components presented here, including the algorithms used to collect metric readings, manage past metric readings and evaluate scaling policies.

4.3.2 Monitoring Master

The *Monitoring Master* (MM) component consists essentially of a concurrent ***TCP Server*** listening for incoming connections from *Monitoring Slave* (MS) instances on a predefined port. In this context, and as will be explained in detail later, the MS acts as a client. A concurrent server, differently from other types of servers, typically creates a new process, task or thread to service a client's request, depending on what the underlying system supports [23]. We will refer to these new processes serving MS instances within the MM as ***Workers***. Hence, whenever a new MS instance establishes a TCP connection to the MM, a new worker is created. Furthermore, for simplicity, each worker is exclusive to one MS and one single underlying TCP connection.

Thus, each worker periodically receives metric readings from one MS. These readings are ultimately obtained from a slave process running a logical operator instance. Metric readings will be forwarded to ***Rule Evaluators*** for further processing, after being deserialised by the worker. The number of evaluators in the system depends on the number of rules in the scaling policy. The diagram below illustrates how workers and evaluators cooperate within the context of the MM component.

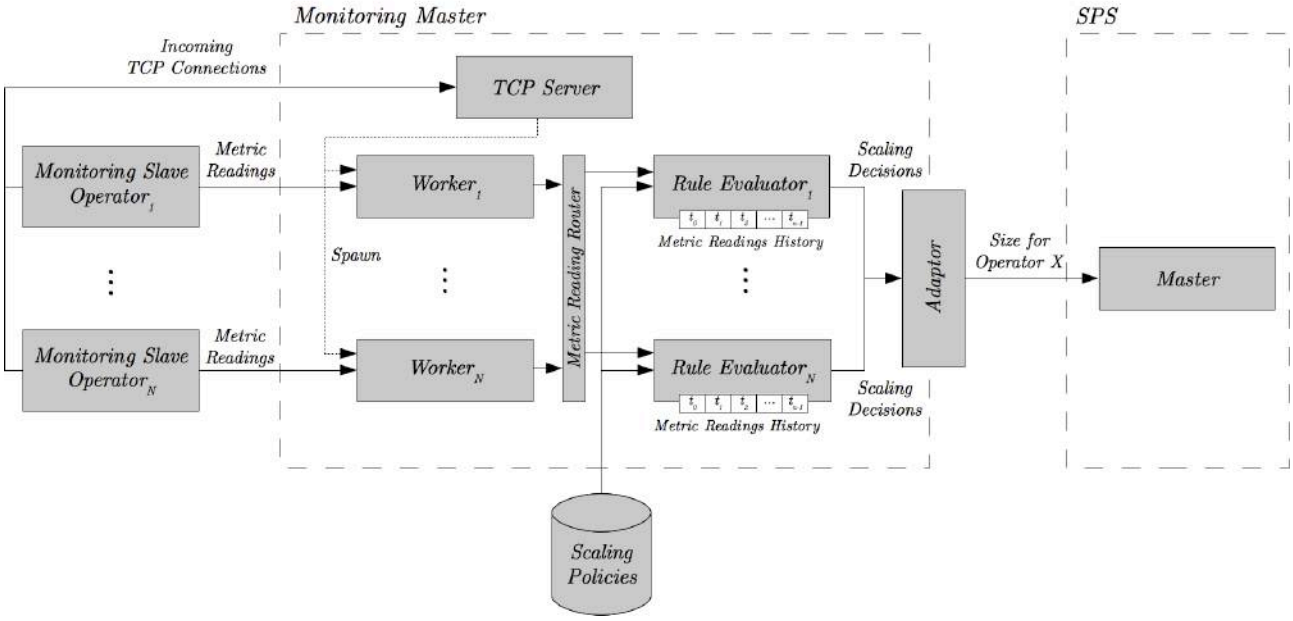

Figure 4.13: Architecture of the Monitoring Master (MM) component

Each worker receives readings from a single MS, associated to one SPS slave process and ultimately, a single logical operator instance. Evaluators are only concerned with a single rule and, given our model for scaling rules, this translates to either one logical operator or all operators in the query. However, the set of policies might contain one or multiple rules per logical operator and therefore, a single metric reading might need to be passed on to multiple evaluators. The ***Metric Reading Router*** component shown above is responsible for deciding which evaluators should process a particular metric. It can be said that metrics map to evaluators in a one-to-many relationship, although in most cases this will reduce to a simpler one-to-one case.

Scaling rules have a temporal dimension (i.e.: time threshold) that determines how long a metric has to meet a certain condition in order for a scaling action to be triggered. To be able to evaluate this time dimension, the corresponding evaluator must maintain a bounded history of past readings. The bound is given by the duration of the rule's time threshold, as it is not necessary to look further back in time to evaluate a rule. This history component is referred to as ***Metric Readings History*** in the diagram above, which holds past readings ordered from most recent to least recent.

Finally, the ***Adaptor*** component provides an abstraction for the MM to interact with a concrete SPS without depending directly on it and allowing the scaling framework to maintain its genericness. Moreover, this additional layer of abstraction provides a place for mappings, translations and any adaptations that might be required, were there any differences between the concrete SPS to which we wish to integrate the scaling framework and our model of an abstract SPS presented in previous sections. The evaluator will output generic scaling decisions (e.g.: increase the size of operator X to double its current size) and the adaptor will translate those to actual instructions supported by the concrete SPS

(e.g.: get number of slaves running operator X instances, calculate desired new size, provision new slaves from idle pool, deploy instances of operator X to new slaves, etc).

4.3.3 Monitoring Slave

The *Monitoring Slave* (MS) component consists primarily of a ***TCP Client***, which will connect to the *Monitoring Master* (MM) on a predefined TCP port. Once a connection is established, the ***Processor*** will start sending metric readings to the MM at regular intervals (e.g.: every 10 seconds, every minute, etc). The more frequently readings are sent to the master, the more accurate its understanding of the SPS status will be. Hence, one of the responsibilities of the processor is to enforce this periodicity and ensure that readings are provided to the MM at the right times. Additionally, it is also responsible for serialising metric readings into a format that can be transmitted over the TCP connection easily. A reading consists essentially of an operator identifier, a timestamp and a set of key-value pairs, where each key is only allowed to appear once in the metric reading. The image below shows a typical reading, including the metrics "CPU Utilisation" and "Input-queue Length":

Operator Identifier	*Timestamp*	*KEY* *CPU Utilisation*	*VALUE* *34%*	*KEY* *Input Queue-length*	*VALUE* *3478 tuples*

Figure 4.14: Metric reading format, with metrics "CPU Utilisation" and "Input-queue Length".

The processor receives unpacked readings from the metric ***Reader*** component, which in turns obtains values from the ***SPS Instrumentation*** and the ***Operating System Instrumentation*** components. The former gives the reader component access to metrics that refer specifically to SPS quantities, such as input-queue length, operator latency, state size, number of out-of-order tuples, etc. The later provides the reader metrics that represent system quantities that are independent of the underlying SPS. Typical examples would be CPU Utilisation, available disk space, number of open file descriptors, etc. In addition, this category also includes any metrics specific to the runtime for the SPS. Specifically, if the SPS along with the monitoring framework are implemented in the Java language, this category would also include metrics for the Java Virtual Machine (JVM).

The following diagram shows all the components described above and how the integrate together to form the MS component of our generic scaling framework:

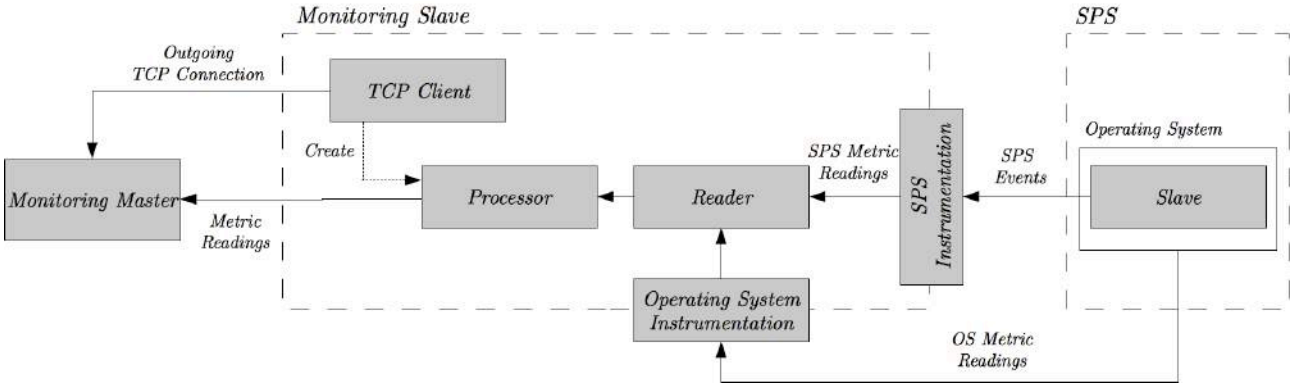

Figure 4.15: Architecture of the Monitoring Slave (MS) component

4.3.4 Integration with Stream Processing Systems

The ***Adaptor*** in the MM component and the ***SPS Instrumentation*** in the MS component are the main integration points between the underlying SPS and our generic scaling framework. Here we provide further details on these two elements work and what type of interfaces these expose to the rest of the framework.

The adaptor receives scaling decisions from the evaluators, as scaling actions are triggered for the different scaling rules that apply to the running query. These scaling decisions are expressed in terms that the adaptor can understand. The main responsibility of this element is to translate these scaling decisions to terms that can be understood by the underlying concrete SPS with which we want to integrate the scaling framework. As mentioned before, the adaptor has to hide or at least make transparent any differences that might exist between the concrete SPS system and the abstract SPS model on which the framework is based.

The adaptor exposes the following functions to evaluators. Again, scaling decisions need to be expressed in terms of these functions and translated accordingly so that these can be carried out and executed by the SPS.

- ***Get list of running operators***: returns a list of the logical operator identifiers running for the current query.
- ***Get current size for a running operator***: returns the number of SPS slave processes currently running a certain logical operator. This function allows the adaptor to support scaling decisions with constraints (e.g.: scale-out operator X but never above this size, scale-in operator Y but never below this size, etc).
- ***Get original size for a running operator***: returns the number of SPS slave processes originally running a certain logical operator. This value is determined by the initial mapping of the logical to the physical query. In most cases, this will be 1 and it enables the adaptor to support scaling decisions expressed in relative terms (e.g.: double the size of operator X, half the size of the operator Y, etc).

- ***Set new desired size for a running operator***: notifies the SPS what the new desired size for a given operator is. Subsequently, the SPS will need to provision new slaves, deploy the corresponding operator and connect it to upstream and downstream operators. Depending on what type of interface the SPS exposes, the adaptor might need to implement more or less complex code to support this function (e.g.: if the SPS only exposes a function to increase the size of an operator by one each time, the adaptor will need to ensure that this function is invoked multiple times to achieve what was originally specified in the scaling decision).

On the other hand, the instrumentation element is intended to receive events from SPS slave processes and generate SPS specific metric readings from received events. Mainly, this consists in being notified whenever a tuple arrives to the slave for processing and conversely, receive a notification when the tuple has been processed by the operator and forwarded to the next downstream operator. With such events, it is relatively straightforward for the instrumentation element to estimate metrics. Initially, we intend to only support the SPS metrics "Input-queue length" and "Operator latency". These two metrics can be derived easily from the events mentioned above. For instance, the length of the input queue for the operator can be estimated by incrementing a counter whenever a tuple arrives and decrementing it when the tuple leaves the slave. In the future, other events can be supported in order to provide a broader variety of SPS metrics.

4.3.5 Policy Evaluation

Scaling policies are evaluated by the MM. As described previously, workers receive a stream of metric readings from a single MS associated to one SPS slave process (i.e.: a single logical operator). In particular, within the MM, policy evaluation is the responsibility of the rule evaluator, supported by the incoming stream of metric readings plus the readings history available at the moment of evaluation. Additionally, policies can define one or many rules for a logical operator and potentially, certain rules can apply to all the operators in a query. Hence, the metric reading router is responsible for deciding which rules from the policy need to be evaluated depending on the source operator for the incoming reading. The router decides which evaluators should receive a copy of the incoming metric and thus, evaluate their associated scaling rule.

In this section, we present the pseudo-code for two algorithms utilised by the monitoring framework. First, we describe the algorithm executed by the router that determines which rules from the policy need to be evaluated based on the source operator for the metric reading and indirectly, which evaluators must receive a copy of the incoming metric reading. Secondly, we present the algorithm executed by evaluators that determines whether to trigger the scaling action of its associated scaling rule, given the present and past metric readings.

When initialised, the router component creates tow data structures that will be later used for the actual routing decision: ***evaluators***, a table indexed by logical operator identifier

containing for each entry a list of all rule evaluators that refer to the operator in question; and ***allEvaluators***, simply a list of all available rule evaluators defined in the scaling policy. The routing algorithm then routes readings based on these two data structures. It is worth mentioning that the data structures are static, in the sense that they do not change at runtime. The shape of the data structures is determined by the rules in the policy, and our proposed scaling framework does not support policy changes at runtime.

Algorithm Routing in Metric Reading Router

```
1:  procedure INITIALIZEEVALUATORS
2:      policy ← Scaling policy
3:      evaluators ← Table of rule evaluators indexed by operator
4:      allEvaluators ← List of all rule evaluators
5:  operator rules:
6:      for all rules in policy do
7:          operator := rule.operator
8:          id := operator.id
9:          if id <> all operators then
10:             evaluator := Create new evaluator for rule
11:             evaluators[id] ← evaluator
12: wildcard rules:
13:     for all rules in policy do
14:         operator := rule.operator
15:         id := operator.id
16:         if id = all operators then
17:             evaluator ← Create new evaluator for rule
18:             allEvaluators ← evaluator
19:     for all operators in evaluators do
20:         id := operator.id
21:         evaluators[id] := allEvaluators

22: procedure ROUTEREADINGTOEVALUATORS
23:     reading ← Metric reading from a Worker
24:     evaluators ← Table of rule evaluators indexed by operator
25:     allEvaluators ← List of all rule evaluators
26: routing:
27:     operator := reading.operator
28:     id := operator.id
29:     if id is a valid key for evaluators then
30:         for all evaluators in evaluators[id] do
31:             Send reading to evaluator
32:     else
33:         for all evaluators in allEvaluators do
34:             Send reading to evaluator
```

Algorithm 4.1: Routing algorithm implemented by the Metric Reading Router

The evaluation algorithm is more complex than the routing algorithm. The first part of the algorithm consists in pruning past metric readings, discarding any that might be too old to be taken into account by the scaling rule. Consecutively, the algorithm evaluates all the triggers associated to the rule. All the triggers, once evaluated, need to be fired and have changed from their previous state, in order for the scaling action to take place.

```
Algorithm  Policy evaluation in Rule Evaluator
 1: procedure EVALUATE
 2:     rule ← Rule from scaling policy
 3:     reading ← Metric reading from a Worker
 4:     history ← Metric readings history
 5: max-age:
 6:     max-age := 0
 7:     for all triggers in rule.triggers do
 8:         if trigger.timeThreshold > max-age then
 9:             max-age := trigger.timeThreshold
10: offer:
11:     Add reading to history
12: prune:
13:     done := False
14:     reading ← history
15:     while reading <> undefined and not done do
16:         age := reading.timestamp − now
17:         if age <= max-age then
18:             done := True
19:         else
20:             Remove reading from history
21:             reading ← history
22: evaluate:
23:     scale := False
24:     for all triggers in rule.triggers do
25:         EvaluateTrigger(trigger, history)
26:         scale := scale and trigger.fired and trigger.changed
27:     if scale = True then
28:         operator := reading.operator
29:         id := operator.id
30:         if id <> all operators then
31:             Send scale command to adaptor for operator id
32:         else
33:             Send scale command to adaptor for all operators
```

Algorithm 4.2: Rule evaluation algorithm implemented by the Rule Evaluator

The trigger evaluation algorithm iterates over past metrics present in the history component, identifying those readings that occurred within the time threshold specified by the trigger. Then, the algorithm checks that the value obtained at the time when the metric was read is either below or above the value threshold for the trigger, depending on its type. If this condition is met for all past metrics within the trigger's time threshold, then the trigger is fired and possibly changed, depending on its state before being evaluated.

```
Algorithm  Trigger evaluation in Rule Evaluator
 1: procedure EVALUATETRIGGER
 2:     trigger ← Trigger from rule in scaling policy
 3:     history ← Metric readings history
 4: init:
 5:     time-threshold := trigger.timeThrshold
 6:     value-threshold := trigger.valueThreshold
 7: check:
 8:     enough-readings := True
 9:     most-recent-time ← history[head].timestamp
10:     least-recent-time ← history[tail].timestamp
11:     if most-recent-time − least-recent-time <= time-threshold then
12:         enough-readings := False
13:     end if
14: evaluate:
15:     past-trigger-state := trigger.fired
16:     if enough-readings = True then
17:         for all readings in history do
18:             if now − reading.timestamp <= time-threshold then
19:                 switch trigger.valueComparator do
20:                     case above
21:                         if reading.value > value-threshold then
22:                             trigger.fired = True
23:                         end if
24:                     case below
25:                         if reading.value <= value-threshold then
26:                             trigger.fired = True
27:                         end if
28:             end if
29:             if trigger.fired <> past-trigger-state then
30:                 trigger.changed := True
31:             end if
32:         end for
33:     end if
34: end procedure
```

Algorithm 4.3: Trigger evaluation algorithm implemented by the Rule Evaluator

4.3.6 Policy Definition

Previously, we described the domain model for scaling rules and policies that we intend to use for our scaling framework. We defined the semantics of the domain and described each one of the parts that form a scaling rule in our model. In this section, we discuss some ideas related to how these policies can be defined and stored, focusing on the particular solution that we propose for our scaling framework.

Assuming that we work in the context of a general purpose programming language (GPL), we need to solve a specific problem in a well-defined application domain: creating rules that determine when to scale operators in the context of a stream processing query executed by an SPS. A few solutions have been attempted to allow solving a specific problem in the context of a GPL [24]:

- *Subroutine libraries*
- *Object-oriented framework*

- *Domain-specific languages*

Subroutine libraries contain functions or subroutines that perform related tasks in a well-defined application domain. This is the classical method to package re-usable domain knowledge. Object-oriented frameworks extend this idea into object-oriented programming. However, differently from classical libraries where the applications call the subroutines, frameworks typically invert control and call the application code instead of being called [24].

Finally, a ***Domain-Specific Language*** (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. In many cases, DSL programs are translated to calls to a common subroutine library and the DSL can be viewed as a means to hide the details of that library [24]. Moreover, DSLs are usually declarative and consequently, they can be viewed as specification languages as well as programming languages [24].

Additionally, by providing notations and constructs tailored toward a particular application domain, DSLs offer substantial gains in expressiveness and ease of use compared with GPLs for the domain in question, with corresponding gains in productivity and reduced maintenance costs. Also, by reducing the amount of domain and programming expertise needed, DSLs open up their application domain to a larger group of software developers compared to GPLs [25]. Further advantages of DSLs include allowing solutions to be expressed in the idiom and at the level of abstraction of the problem domain, producing concise, self-documenting and re-usable programs [24].

DSLs can be further subdivided into two categories. An external DSL is a language separate from the main language the application it works with, possibly with its own custom syntax requiring parsing. In contrast, an internal or embedded DSL (EDSL) is a particular way of using a GPL, using only a subset of the language features in a particular style. The result should have the feel of a custom language rather than the host language [26]. With an EDSL, domain-specific features can be embedded into an existing GPL to take advantage of its implementation and tools. The conventions of the host language are unlikely to apply to an EDSL, given that the motivation for writing an EDSL is to overcome limitations in the host. To make the EDSL readable, it may need to break conventions such as capitalisation, formatting, and naming for classes and methods [27].

We believe that the problem of creating scaling rules and policies is quite specific and it applies to a well-defined application domain. Therefore, we think that our problem can be better solved by means of a DSL that allows query developers to easily write scaling rules using notations and abstractions pertinent to this particular domain. Particularly, we want to create an expressive DSL that, by means of a non-imperative declarative style, acts as a specification language for scaling rules and policies. The language would obviously be based on the model for scaling rules that we presented previously as one of our assumptions for our framework. Furthermore, to leverage existing tools and facilitate

integration, we aim at implementing our DSL for scaling rules as an EDSL for the language in which stream queries and the target SPS are developed.

4.3.7 Summary

In this chapter, we described in detail the architecture and design of our proposed scaling framework, which runs on top of a generic SPS described by our abstract model. The architecture was conceived on the basis that all the assumptions of the generic SPS model are true.

Initially, we presented an overview of the architecture as a whole and how it integrates with an abstract SPS. Then, we split the architecture roughly into three areas of concerns:

- ***Monitoring Master***: we described all the components that run alongside the SPS Master process.
- ***Monitoring Slave***: we described all the components that run alongside the SPS Slave processes.
- ***Scaling Policies***: we described how policies are defined and evaluated. We defined what an embedded DSL (EDSL) is and we justified our decision to create one for the definition of scaling policies under our proposed framework. We also presented the algorithm that the framework uses to evaluate policies in pseudo-code form.

The next section presents the implementation of the proposed architecture.

4.4 Implementation

In this section, we present a detailed description of the implementation of our proposed framework. We first describe the implementation of scaling policies and how the abstract model we defined before maps to an actual implementation. Secondly, we show the internal DSL that our framework exposes for the definition of scaling policies. Then, we continue with the implementation details of both the Monitoring Master and the Monitoring Slave components. Lastly, we cover the implementation details of our generic framework with an existing SPS, the SEEP system.

Whenever possible, we will use UML class and sequence diagrams to describe the implementation of a particular aspect of our framework. However, we do intend to omit certain implementation aspects that relate more to conventions and techniques of the target implementation language (e.g.: accessor methods, inner classes, etc) rather than the core functional features of our framework. In spite of this, we might include small source code examples when this adds clarity to the description of a particular feature.

4.4.1 Implementation Language

Up to this point, we have described an abstract scaling framework, avoiding making any assumptions in terms of the implementation language or platform. Potentially, the

proposed design and architecture could be implemented using any language that provides bindings or a library for the underlying operating system's TCP/IP stack. However, for this project in particular, we have decided to base our implementation on the Java programming language. The reason for this is two folded.

In first place, Java is consistently ranked as one of the most popular languages, as shown in [28] and [29]. The more popular a language is, the easier it should be to find well maintained open-source libraries and tools that address common programming problems (e.g.: logging, date handling, configuration management, build tools, automatic documentation generation, etc). We can leverage these libraries and somewhat reduce the development effort, focusing on the core features of the system rather than common concerns.

Secondly, as we mentioned before, we intend to integrate our generic scaling framework with SEEP. This system is implemented in Java and therefore, choosing the same implementation language for our framework seems like a reasonable decision. Using the same language will minimise integration problems, completely avoiding problems that might arise due to language and platform differences and incompatibilities.

4.4.2 Scaling Policies

The class hierarchy shown in the diagram below represents our model for scaling rules. In general, these classes are a direct mapping of the elements defined by the abstract model described previously. Rules, actions, triggers and constraints are mapped directly to a corresponding equivalent Java class. The ***EvaluateTrigger*** algorithm presented previously is implemented by the ***evaluate()*** method in the ***ActionTrigger*** class.

We rely on abstract classes, interfaces and generalisation to make the implementation of policy rules and action triggers as generic as possible. Thus, the same code is capable of dealing with abstract types of value thresholds, scale factors and scale constraints, without being aware of the concrete type the code is actually dealing with. Moreover, the ***EvaluateTrigger*** algorithm is implemented in terms of abstractions rather than concretions.

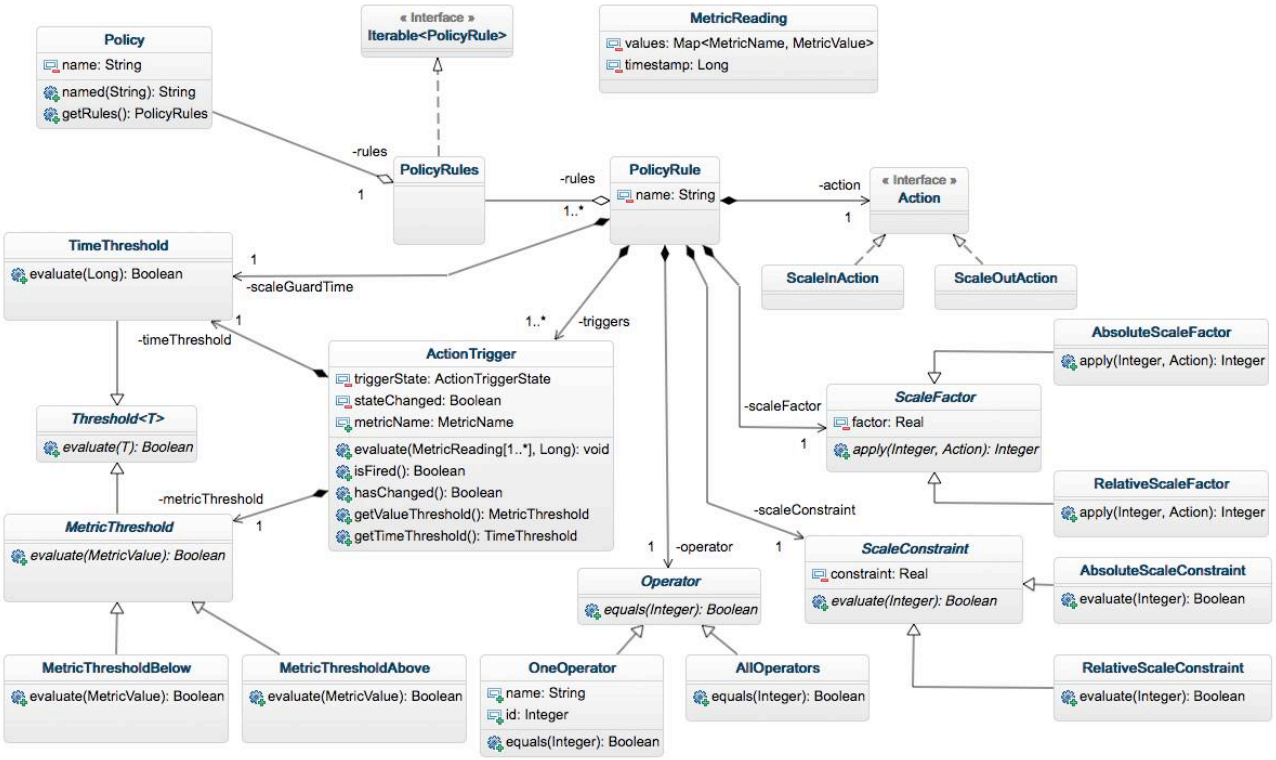

Figure 4.16: UML class diagram for Scaling Policies

Similarly, we want to avoid the direct creation of new instances of the different concrete classes in our model. In particular, we want to avoid client code calling the ***new*** Java operator on classes such as ***MetricThresholdBelow***, ***AbsoluteScaleConstraint*** **and** ***RelativeScaleFactor***. In order to achieve this, we implement a common pattern across most of the classes. Our pattern has certain points in common with the creational patterns ***Factory Method*** and ***Abstract Factory,*** as defined in [30], however it does not constitute an actual realisation of any of these two at least in their classic form.

Our pattern is structured around two main concepts:

- We make the constructors for any concrete subclasses in our model package-private by using the default Java access modifier (e.g.: ***MetricThresholdBelow***). This effectively prevents the creation of instances from outside the package where our classes are defined.
- The base abstract class of each concrete class defines a public static factory method that create and returns instances of the corresponding concrete sub-class. The factory method is named as the aspect that differentiates the concrete sub-class from the rest. For instance, the abstract class ***MetricThreshold***, which is the super class of ***MetricThresholdBelow***, defines a factory method named ***below()***.

The example below shows this pattern applied to ***MetricThreshold*** and ***MetricThresholdBelow***:

```
/**
 * Base abstract class for metric value thresholds in scaling policies.
 */
public abstract class MetricThreshold extends Threshold<MetricValue> {

    public static MetricThresholdBelow below(MetricValue threshold) {
        return new MetricThresholdBelow(threshold);
    }

    ...
}

/**
 * Concrete metric value threshold that evaluates to true when a
 * particular metric value is below the threshold.
 */
public class MetricThresholdBelow extends MetricThreshold {

    MetricThresholdBelow(MetricValue threshold) {
        super(threshold);
    }

    ...
}
```

Source Code 4.4: Creational pattern applied to MetricThreshold and MetricThresholdBelow

```
import static uk.ac.imperial.lsds.monitor.policy.threshold.MetricThreshold.*;
import static uk.ac.imperial.lsds.monitor.policy.metric.MetricValue.*;

// Create a threshold that evaluates to true when a metric is below 50%
// We also use the factory method for percent values.
MetricThreshold t = below(percent(50));
```

Source Code 4.5: Use of proposed creational pattern for object creation

This implementation approach, coupled with the Java static imports, provides one of the foundations for our internal DSL for the definition of scaling policies, which will be described in depth in the following section.

4.4.3 Internal DSL for Scaling Policies

As we stated before, our intention is to implement an EDSL for the definition of scaling policies and scaling rules. Our approach is similar and relies on several of the techniques described in [27] to write a language in Java, which acts as the host language for our EDSL. The following UML class diagram illustrates the classes that support our EDSL.

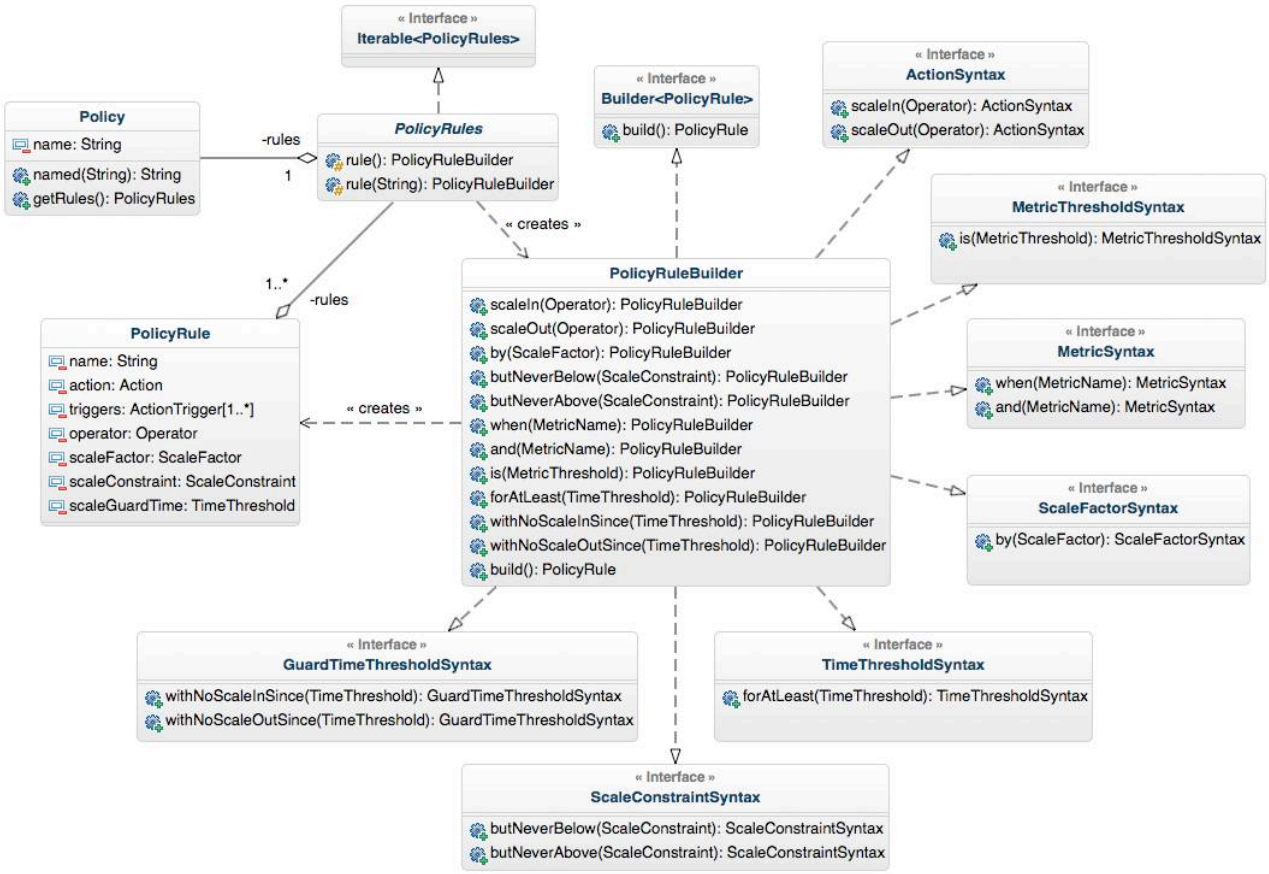

Figure 4.17: UML class diagram for classes supporting the EDSL

In first place, we aim to keep the syntax of the EDSL and its interpretation into separate layers. We define several Java interfaces, such as ***ActionSyntax***, ***ScaleFactorSyntax*** and ***MetricSyntax***, that represent the syntax of our embedded domain language for scaling policies. These syntax interfaces simply define methods to specify the different elements of scaling rules, according to our abstract model presented earlier.

Secondly, all these syntax interfaces are implemented by a realisation of the Builder pattern, as advised by the authors in [27]. The Builder pattern intent is to separate the construction of a complex object from its representation so that the same construction process can create different representations [30]. In our case, the introduction of the ***PolicyRuleBuilder*** class allows us to simplify the creation of ***PolicyRule*** instances and also, to have a more clear separation of concerns. We encapsulate the rule creation process in its own separate class, apart from the internal representation of rules. Furthermore, the introduction of a builder allows to specify the various elements of a rule cleanly, simply by calling and combining methods on the builder. Typically, without a builder and considering that not all scaling rules are made of the same elements, we would need to define a multitude of constructor overloads in ***PolicyRule*** in order to support different combinations of parts that can form a scaling rule (this is usually known as the "telescoping constructor" problem).

Finally, as suggested in [27], in order to make the EDSL readable, we might need to use

and abuse the host language, breaking conventions such as capitalisation, formatting and naming of classes and methods. In particular, there are three areas where we abuse the usual Java conventions and accepted practices to make our EDSL fluent and readable:

- ***Initialiser blocks***: we rely on instance initialiser blocks to provide scope for and access to our EDSL. Within these blocks, we expose the methods that provide the syntactic sugar needed to create rule definitions. Usually, an anonymous subclass of ***PolicyRules*** with an initialiser block would be the place where we would embed the definitions for scaling rules. We enforce that the EDSL is not use anywhere else by ensuring that ***PolicyRuleBuilder*** cannot be instantiated outside these blocks (e.g.: package-private constructors in conjunction with a protected factory method in ***PolicyRules***).
- ***Method-chaining***: this consists in chaining method calls one after the other, typically for ***PolicyRuleBuilder*** methods. In [27], the authors state that this would be normally regarded as very bad practice in object-oriented code. However, chained statements seem to be the only way they found to emulate a new syntax in Java. We follow the same approach and structure rule creation around chained method calls to builder objects.
- ***Static imports***: in the previous section, we mentioned that the classes modelling rule elements protect constructors and expose public static factory methods to create concrete instances. We rely heavily on Java static imports to make these methods accessible without specifying the class in which they were originally defined. Thus, we can simply call these factory methods and pass the results directly to some of the ***PolicyRuleBuilder*** methods as a single expression. It is worth mentioning that we have chosen the names of these static method carefully so that expressions in our EDSL read as natural and as close to English as possible, e.g.: *builder.is(below(percent(50)))* or *builder.is(above(percent(50)))*.

This example shows the EDSL in use, specifying scale-in and scale-out rules. Note again how an anonymous subclass of the abstract ***PolicyRules*** is created to provide scope for our embedded language and granting access to instances of the ***PolicyRuleBuilder*** class.

```
import uk.ac.imperial.lsds.monitor.policy.PolicyRules;
import static uk.ac.imperial.lsds.monitor.policy.metric.MetricName.metric;
import static uk.ac.imperial.lsds.monitor.policy.metric.MetricValue.tuples;
import static uk.ac.imperial.lsds.monitor.policy.operator.Operator.operator;
import static uk.ac.imperial.lsds.monitor.policy.scale.factor.ScaleFactor.absolute;
import static uk.ac.imperial.lsds.monitor.policy.threshold.MetricThreshold.above;
import static uk.ac.imperial.lsds.monitor.policy.threshold.MetricThreshold.below;
import static uk.ac.imperial.lsds.monitor.policy.threshold.TimeThreshold.minutes;

...

// Create anonymous subclass of PolicyRules with initialiser block
PolicyRules rules = new PolicyRules() {{

    // Call the rule() protected method to get a PolicyRuleBuilder instance
    // Then just chain method calls to the builder to define the rule
    rule("Queue length is above 1000 for 5 minutes")
        .scaleOut(operator("Operator", 1))
        .by(relative(2))
        .when(metric("queue-length"))
        .is(above(tuples(1000)))
        .forAtLeast(minutes(5)).build();

    rule("Queue length is below 1 for 5 minutes")
        .scaleIn(operator("Operator", 1))
        .by(relative(2))
        .when(metric("queue-length"))
        .is(below(tuples(1)))
        .forAtLeast(minutes(5)).build();
}};
```

Source Code 4.6: Example policy rule creation using our EDSL

All of the above allows us to expose a clean embedded DSL for the definition of scaling rues, where most of the boiler plate code, usually needed to create and initialise objects, is hidden. Only concepts that are semantically meaningful at the domain-level are exposed. We believe that this is a valuable addition to our proposed scaling framework, as it allows stream query developers to concentrate on domain-level problems and understanding what scaling policies are best suited for their queries.

4.4.4 Monitoring Master

In this section, we describe the various classes that implement the *Monitoring Master* (MM) component. As defined in the framework's architecture section, the MM is formed by a series of subcomponents that collaborate and cooperate to receive metric readings from *Monitoring Slave* (MS) instances. We will now review these subcomponents and briefly mention which classes provide the supporting implementation for them. All the classes referred below can be found in the UML class diagram shown in Figure 4.19.

- ***TCP Server***: this is the concurrent TCP server that listens for incoming connections from slave instances and spawns new workers to handle those connections. The ***MonitorMaster*** class contains a ***ServerSocket*** instance, which listens for incoming connections on a given TCP port. Whenever a slave connects to the server, a ***MonitorMasterWorker*** instance is created. It is also worth mentioning that both ***MonitorMaster*** and ***MonitorMasterWorker*** implement the ***Runnable*** and ***Stoppable*** interfaces. The first one allows instances of these

classes to run in their own separate threads, whereas the second one provides a common interface to stop the threads gracefully.

- ***Workers***: these are the processes serving specific MS instances. As we stated previously, each worker is dedicated to one MS instance and one underlying TCP connection. In terms of implementation, each ***MonitorMasterWorker*** receives a ***BinaryMetricsDeserializer*** instance upon creation. All instances of this class have a ***Socket*** instance, originally created by ***ServerSocket*** when accepting the underlying incoming TCP connection. Thus, workers read metrics from a slave through a TCP connection, wrapped in a deserialiser object which is aware of the binary representation of metrics on the wire. Deserialised metrics are then wrapped in a ***MetricReadingProvider*** object before being handed to the router subcomponent. In conclusion, this approach ensures that workers and their dependencies are agnostic in terms of transport.
- ***Metric Reading Router***: this subcomponent routes incoming metric readings to the evaluators associated to scaling rules that reference the source logical operator. It implements the routing algorithm we presented before. In particular, the ***PolicyRulesEvaluator*** class is instantiated by ***MonitorMaster*** at a rate of one instance per worker. Then, instances of ***MonitorMasterWorker*** have a reference to their associated rules evaluator, delegating the processing of incoming metrics to it. The routing algorithm in the rules evaluator will then decide which individual ***PolicyRuleEvaluator*** is to receive and evaluate the incoming metric.
- ***Rule Evaluators***: this subcomponent is responsible for evaluating a scaling rule and deciding what scaling action to trigger, if any. The ***PolicyRuleEvaluator*** class implements the metric evaluation algorithm we outlined previously. Instances of this class have references to the actual ***PolicyRule*** they need to asses and also, to the ***InfrastructureAdaptor*** to inform scaling decisions to the underlying SPS. At any given time, there will always be one evaluator per rule.
- ***Metric Readings History***: holds past readings ordered from most recent to least recent. ***PolicyRuleEvaluator*** instances contain a queue of ***MetricReading*** objects, which is responsible for maintaining past readings ordered from most recent to least recent. The evaluator, depending on the time threshold of the underlying scaling rule, will prune readings and discard those that are no longer relevant.
- ***Adaptor***: this subcomponent provides an abstraction layer for the MM to interact with a concrete SPS. The ***InfrastructureAdaptor*** interface fulfils this function in our implementation, exposing methods needed by rule evaluators to inform their scaling decisions to the SPS. Which implementation of this interface the framework uses depends on the concrete underlying SPS. Particularly, we will describe in detail the ***SeepInfrastructureAdaptor*** class later on, when we review the integration of our scaling framework with the SEEP system.

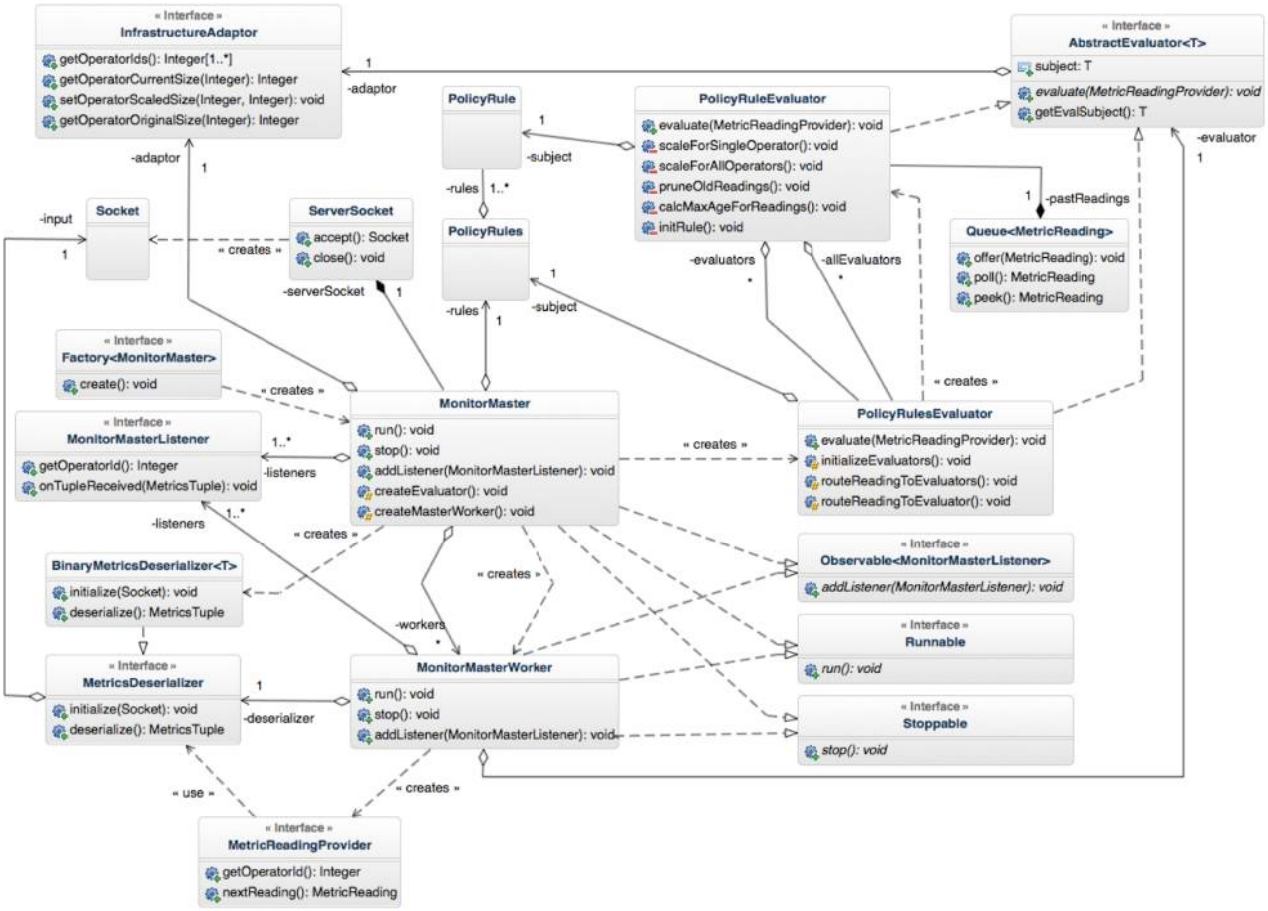

Figure 4.18: UML class diagram for the Monitoring Master (MM) component

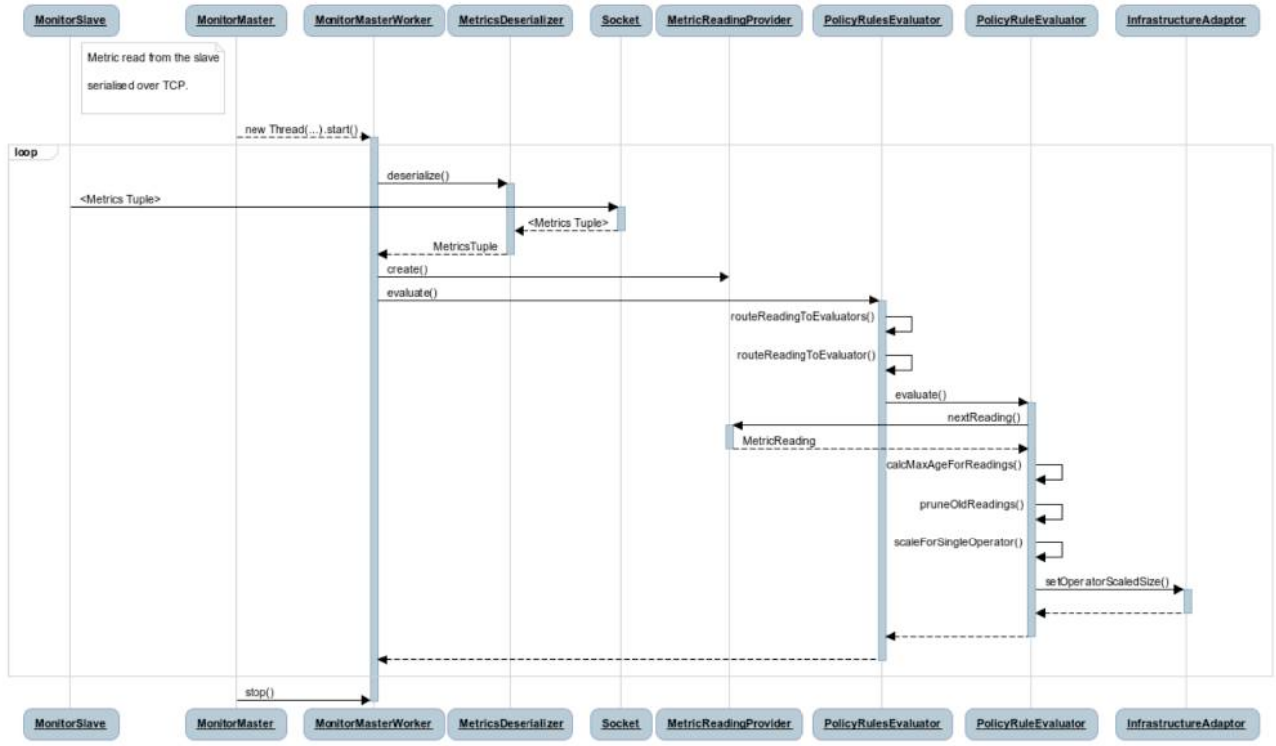

Figure 4.19: UML sequence diagram for the Monitoring Master (MM) component

The UML sequence diagram above illustrates how the classes that participate in the implementation of the MM interact when a metric reading is received from a given MS instance. In particular, the diagram shows how ***MonitorMasterWorker***, ***PolicyRulesEvaluator*** and ***PolicyRuleEvaluator*** collaborate to receive, deserialise and evaluate a metric against a particular rule. Certain classes, such as ***PolicyRule***, are omitted in order to simplify the sequence diagram.

4.4.5 Monitoring Slave

We now describe the different classes that implement the *Monitoring Slave* (MS) component. Analogously to the *Monitoring Master* (MM), the MS component is formed by a series of subcomponents that work together to collect and send metric readings to the MM instance. As we did in the previous section, we will review the subcomponents that form the MS and enumerate the classes that implement each of them. The UML class diagram in Figure 4.20 shows all the classes that conform the implementation of the MS.

- ***TCP Client***: responsible for setting up a TCP connection to the MM, which is listening for incoming connections from slaves on a predefined TCP port. A ***Socket*** instance will be created by the ***MonitorSlave*** class, through which it will attempt to connect to the master.

  The ***MonitorSlave*** class also implements the ***Runnable*** and ***Stoppable*** interfaces, which allow implementing classes to run in their own separate threads and to be stopped in a graceful manner, respectively. The framework ensures that the MM runs in its own thread separate from the SPS slave instance, in order to improve reliability (e.g.: if the MS crashes, the operator itself is not affected) and to avoid contention and locking issues.

- ***Processor***: this component is responsible for sending metric readings to the MM at regular intervals (e.g.: every 10 seconds, every minute, etc). This functionality is implemented by the ***MonitorSlave*** class and ***MonitorSlaveProcessor***, with the former ensuring that the later is invoked regularly at a predefined frequency. On each invocation, ***MonitorSlaveProcessor*** will iterate over all its ***MetricsReader*** instances, read all the metrics returned by these objects and then, iterate once again over all its ***MetricsSerializer*** instances in order to serialise the metrics. This implementation design allows for a single processor to read from multiple sources of metric readings and to forward these values to several destinations, due to its ability to write to multiple serialisers. The processor is implemented against abstractions rather than concrete classes, hence being more generic.

  The framework currently defines two concrete implementations of ***MetricsSerializer:*** ***BinaryMetricsSerializer*** and ***StringMetricsSerializer***.

The later is intended to serialize tuples to strings in order to facilitate logging. On the other hand, ***BinaryMetricsSerializer*** converts ***MetricsTuple*** instances to a binary representation that is understood by the de-serializing end-point on the MM, implemented by the ***BinaryMetricsDeserializer*** class that we described before. Both the binary serialiser and the de-serialiser can write and read directly to ***Socket*** instances, via Java input and output streams. The framework relies on the Kryo library in order to convert Java objects into a binary representation that can be written directly to a stream. This library is used by SEEP and also other systems related to stream processing, such as Storm and Yahoo's S4 [31].

- ***Reader***: this component reads various metrics from the instrumentation components. The ***MetricsReader*** interface defines the common interface required from concrete implementations. ***DefaultMetricsReader*** is the actual reader implementation that the MS currently uses to obtain various metrics, such as CPU utilisation, heap size and utilisation, etc.
- ***SPS Instrumentation and Operating System Instrumentation***: the role of these two components is to provide instrumentation and to expose to other classes quantities that represent the state of the underlying SPS and operating system (or the JVM in our case too). We rely on a few third-party libraries for the framework's instrumentation needs. In particular, the JVM heap instrumentation is provided by the Coda Hale Metrics library [32], shown as package ***com.codahale.metrics.jvm*** in the diagram below. The CPU utilisation metric is obtained using the Java Management API [33] and some Oracle extensions to it [34].

  Lastly, the SPS instrumentation, to collect metrics such as input-queue length or operator latency, is supported by the ***MetricsNotifier*** interface and the ***DefaultMetricsNotifier*** class. The SPS needs to notify the MS when the operator running locally starts and completes the processing of a tuple, in addition to its arrival and departure from the input queue.

Finally, in Figure 4.21, we present an UML sequence diagram that shows how the classes that participate in the implementation of the MS interact to periodically read metrics from the instrumentation components, serialise these in a format that is appropriate for the MM component and lastly, transmit them over a TCP socket connected to the master. We also show how metric values flow from the instrumentation to the ***MetricsReader*** class, either directly for JVM and operating system metrics or via a ***MetricsNotifier*** subclass for SPS specific metrics.

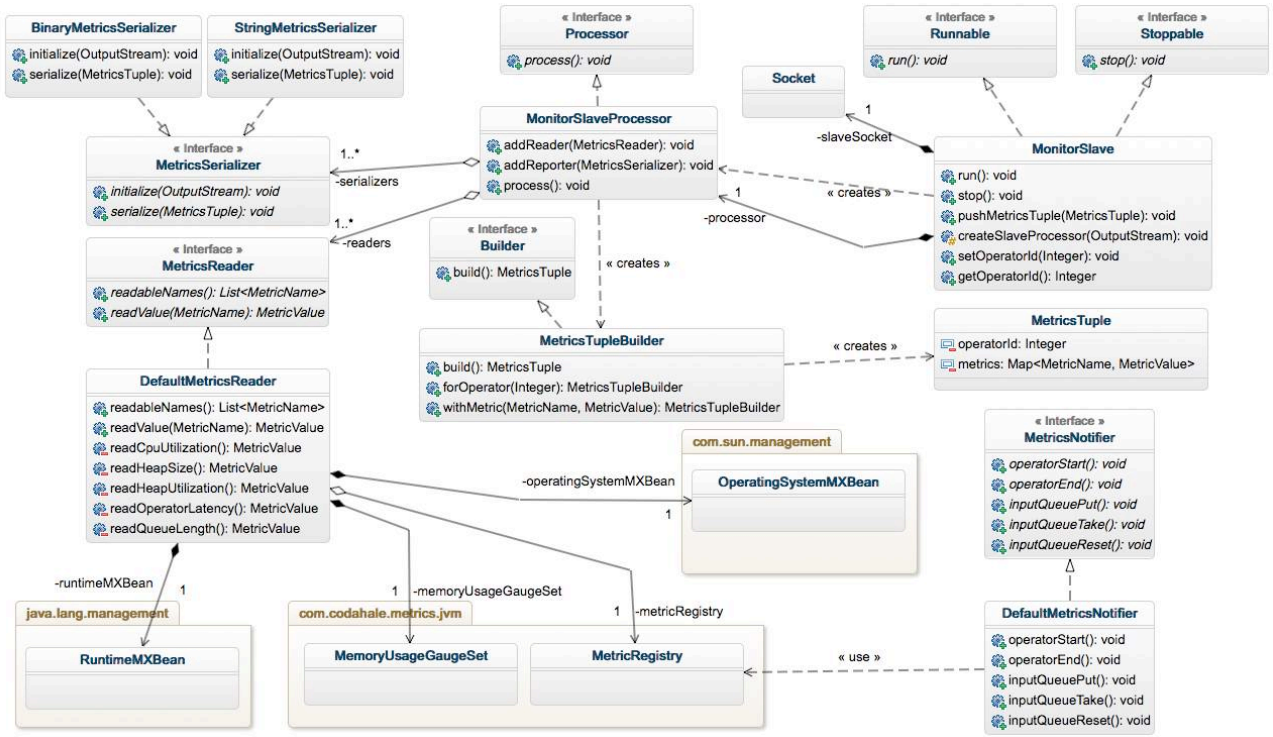

Figure 4.20: UML class diagram for the Monitoring Slave (MS) component

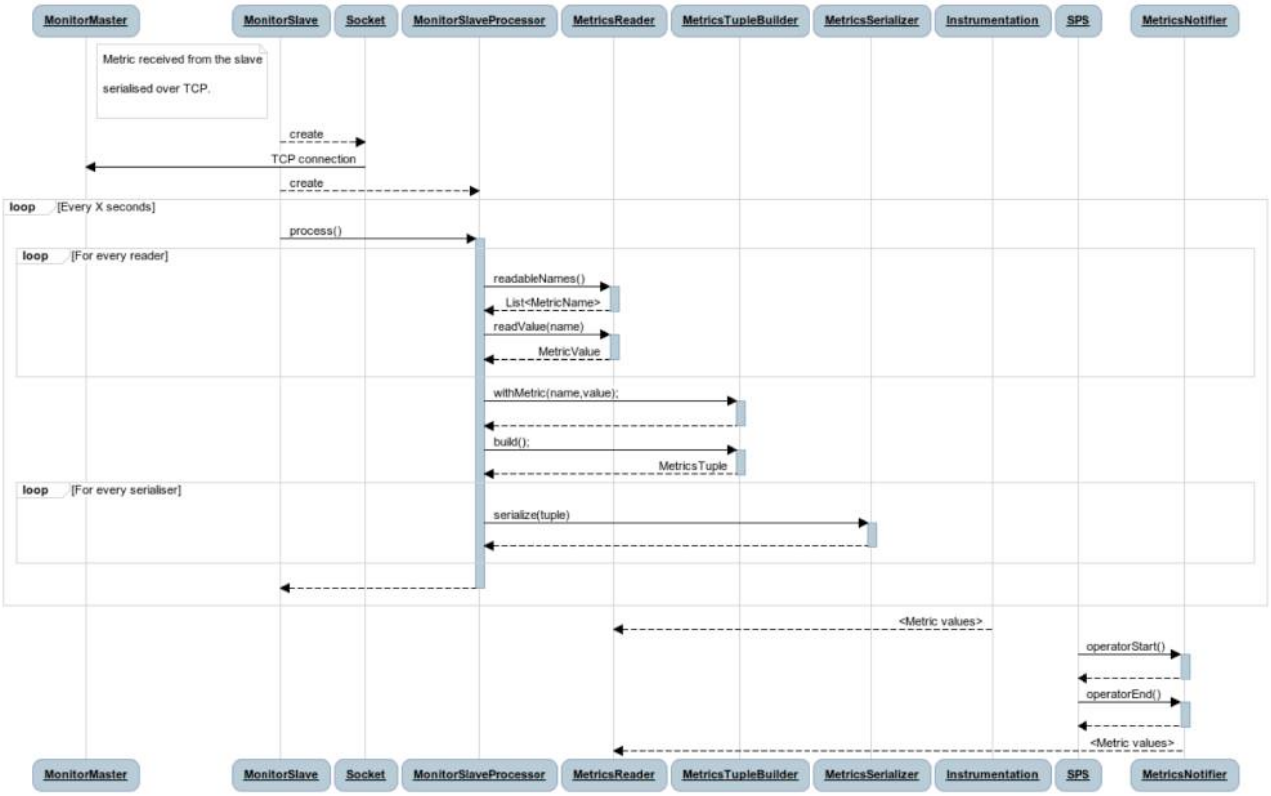

Figure 4.21: UML sequence diagram for the Monitoring Slave (MS) component

4.4.6 Integration with SEEP

So far, we have presented the implementation of the scaling framework adopting an agnostic approach in terms of the supporting SPS. We will now describe the implementation of the integration of our framework with the SEEP system. As mentioned before, SEEP is relevant in the context of our work as it is the concrete SPS that we will use to evaluate the performance of the scaling framework. The result of this work will be presented in the next chapter.

We can broadly divide the integration work into three areas. First, we will explain how the *Monitoring Master* (MM) and *Monitoring Slave* (MS) components of the framework are instantiated and attached to the master and slave processes of SEEP, respectively. In second place, we will present the ***SeepInfrastructureAdaptor*** class which implements the ***InfrastructureAdaptor*** interface, proving the framework with a mechanism to execute scaling actions. Lastly, we will describe how the ***DefaultMetricsNotifier*** is invoked to notify the framework of certain relevant events (e.g.: arrival and departure of tuple, processing of a tuple started, etc). Part of the explanation that follows refers to some SEEP implementation details, its source code is publicly available [35].

The ***MonitorMaster*** and ***MonitorSlave*** classes are created by means of a pair of factory classes, named ***MonitorMasterFactory*** and ***MonitorSlaveFactory*** respectively. This a variation of the abstract factory pattern described in [30]. There are two integration points in SEEP where our monitoring classes above need to be instantiated. The SEEP ***Infrastructure*** class needs to create a ***MonitorMaster*** instance when loading the stream query to execute. Similarly, the ***NodeManager*** class needs to instantiate the ***MonitorSlave*** class when starting an instance of a logical operator on the slave node. In both cases, this is done by means of the respective factory class. The UML class diagram below illustrates this instantiation process.

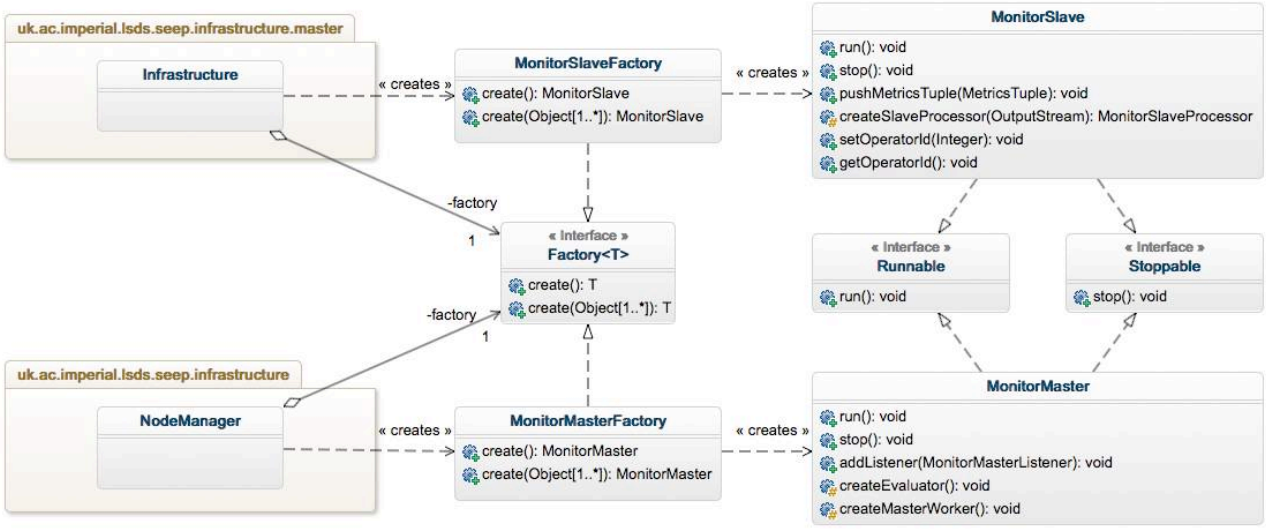

Figure 4.22: UML class diagram of the integration with SEEP (partial)

The ***SeepInfrastructureAdaptor*** exposes a series of methods that are needed by the monitoring framework in order to execute scaling actions. In line with the original intent of the Adapter pattern [30], the main goal of our adaptor is to isolate the framework from SEEP and enable it to use the SEEP interfaces that manage the infrastructure, such as ***Infrastructure*** and ***ElasticInfrastructureUtils***, without requiring any code changes. Obviously, our adaptor needs to smooth out any mismatches between the expectations of the framework and what SEEP can actually deliver. One typical example of this is the scale factor for actions: the framework expects to be able to start X slave instances with a single method call, but the SEEP interfaces only allow for one slave to be started at a time. Some other mismatches require the adaptor to behave as a stateful object, maintaining a valid copy of the state of the SEEP system at any given time.

Internally, both ***Infrastructure*** and ***ElasticInfrastructureUtils*** interact with SEEP components such as the ***Deployment Manager***, the ***VM Pool***, the ***Scale-out Coordinator*** and the ***Recovery Coordinator*** [36]. Consequently, the operations exposed by our adaptor interface are implemented in terms of the lower-level primitives that the SEEP components support, which as mentioned before can be combined to provide fault-tolerant dynamic scale-out of query operators [11].

Finally, various method on the ***DefaultMetricsNotifier*** class need to be invoked by SEEP in order to let the framework know when a tuple arrives for processing and departs the node after being processed. This enables the framework to expose metrics such as input queue length and average operator latency. Unfortunately, SEEP doesn't currently expose an interface for observers to register and observe certain system events. In this case, the integration required certain SEEP classes to be modified to call the required methods explicitly (e.g.: ***InputQueue***, ***StatelessProcessingUnit*** and ***StatefulProcessingUnit***).

4.4.7 Summary

We have presented an overview of the implementation of our scaling framework. We have described the implementation of scaling rules and scaling policies, in addition to the embedded domain-specific language that we propose for their definition. We have also described the implementation of the *Monitoring Master* and *Monitoring Slave* framework components, including UML class and sequence diagrams where necessary. Finally, we have given a brief overview of the integration with the SEEP system.

5 Evaluation and Discussion

This chapter is aimed at evaluating the proposed scaling framework, running on top of a specific SPS such as the SEEP System. As mentioned previously, the evaluation chapter is divided into two parts:

- First, we want to proof that the behaviour of the scaling framework is correct for simple scenarios where the outcome is predictable. The predictability of the results is important in this case, as we want to ascertain the correctness of the scaling decisions taken by the framework. The intention is to confirm that the framework can accurately decide when and how to scale-in and scale-out.
- Secondly, the remaining part of the chapter takes a different approach and aims at testing the framework with real-world data. In order to do so, we created a logical query that extracts analytics from a Google cluster trace and tested it with various scaling policies. The intention in this case is to confirm that the scaling framework does not hamper the scalability of the SEEP system and can actually help in achieving higher throughputs.

5.1 Test Environment

All the tests presented in this section were executed on a cluster of Linux servers with the following hardware and software configuration:

- Quad-Core AMD Opteron™ Processor 2346 HE 1.8GHz.
- 4GB of RAM memory on each cluster node.
- Ubuntu version 12.04.2 LTS, with a Linux kernel version 3.2.0-58-generic-pae.
- OpenJDK Runtime Environment version 1.7.0_55 (IcedTea 2.4.7).

We configured the test system as follows:

- One of the servers executes SEEP as master. This means that any non-distributed parts of SEEP plus the *Monitoring Master* component of our scaling framework run on this server. Furthermore, the logical query to run and its associates scaling policies also reside on this server.
- Multiple servers execute SEEP as slave, together with an instance of our *Monitoring Master* component per server. As described in the architecture chapter before, slaves receive the query to run from the master and periodically report metrics back to it. All slaves automatically register themselves with the SEEP *VM Pool* on startup. Servers will be provisioned from or returned to this pool when dynamically scaling operators up or down, respectively.

See Appendix: Test Environment for further details on how SEEP was configured and executed in the test environment.

5.2 Evaluation with Simulated Streams

In this section, we present the results for various simple tests cases using simulated streams of events. These streams are generated so that they present specific characteristics in terms of shape, throughput, etc; which will eventually trigger certain behaviours of the dynamic scaling framework. We first provide an overview of the testing methodology and a systematic way of presenting and evaluating results. Then, we go through various test cases that verify the correctness of the scaling framework under certain conditions and assumptions. It is worth mentioning that this first phase of testing was completed before attempting any real workloads.

5.2.1 Test plan

We want to define test cases for which we can easily predict what the correct outcome should be and compare that with the actual result. Hence, most of the tests presented here will be based on a simple metric that exists for SEEP query operators, the length of the input queue, measured as the number of tuples queued up on a given operator waiting to be processed. The reason is that this metric is deterministic and, thanks to queueing theory and Little's law, we can predict the expected length of the queue knowing the arrival and departure rate for each operator.

Furthermore, to avoid adding unnecessary complexity to the test cases, we will use test queries based on stateless SEEP operators. Stateful operators will be used later on, when testing the scaling framework in conjunction with SEEP subject a real workload.

As a bare minimum, we aim at validating the framework under the following scenarios:

- Constant arrival rate λ and departure rate μ, $\lambda \leq \mu$. No queueing at processor and hence no scaling.
- Constant arrival rate λ and departure rate μ, but with $\lambda > \mu$. A queue of tuples should build up at the operator. Scaling rules should cause the operator to be scaled out by the framework but only up to a point where $\lambda = \Sigma\mu_i$. This means that the aggregated departure rate for of all instances of the scaled operator matches the arrival rate, causing the length of the input queue to stabilise.
- Constant arrival rate λ and departure rate μ, but with $\lambda > \mu$. A queue of tuples should build up at the operator. Scaling rules should cause the operator to be scaled out by the framework so that $\lambda < \Sigma\mu_i$. This means that the aggregated departure rate for of all instances of the scaled operator (labeled as i in the previous expression) exceeds the arrival rate, causing the length of the input queue to decrease and eventually become 0.
- Periodic arrival rate λ and constant departure rate μ, with $\lambda > \mu$ during peaks. A queue of tuples should build up at the operator during peaks. Scaling rules should cause the operator to be scaled out by the framework during peaks, causing the

length of the input queue to decrease or remain stable at the least. Additionally, the rules can introduce a certain hysteresis the prevent the operator from scaling in immediately after the peak, thus preventing the system from going into a thrashing state where it keeps scaling in and out.

For each test case presented in the following sections, we outline the following aspects:

- ***Intent***: brief description of the purpose of the test and what aspect of the monitoring framework we intend to demonstrate.
- ***Logical query***: a simple diagram and a brief description of the logical query that SEEP will execute and to which scaling rules will be applied by the monitoring framework during the test.
- ***Scaling policy***: Java code snippet presenting the policy rule that will govern scaling during the test. Rather than providing a textual description of the rule, we believe it is better to just show the Java code as the framework's fluent interface is simple to understand and might be more accurate and less vague.
- ***Results***: charts and figures depicting the results of the test.
- ***Conclusion***: brief discussion of the results and aspects of the test that might motivate further testing or exploring different behaviours of the monitoring framework.

Having explained the testing methodology, we can proceed to present the test cases and corresponding test results.

5.2.2 Single Operator Scale-out

Intent

Verify that the monitoring framework is capable of detecting when a particular metric (input queue length) exceeds a threshold for a given operator and execute the associated scaling action.

Logical query

Figure 5.1: Simple query with a single stateless operator

The query is composed of the following operators:

1. *ConstantRateRandomSource*: source that emits random data tuples at a constant parametrisable rate. The emission rate λ will be set to 10 tuples/second.

2. *SimpleStatelessOperator*: stateless operator that performs some computation on tuples, such as swapping components in the tuple or performing arithmetical operations on them. The processing rate μ will be set to 5 tuples/second.
3. *LoggingSink*: sink that that simply logs received tuples to the standard output, without performing any additional computation on them.

Scaling policy

The scaling policy is defined as follows:

```
QueryBuilder.withPolicyRules(
    new PolicyRules() {{
        rule("Queue length is above 300 for 30 seconds")
                .scaleOut(operator("SimpleStatelessOperator", OP_ID_1))
                .by(absolute(1)).butNeverAbove(nodes(2))
                .when(metric("queue-length"))
                .is(above(tuples(300)))
                .forAtLeast(seconds(30))
               .withNoScaleOutSince(minutes(5)).build();
    }});
```

Source Code 5.1: Simple scaling rule based on input queue length

Results

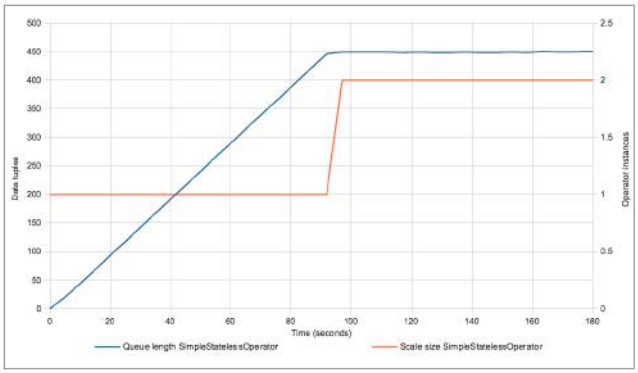

Figure 5.2: Single operator scale-out results.
Input queue length vs. operator scale size.

Conclusion

We can see that approximately 60 seconds into the test, the length of the input queue for the *SimpleStatelessOperator* operator goes above the 300 tuples threshold defined in the scaling policy. The trigger associated to this threshold is fired when the queue length exceeds the threshold for at least 30 seconds. This can be confirmed by the experimental results presented above, as the number of instances allocated to the operator goes from 1 to 2 at about 92 seconds into the test. Thus, at this time, the aggregate processing rate of the query matches the rate of the source, preventing the input queue at *SimpleStatelessOperator* from growing further.

5.2.3 Single Operator Scale-out with Additional Capacity

Intent

Verify that the monitoring framework is capable of detecting when a particular metric (input queue length) exceeds a threshold for a given operator and execute the associated scaling action. The action involves provisioning multiple instances of the operator, in order to increase the query processing capacity and clear the backlog of tuples.

Logical query

The query is composed of the same operators as the previous test. The emission rate λ and the processing rate μ for the *SimpleStatelessOperator* are again set to 10 tuples/second and 5 tuples/second, respectively.

Scaling policy

The scaling policy is defined below. Note this policy rule is similar to the one defined in the previous test with the exception that now the action triggered involves scaling up the *SimpleStatelessOperator* by two instances rather than one. This will ensure that the processing capacity of the query exceeds that of the source.

```
QueryBuilder.withPolicyRules(
    new PolicyRules() {{
        rule("Queue length is above 300 for 30 seconds")
                .scaleOut(operator("SimpleStatelessOperator", OP_ID_1))
                .by(absolute(2)).butNeverAbove(nodes(3))
                .when(metric("queue-length"))
                .is(above(tuples(300)))
                .forAtLeast(seconds(30))
               .withNoScaleOutSince(minutes(5)).build();
    }});
```

Source Code 5.2: Simple scaling rule based on input queue length, with a higher scale-out factor.

Results

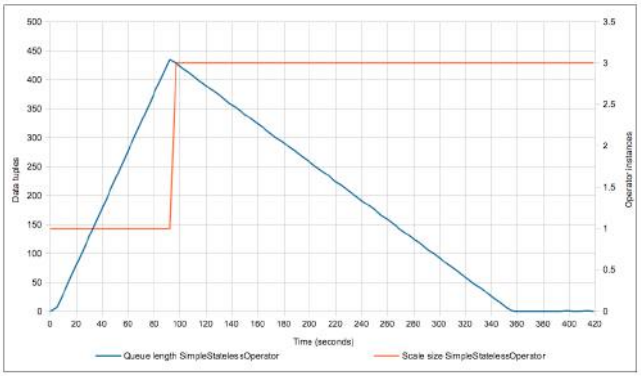

Figure 5.3: Single operator scale-out with additional capacity. Input queue length vs. operator scale size.

Conclusion

As in the previous test, we can see that approximately 60 seconds since the start of the test, the length of the input queue for the *SimpleStatelessOperator* operator goes above the 300 tuples threshold defined in the scaling policy. The trigger will fire when the queue length exceeds the threshold for at least 30 seconds. This can be confirmed by the experimental results presented above, as the number of instances allocated to the operator goes from 1 to 3 at around 90 seconds. From this point on, the aggregated processing capacity of the operator instances becomes 3 times the value of our initial μ, or equivalently 15 tuples/second. Since the rate of the source remains constant, the queue that built up on the initial *SimpleStatelessOperator* instance is progressively emptied.

5.2.4 Multiple Operator Scale-out with Additional Capacity

Intent

Verify that the scaling framework is capable of detecting when a particular metric exceeds a threshold for a given operator in a multi-operator query and execute the associated scaling action. The action involves provisioning multiple instances of the operator, in order to increase the query processing capacity and clear the backlog of tuples. The main goal is to prove that scaling policies can accurately detect bottlenecks in a stream query.

Logical query

Figure 5.4: Simple query with multiple stateless operators

The source and sink operators are similar to the ones described in previous tests, with a constant emission rate λ of 10 tuples/second. The intermediate operators are instances of the same *SimpleStatelessOperator* already used in previous tests. However, these instances are parametrised slightly different:

1. *SimpleStatelessOperator (1)*: stateless operator that performs some simple computation on incoming tuples with a processing rate μ_1 of 20 tuples/second. Note that the arrival rate for this operator is half the departure rate ($2\lambda = \mu_1$). The operator has spare capacity to cope with the incoming stream of tuples.
2. *SimpleStatelessOperator (2)*: this is also a stateless operator that performs some simple computation on incoming tuples, with a processing rate μ_2 of 5 tuples/second. Note that in this case the arrival rate doubles the departure rate for

the operator ($\lambda = 2\mu_2$). The operator doesn't have enough capacity to handle the incoming stream of tuples.

Scaling policy

We define the same scaling policy rules for both operators, although *SimpleStatelessOperator (1)* is not expected to scale during this test given its arrival and departure rates. The rule is simply defined to prove that the operator does not scale because there are no rules defined but because it doesn't need to.

```
QueryBuilder.withPolicyRules(
    new PolicyRules() {{
        rule("Queue length is above 300 for 30 seconds")
            .scaleOut(operator("SimpleStatelessOperator (1)", OP_ID_1))
            .by(absolute(2)).butNeverAbove(nodes(3))
            .when(metric("queue-length"))
            .is(above(tuples(300)))
            .forAtLeast(seconds(30))
            .withNoScaleOutSince(minutes(5)).build();

        rule("Queue length is above 300 for 30 seconds")
            .scaleOut(operator("SimpleStatelessOperator (2)", OP_ID_2))
            .by(absolute(2)).butNeverAbove(nodes(3))
            .when(metric("queue-length"))
            .is(above(tuples(300)))
            .forAtLeast(seconds(30))
            .withNoScaleOutSince(minutes(5)).build();
    }});
```

Source Code 5.3: Simple scaling rule based on input queue length, with a higher scale-out factor.

Results

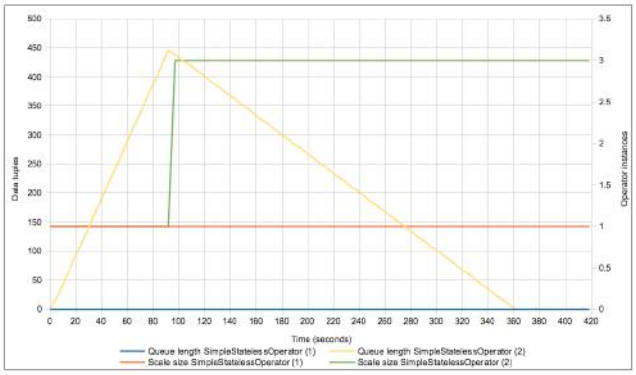

Figure 5.5: Multiple operator scale-out with additional capacity. Input queue length vs. operator scale size.

Conclusion

The operator *SimpleStatelessOperator (1)* does not scale as it shows a processing rate that is above the arrival rate from the random source. In contrast, *SimpleStatelessOperator (2)* does scale out as its processing capability is half the arrival rate from its upstream

operator. Again, approximately after 90 seconds into the test, two additional instances are provisioned for this operator, which makes its aggregated departure rate equal to 15 tuples/second. This exceeds the rate of the source and prevents a queue from continuing to build up on this operator. In fact, the additional capacity enables the operator to eventually clear its backlog of queued tuples.

5.2.5 Single Operator with Periodic Source and no Scale-out

Intent

Verify the behaviour of the scaling framework in terms of its responsiveness to the periodicity of the input work load. As usual, the scaling policy will define how long a metric needs to be exceeded before triggering a scaling action. In this test, the source will ensure that tuples are emitted at different rates, with the peak emission rate λ exceeding the aggregated processing capacity of the query and thus causing queuing to happen.

Logical query

Figure 5.6: Simple query with a periodic random source

The stream query used in this test is the same as the ones from previous test cases. The main difference is the source operator, which is now a *PeriodicRandomSource* instead of *ConstantRateRandomSource*. The new source emits tuples at two rates: a lower base rate and a higher peak rate. The source will introduce higher rate peaks of fixed or random duration on top of the base rate, at either a fixed or a random frequency.

Particularly for this case, we will use random durations for peaks but constrained in such a way that the thresholds defined in the scaling policy are never exceeded and thus, scaling actions are not triggered. The goal of this test if to prove that no scale-out will take place for transient peaks (albeit rules are expressed indirectly, using the input queue length on *SimpleStatelessOperator* as a proxy).

Thus, the base emission rate λ and the peak rate λ_p for the *PeriodicRandomSource* are set to 10 tuples/second and 20 tuples/second respectively. The processing rate μ for the *SimpleStatelessOperator* operator is 10 tuples/second. This means that when the source is emitting at the base rate, the query capacity matches the rate of the source operator.

Scaling policy

The scaling policy is defined as follows:

```
QueryBuilder.withPolicyRules(
    new PolicyRules() {{
        rule("Queue length is above 10000 for 60 seconds")
                .scaleOut(operator("SimpleStatelessOperator", OP_ID_1))
                .by(absolute(2)).butNeverAbove(nodes(3))
                .when(metric("queue-length"))
                .is(above(tuples(10000)))
                .forAtLeast(seconds(60)).build();
    }});
```

Source Code 5.4: Simple scaling rule based on input queue length, with a very high threshold

Results

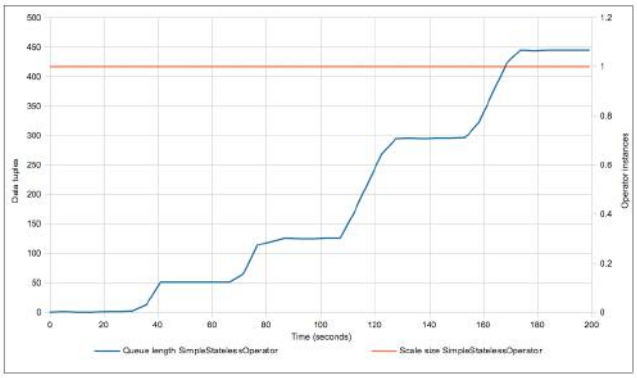

Figure 5.7: Single operator with periodic source. Input queue length vs. operator scale size.

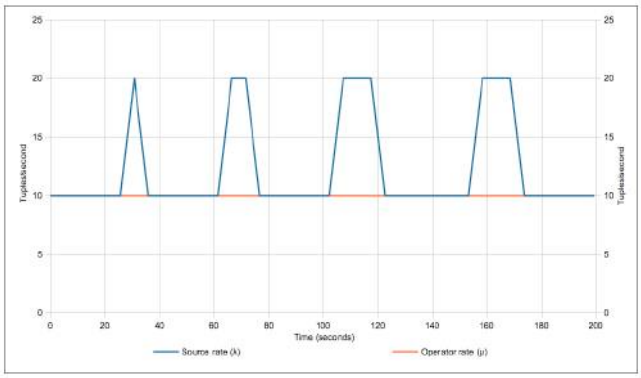

Figure 5.8: Single operator with periodic source. Periodic source rate vs. operator rate.

Conclusion

We can observe that, whenever there is a peak on the emission rate λ for the *PeriodicRandomSource* operator, the length of the input queue on *SimpleStatelessOperator* increases as expected. If the source is working at the peak rate of 20 tuples/second and *SimpleStatelessOperator* can only handle 10 tuples/second, then for every second spent at peak rate 10 tuples are queued up. The first peak lasts for roughly 5 seconds (although this is not seen clearly in the graph as the sampling resolution is exactly 5 seconds). This corresponds with the behaviour observed for the input queue on the stateless operator, with its length jumping from 0 to 50 tuples during the same period of time. The same can be said for subsequent peaks during the length of this test.

No scale-out happens given that the threshold is set sufficiently high so that it is never exceeded during the test.

5.2.6 Single Operator Scale-out and Scale-in with Periodic Source

Intent

Verify the behaviour of the scaling framework in terms of its responsiveness to the periodicity of the input work load. As in the previous test, the source will ensure that tuples are emitted at different rates, with the peak emission rate λ exceeding the aggregated processing capacity of the query and thus causing queuing to happen. The peaks will last long enough to ensure that the length of the input queue exceeds the threshold defined in the corresponding scaling rules. Additionally, the test will also verify that the operator is scaled-in once the backlog of tuples is cleared, thus allowing the operator size to follow the input work load quite closely.

Logical query

The query is composed of the same operators as the previous test, again with a periodic random source instead of a constant one. Thus, the *PeriodicRandomSource* operator is configured as follows:

- Base emission rate λ of 10 tuples/second.
- Peak emission rate λ_p of 20 tuples/second.
- Peak duration of 20 seconds.
- Trough duration of 40 seconds.

The processing rate μ for the *SimpleStatelessOperator* is set to 10 tuples/second, matching the base rate for the periodic source.

Scaling policy

The scaling policy defines two rules that determine when to scale-out and when to scale-in the *SimpleStatelessOperator* operator.

```
QueryBuilder.withPolicyRules(
    new PolicyRules() {{

        rule("Queue length is above 100 for 5 seconds")
                .scaleOut(operator("SimpleStatelessOperator", OP_ID))
                .by(absolute(2)).butNeverAbove(nodes(3))
                .when(metric("queue-length"))
                .is(above(tuples(100)))
                .forAtLeast(seconds(5)).build();

        rule("Queue length is below 1 for 5 seconds")
                .scaleIn(operator("SimpleStatelessOperator", OP_ID))
                .by(absolute(2)).butNeverBelow(nodes(1))
                .when(metric("queue-length"))
                .is(below(tuples(1)))
                .forAtLeast(seconds(5)).build();

    }});
```

Source Code 5.5: Scale-out and scale-in policy based on input queue length.

Results

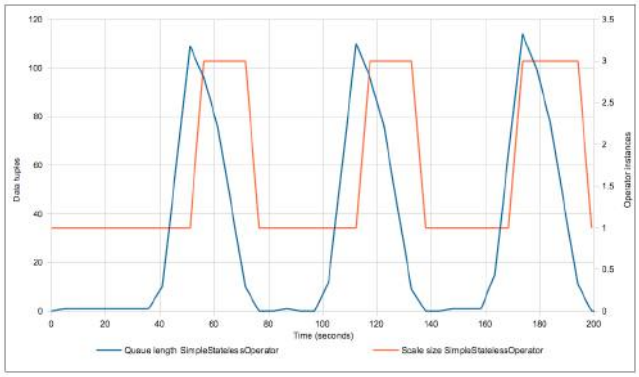

Figure 5.9: Single operator with periodic source. Input queue length vs. operator scale size.

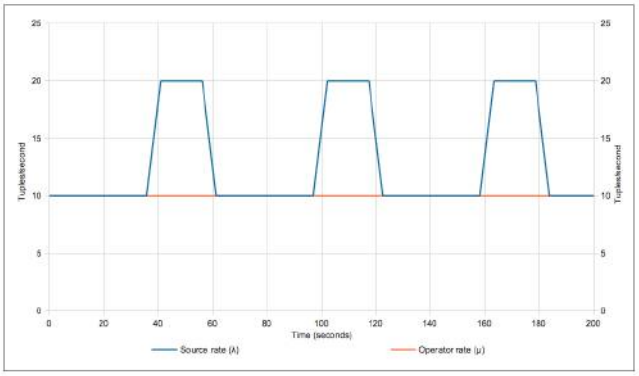

Figure 5.10: Single operator with periodic source. Periodic source rate vs. operator rate.

Conclusion

We can observe that when the *PeriodicRandomSource* operator emits tuples at the peak rate λ_p, the length of the input queue starts to grow on the *SimpleStatelessOperator* operator. Given that the difference between the arrival rate and the departure rate for *SimpleStatelessOperator* during a peak is 10 tuples/seconds, after roughly 10 seconds the length of the input queue exceeds the threshold of 100 tuples defined in the scaling rules. At this time, the operator is scaled-out from 1 to 3 instances (see *Figure 5.9* above), hence causing the aggregated processing capacity of the query to be 30 tuples/second. Since this rate is significantly above the emission rate, even during a peak, the backlog of tuples in *SimpleStatelessOperator* is progressively cleared.

We can also confirm that once the backlog of tuples is cleared and the length of the input queue drops below the threshold defined for the scale-in rule (e.g.: less than 1 tuple for at least 5 seconds and no scale-out action during the last 60 seconds), the *SimpleStatelessOperator* is scaled down from 3 back to 1 single instance.

In conclusion, the framework allows defining rules that enable queries to dynamically adjust their processing capacity up or down, as necessary, to follow closely the input work load. This particularly convenient to deal with workloads that exhibit certain periodicity or transient spikes in load that needed additional capacity but only for a short period of time.

5.2.7 Single Operator Scale-out and Scale-in hysteresis with Periodic Source

Intent

Similarly to the previous test, we want to verify the behaviour of the scaling framework in terms of its responsiveness to the periodicity of the input work load. However, in this case, we want to check that the introduction of certain hysteresis to the scaling rules will prevent the framework from triggering a scale-in action after a peak in the input work load. Hysteresis will be expressed as a guard time after a scale-out action during which no scale-

in actions are allowed. The ability to scale-out quickly and be more reluctant to scale-in is relevant when working on environments where the provisioning of hosts for operators to run is expensive. Overall, in such an environment, we would want to avoid incurring in additional costs by scaling in when it is likely that another peak will occur after the peak that caused the current scale-out.

Logical query

The query is composed of the same operators as in the previous test. The various emission rates and other parameters for the *PeriodicRandomSource* operator remain the same. The processing rate μ for the *SimpleStatelessOperator* operator is again set to 10 tuples/second.

Scaling policy

As before, the scaling policy defines two rules that determine when to scale-out and when to scale-in the *SimpleStatelessOperator* operator. Note that the scale-in rule defines a guard time of at least 5 minutes since the last scale-out before permitting any scale-in action. This provides the rule with hysteresis: once the operator is scaled to 3 instances, the framework will be more reluctant to scale-in. It is not enough with the backlog of tuples being cleared from the input queue for the rule to be triggered, it is also required that the last scale-out took place at least 5 minutes ago.

```
QueryBuilder.withPolicyRules(
    new PolicyRules() {{

        rule("Queue length is above 100 for 5 seconds")
                .scaleOut(operator("SimpleStatelessOperator", OP_ID))
                .by(absolute(2)).butNeverAbove(nodes(3))
                .when(metric("queue-length"))
                .is(above(tuples(100)))
                .forAtLeast(seconds(5)).build();

        rule("Queue length is below 1 for 5 seconds")
                .scaleIn(operator("SimpleStatelessOperator", OP_ID))
                .by(absolute(2)).butNeverBelow(nodes(1))
                .when(metric("queue-length"))
                .is(below(tuples(1)))
                .forAtLeast(seconds(5))
                .withNoScaleOutSince(minutes(5)).build();
    }});
```

Source Code 5.6: Scale-out and scale-in policy based on input queue length.

Results

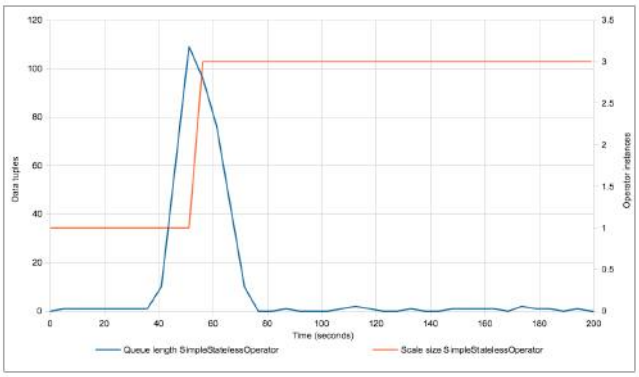

Figure 5.11: Single operator with periodic source and hysteresis. Input queue length vs. operator scale size.

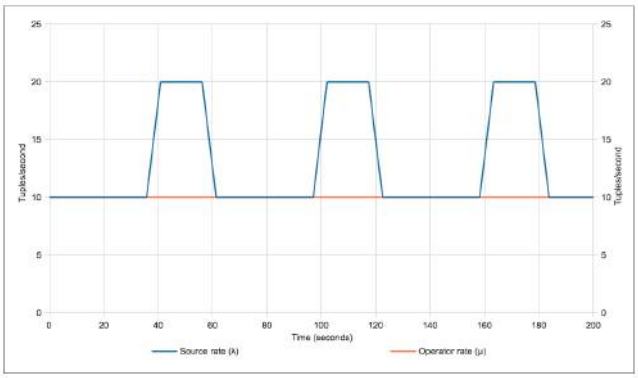

Figure 5.12: Single operator with periodic source and hysteresis. Periodic source rate vs. operator rate.

Conclusion

As in the last test, we can observe that when the *PeriodicRandomSource* operator emits tuples at the peak rate λ_p, the length of the input queue starts to grow on the *SimpleStatelessOperator* operator. Given the differences between the arrival and departure rates for this operator, after roughly 10 seconds the length of the input queue exceeds the threshold of 100 tuples defined in the scaling rules. As expected, at this time, the operator is scaled-out from 1 to 3 instances. Since the aggregated processing capacity of the query becomes 30 tuples/second, the backlog of tuples in *SimpleStatelessOperator* is progressively cleared.

In this test, however, even though the input queue length drops below the threshold defined for the scale-in rule, the size of the operator remains unchanged and no scaling actions take place. This is due to the guard time of 5 minutes introduced for the scale-in rule, which makes the system reluctant to reduce the number of instances immediately after a peak in the input work load. The logic behind this behaviour is that, under certain conditions, it might actually be less costly to continue running with an unnecessarily high number of operator instances rather than scaling in and out again in a short period of time.

To summarise, the framework allows defining rules that enable queries to dynamically adjust their processing capacity up or down, as necessary. Simultaneously, the framework provides mechanisms to introduce certain hysteresis after scaling actions, in the form of guard time periods during which no scaling actions are allowed.

5.2.8 Summary

We have presented a set of simple use cases that demonstrate the behaviour of our scaling framework, showing how different scaling actions are executed in response to a changing system metric. Particularly, we have used simulated streams of tuples that highlight different traits of real-world workloads (tuples arriving in bursts, workload periodicity, etc), coupled with scaling policies based on operator queue length. We chose to use this metric

for these tests given that it behaves deterministically for constant operator processing delays. Thus, we can easily verify and assert the correct behaviour of our proposed scaling framework.

5.3 Evaluation with Real Streams

In contrast with the previous section where we used simulated streams of events, we now intend to evaluate our scaling framework on the SEEP system using real-world workloads, processing streams of events originated from real systems. In particular, as mentioned before, we have decided to use job and task scheduling traces provided by Google for one of its computing clusters. We will create a logical query that derives some analytical information from these traces, representing the state of the cluster at various moments in time.

We will first run a series of tests where we scale the logical query almost statically. We achieve this by means of a simple rule that scales-out a stateful SEEP operator as quickly as possible to its maximum intended size. Once the operator reaches this state, it remains in its final state for the duration of the test. We repeat the test with several maximum sizes for the aforementioned operator, always replaying trace events as fast as possible. The ultimate goal of this approach is two folded. First, we want to learn which operators in our query constitute bottlenecks and can potentially reduce the overall throughput and scalability of the test query. This will allow us to define effective rules for dynamic scaling later on. Secondly, we want to demonstrate that the addition of our scaling framework to the SEEP system does not hamper the scalability of queries.

The second and final set of tests consists in defining scaling rules that allow for more dynamic scale-out and scale-in of query operators. In contrast with the previous set of tests, trace events in this case will be played back in accelerated real-time. Thus, we can preserve the shape of the workload and its dynamic profile, but putting a higher load on the system by proportionally compressing time between trace events. In conclusion, these final tests will enable us to observe the behaviour of the system in the presence of a dynamic workload from a real system and assert certain aspects of the scaling framework that have not been analysed in detail in this project up to this point.

5.3.1 Google Cluster Traces

In order to understand the meaning of cluster in the context of this test and what streams of events can be obtained from a cluster trace, we can use the definitions from [37]:

- A **Google cluster** is a set of machines, packed into racks, and connected by a high-bandwidth cluster network.
- A **cell** is a set of machines, typically all in a single cluster, that share a common cluster-management system that allocates work to machines.

- Work arrives at a cell in the form of **jobs**.
- A job is comprised of one or more **tasks**, each of which is accompanied by a set of resource requirements used for scheduling the tasks onto machines. Each task represents a Linux program, possibly consisting of multiple processes, to be run on a single machine.

A single usage trace typically describes several days of the workload on one of these compute cells. A trace is made up of several **datasets**. A dataset contains a single **table**, indexed by a primary key that typically includes a timestamp. Each dataset is packaged as a set of one or more files, each provided in a compressed CSV format [37].

In particular for this test, we have obtained a cluster trace that provides job and task scheduling data from a computing cell made out of approximately 12000 machines. The trace events were collected over a period of about a month during May 2011. This cluster trace is publicly available for research purposes and can be downloaded from the *Google Code* website. However, some of the data in the trace tables, such as usernames or machine names, have been obfuscated for confidentiality reasons. Obfuscation techniques include random hashing, linear scaling and normalisation of quantities, all of which is described in detail in [37].

The trace contains the following datasets:

- ***Machine Events***: events that indicate when a machine is added to the cluster or removed from it (due to failure or maintenance). There are also events that indicate when machines are updated (change in CPU or memory resources).
- ***Machine Attributes***: key-value pairs representing machine properties such as kernel version, clock speed, external IP address allocated, etc. Attribute names and values in this table are obfuscated.
- ***Job Events***: events related to jobs and their life cycles.
- ***Task Events***: events related to tasks and their life cycles.
- ***Task Constraints***: resource requirements for the tasks submitted to the cluster for execution. These are effectively placement constraints that restrict the machines on which the tasks can run.
- ***Resource Usage***: resource utilisation measurements from Linux containers, reported at periodic intervals (typically every 300 seconds although occasionally can be less time).

For the tests to be conducted on our scaling framework running on top of SEEP, we could potentially choose any of the datasets included in the Google cluster trace as the source for our stream of events. However, there are certain requirements that the dataset must meet in order to be suitable for our purposes:

1. The candidate dataset must include temporal information. Given that we want to

generate a stream of events from the dataset and therefore, we require each row to have a timestamp.

2. The dataset should be significantly large and consists of several million events taking place over the entire duration of the trace (we require high volume data). Processing the dataset in a sequential manner should not be trivial.

3. Ideally, we would like events to happen in the dataset at rates sufficiently high to represent a challenge for SEEP. In other words, we want to the stream to load the system in order to be able to define scaling rules and trigger them easily.

4. We intend to test the dynamic nature of the scaling framework. We want to exercise its ability to adjust the physical query, by means of scale-out and scale-in actions, depending on the incoming stream of events. Therefore, we would like the events in the dataset to exhibit some periodicity and the rate at which these are recorded in the trace should vary over time, preferably in a none predictable way.

All the datasets included in the trace meet the first requirement, as all the tables are indexed by a primary key that includes a timestamp (all table definitions are available in [37]). The second requirement rules out the *Machine Events* table, since it only consists of 37780 rows which can be processed sequentially in a relatively short period of time. Similarly, the same requirement rules out the *Machine Attributes* table as it contains 10748566 events, but roughly 6% of those occurring even before the trace starts (i.e.: the timestamps for these events is always 0, which indicates an event that occurred before the beginning of the trace window [37]). Finally, the fourth requirement is contrary to the nature of the ***Resource Usage*** table, as events in this table table place at regular intervals making

The group of potential candidates is then reduced to *Job Events*, *Task Events* and *Task Constraints*. Furthermore, in order for an event to exist in this last table expressing a placement constraint for a task, there has to be a corresponding cluster submission or task update event in the parent task table [37]. Therefore, *Task Events* will always have the same number or more events than *Task Constraints*. Thus, we can also discard the constraints table and focus on the main job and task event datasets. We can then compare the two remaining candidate datasets:

	Job Events	*Task Events*
Total events	*1,186,101*	*144,328,173*
Median rate (events/second)	*0.42*	*31.6*
75^{th} Percentile rate (events/second)	*0.60*	*62.8*
90^{th} Percentile rate (events/second)	*0.80*	*116.1*
95^{th} Percentile rate (events/second)	*0.95*	*179.5*

Figure 5.13: Comparison of Job and Task Events datasets from a Google cluster trace

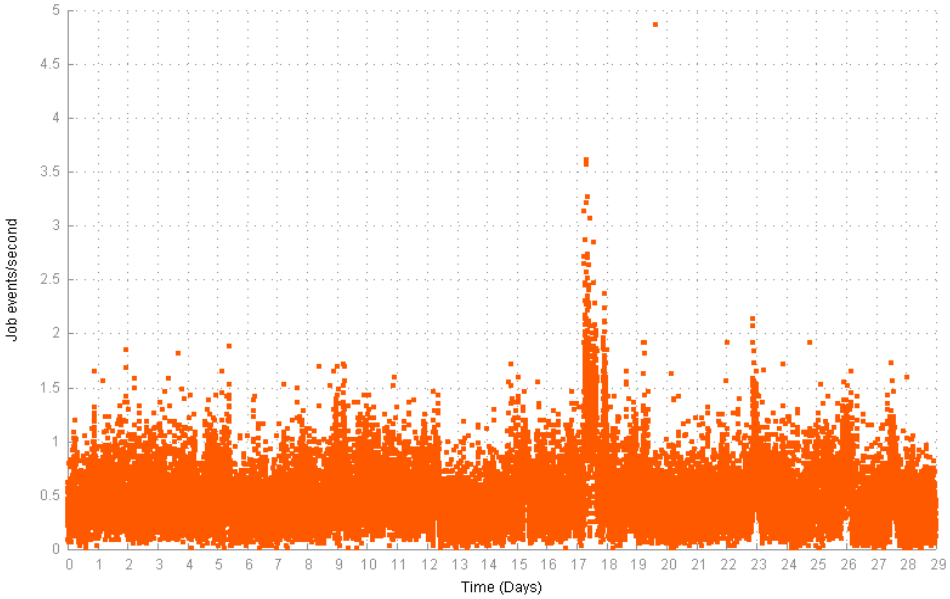

Figure 5.14: Job event rate for a month-long Google cluster trace

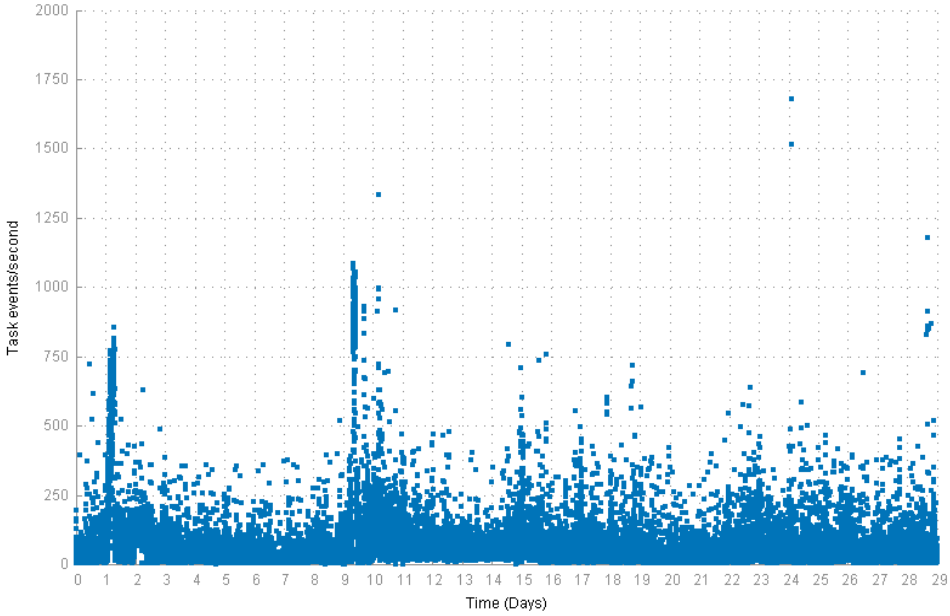

Figure 5.15: Task event rate for a month-long Google cluster trace

Each job submitted to the cluster typically consists of multiple tasks. Tasks are therefore finer-grained entities compared with coarser jobs. It can be expected that there will be more cluster events associated to tasks than jobs, which is confirmed by the graphs and tables above. Moreover, the rate at which task events occur is significantly higher than that observed for jobs, at least two orders of magnitude.

In conclusion, the best fit for our set of requirements on the event stream is the *Task Events* dataset. In the next section, we will present a SEEP query that is capable of deriving useful metrics and analytics from this dataset, consisting mainly of stateful operators. We will describe in detail the implementation of each operator, in particular the source operator that reads compressed comma-separated values (CSV) files from the Google cluster trace and transforms them into an event stream, by re-playing events in real-time or accelerated real-time.

5.3.2 Task Events Dataset

We will build a query that processes the stream of events contained in the Task Events table. This table is made of events related to tasks and their lifecycle, such as an event indicating that a task has been submitted to the cluster or that a particular task has been scheduled for execution. These are the fields contained in this table, as defined in [37], also listed here in the exact order in which they appear in the CSV files:

1. ***Timestamp***: 64-bit integer representing time in microseconds elapsed since 600 seconds before the beginning of the trace period. Hence, all timestamps need to be adjusted by an offset of 600 seconds (an event that happened at T = 20 seconds has a timestamp of 620 seconds in the trace files).
2. ***Missing information***
3. ***Job identifier***: 64-bit integer, this is the unique identifier for the job.
4. ***Task index***: this is a 0-based index for the task within the job. Therefore, tasks can be uniquely identified by the tuple *(job identifier, task index)*. These unique identifiers can be reused if a task is stopped and then re-started at a later time.
5. ***Machine identifier***: 64-bit integer, this is the unique identifier for the machine. Not every event has a machine identifier, as tasks might have not been yet assigned to a machine in the cluster. We will describe the lifecycle of tasks in detail below.
6. ***Event type***: small integer indicating the type of the event. An event is recorded to the cluster trace whenever there is a change in the state of a task. The event type indicates the type of state transition that took place. We will describe the lifecycle for tasks in the cluster below.
7. ***Username***: name of the engineer or the service that started the job to which this task belongs. This is likely to be the same for all tasks spawned by the same job.
8. ***Scheduling class***: integer value indicating how latency-sensitive the task is. Higher values are assigned to sensitive tasks, i.e.: revenue generating tasks. These tasks have preference for local resources over others running on the same machine.
9. ***Priority***: small integer indicating the priority of the task, with 0 representing the lowest possible priority. This value determines preference for resources and some priorities might prevent the task from being evicted by the scheduler when there is contention. Priority determines whether a task is scheduled on a machine.
10. ***Resource request for CPU cores***: normalised value for the task's CPU requirements (with 1 representing all the CPU cores available on the machine).
11. ***Resource request for RAM***: normalised value for the task's memory requirements (with 1 representing all the RAM memory available on the machine).
12. ***Resource request for local disk space***: normalised value for the task's local disk requirements (with 1 representing all the disk space available on the machine).

13. ***Different machine constraint***: special type of constraint that prevents the task from running on the same machine as any other tasks that are part of the same job.

The type of an event is given by a state transition affecting a task. Mainly, there are two types of events: those affecting the scheduling state (e.g., a task gets scheduled and its state changes to "running"), and ones that reflect state changes originating in the task itself (e.g., the task finishes executing normally and its state changes to "dead"). Depending on the test query we want to implement, we might need to treat events different depending on their type. We reproduce the simplified state diagram from [37], showing the lifecycle of a task where event types are the actual names of the allowed state transitions. Furthermore, we also include a brief description of each state transition representing an event type.

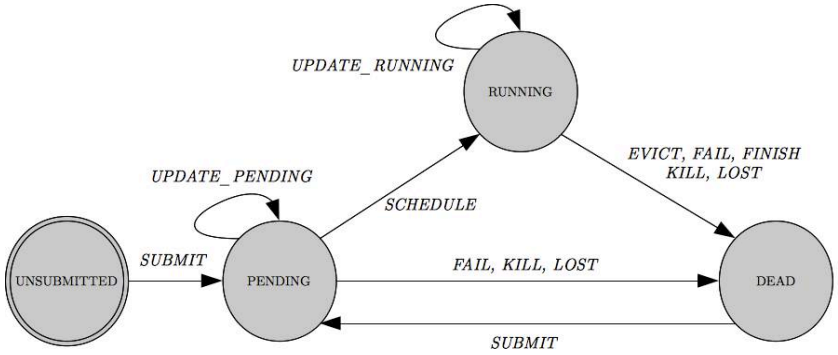

Figure 5.16: State diagram showing the lifecycle of a task in a Google cluster

Event Type	*Value*	*Description*
SUBMIT	*0*	The task becomes eligible for scheduling.
SCHEDULE	*1*	The task is scheduled to run on a machine.
EVICT	*2*	The task was de-scheduled because a higher priority task took precedence or the scheduler over-allocated the machine and resource requirements exceeded availability, etc.
FAIL	*3*	The task was de-scheduled due to a task failure.
FINISH	*4*	The task finished execution normally.
KILL	*5*	The task was cancelled by the user or a driver program.
LOST	*6*	The task was presumably terminated.
UPDATE_PENDING	*7*	The task's scheduling class, resource requirements or constraints were updated before the task being scheduled.
UPDATE_RUNNING	*8*	The task's scheduling class, resource requirements or constraints were updated once the task was scheduled and already running.

Figure 5.17: Event types for a task in a Google cluster trace

5.3.3 Test Query

Considering the data available in the selected table and the different event types, we intend our test query to produce the following statistics at regular intervals:

- Number of tasks running per machine.

- Aggregated CPU requirements for all tasks per machine.
- Aggregated memory requirements for all tasks per machine.
- Aggregated local storage requirements for all tasks per machine.
- Total number of tasks running at cluster level.
- Average, median, 75^{th}, 90^{th} and 95^{th} percentile for the number of tasks per machine, across the cluster.
- Average, median, 75^{th}, 90^{th} and 95^{th} percentile for CPU requirements per machine, across the cluster.
- Average, median, 75^{th}, 90^{th} and 95^{th} percentile for memory requirements per machine, across the cluster.
- Average, median, 75^{th}, 90^{th} and 95^{th} percentile for local storage requirements per machine, across the cluster.

Some of these statistics can be obtained on a per-machine basis, however others require the entire cluster state to be known. For instance, in order to calculate the median number of tasks running per machine across the entire cluster, we need to know how many tasks are running on each machine at a given time. This indicates that the query will need to store some state for each machine on the cluster and possibly for the cluster as a whole.

Following an approach similar to that described in [38], our test query can be structured around to SEEP stateful operators: an operator holding the state for each machine (capable of tracking the number of running tasks plus aggregated amount of CPU, memory and I/O resources per machine) and a global operator that is aware of the state of all the machines in the cluster (capable of calculating medians and percentiles for different quantities across the cluster). We will name the first operator *GoogleTaskMachineOperator* and the later *GoogleTaskClusterOperator*. In addition to these two operators, we will also need a source capable of streaming the trace files to the other operators and a logging sink, that periodically outputs the calculated statistics to an output file. Thus, our test query would be composed of these operators:

- *GoogleTaskLocalSource*: this is a stateless SEEP operator acting as the source of events for our test query. The operator reads a row from one of the CSV files in the Google cluster trace, parses it and packs it as a SEEP tuple, before forwarding it to the downstream operator. The operator can replay events in real-time, calculating the time difference between events from their timestamps and waiting before forwarding the next tuple. Additionally, the operator supports playback of events in accelerated real-time with an acceleration factor between 1 and 100. If this factor is above 100, then the operator simply streams tuples as fast as the downstream operator can accept them.

  Only events that refer to a task running on a machine are relevant to the statistics we want to produce. Therefore, we apply semantic load-shedding similar to that

described in [38], aiming to reduce load on downstream operators and increase the overall throughput of our test query.

- *GoogleTaskMachineOperator*: stateful SEEP operator that keeps track of the tasks running on each machine, aggregating resource requests and applying task events to the machine's state. This operator's state consists of a data structure indexable by machine identifier, the operator only needs to know the state of one machine in order to produce results and send tuples to downstream operators. Hence, its state can be partitioned by machine.
- *GoogleTaskClusterOperator*: stateful SEEP operator that calculates global statistics by processing the state received from its upstream operator for each machine. As it produces global cluster-wide statistics, the operator's state spans the entire cluster and it can't be partitioned spatially (i.e.: by machine) nor temporally (i.e.: the operator does not keep a history of past events, as all past events are summarised in the current state of each machine by the *GoogleTaskMachineOperator* operator).
- *GoogleTaskLoggingSink*: this is also a stateless SEEP operator acting as the sink for our test query. It receives tuples from the upstream operator and writes out results at periodic intervals, either for the cluster or for particular machines. In our tests below, the operator will output results every 15 minutes of trace time. The operator determines how much trace time has elapsed between two tuples by comparing the trace timestamps attached to them. In total, it will produce 96 results per trace day per machine, so for the entire file this amounts to 2,784 results per machine.

The operators are connected sequentially as shown in the logical query below:

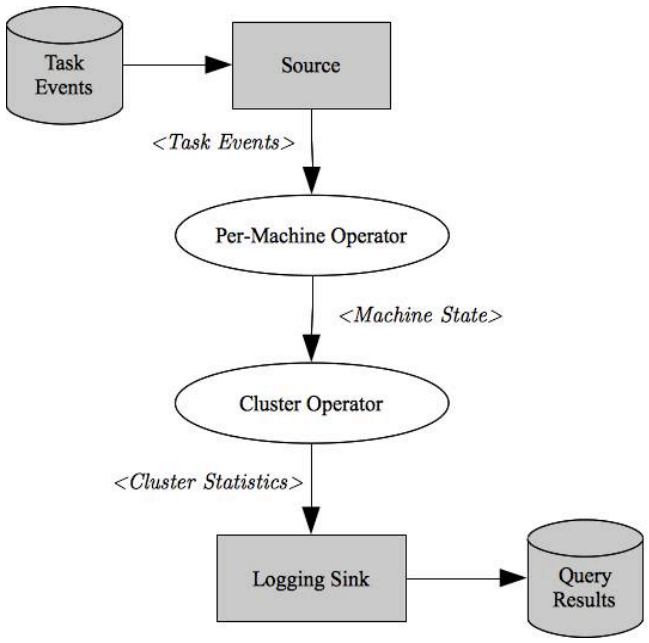

Figure 5.18: Diagram showing the logical query and its operators

Finally, the Java source code for the logical query is:

```
// Declare Source
List<String> srcFields = new ArrayList<String>();
Connectable src = QueryBuilder.newStatelessSource(
    new GoogleTaskLocalSource(tracePath, Integer.MAX_VALUE,
       true), OP_ID_SRC, srcFields);

// Declare per-machine operator
StateWrapper machineStateWrapper = new PerMachineStateWrapper(
    OP_ID_MACHINE, 30000, new PerMachineStateWrapper.StateImpl());

List<String> op1Fields = new ArrayList<String>();
op1Fields.add("timestamp");
op1Fields.add("jobId");
op1Fields.add("taskIndex");
op1Fields.add("machineId");
op1Fields.add("eventType");
op1Fields.add("userName");
op1Fields.add("schedulingClass");
op1Fields.add("priority");
op1Fields.add("resourceReqCpu");
op1Fields.add("resourceReqMemory");
op1Fields.add("resourceReqDisk");
op1Fields.add("machineConstraint");
Connectable op1 = QueryBuilder.newStatefulOperator(
            new GoogleTaskMachineOperator(machineStateWrapper),
            OP_ID_MACHINE, machineStateWrapper, op1Fields);

// Declare cluster operator
StateWrapper clusterStateWrapper = new ClusterStateWrapper(
    OP_ID_CLUSTER, 30000, new ClusterStateWrapper.StateImpl());

List<String> op2Fields = new ArrayList<String>();
op2Fields.add("machineId");
op2Fields.add("timestamp");
op2Fields.add("numTasks");
op2Fields.add("resourceReqCpu");
op2Fields.add("resourceReqMemory");
op2Fields.add("resourceReqDisk");

Connectable op2 = QueryBuilder.newStatefulOperator(
            new GoogleTaskClusterOperator(clusterStateWrapper),
            OP_ID_CLUSTER, clusterStateWrapper, op2Fields);

// Declare sink
List<String> snkFields = new ArrayList<String>();
snkFields.add("timestamp");
snkFields.add("totalTasks");

snkFields.add("averageCpu");
snkFields.add("medianCpu");
snkFields.add("percentile75Cpu");
snkFields.add("percentile90Cpu");
snkFields.add("percentile95Cpu");
snkFields.add("percentile99Cpu");
...
snkFields.add("machineId");
snkFields.add("numTasks");
snkFields.add("resourceReqCpu");
snkFields.add("resourceReqMemory");
snkFields.add("resourceReqDisk");

Connectable snk = QueryBuilder.newStatelessSink(
            new GoogleTaskLoggingSink(), OP_ID_SINK, snkFields);

// Connect operators
src.connectTo(op1, true, 0);
op1.connectTo(op2, true, 0);
op2.connectTo(snk, true, 0);

return QueryBuilder.build();
```

Source Code 5.7: Source code for SEEP query to process a Google cluster trace.

We utilised some third party libraries to implement the operators and the test query described here. In particular, *GoogleTaskLocalSource* uses the Commons Compress [39] and Commons CSV libraries [40] from Apache. Similarly, all statistical calculations performed by the *GoogleTaskClusterOperator* operator, such as median and various percentile calculations, are delegated to the Apache Commons Math library [41].

5.3.4 Static Scaling Test

In this section, we present the results obtained while scaling the logical query statically and re-playing the trace events as quickly as possible.

We first started by running the query without any scaling policies, meaning that the physical query was a one-to-one mapping to the logical query. At any time during the test, there was a single instance of *GoogleTaskMachineOperator* and *GoogleTaskClusterOperator*. We want to identify which of these two operators constitutes a bottleneck and acts as the limiting factor for the overall throughput that can be potentially achieved by our test query. Additionally, knowing which operator is a bottleneck in the base case can facilitate writing effective scaling rules for other tests.

Processing the entire trace took slightly longer than 2 hours, $T_1 = 129.7$ minutes to be more precise. The average throughput, measured at the sink, was 18,577.8 tuples/second. Simultaneously, we collected metrics for the SEEP slave processes running instances of *GoogleTaskMachineOperator* and *GoogleTaskClusterOperator*. In both cases, we recorded average CPU utilisation, length of the input queue for the operator and memory utilisation (as a percentage of the JVM heap utilised by the SEEP runtime and the operator code).

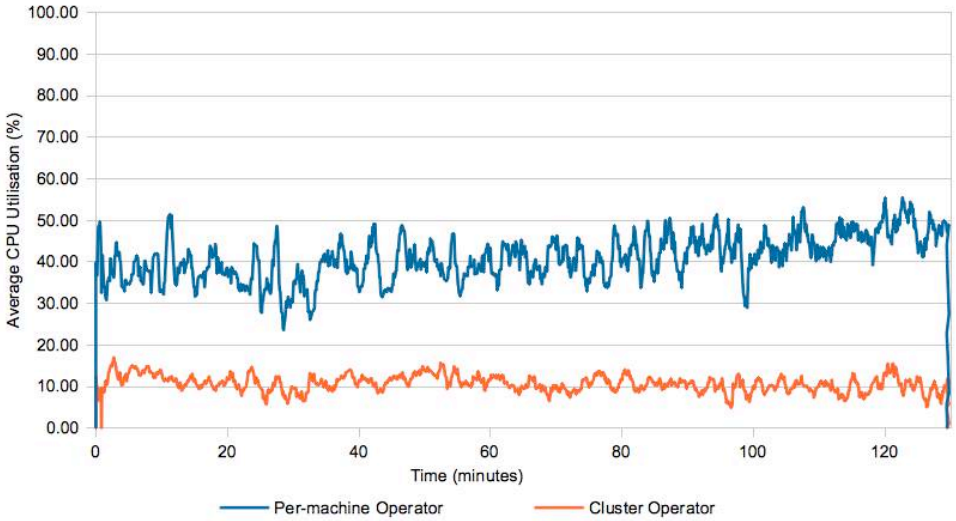

Figure 5.19: Average CPU utilisation. Per-machine operator vs. Cluster operator.

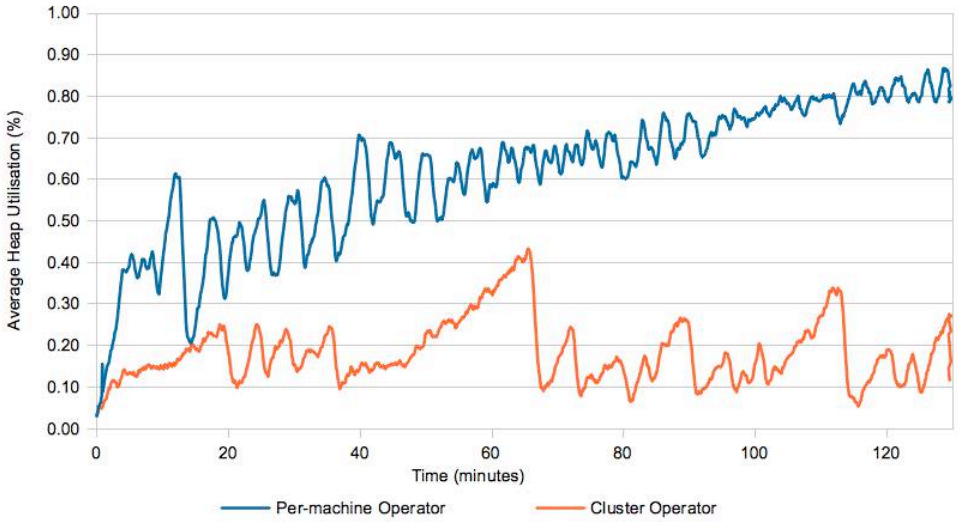

Figure 5.20: Average heap utilisation. Per-machine operator vs. Cluster operator.

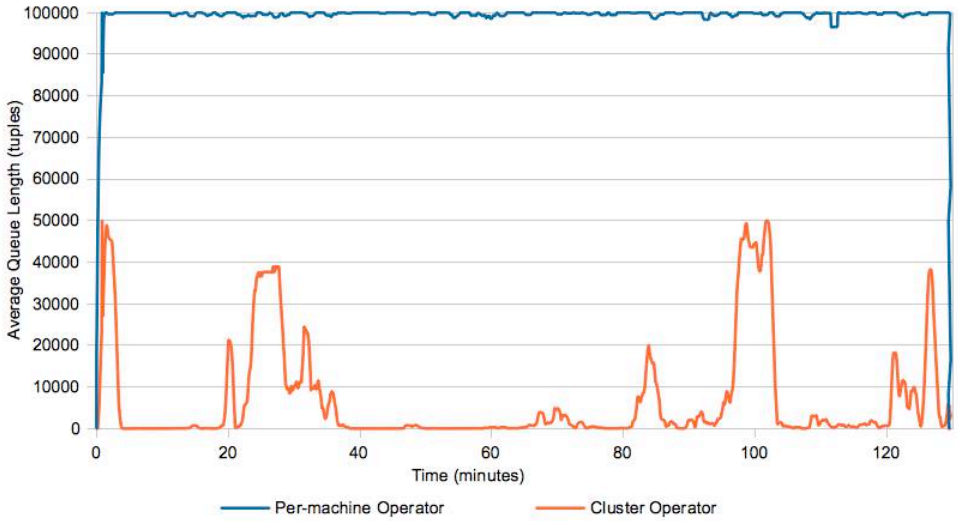

Figure 5.21: Average queue length. Per-machine operator vs. Cluster operator.

When we compare CPU utilisation vs. memory utilisation, we can first observe that for both operators the computation is CPU-bound rather than memory-bound. The heap utilisation metric oscillates, as one would expect in an environment with managed memory and garbage collection. However, it does not appear to be the dominant factor. There is free memory available for both operators at all times during the test, significantly more for the cluster operator.

In contrast, the average CPU utilisation is significantly higher on the per-machine operator, hovering around 50% and about 5 times higher than the CPU utilisation recorded for the cluster operator. It could be argued that if the CPU was in deed the limiting factor, the CPU should be pegged higher, closer to 100%. Even though this is true to some extent, it does not apply in this case chiefly because the SEEP slaves were running on multi-core machines but they do not spawn a sufficiently large number of threads to occupy all cores. SEEP does have some experimental support for multi-core architectures, however we opted to disable it for our tests.

Finally, the length of the input queue confirms that the bottleneck for our test query is happening at the *GoogleTaskMachineOperator* operator. SEEP defines a limit of 100,000 on the number of tuples that can be queue for processing on any particular operator. It can be seen quite clearly that the length of the input queue for the per-machine operator is close to this figure for the entire duration of the test. This indicates that the operator can't cope with the rate at which tuples are arriving from its upstream operators, the Google trace source in this case. There is some queueing happening on the cluster operator at specific times, however it manages to recover as the length of the queue eventually drops to 0.

The bottleneck can be alleviated by scaling out the *GoogleTaskMachineOperator* operator. This is convenient as this state's operator can be partitioned. In contrast, if the bottleneck for the base case happened on *GoogleTaskClusterOperator,* we wouldn't have much choice as the state for this operator is global and cannot be partitioned. The query wouldn't be scalable and its performance would be bound by cluster operator's throughput.

As mentioned before, we scaled the query statically by means of a simple rule that scaled-out the per-machine operator as quickly as possible to its maximum intended size. Once the operator reached this state, it remained in its final state for the duration of the test. We then repeated the test with several absolute maximum sizes ($n = 2$, $n = 4$, $n = 8$, etc; with n being the number of *GoogleTaskMachineOperator* instances), measuring the runtime of the query and comparing the speed-up and efficiency in each case. For instance, the scaling rule for $n = 4$ would be defined in Java as:

```
QueryBuilder.withPolicyRules(
    new PolicyRules() {{
        rule("CPU utilisation is above 10% for 30 seconds")
            .scaleOut(operator("GoogleTaskMachineOperator", OP_ID_MACHINE))
            .by(relative(4)).butNeverAbove(nodes(4))
            .when(MetricName.CPU_UTILIZATION)
            .is(above(percent(10)))
            .forAtLeast(seconds(30)).build();
}});
```

Source Code 5.8: Sample scaling rule for $n = 4$ and operator *GoogleTaskMachineOperator.*

The following execution times were measured, ranging from $n = 1$ to $n = 16$:

n	T_n *(minutes)*	S_n *(Speed-up)*	E_n *(Efficiency)*
1	129.69	1.00	1.00
2	71.06	1.83	0.91
4	40.35	3.21	0.80
8	26.15	4.96	0.62
16	25.00	5.19	0.32

Figure 5.22: Execution times from $n = 1$ to $n = 16$

The speed-up and efficiency can be plotted, together with the values corresponding to linear scaling:

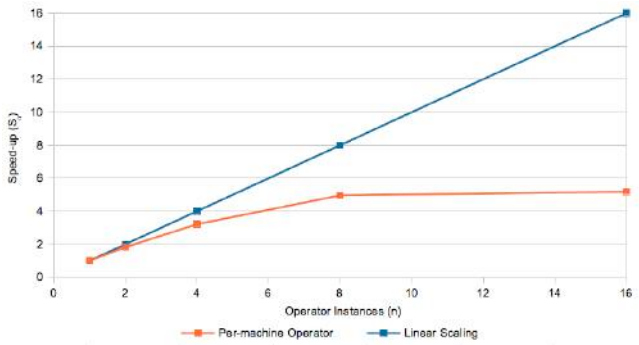

Figure 5.23: Speed-up for $n = 1$ to $n = 16$.

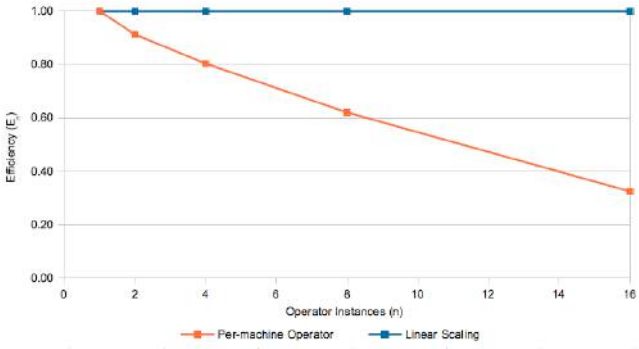

Figure 5.24: Efficiency for $n = 1$ to $n = 16$.

Furthermore, these are throughput achieved during the test for various n values, again ranging from $n = 1$ to $n = 16$:

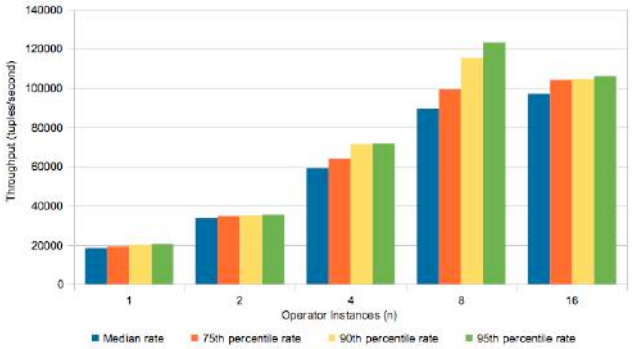

Figure 5.25: Throughput for $n = 1$ to $n = 16$ (chart).

n	*Median (tuples/sec)*	*75^{th} percentile (tuples/sec)*	*90^{th} percentile (tuples/sec)*	*95^{th} percentile (tuples/sec)*
1	18405	19219	19955	20433
2	33779	34609	35010	35475
4	59118	63941	71445	71579
8	89329	99253	115291	123022
16	96985	104094	104575	106080

Figure 5.26: Throughput for $n = 1$ to $n = 16$ (table).

The results show that our test query scales sub-linearly for any value of n, with the speed-up S_n below the linear expected value and the efficiency E_n dropping quite quickly as the value of n increases. In spite of these results, the efficiency remains above 60% for $n = 8$ and we manage to reduce the execution time T_n for the query by a factor of approximately 5, achieving a median throughput of roughly 90,000 tuples/second. The results also confirm that there is little gain from scaling the per-machine operator above $n = 8$. The execution time is almost the same for $n = 16$. Any dynamic scaling policy for our query should avoid scaling the *GoogleTaskMachineOperator* by a relative factor greater than 8, as this has no effect in terms of execution time and can be counterproductive when it comes to efficient use of allocated SEEP slaves.

Finally, the reason why we were unable to scale any further is because the bottleneck migrated from *GoogleTaskMachineOperator* to *GoogleTaskClusterOperator*. This can be confirmed by analysing the CPU utilisation metric for $n = 8$ across all SEEP slave processes and comparing with previous observations for $n = 1$. In a way, this type of bottleneck could be expected as the scaling actions have an effect on the processing capacity of operators in the logical query. As we increased the value of n, it became increasingly hard for *GoogleTaskClusterOperator* to cope with higher input rates from upstream operators.

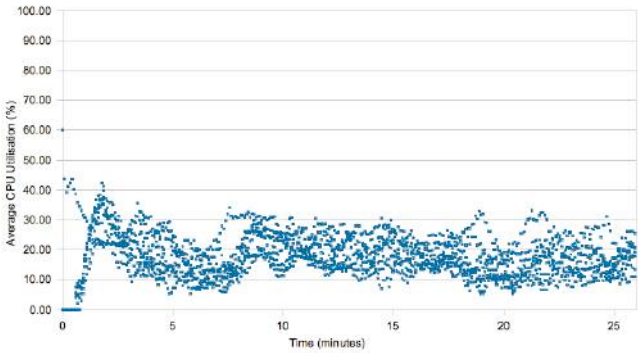

Figure 5.27: Average CPU Utilisation. Per-machine operators (n = 8).

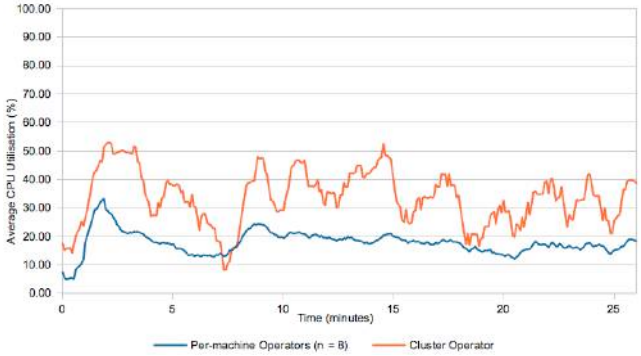

Figure 5.28: Average CPU Utilisation. Per-machine operator vs. Cluster operator (n = 8).

5.3.5 Dynamic Scaling Test

The goal of the previous section was to determine how scalable our test query is, find out which operators act as a bottleneck in the base case where the logical query is mapped one-to-one to the physical query and finally, understand which operators can be scaled and what type of throughput improvement this would yield. In order to achieve these goals, we progressively increased the maximum scaling factor for our per-machine logical operator and measured the processing time in each case. Once this operator was scaled out to its maximum size, we would not allow any further scaling actions during the same test.

In contrast, in this section, we intend to show the results of processing the Google cluster trace with the same logical query but introducing scaling rules that allow the operator size to change dynamically during runtime. This will require us to define rules that trigger scale-in actions (reducing the number of operators instances) and scale-out actions (increasing the number of instances). Again, we will re-play the task events from the Google cluster trace but, differently from our previous test, we will do so in accelerated real-time. In other words, we compress the time between task events by a given factor, resulting in the stream of tuples flowing through the system at a higher pace. For instance, if the timestamps for two events are 1 second apart and we apply a factor of 1000, then in our test 1 real-time millisecond would elapse between these two events.

The following charts show the rates of events when replaying the cluster traces 1000 times faster than real-time. In particular, figure 5.30 shows a 1-minute rolling average of the rate for task events in the cluster trace when applying an acceleration factor of 1000.

The entire cluster trace, which spans for roughly a month, contains job and tasks events that took place over a period of approximately 43200 minutes. If the tuples in the traces are replayed with a time acceleration factor of 1000, then it is reasonable to expect this dynamic test to take around 43 minutes to complete. We can see in the charts below that this is not the case, as the test executed for over 50 minutes. This can be attributed, at least in part, to implementation inefficiencies and various errors, especially when the time between two consecutive events is below the resolution of Java timers.

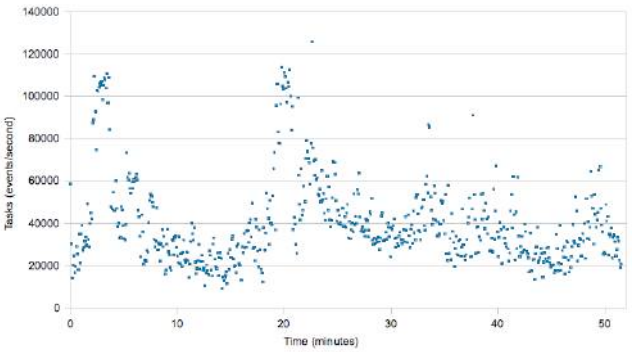

Figure 5.29: Task event rate for the cluster trace, played back in accelerated real-time (x100).

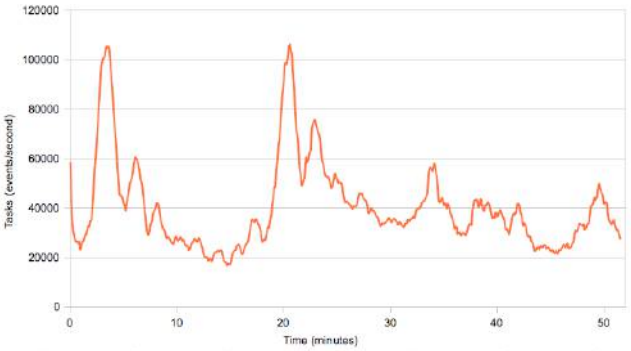

Figure 5.30: Average task event rate for the cluster trace, played back in accelerated real-time (x100).

We defined a different policy for our dynamic scaling test, including both scale-in and scale-out rules for *GoogleTaskMachineOperator*. As shown in the Java excerpt below, the policy specifies two absolute scale-out rules: one that adds instances more aggressively if CPU utilisation exceeds a higher threshold and another one that does so more conservatively, for a lower CPU utilisation threshold. We also establish a limit on the maximum scale size of the operator, which is given by the results observed in the static scaling test. We do not allow more than 8 instances of *GoogleTaskMachineOperator*, as efficiency drops dramatically beyond this value.

Conversely, the scale-in rule terminates instances whenever the CPU utilisation for the operator in question is below a given threshold for a sufficiently long period of time. Lastly, we protect against abrupt oscillations by defining guard times for all rules.

```
QueryBuilder.withPolicyRules(
    new PolicyRules() {{
        rule("CPU is above 10% for 30 seconds, scale-out by 1")
            .scaleOut(operator("GoogleTaskMachineOperator", OP_ID_MACHINE))
            .by(absolute(1)).butNeverAbove(nodes(8))
            .when(MetricName.CPU_UTILIZATION)
            .is(above(percent(10)))
            .forAtLeast(seconds(30))
            .withNoScaleInSince(minutes(1)).build();

        rule("CPU is above 20% for 60 seconds, scale-out by 2")
            .scaleOut(operator("GoogleTaskMachineOperator", OP_ID_MACHINE))
            .by(absolute(2)).butNeverAbove(nodes(8))
            .when(MetricName.CPU_UTILIZATION)
            .is(above(percent(20)))
            .forAtLeast(seconds(60))
            .withNoScaleInSince(minutes(1)).build();

        rule("CPU is below 10% for 60 seconds, scale-in by 1")
            .scaleIn(operator("GoogleTaskMachineOperator", OP_ID_MACHINE))
            .by(absolute(1))
            .when(MetricName.CPU_UTILIZATION)
            .is(below(percent(10)))
            .forAtLeast(seconds(60))
            .withNoScaleOutSince(minutes(1)).build();
}});
```

Source Code 5.9: Dynamic scaling policy for the operator *GoogleTaskMachineOperator.*

The chart below illustrates the results of this test. We plot average CPU utilisation on the initial *GoogleTaskMachineOperator* instance and current scaling size of this operator. It can be seen clearly that when the CPU utilisation exceeds the threshold established by the scaling rules for a sufficiently long period of time, additional instances are provisioned and started. Furthermore, when the CPU utilisation drops down, we can also see that the scale-in action is triggered and some *GoogleTaskMachineOperator* instances are terminated.

Finally, if we compare charts 5.30 and 5.31, we can see that the shape of the workload defined by the tuple arrival rate in the former chart roughly resembles that of the average CPU utilisation shown in the later graph, which confirms that the *GoogleTaskMachineOperator* is CPU-bound. Furthermore, the reactive nature of our scaling framework is confirmed by the slight delay seen between the peaks in CPU utilisation that are meant to trigger a scaling action and the actual change in the size of the operator. A bottleneck has to occur first in order for the framework's monitoring components to detect it and act accordingly.

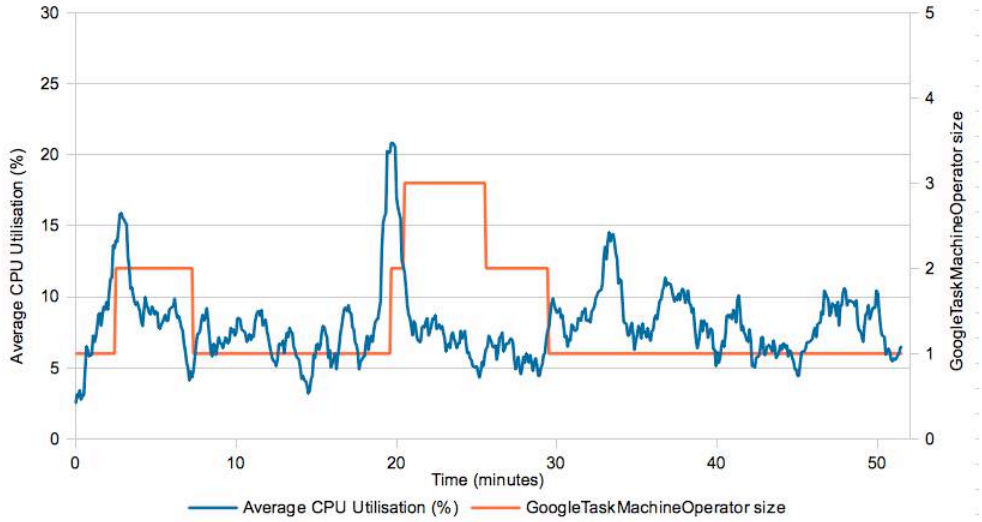

Figure 5.31: Average CPU Utilisation vs. size for the per-machine operator.

5.3.6 Query Results

We now present the cluster statistics obtained from our dynamic scaling test. As mentioned before, the task events from the Google cluster trace were played in accelerated real-time for the query to process them. In order to preserve the original time scale for the traces, the tuples exchanged internally by the operators in our test query forward the original task event timestamp. Thus, we can reconstruct and present the results in the original time scale while still process the events at a faster pace (e.g.: using a time acceleration factor of 1000).

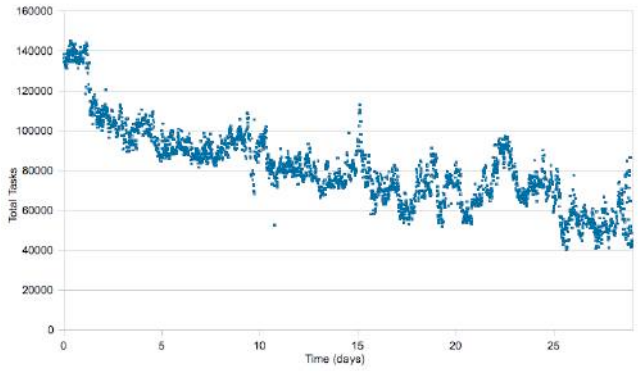

Figure 5.32: Total running tasks in a Google cluster.

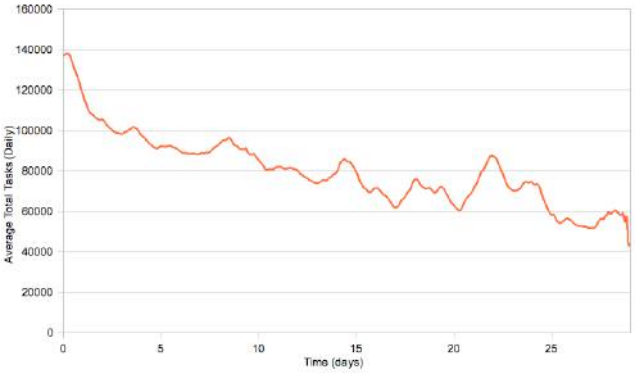

Figure 5.33: Total running tasks in a Google cluster (24-hour rolling average).

First, let's consider the total number of tasks running on the cluster during the month when the traces were obtained. We can see that at the start of the month the number of running tasks was well above 100000, slowly dropping and oscillating as the month progresses and finally, tailing at roughly 40000 at the end of the trace. The charts above show the number of running tasks, with figure 5.33 corresponding to a 24-hour rolling average of the total number of tasks.

The test query executed in the dynamic scaling test also produces median, 75^{th}, 90^{th} and 95^{th} percentile for various resource statistics, including number of tasks per machine and aggregated resource requirements (e.g.: CPU, memory and local disk storage). In particular, we are going to present these statistics for the number of running tasks and CPU requirements per machine.

To facilitate the interpretation of the results, let's consider the 90^{th} percentile for CPU requirements. A value of 0.8 indicates that 90% of the machines in the cluster have committed up to 80% of their available CPU time to some running tasks. Similarly, a value of 5 for the 75^{th} percentile of running tasks represents 75% of the machines in the cluster having up to 5 tasks running at any time. Thus, an analysis of the above percentiles can indicate how over or under committed the cluster is at any one time, assuming that an over-committed cluster is one where resources are almost completely allocated to tasks.

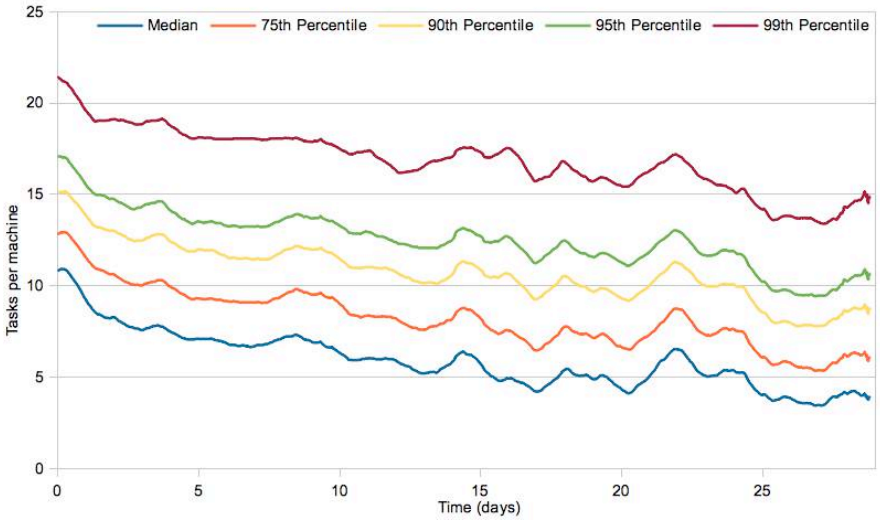

Figure 5.34: Median and percentiles for the number of tasks running on a machine in a Google cluster.

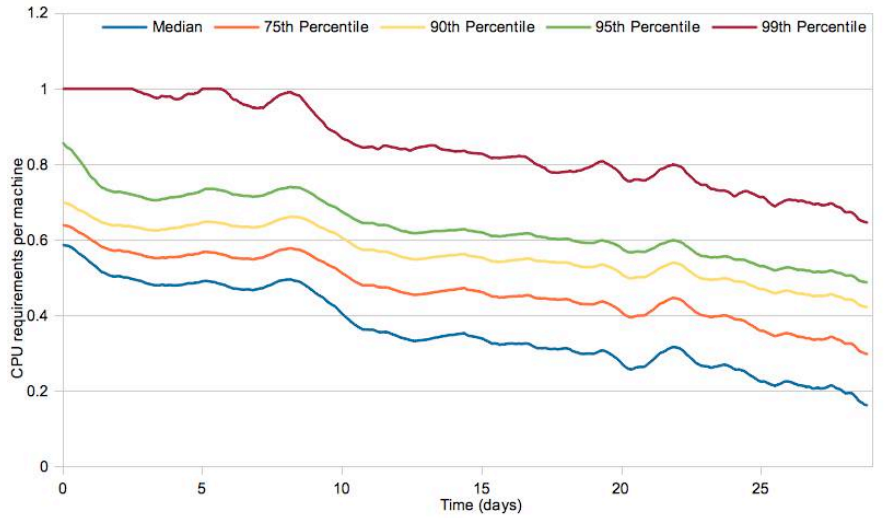

Figure 5.35: Median and percentiles for the CPU requirements of tasks running on a machine in a Google cluster.

These last statistics seem to confirm the trend previously shown for the total number of tasks. The Google cluster for which traces were analysed seems to have been busier at the start of the month. Both the number of tasks per machine and the CPU requirements drop steadily as the month progresses across all the considered percentiles.

In addition to the aggregated cluster results shown so far, our test query outputs the following statistics at regular intervals for each individual machine in the cluster:

- Number of running tasks.
- CPU requirements.
- Memory requirements.
- Local storage requirements.

5.3.7 Summary

In this section, we evaluated our scaling framework running on top of the SEEP system and processing a real-world workload. For this purpose, we have created a simple query that combines stateful and stateless SEEP operators and derives some analytical information from a set of job and task scheduling traces provided by Google for one of its computing clusters.

Firstly, we studied the scalability of the query by attempting to identify which operators behaved as bottlenecks under different conditions. Then, we scaled the query statically, defining a simple policy that scales out to the desired number of instances for a particular operator and doesn't allow anymore changes for the duration of the test.

Secondly, we defined a more complex scaling policy for the same query, allowing for more dynamic scale-out and scale-in of query operators. We then replayed the stream of events from the Google traces but in accelerated real-time, thus preserving the shape of the workload but putting a higher load on the SEEP operators by means of higher tuple rates.

Lastly, for the sake of completeness, we showed some of the analytical results derived by our test query from the Google cluster traces.

5.4 Known Limitations and Possible Improvements

We have so far utilised our proposed scaling framework to dynamically scale SEEP query operators, both with simulated workloads and streams of tuples obtained from real-world systems. Even though we have so far achieved our stated goals, our framework does present certain shortcomings and limitations. We will now describe these known limitations in some detail, concluding with some possible improvements to address such limitations.

We consider the following to be limitations of our proposed solution for dynamic scaling in the context of distributed master-slave SPS systems:

- ***Reactive policy evaluation algorithm***: our scaling framework is purely reactive. Bottlenecks need to happen first in order for the framework to detect them and take scaling decisions to alleviate the situation. This can be seen quite clearly in the dynamic scaling results presented in the previous section for the Google cluster traces, where additional operator instances are only provisioned after the monitored metric has peaked.
- ***Naive evaluation of metric samples***: our implementation of the *EvaluateTrigger* algorithm relies exclusively on the value of past metric samples stored by the *Metric Readings History* component. The fact that all historic samples need to be above or below a certain value threshold for a sufficiently long period of time can occasionally lead to inaccurate scaling decisions. Consider, for example, the case of a single scale out rule in our policy, where all historic samples but the most recent one are significantly above the value threshold. In this case, our

implementation of the evaluation algorithm will decide not to scale out, whereas possibly it might be better to do exactly the opposite. In other words, our policy evaluation implementation lacks the ability to detect and ignore outlier samples.

- ***Centralised evaluation of scaling policies***: the *Monitoring Master* acts as a single point of failure for the monitoring framework. This is obviously less than ideal in terms of fault tolerance and resilience. A crash on the master would essentially mean that scaling policies are no longer evaluated, which can bring about two undesired consequences. On the one hand, the SPS might end up with more instances of a particular operator than needed for the workload at the time. In a cloud-computing environment, this might translate into higher operational costs due to the inability to scale in. On the other hand, if the arrival rate increases and a bottleneck occurs, this will go undetected and the SPS will fail to deliver the desired throughput as it is unable to scale out.
- ***SPS operational support***: the SPS type of systems is characterised by queries that execute continuously and uninterruptedly. The operational aspects of a continuously running system are neglected by our scaling framework. In particular, there are no readily available interfaces to determine the state of a running query by means of the metrics transmitted by *Monitoring Slave* instances and collected by the *Monitoring Master*. Furthermore, the framework does not provide any mechanism to alter scaling policies for running queries. The ability to make policy changes without stopping the SPS is quite relevant in this context.
- ***Bulk transmission of metric samples***: the current implementation of the framework does not adjust the metrics sent by *Monitoring Slave* instances to the *Monitoring Master*. All metric values samples by the slaves are eventually transmitted to the master, even if no scaling policies reference them (e.g.: memory utilisation metrics are transmitted even if the current scaling policy refers exclusively to CPU utilisation). This behaviour results in higher network bandwidth, which might be paid for based on usage especially in cloud-computing environments.

As can be seen from the points stated above, there are some limitations that affect our scaling framework. We will now consider some possible improvements that address some of these limitations and other related problems:

- ***Predictive policy evaluation algorithm***: instead of reactively detecting bottlenecks, the scaling framework could proactively attempt to predict when such events are expected to happen and adopt scaling decisions early on. Thus, the SPS is in a better to position to deal with a change in the workload before this change actually happens. We have presented systems that follow this approach before, particularly AGILE [12] and PRESS [13]. Both attempt to use Fourier transforms, wavelets and Markov models, with some variations, to predict peaks and troughs in period and aperiodic workloads.

- ***Detection and countering of outlier samples***: the *EvaluateTrigger* algorithm could be improved to avoid outliers from aborting scaling decisions that would have otherwise been taken. The approach can range from something simple, where a rolling average or mean is used instead of testing past readings individually, to more complex statistical models that provide for the detection of outliers.
- ***Distributed evaluation of policies***: instead of forwarding all metric readings to a single *Monitoring Master*, slave instances could forward them to local masters who make local scaling decisions. This approach has some similarities with that of local collaborating optimizers proposed for the Borealis system [8]. For example, all the *Monitoring Slave* instances associated to a logical query operator could elect one as the leader. Then, metric readings would be forwarded to this leader instead of the centralised *Monitoring Master*. Thus, the leader, acting as a local master, would make local scaling decisions pertaining only to one of the query operators. Such an approach would improve the isolation of operators and the resilience of the system as a whole. A crash in a local master would only affect one of the query operators and not the entire scaling function across the SPS.
- ***External interface for on-line monitoring of queries***: the monitoring framework could expose a well defined interface for others to find out the state of a running query, as reflected by recent metric readings. As the *Monitoring Master* is aware of the state of the entire SPS, it could potentially expose a REST API or a JMX interface for others to leverage on this fact and obtain a snapshot of the system's state.
- ***Ability to modify scaling policies while continuos queries are running***: as we stated before, queries in a DSPS need to run continuously. In this context, it is important for query developers to be able to make live changes to their scaling policies while queries are running. The monitoring framework should expose a mechanism to refresh the scaling policy and its rules without stopping the processing of the data stream.
- ***Publisher-subscriber model for metrics***: typically, a scaling policy for a given operator does not reference all the metrics collected by *Monitoring Slave* instances. The network bandwidth requirements of our scaling framework can be reduced if we implement a publisher-subscriber model for metrics. All *Monitoring Slave* instances would advertise the metrics that they can sample and transmit, which depend ultimately on the associated operator. Conversely, the *Monitoring Master* would only subscribe to those metrics that are relevant for the rules that apply to each operator, but not other metrics (e.g.: if the rule for Operator X refers to CPU utilisation, then the master would subscribe to this metric from this operator's slave but not memory utilisation or input queue length, for instance). Thus, at runtime, only the relevant metrics are transmitted by slaves to the master.

6 Conclusion

As we defined at the beginning of this work, a SPS executes continuous queries over unbounded non-persistent data streams that produce unbounded results to be returned to users. Unfortunately, in many cases, it is impossible to predict the rate at which data stream items might arrive into the SPS for processing. Hence, the difficulty resides in how to determine the amount of resources to provision for a query and critically, for each one of its parts. Approaches that might seem simple at first sight, such as estimating for peak load or average load, are actually difficult to implement and present serious disadvantages both in terms of cost and performance.

We have created a generic framework that enables dynamic reactive scaling of queries running on SPS systems, when these meet certain architectural assumptions. We believe that dynamic scaling offers an alternative solution to the problem of resource provisioning and allocation when facing unknown workloads, as is typically the case for a SPS. Our framework proposes a model for scaling policies and rules that derives from that defined by existing dynamic scaling solutions in commercial cloud-computing environments, such as Amazon Web Services and Microsoft Windows Azure, albeit tailored to the specific requirements of stream processing. Thus, our dynamic scaling framework allows a SPS to execute queries continuously without concerning with initial resource allocation, leaving the framework to decide at runtime when to provision additional resources.

We have also proposed an embedded domain-specific language for the definition of scaling rules for SPS queries. Our embedded language relies on various programming techniques, such as method chaining, initialiser blocks and static imports, to offer fluent scaling rules that are readable and can be easily understood. We believe that such an approach enables query developers to easily define scaling rules for their policies and test their queries with various alternatives.

Finally, we have integrated our generic scaling framework with the SEEP system. We have executed a series of tests with simulated workloads and with real-world workloads obtained from a Google cluster trace, running stateless and stateful queries respectively. The first set of tests confirmed that the framework took the correct scaling decisions under different conditions, including constant, periodic and aperiodic simulated workloads. We defined simple scaling rules in each test case, always based on the input queue length for a particular query operator. This metric provides a deterministic behaviour and it was possible to analytically calculate the correct state of the system. Secondly, we performed a set of static and dynamic scaling tests against a real-world workload. We confirmed that the framework can dynamically scale in and out a query composed of stateful partitionable operators.

We believe that the objectives laid out initially for this project have been met and that our dynamic scaling framework contributes to the solution for the problem of resource allocation and provisioning for SPS systems facing unknown workloads.

7 References

[1] P. Pietzuch, *Drinking From The Fire Hose: The Rise of Scalable Stream Processing Systems*, Large-Scale Distributed Systems Group, Imperial College, London, 2013.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, *Models and Issues in Data Stream Systems*, ACM SIGMOD/PODS Conference, Madison, Wisconsin, 2002.

[3] Google, *Google Cloud Platfrom Documentation*, October 2014, https://cloud.google.com/docs/

[4] L. Golab, M. T. Ozsu, *Data Stream Management Issues – A Survey*, University of Waterloo, Canada, 2003.

[5] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, *Aurora: a new model and architecture for data stream management*, The International Journal on Very Large Data Bases, Springer-Verlag, Volume 12 Issue 2 August, New Jersey, 2003.

[6] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, M. J. Franklin, *Flux: An Adaptive Partitioning Operator for Continuous Query Systems*, Proceedings of the 19th International Conference on Data Engineering, Bangalore, India, 2003.

[7] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava and J. Widom, *STREAM: The Stanford Data Stream Management System*, Stanford University, Stanford, CA, 2003.

[8] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing and S. Zdonik, *The design of the Borealis stream processing engine*, Proceedings of the 2005 CIDR Conference, Asilomar, CA, 2005.

[9] L. Neumeyer, B. Robbins, A. Nair and A. Kesari, *S4: Distributed Stream Computing Platform*, 2010 IEEE International Conference on Data Mining Workshops, Sydney, Australia, 2010.

[10] N. Marz, *Storm: distributed and fault-tolerant realtime computation*, Apache Software Foundation, Maryland, USA, 2014.

[11] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch, *Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management*, ACM International Conference on Management of Data (SIGMOD), New York, NY, 2013.

[12] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, J. Wilkes, *AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service*, USENIX Association, 10th International Conference on Autonomic Computing (ICAC '13), San Jose, CA, 2013.

[13] Z. Gong, X. Gu, J. Wilkes, *PRESS: Predictive Elastic Resource Scaling for Cloud Systems*, International Conference on Network and Service Management (CNSM), Niagara Falls, ON, Canada, 2010.

[14] Z. Shen, S. Subbiah, X. Gu, J. Wilkes, *CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems*, ACM Symposium on Cloud Computing 2011, Cascais, Portugal, 2011.

[15] J. Varia, S. Mathew, *Overview of Amazon Web Services*, 2014, https://media.amazonwebservices.com/AWS_Overview.pdf
[16] Amazon Web Services, *Amazon Elastic Compute Cloud, User Guide for Linux*, 2014, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
[17] Amazon Web Services, *Amazon CloudWatch, Developer Guide*, 2010, http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/WhatIsCloudWatch.html
[18] Amazon Web Services, *Amazon Auto Scaling, Developer Guide*, 2011, http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/WhatIsAutoScaling.html
[19] David Chappell, *Introducing Windows Azure*, October 2010, http://go.microsoft.com/?linkid=9682907
[20] Microsoft Corporation, *Microsoft Developer Network Library, Patterns & Practices, Enterprise Library 5.0*, May 2011, http://msdn.microsoft.com/en-us/library/ff632023.aspx
[21] Microsoft Corporation, *Microsoft Developer Network Library, Patterns & Practices, Enterprise Library 5.0, Integration Pack for Windows Azure*, June 2012, http://msdn.microsoft.com/en-us/library/hh680918.aspx
[22] Google, *Auto Scaling on the Google Cloud Platform*, 2013, https://cloud.google.com/developers/articles/auto-scaling-on-the-google-cloud-platform/
[23] K. R. Fall, W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Second Edition, Addison-Wesley Professional, Berkeley, CA, 2011
[24] A. van Deursen, P. Klint, J. Visser, *Domain-Specific Languages: An Annotated Bibliography*, ACM SIGPLAN Notices Volume 35 Issue 6, New York, NY, 2000.
[25] M. Mernik, J. Heering, A. M. Sloane, *When and How to Develop Domain-Specific Languages*, ACM Computing Surveys (CSUR), New York, NY, 2005.
[26] M. Fowler, R. Parsons, *Domain-Specific Language*, , Addison-Wesley Professional, Boston, MA, 2010
[27] S. Freeman, N. Pryce, *Evolving an Embedded Domain-Specific Language in Java*, ACM OOPSLA'06, Portland, OR, 2006.
[28] Stephen O'Grady, *The RedMonk Programming Language Rankings*, 2014, http://redmonk.com/sogrady/2014/01/22/language-rankings-1-14/
[29] TIOBE, *Programming Community Index*, 2014, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[30] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, 37th Edition, Addison-Wesley Professional Computing Series, Boston, MA, 2009
[31] Esoteric Software, *Kryo*, 2013-2014, https://github.com/EsotericSoftware/kryo
[32] Coda Hale, Yammer Inc, *Metrics*, 2010-2014, https://dropwizard.github.io/metrics/3.1.0/
[33] Oracle Corporation, *Java 7 Standard Edition, Monitoring and management of the Java virtual machine*, 1993-2014, http://docs.oracle.com/javase/7/docs/api/java/lang/management/package-summary.html
[34] Oracle Corporation, *Java 7 Standard Edition, Platform extension to the*

implementation of the java.lang.management API, 2003-2014, http://docs.oracle.com/javase/7/docs/jre/api/management/extension/com/sun/management/package-summary.html
[35] Large-Scale Distributed Systems (LSDS) research group, *SEEP Source Code*, 2014, https://github.com/lsds/seep/
[36] Large-Scale Distributed Systems Group, Imperial College, London, *Source Code for the SEEP System*, 2014, https://github.com/lsds/Seep/
[37] C. Reiss, J. Wilkes, J. Hellerstein, *Google cluster-usage traces: format + schema*, 2011, https://code.google.com/p/googleclusterdata/
[38] R. C. Fernandez, M. Weidlich, P. Pietzuch, A. Gal, *DEBS Grand Challenge: Scalable Stateful Stream Processing for Smart Grids*, , Mumbai, India, 2014.
[39] The Apache Software Foundation, *Commons Compress*, 2014, http://commons.apache.org/proper/commons-compress/
[40] The Apache Software Foundation, *Commons CSV*, 2014, http://commons.apache.org/proper/commons-csv/
[41] The Apache Software Foundation, *Commons Math: The Apache Commons Mathematics Library*, 2003-2014, http://commons.apache.org/proper/commons-math/

8 Appendix: Test Environment

All the tests presented in this work were executed on a cluster of Linux servers with the following hardware and software configuration:

- Quad-Core AMD Opteron™ Processor 2346 HE 1.8GHz.
- 4GB of RAM memory on each cluster node.
- Ubuntu version 12.04.2 LTS, with a Linux kernel version 3.2.0-58-generic-pae.
- OpenJDK Runtime Environment version 1.7.0_55 (IcedTea 2.4.7).

As we mentioned previously, we executed SEEP in a master-slave arrangement where one of the servers in the cluster acts as the master and the remaining ones act as slaves. The *Monitoring Master* component of our scaling framework run on the same server as the SEEP master. Conversely, the multiple SEEP slaves run alongside instances our *Monitoring Master* component on each server.

SEEP can be executed as master with the following Java command, which can be executed directly from the shell on a Linux server:

```
mr2111@wombat01:~$ java -Xmx2048m -Xms2048m -XX:+UseConcMarkSweepGC -XX:+CMSIncrementalMode -XX:
+ForceTimeHighResolution -jar seep-system-0.0.1-SNAPSHOT.jar Master "${PWD}/seep-system-test-
queries-1.0.0-SNAPSHOT.jar" google.query.GoogleTaskDataSetQuery |& tee master.out
```

Similarly, SEEP can be started as a slave with the command shown below. The slaves need to know where the master is running. Hence, the IP address and the port where the SEEP master is listening for slave connections need to be specified as part of the command-line arguments for the Java Virtual Machine. So, a slave node would be started as follows:

```
mr2111@wombat02:~$ java -Xmx2048m -Xms2048m -XX:+UseConcMarkSweepGC -XX:+CMSIncrementalMode -XX:
+ForceTimeHighResolution -jar "${PWD}/seep-system-0.0.1-SNAPSHOT.jar" Worker 63000 146.179.131.131
2> slave.err |& tee slave.out
```

In both cases we specify the desired memory configuration for the Java Virtual Machine, including initial and maximum heap size of 2GB plus the preferred garbage collection algorithm (i.e.: incremental concurrent mark-and-sweep). We also rely on the standard Unix *tee* command to read SEEP logs from the standard input and write them back to the standard output and a log file.

9 Appendix: Source Code

The source code for the dynamic scaling framework described in this work has been uploaded and made publicly available on Github. The repository can be found at https://github.com/mrouaux/sps-monitor-framework. Using the git command line tools it is possible to clone the repository and thus obtain a local copy of the source code:

```
mr2111@wombat01:~$ git clone https://github.com/mrouaux/sps-monitor-framework.git
Cloning into 'sps-monitor-framework'...
remote: Counting objects: 224, done.
remote: Compressing objects: 100% (133/133), done.
remote: Total 224 (delta 73), reused 212 (delta 71)
Receiving objects: 100% (224/224), 77.70 KiB | 0 bytes/s, done.
Resolving deltas: 100% (73/73), done.
Checking connectivity... done.
mr2111@wombat01:~$
```

The framework can be built using Apache Maven version 3.0.5 or later (further information can be found at http://maven.apache.org/). Maven is a build and dependency management tool that automatically downloads all third party libraries our code depends on, compiles all Java classes and executes all unit and integration tests. It can be invoked as shown here:

```
mr2111@wombat01:sps-monitor-framework$ mvn clean install
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Dynamic Scaling Framework 1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ monitor ---
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ monitor ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 2 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ monitor ---
[INFO] Changes detected - recompiling the module!

...

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 12.238s
[INFO] Finished at: Tue Nov 11 21:55:03 GMT 2014
[INFO] Final Memory: 18M/43M
[INFO] ------------------------------------------------------------------------
```

If the build is successful a Java archive file named *sps-monitor-framework-{version}* will be created, where *{version}* represents the actual version of the software being built. The example above shows a successful build for version *1.0.0-SNAPSHOT* (i.e.: a development version).

Finally, the integration work with SEEP can be found on the Github repository for the SEEP project itself, which can be found at https://github.com/lsds/SEEP. The integration code can be found in the package *uk.ac.imperial.lsds.seep.infrastructure.monitor*, where the most relevant class is probably *SeepInfrastructureAdaptor*.