Moira: A Goal-Oriented Incremental Machine Learning Approach to Dynamic Resource Cost Estimation in Distributed Stream Processing Systems

Daniele Foroni
University of Trento
daniele.foroni@unitn.it

Cristian Axenie
Huawei ERC Munich
cristian.axenie@huawei.com

Stefano Bortoli
Huawei ERC Munich
stefano.bortoli@huawei.com

Mohamad Al Hajj Hassan
Huawei ERC Munich
mohamad.alhajjhassan@huawei.com

Ralph Acker
Huawei ERC Munich
ralph.acker@huawei.com

Radu Tudoran
Huawei ERC Munich
radu.tudoran@huawei.com

Goetz Brasche
Huawei ERC Munich
goetz.brasche@huawei.com

Yannis Velegrakis
University of Trento
velgias@unitn.it

ABSTRACT

The need for real-time analysis is still spreading and the number of available streaming sources is increasing. The recent literature has plenty of works on Data Stream Processing (DSP). In a streaming environment, the data incoming rate varies over time. The challenge is how to efficiently deploy these applications in a cluster. Several works have been conducted on improving the latency of the system or to minimize the allocated resources per application through time. However, to the best of our knowledge, none of the existing works takes into consideration the user needs for a specific application, which is different from one user to another. In this paper, we propose Moria, a goal-oriented framework for dynamically optimizing the resource allocation built on top of Apache Flink.

The system takes actions based on the user application and on the incoming data characteristics (i.e., input rate and window size). Starting from an initial estimation of the resources needed for the user query, at each iteration we improve our cost function with the collected metrics from the monitored system about the incoming data, to fulfill the user needs. We present a series of experiments that show in which cases our dynamic estimation outperforms the baseline Apache Flink and the thumb rule estimation alone performed at the deployment of the applications.

ACM ISBN 978-1-4503-6607-6/18/08…$15.00
https://doi.org/10.1145/3242153.3242160

CCS CONCEPTS

• **Information systems → Stream management**; **Process control systems**; **Data stream mining**;

ACM Reference Format:
Daniele Foroni, Cristian Axenie, Stefano Bortoli, Mohamad Al Hajj Hassan, Ralph Acker, Radu Tudoran, Goetz Brasche, and Yannis Velegrakis. 2018. Moira: A Goal-Oriented Incremental Machine Learning Approach to Dynamic Resource Cost Estimation in Distributed Stream Processing Systems. In *International Workshop on Real-Time Business Intelligence and Analytics (BIRTE '18), August 27, 2018, Rio de Janeiro, Brazil.* ACM, Rio de Janeiro, Brazil, 10 pages. https://doi.org/10.1145/3242153.3242160

1 INTRODUCTION

Nowadays, the amount of available data sources is cumbersome, which is reflected in the so-called data deluge. In fact, the data produced at any moment is continuously growing. This data can be of two kinds: batch and streaming data. While data batch has fixed size, data streams receive data of an unbounded length and process it in real-time. This leads to several additional issues.

A data stream is continuous and has no fixed length. Furthermore, the number of elements received in a time window, i.e., the input rate, is prone to vary over time. This fluctuation of the input rate may lead to overallocation or underutilization of the system resources since the amount of data that is processed is changing as the input rate fluctuates. Data Stream Processing (DSP) applications, i.e., queries or analytics over the data, are commonly represented as a directed acyclic graph (DAG), where the nodes are the operations to be performed and the edges serve as data streams. Hence, the needed optimization has to be performed on the topology generated by the user query. In particular, using the example of Apache Flink, we present a topology optimization that this streaming framework enables by default. Each node of the DAG can be replicated several times to parallelize the task operation through the cluster where the application is deployed for improving the performance of the user query. Furthermore, one node can be chained with the succeeding operator, which means that they will be executed on the same machine and in the same thread, avoiding thread-to-thread handover. The greedy approach would be to parallelize each operator as much as possible and to chain them likewise. However, splitting the work of an operator on multiple machines allocating more resources than those needed has still a cost and we may want to actually reduce it, or we may keep a fine-grained control of the operators while chaining two or more leads to coarse-grained control of the operators. Hence, the deployment of the topology for the DSP applications is significant in order to allocate the right amount of resources and deploy the best topology.

Several works have been proposed to bind the used resources with the incoming data rate. Dhalion is a system built on top of Heron [2] that "heals" the running application, enabling the self-regulation of the system, detecting symptoms into the system metrics and applying a number of policies to handle the rescheduling or tuning of the topology [9]. Elastic Allocator gathers information from the cluster usage and exploits a high resource allocation through a greedy-based algorithm [11]. Another work models the problem of the topology that fits better the incoming data as a Markov decision process, with a model based on Reinforcement learning [17].

However, none of these works consider the user goal for the analytics that has to be performed. Hence, we introduce Moira, which is a character of the Greek mythology with decision power on the faith of humans, a dynamic cost estimator system, that through the monitoring of the systems decides the "faith" of the running applications, deciding if a redeployment or a tuning is needed for each application to meet its user-defined goal. A user has to specify the query she wants to perform over the data and an optimization goal, weighing three parameters, i.e., throughput, latency, and cost. Before the deployment to the streaming framework, Moira applies a cost-based estimation of the given analysis, to optimize its deployment and to get closer to the user goal. Then, after the submission of the application to the streaming framework, the dynamic cost estimator monitors multiple cluster metrics and other data taken from the incoming data and, at every defined interval, it triggers a new cost estimation aware of these parameters. If the built topology does not fit the user requirements and a new and better topology can be deployed, then the running application is stopped and the new topology is actually used.

In this paper, we focus on the optimal solution for the user goal, given the characteristics of the incoming data and the analysis that the user has to perform. Hence, we provide the following contributions:

- To our knowledge, this is the first paper that takes into consideration the user goal for an application for a dynamic resource allocation in a streaming environment.
- We provide a formal definition of the optimal solution for the user goal, knowing information about the incoming data, cluster usage metrics, and being aware of the user query. This is also presented in a framework built on top of Apache Flink that enables this kind of analysis.
- We conduct a series of experiments to compare our method with a single static estimation and without any cost estimation, presenting the advantages and the issues for each case.

The remainder of this paper is organized as follows. We present a use case scenario to allow the readers to understand the needs for such a system in Section 2. Then, a literature overview of the related work is conducted in Section 3. Section 4 exposes the formal definition of the problem we are dealing with, and in Section 5 we show the insights of our solution. Section 6 presents the experiments performed and the comparison with and without our framework, and with only a cost-based estimation at the deployment of the application. Then, we conclude with some final remarks in Section 7.

2 MOTIVATING EXAMPLE

A startup has just published a new mobile phone game on the market. They developed a logging system in their mobile application to receive feedback from the users while they are playing their game. The logs have several objectives, such as debugging purpose or reporting, so each kind of log needs a specific management. Moreover, the incoming data to the logging system will change its input rate through time, since people play their game but only for a small amount of time. Even more, if we consider that the game has been purchased more in Europe than in the rest of the world, for sure we will have fluctuating data, with some peaks, for example, in the European evening time and some drops during the European nights. So, in this context, the startup wants to perform some kind of analysis on the logs, to improve the game experience of their customer. This analysis is translated into a DAG (Directed Acyclic Graph), where each node represents an operation of the analysis (e.g., a map/reduce operation), while the edges represent the data flow path. Intuitively, each node will be deployed on a slot placed in a machine of the cluster. A machine has multiple slots available for the nodes of the DAG. However, this DAG is not deployed as it is on the cluster, but it can be optimized in different ways to improve the performance of the analytics. For example, multiple nodes (operations) of the DAG can be performed in the same slot, avoiding the cost of moving the results of a previous operator to the next operator through the cluster slots that can even be on different machines, or one node can be deployed multiple times to parallelize the operation and thus to improve the performance of that operation.

Let's consider the following analytic described by the startup: get as input the logs, perform a word count of the error code, filter the elements that appear a fewer number of times than a predefined threshold, try to replicate the error with a custom code taken as a black-box, and then save the results into a text file. The analytics will be translated into the DAG pictured in Figure 1.

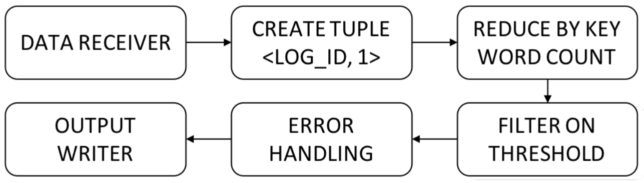

Figure 1: DAG example for a use case application

So, with such an analysis, other available systems would optimize the resource allocation or the latency. In our system, it is the user that specifies the optimization goal for the given query. Then, the system will start from it using the gathered information about the incoming data and the cluster usage to deploy the topology that fits better the user's goal. For example, if the user wants to improve latency, one choice can be to chain together the whole process, parallelizing the operators as much as possible. However, if it may be useful while the data is at its peak, when the income rate is on a drop, it becomes not worthy to have the whole cluster filled. Hence, it will be worthy to have a system that takes care of it and decreases the parallelism, which is our aim, even if the goal is to have low latency, since even with fewer resources the goal is still fulfilled.

On the other hand, if the custom error handling policy is a slow operation that needs an amount of resources, for optimizing latency then it will be better to chain the first four nodes and have a bigger parallelism for the error handling task.

Hence, as it is shown, each query has to be handled in different ways given the goal defined by the user. Moreover, since in such a context the data incoming rate might vary through time, a dynamic estimation is needed to improve performance and resource allocation, even better if it follows the user needs and allow the user to fulfill her goal.

3 RELATED WORK

Streaming analysis has been a trending topic over the last years and many frameworks for such applications have been proposed. Among them, the most known are Apache Flink [1], the streaming library of Apache Spark [4], Apache Storm [5], and Apache Heron [2]. Given the number of available tools, one of the main questions on the hype recently is about identifying the best available streaming system. To answer this question, several works compare these frameworks to find the most reliable and fastest system available on the market [8, 16]. Moreover, a positive aspect of these systems is that they allow the deployment of the applications over a cluster of machines, enabling scalable prone analysis. Hence, in this context, resource allocation and how to deploy an application over multiple machines are becoming even more important.

The first investigation track leverages on running multiple applications in the same cloud system, where the resources are limited and the applications have to compete with each other to assure them. From this group, YARN [18], Mesos [13], and Abacus [19] are some notable examples. The basic idea behind these systems is that each application knows the needed resources and the framework takes care of its scheduling and deployment among the nodes of the cluster. However, these systems do not provide an analysis bounded to the application resource allocation during its whole cycle.

There are many works on this side, and among them, Dhalion is a self-regulating system built on top of Apache Heron [9]. It is a system implemented through a set of Symptom Detectors that check the status of both the incoming stream of data and the allocated resources. Then, the symptoms are used by the so-called Diagnosers, which aim at diagnosing problems from those detected symptoms, e.g., back pressure and over-provisioning. Finally, the performed diagnoses are examined by the Resolvers that take the appropriate decision, e.g., allocate more resources or change the location of a task to avoid back pressure. This analysis continues through all the application running period.

An older approach for Hadoop Map/Reduce applications is Starfish [12]. It proposes a self-tuning system that handles the Hadoop configuration without the need for the users to tune it by hand. This automatic tuning can be performed at different granularity levels, from job-level tuning (fine-grained) to workflow-level (coarse-grained) tuning, to fulfill different needs. In addition, the framework presents a language to specify a workload (i.e., a sequence of workflows) along with some metadata, which is added to those automatically gathered from the system in order to improve the configuration performance. Furthermore, this language system works as a recommendation engine for configuring Hadoop applications.

Elastic Allocator is another adaptive system that gathers information about the cluster both from what concerns the CPU usage and the bandwidth usage [11]. It is claimed to be the first system to use the latter metric for this kind of analysis. The framework is built on top of Apache Storm and it aims at solving the problem of assigning the task operators to the appropriate node of the cluster to improve the performance of the application. It takes the decision, knowing the collected information metrics, through a greedy-based algorithm.

Another work that performs dynamic resource provisioning is Flower (Flow Elasticity Manager) [14, 15]. The framework collects information from multiple monitoring systems of the cluster at different layers, e.g., data ingestion, analytics, and storage, that are later fit into the control system. This module takes as input the history of the sensor values, which are the measured values and the desired values of each monitored element at a specific time, and dynamically updates the value of the actuator to reach the desired results.

A different approach models the problem as a Markov Decision Process (MDP) [17]. However, usual cases do not have a full knowledge of the system, so the approach exploits a reinforcement learning algorithm to overcome this problem. At each iteration the model checks for each task operator of the application if it has to increase its number of parallel running instances, decrease it, or if the actual value suits the requirements. In addition, it provides an analysis on both a centralized approach, where the system runs in the master node and coordinates all the others, and a decentralized approach, where each node is aware only of the tasks running on it and the parallelism can be increased only on the node's available resources.

DRS (Dynamic Resource Scheduling) is yet another application for dynamic rescheduling the resources assigned to an application [10]. It comprises two layers, the DRS layer, and the CSP layer. The former contains the monitoring system, which runs the resource optimizer algorithm and performs the actual resource allocation, while the latter is just a framework deployed on top of the streaming processing system. Hence, the CSP (Cloud-based Streaming Processing) layer acts as a middleware to allow the communication between the DRS layer and the streaming system adopted. It proposes a solution for allocating the right amount of resources and assigning them to the right cluster nodes under the constraint of a low-latency application.

4 PROBLEM STATEMENT

We assume the existence of a countable set of records $\mathcal{R}$ and a countable ordered set of timestamps $\mathcal{T}$. A sequence of records, each with an assigned timestamp, is referred as a *data stream* or simply a *stream*. Let $\mathcal{S}$ represent the set of all possible streams. We denote as $\tau(r)$ the timestamp of a record r in a stream. The rate of a stream $s \in \mathcal{S}$, at a time t, and for a temporal window w, is the number $\frac{|\{r \mid r \in \mathcal{R} \wedge (t-w) \leq \tau(r) \leq t\}|}{w}$. Note that the rate of a stream may be different over time.

The records of one or more streams can be processed to produce new objects. There is a number of primitive processing tasks that can be performed on streams. These tasks are referred to as *operators*. The output of an operator is a stream itself.

*Definition 4.1 (**Operator**).* A *stream operator* is a function $o : \mathcal{P}(\mathcal{S}) \rightarrow \mathcal{S}$. The set of all possible operators is denoted as $\mathcal{O}$.

Since the output of an operator is a stream, it can be used as input to another operator. In this way, operators can be combined to form more complex processing tasks. Such tasks are referred to as *queries*.

*Definition 4.2 (**Query**).* A k-input query is a tuple $q = \langle \bullet, N, E, I, n_e \rangle$, where $N \subset \mathcal{O}$ and is finite, $\bullet$ is a partial order over N, $n_e \in N$, I is an assignment $[1..k] \rightarrow N$ and $E \subseteq N \times N$ such that $\forall \langle n_1, n_2 \rangle \in E : \bullet(n_1) \leq \bullet(n_2)$.

A query is actually a function that accepts as input k streams, and produces a single output stream. The output stream is the output of the operator n_e. The assignment I assigns each of the k input streams to one or more operators in N. Intuitively, a query can be seen as a directed acyclic graph that has a node for every operator in N, and an edge for every entry in E. The node n_e is referred to as the *output* node, and every node that has been assigned at least one input in I, as an *input* node. Input nodes are annotated also with the number of the input stream they have been assigned. For instance, if the assignment I contains the assignment $\langle 3, n \rangle$, it means that the third input stream is among the inputs of the operator n. Input nodes are annotated with the numbers of the inputs that they have been assigned to them. In what follows, when we refer to a query, we will refer to its equivalent graph representation.

Example 4.3. Figure 2 illustrates the graph representation of a query that takes as input 3 input streams. The input nodes are those without incoming edges and their annotations on the left of the identifiers indicate which of the three input streams s_1, s_2, and s_3 they use as input. Although not shown in the graph, note that an input node can have more than one input streams. Node 10 is the output node, i.e., the node of the output stream s_o, which is considered the output of the query.

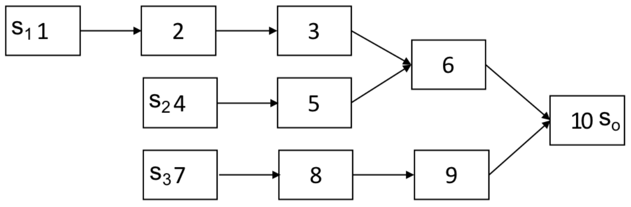

Figure 2: Query graph

Since a query is a combination of individual operators, it is possible that different operators are executed on different machines, such that the data produced by one operator is immediately fed to the next operator that can start processing it. Of course, if two consecutive operators are in different machines, then, some cost needs to be paid to transfer data from one to another. If that cost is high, then it may be better to restrict these operators, by bounding them together. This is known as *chain*.

*Definition 4.4 (**Chain**).* Given a query $\langle \bullet, N, E, I, n_e \rangle$, a chain in a sequence of operators $n_1, n_2, \ldots, n_k$, such that for each i=1. . .k−1:

- $\langle n_i, n_{i+1} \rangle \in E$
- $\nexists \langle n_i, x \rangle \in E$ such that $x \neq n_{i+1}$,
- $\nexists \langle x, n_{i+1} \rangle \in E$ such that $x \neq n_i$, and
- $\nexists \langle s, n_{i+1} \rangle \in I$

Intuitively, a sequence of two or more operations can form a chain only if the succeeding operator has only one input and it comes from the previous the previous one, apart from the first operator of the chain that can have more than one inputs, and the last element, whose output is not a member of the chain.

Example 4.5. In the query presented in Example 4.3, the nodes 7, 8 and 9 can form a chain, since they are consecutive and the only edge that has one endpoint among them is one incoming from 7 and one outgoing to 9. The nodes 3, 6, and 10 cannot form a chain because 6 has an incoming edge that originates from node 5 that is not part of the group.

The nodes of a chain can be collapsed to one node that has as input the input of the first node, and as output the output of the last node. In other words, a chain can be treated as a single operator, and the resulted graph is again a query graph.

The task of an operator can be replicated across different machines such that the different replicas are processing different parts of the records of the input stream. This process is known as parallelization. The parallel versions of the operator are replacing the original operator they parallelize and are called *replicas*. All the replicas of an operator have the same input as the original operator and the same output.

*Definition 4.6 (**Parallelization**).* A k-parallelization of an operator $n \in N$ of a query $\langle \bullet, N, E, I, n_e \rangle$, is a set of replicas $n_1, n_2, \ldots, n_k$ of the operator n, such that a new query can be created of the form $\langle \bullet, N', E', I', n_e \rangle$ for which:

$N' = (N - \{n\}) \cup \{n_1, n_2, \ldots, n_k\}$,

$E' = \{\langle n', x \rangle \mid n' \in \{n_1, \ldots, n_k\} \wedge \langle n, x \rangle \in E\}$
$\cup \{\langle x, n' \rangle \mid n' \in \{n_1, \ldots, n_k\} \wedge \langle x, n \rangle \in E\}$
$\cup \{\langle x, y \rangle \mid \langle x, y \rangle \in E \wedge y \neq n \wedge x \neq n\}$,

$I' = \{\langle s, x \rangle \mid \langle s, x \rangle \in I \wedge x \neq n\}$
$\cup \{\langle s, n' \rangle \mid \langle s, n \rangle \in I \wedge n' \in \{n_1, \ldots, n_k\}$

There are different ways to execute a query depending on what operators are parallelized and what are executed in sequence on the same machine. Each different way of doing this has a different cost. Parallel executions can exploit different cores at the same time, but increase the data communication cost. Execution on the same machine, on the other hand, is increasing the time since the operators are executed in sequence, but saves communication cost. To model the way a query can be executed, we define the notion of an execution plan. Intuitively, the execution plan is a specification of what operators should be chained and what should be parallelized.

Given a query $\langle \bullet, N, E, I, n_e \rangle$, we consider a number of equivalent classes, as many as the number operators, i.e., $|N|$. Each equivalent class is modeled by its representative. By default we assume that every operator belongs to a different equivalent class, which means that each operator is also the representative of the class to which it belongs. Deciding that two or more operators need to be executed on the same machine, can be modeled by simply putting the two operators in the same equivalent class. Merging two equivalent classes is as simple as making the members of the second class have as a representative the one from the first class. This means that we can model the chains by a vector that consists of as many elements as the number of operators in the query, and each element indicates the representative of the equivalent class in which the respective operator belongs.

In a similar fashion we can model the parallelization by indicating the degree of replications that we need to achieve for each operator we need to parallelize. This means that we can also represent the parallelization as a vector of integers, one for each operator. Note that parallelization is done only for operators that are not chained, which means that for such a vector to be valid, an element can have a value more than 1 only if the respective operator is the only member of its equivalent class (which intuitively translates to not being part of a chain).

The combination of the two vectors, one for the chains and one for the parallelization, is what we refer to as a *execution plan.*

*Definition 4.7 (**Execution Plan**).* Given a query $\langle \bullet, N, E, I, n_e \rangle$, an execution plan is a tuple $\langle C, P \rangle$, where C is a vector of $|N|$ elements, each one with a value from N, and P is a vector of positive integers.

An execution plan $\langle C, P \rangle$ is said to be valid if for every i=1..$|P|$ with $P[i]$>1, it holds that $|C[i]|$=1. By abuse of notation, we use $C[i]$ to denote the equivalent class of the operator i-th operator, and the $|C[i]|$ to denote the cardinality of that equivalent class.

The performance of the execution of a query at any given time depends in the data that arrives into its input stream and the execution plan that has been followed. Our goal is to be able to decide at any given moment, given the input stream data, what the best execution plan is.

[Problem Statement]: Given a query $\langle \bullet, N, E, I, n_e \rangle$, and a series of tuples of the form $\langle S,\ p,\ c \rangle$, where S is a set of input streams, p is an execution plan and c is a cost of that plan, we would like to find the best execution plan when the set of input streams is S'.

5 MOIRA ARCHITECTURE

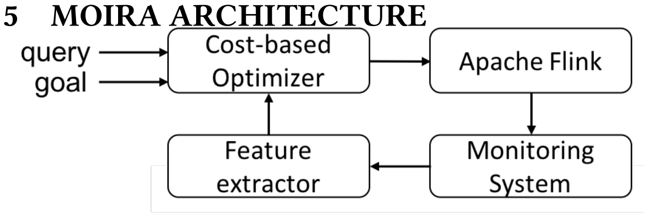

Figure 3: Moira framework

In this section, we present Moira, the framework we propose as a solution to the problem of dynamically applying the best execution plan to the user queries. Moira is a system built, for our scope, on top of Apache Flink, but it can be adapted to any other available streaming framework. As shown in Figure 3, it is composed of three main components and it strictly interacts with Apache Flink. The entry point of the system is the cost-based optimizer component, which takes as input the user query over the available data and her goal. Note that our framework is goal-centered since each query has to be optimized differently according to the user needs. This component finds the execution plan that fits mostly the given user goal and sends the built application to Apache Flink in order to run it. The streaming framework will then deploy the application through the cluster machine.

Meanwhile, the monitoring system is actually polling the system to gather information about the cluster usage, the characteristics of the data, e.g., the income rate, and the status of the running applications. This information will be then sent to the dynamic cost-estimator that will build the features to use in the cost estimation that will check again if the already deployed topology is the best one or if the newly gathered information will allow a better execution plan to be used. In the following, we present each component in details.

5.1 Cost-based Optimizer

The purpose of the cost-based optimizer component is to derive the optimal topology for executing a query based on available resources and an optimization goal that was defined together with the query by the DSP application programmer.

The optimization goal describes how resources should be allocated by the cost-based optimizer. It is defined as a weighted triple of three conflicting optimization areas: cost, latency, and throughput, i.e. $\langle W_{cost}, W_{latency}, W_{throughput} \rangle$. Here the term *cost* refers to the economical allocation of resources, allowing to execute the query using fewer slots and consequently fewer machines in a cluster. This allows either to limit expenses for compute nodes, e.g. if deployed in public cloud, or to reserve resources for deploying other topologies in parallel on a cluster with limited resources. The latency goal quantifies the importance of generating output with a small delay after an event is entering the topology through an input stream, while throughput emphasizes a topology's ability to absorb high input rates. The optimization goal gives the DSP application programmer the possibility to manage resource allocation of a high level of abstraction and at the same time to establish guarantees for throughput and latency. Note that the goal remains constant for the duration of the query execution.

In addition, and regardless of the actual balance of the optimization goal, the cost-based optimizer always aims to prevent resource shortage (processing stalls, out-of-memory) and back-pressure in the processing pipelines, allowing for the unobstructed and continuous execution of the stream topology.

For accomplishing these tasks, the cost-based optimizer determines the resource requirements of every stream operator, based on the operator's algorithm and configuration, and on the data characteristics of the operator's input streams. Cost estimation assumes correlation of data characteristics, i.e. of the characteristics of the flow of stream events, where a finite set of events from the input streams generates a finite set of output events. Formally, the data characteristics of a data stream s are composed of an event rate r_s and a temporal window size w_s. So, the cost function of any operator $n{\in}N$ with i input streams $s_1, s_2, \ldots, s_i$ a parallelism of p, is calculated as

$$Cost_n(\{r_{s_i}, w_{s_i}\}, p) = (cpu_w, size_w, r_{out}, w_{out})$$

where cpu_w denotes the computation complexity of n for handling one temporal window, and $size_w$ corresponds to the storage complexity that n requires internally to manage the state of that window. Computation complexity, storage complexity, and output rate serve as indicators for resource requirements, as well as for latency and throughout estimation.

With this estimation, the optimizer can associate the operators with resources, by (1) aggregating operators into chains, which are deployed into individual slots for execution, and by (2) setting the chain parallelism, such that replica of a particular chain are executed in separate slots.

With this approach, it is possible to propagate the costs bottom-up through the DAG that represents the query. The data characteristics of data sources (DAG leaves) are determined before cost estimation can start. Note that data characteristics of data sources may vary over time. Ideally, the data source characteristics are known upfront, e.g. consulting statistics from previous accesses. Alternatively, sampling is applied to gauge the characteristics of the current data flow. Finally, the API allows the DSP programmer to manually set/override the data characteristics of individual data sources.

The basic strategy of optimization is the following: Start at the DAG leaves and create chains initially containing only a data source. Eventually each chain C will be allocated to a slot, an execution container, which is a unit for resource management, having a single CPU core and a fixed amount of memory, hence any chain's resource requirements may not exceed the resource capacity offered by its allocated slot. Recursively try to extend the current chain across the subsequent operator n, by checking the following conditions.

(1) n is chainable
(2) n supports the parallelism of C
(3) Chaining creates no back-pressure
(4) Chaining creates no resource shortage (e.g. CPU / memory of execution container)
(5) Chaining complies latency constraint of optimization goal
(6) Chaining complies throughput constraint of optimization goal

As a consequence, new chains are started whenever a conflicting condition is detected. If all conditions apply, the chain is extended, thereby minimizing the total number of chains in a topology, and ultimately controlling the cost of execution.

The amortized complexity of cost-based stream optimization is $O(|N|log|N|)$, as it corresponds to one traversal of the DAG with eventual backtracking for adapting chain parallelism.

As mentioned before, the characteristics of data sources are not constant while stream topologies tend to be deployed for a long period of time. In the following we describe how monitoring continuously re-validates the topology against the optimization goals and re-deploys the topology if significant changes are detected.

5.2 Monitoring System

Apache Flink exposes by default a number of metrics on the cluster usage, both for the master node (Job Manager) and for the slaves (Task Manager). It is possible to receive these metrics through JMX, and we store all the available metrics in Apache Lucene [3], which allows the user to perform range queries, which we use for getting the history of the needed metrics. We take into consideration the metrics related to the amount of resources used (e.g., RAM, CPU), and we monitor possible problems (e.g., back-pressure). But among all, the most important for our algorithm is the knowledge of the input and output rate, in both forms of size and number of events. The collected metrics are then forwarded to the feature extractor, which processes them to provide the cost-estimation the right parameter to actually check the redeployment of the topology.

5.3 Incremental learning for Dynamic Cost Estimation

Our solution for dynamic cost estimation unfolds as a learning problem. We formulate it as such to offer a generic, robust and flexible approach suitable for such massive distributed systems.

Dynamic cost estimation is an incremental process. Starting from an initial topology our system employs a machine learning algorithm that uses recent history to build a model of the data (i.e. query specific data: input rate, window size; goal: cost, latency, throughput; topology) and its evolution to estimate predictions. Such a model is updated for each new query, such that the model evolves with the data it represents and is able to accurately trigger a topology reconfiguration. Moreover, this kind of model needs to capture and accommodate the changes in the process generating the data (e.g. learning the correlation among input rate and the topology) to either generate a prediction conditioned on history or to trigger a full model retraining. Of course, this implies either the use of incremental algorithms or the periodic retraining with batch algorithms (expensive in terms of time and resource consumption - critical in such a dynamic cost estimation problem). As a first implemented method, we take care of the history of the input rate for the operator, we perform the average, and we send it to the cost-estimation function for building the new topology.

In addition, we propose a new approach using incremental learning, which is a learning paradigm where computations adjust to any external change to their data automatically. As is it the case in online machine learning, applications need to respond to incremental modifications to data (i.e. update the topology based on the desired cost, throughput and latency). Being incremental, such modifications often require incremental modifications to the output, making it possible to respond to them asymptotically faster than recomputing from scratch.

In such cases, taking advantage of incremental behavior, dramatically improves performance, especially as the system evolves in time for subsequent queries.

In order to decide on a topology change, the Feature Extractor component receives as input the query and the stream parameters (i.e. input rate and window size), the goal (i.e. the desired cost, latency, and throughput) and the measured metrics (i.e. usage, measured input rate etc.) and when needed triggers a topology change through to the cost-estimation function. In order to make the solution flexible and adaptive, we extend from a static policy switch to an incremental learning approach. Such an approach assumes learning the dependencies among the input and output variables of the cost-estimation function, in a pairwise fashion to exploit all the underlying correlations among the measured metrics of the current topology and the current query parameters, for example.

Learning such pairwise functional dependencies among the variables is basically a regression problem. Various methods for both univariate and multivariate regression have been developed, yet it is not trivial how to extend them to cope with the evolving nature of the problem. In other words, there is not a straightforward approach to incremental regression.

In order to explore the underlying relations among the variables in our problem, we started by exploring a linear regression problem, employing a simple incremental Linear Least Squares (LLS) model (described in Figure 4). The linear least squares fitting technique is the simplest and most commonly applied for linear regression and provides a solution to the problem of finding the best fitting straight line through a set of points. It tries to minimize the sum squares of the deviations of a set of n data points:

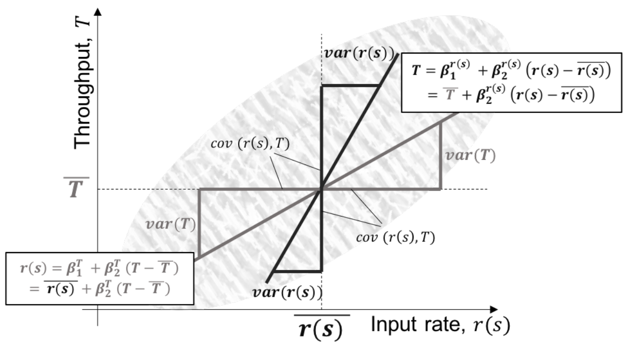
Throughput, T
$\overline{T}$
$var(r(s))$
$T = \beta_1^{r(s)} + \beta_2^{r(s)}(r(s) - \overline{r(s)})$
$= \overline{T} + \beta_2^{r(s)}(r(s) - \overline{r(s)})$
$cov\ (r(s), T)$
$var(T)$
$var(T)$
$cov\ (r(s), T)$
$r(s) = \beta_1^T + \beta_2^T(T - \overline{T})$
$= \overline{r(s)} + \beta_2^T(T - \overline{T})$
$var(r(s))$
$\overline{r(s)}$
Input rate, $r(s)$

Figure 4: Linear Least Squares (LLS) model example

In our case, for example, for the input rate to throughput dependency we can incrementally calculate:

- mean: $\overline{r_n(s)} = \overline{r_{n-1}(s)} + \frac{1}{n}((s) - \overline{r_{n-1}(s)})$, where n is the window size, so $\overline{r_{n-1}(s)}$ is the mean over $n-1$ readings (without the current reading), and $\overline{r_n(s)}$ is the mean of window with the current read
- variance (2nd moment): $m_{2,n} = m_{2,n-1} + (r_n(s) - \overline{r_{n-1}(s)})(r_n(s) - \overline{r_n(s)})$
- covariance: $s_{r(s)T,n} = \frac{n-2}{n-1}s_{r(s)T,n-q} + \frac{1}{n}(r_n(s) - \overline{r_{n-1}(s)})(T_n - \overline{T_{n-1}})$

In general, the nonlinear regression problem assumes a loss function $L^2 = \sum[T - f(r(s), \beta_1, \ldots, \beta_n)]^2$. In particular, for the linear case we have:

$$L^2 = \sum[T - f(r(s)\beta_2 + \beta_1)]^2$$

We learn incrementally the coefficients:

$$\beta_1 = \overline{T} - \beta_2 r(s) \text{ and } \beta_2 = \frac{s_{r(s)T,n}}{m_{2,n}^2}$$

Such a model uses incrementally calculated descriptive statistics and is able to cope with the dynamic updates of topology in our solution. At the moment, the model is considering linear functional relations among the variables, for example, window size and topology equivalence class. For more complex, nonlinear dependencies such a model will fail to capture the underlying relations. In order to cope with such a problem, the regression mechanism could be extended to incrementally approximate also the nonlinearity dependencies among the variables. For example, we are planning to use an incremental Support Vector Machine (SVM). Support vector machines (SVMs) are supervised learning methods used for classification, regression and outliers detection. Among the advantages of support vector machines are: the effective in high dimensional spaces, such as the dynamic cost estimation for topology reconfiguration; the effectiveness in cases where number of dimensions is greater than the number of samples; the use of a subset of training points in the decision function (i.e. support vectors), so it is also memory efficient and suitable for dynamic cost estimation, and can learn highly nonlinear dependencies by using nonlinear kernels to encode the input data.

Given training vectors $r_i(s) \in \mathbb{R}, i = 1, \ldots, n$, and a vector $T \in \mathbb{R}$, SVM solves the following optimal problem:

$$\min_{w,b,\zeta,\zeta^*} \frac{1}{2}w^T w + c\sum_{i=1}^{n}(\zeta_i + \zeta_i^*)$$

$$\begin{aligned} \text{subject to } T_i - w^T\phi(x_i) - b &\le \epsilon + \zeta_i, \\ w^T\phi(r_i(s)) + b - T_i &\le \epsilon + \zeta_i^*, \\ \zeta_i, \zeta_i^* &\ge 0, i = 1, \ldots, n \end{aligned}$$

where ζ_i, ζ_i^* are the slack variables and $\phi(r(s))$ is the kernel. Here training vectors are implicitly mapped into a higher dimensional space by the function ϕ.

Training a support vector machine (SVM) requires solving a quadratic programming (QP) problem in a number of coefficients equal to the number of training examples. For very large datasets, standard numeric techniques for QP become infeasible, this is why an incremental approach is definitely an approach to follow. As an extension to our linear regressor using LLS, we will propose an on-line, incremental SVM alternative, that formulates the (exact) solution for n+1 training data in terms of that for n data and the current data point (i.e. measurement). The incremental procedure would also be reversible and allow "unlearning" of each training sample to remove the impact it had on the dynamic estimation of the functional relations in the topology modifications decision.

The ultimate goal of the incremental learning component of our system is to learn the relevant pair of variables which have a strong contribution to the decision to change the topology. This assumes learning pairwise functions among them while making sure that consensus is reached among any variable. Such an approach assumes building a network (i.e. a graph representation) in which each vertex is a variable (i.e. input rate, window size, topology parallelism, topology equivalence class entries etc.) and connections among vertices (i.e. edges) represent the functional relationship connecting those variables, as learned by LLS and SVM.

Such a method would use an entropy reduction technique [7] to actually select which variable pairs are the most informative and have a strong correlation. The system would allow to actually have a consistent state in the estimation process, i.e. all relations would be satisfied and the topology adjusted accordingly. A sample depiction of the system is provided in Figure 5.

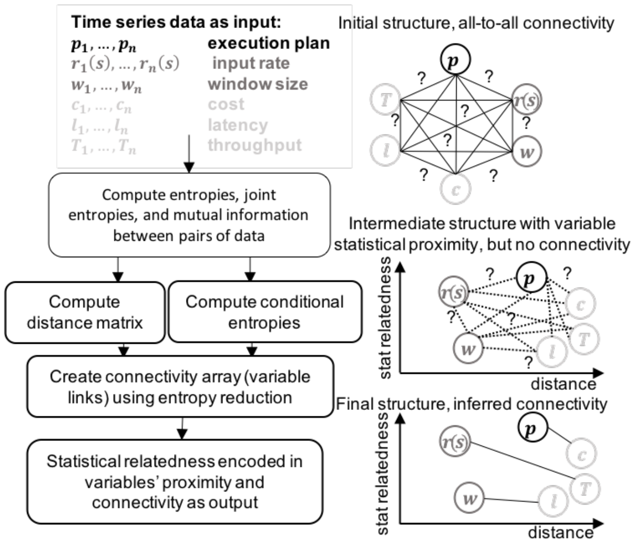
Time series data as input:
$p_1, \dots, p_n$ execution plan
$r_1(s), \dots, r_n(s)$ input rate
$w_1, \dots, w_n$ window size
$c_1, \dots, c_n$ cost
$l_1, \dots, l_n$ latency
$T_1, \dots, T_n$ throughput
Compute entropies, joint entropies, and mutual information between pairs of data
Compute distance matrix
Compute conditional entropies
Create connectivity array (variable links) using entropy reduction
Statistical relatedness encoded in variables' proximity and connectivity as output
Initial structure, all-to-all connectivity
Intermediate structure with variable statistical proximity, but no connectivity
Final structure, inferred connectivity
stat relatedness
distance

Figure 5: Entropy reduction technique for correlation learning

This system is composed of a pipeline which feeds time-series sensory input; compute statistics for individual and pairs of sensors (entropy and mutual information); computes statistical distance and conditional entropies to extract statistical relatedness; creates a connectivity array using entropy reduction (minimization), as shown in the left panel. In the right panel, the underlying functionality behind the correlation learning is depicted.

Such an approach will enable the dynamic cost estimator to learn which of the available variables are correlated, learn the pairwise relations among them and then use these learned relations to take a decision on the topology update policy based on the consensus of the learned functions.

6 EXPERIMENTS

We present here a set of experiments to validate our framework for a dynamic cost estimation of the resources given the user goal. We show the results with respect to the static cost estimation and to Apache Flink by itself, describing the advantages and disadvantages of our system. We analyze both latency and throughput, which are the most used metrics to check the features of a system.

[System Description] The experiments were run over a cluster of 4 machines running RedHat 6.5. All the machines were featured with 126 GB of main memory and 24 cores. We perform our experiments with the standard version of Apache Flink 1.4.2, and we apply our framework on top of it. *base* is the original 1.4.2 version, while in *static* is enabled the static estimation at the deployment of the application, and *dynamic* describes the performance of the enabled dynamic goal-oriented cost-based estimation. If *static* performs a standard starting estimation, with *dynamic* is running another application which actually takes care of the rescheduling of the jobs. We perform our evaluation with the TPC-H benchmark [6], creating the data through the data generator and applying some of the queries they propose. We tested our framework with a constant input rate of 4000 elements/second for 2 hours.

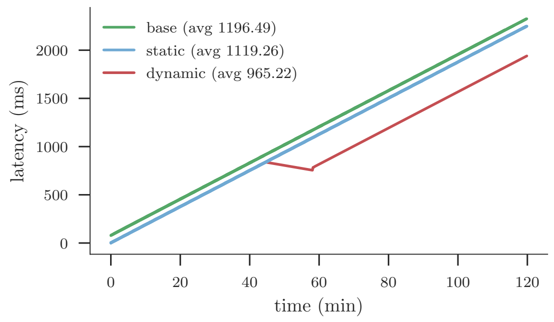

Figure 6: Latency evaluation for TPC-H query

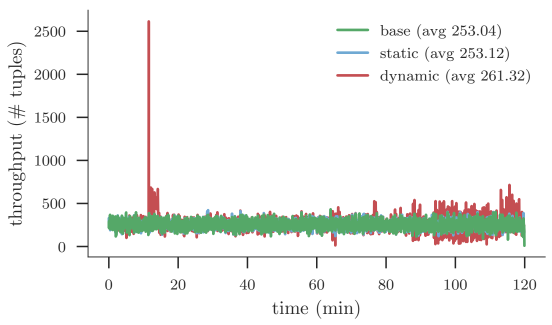

Figure 7: Throughput evaluation for TPC-H query

Figure 6 and Figure 7 show the performance of our framework with the TPC-H query number 3. The former presents an analysis on the latency, while the latter on the throughput of the query. For what concerns the *dynamic* estimation, we optimized the latency for the first plot, while for the throughput figure, the optimization goal was the throughput. For the latency chart, first of all, we see that the latency is high, and it is still increasing. That is because the number of elements sent to Flink was high for such a query, and its execution was computationally intensive. But this is not a problem caused by our implementation since even the base version reports the same behavior. We can see that the *static* estimation is giving a small advantage w.r.t. the *base* version, which proves that the cost function we implemented is a good starting point. Moreover, the *dynamic* estimation shows better performance after 45 minutes, when the job was rescheduled. We were using a constant input rate, so the job was rescheduled only once. But, with a fluctuating input rate, the improvement would have been even bigger, since the job resources would have followed the input rate.

For what concerns the throughput, we see that both the *static* estimation almost follows the *base* version, which is stable during all the execution, but with a slightly higher the number of output tuples per second. The *dynamic* estimation shows the same behavior of the *static* up to the rescheduling (after 10 minutes). Then, it presents a peak, due to the new configuration that allows it to output the queued elements. After that, it maintains a constantly higher throughput for the remaining execution time.

7 CONCLUSIONS

In this work, we considered the problem of dynamically allocating the resources for a streaming application in a cluster environment. The novelty of this work is that the amount of allocated resources is bound to a goal defined by the user, which consists of a value for cost, latency, and throughput. Moira, the framework we propose, is a system built on top of Apache Flink, and from an initial static estimation of the user query which deploys the first execution plan, it dynamically accesses the metrics exposed by Flink and the characteristics of the incoming data, and then it will use them as features to reach the user goal. If the new cost-estimation suggests the deployment of a new plan, Moira actually reschedules the job with the new topology. We propose an insight of our solution and then we discuss deeply the idea we have to improve the system as a future work.

REFERENCES

[1] Apache Flink. http://flink.apache.org.
[2] Apache Heron. http://heronstreaming.io.
[3] Apache Lucene. http://lucene.apache.org.
[4] Apache Spark. http://spark.apache.org/streaming/.
[5] Apache Storm. http://storm.apache.org.
[6] TPC-H. http://www.tpc.org/tpch/.
[7] C. Axenie, C. Richter, and J. Conradt. A self-synthesis approach to perceptual learning for multisensory fusion in robotics. *Sensors*, 16(10):1751, 2016.
[8] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. Peng, and P. Poulosky. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1789–1792, 2016.
[9] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. *PVLDB*, 10(12):1825–1836, 2017.
[10] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. DRS: dynamic resource scheduling for real-time analytics over fast streams. In *35th IEEE ICDCS 2015, Columbus, OH, USA, June 29 - July 2, 2015*, pages 411–420, 2015.
[11] Z. Han, R. Chu, H. Mi, and H. Wang. Elastic allocator: An adaptive task scheduler for streaming query in the cloud. In *8th IEEE International Symposium on Service Oriented System Engineering, SOSE 2014, Oxford, United Kingdom, April 7-11, 2014*, pages 284–289, 2014.
[12] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR 2011, Asilomar, CA, USA, January 9-12, 2011*, pages 261–272, 2011.
[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, 2011.
[14] A. Khoshkbarforoushha, A. Khosravian, and R. Ranjan. Elasticity management of streaming data analytics flows on clouds. *J. Comput. Syst. Sci.*, 89:24–40, 2017.
[15] A. Khoshkbarforoushha, R. Ranjan, Q. Wang, and C. Friedrich. Flower: A data analytics flow elasticity manager. *PVLDB*, 10(12):1893–1896, 2017.
[16] S. Perera, A. Perera, and K. Hakimzadeh. Reproducible experiments for comparing apache flink and apache spark on public clouds. *CoRR*, abs/1610.04493, 2016.
[17] G. R. Russo. Towards decentralized auto-scaling policies for data stream processing applications. In *Proceedings of the 10th Central European Workshop on Services and their Composition, Dresden, Germany, February 8-9, 2018.*, pages 47–54, 2018.
[18] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop YARN: yet another resource negotiator. In *ACM SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 5:1–5:16, 2013.
[19] Z. Zhang, R. T. B. Ma, J. Ding, and Y. Yang. ABACUS: an auction-based approach to cloud service differentiation. In *2013 IEEE IC2E 2013, San Francisco, CA, USA, March 25-27, 2013*, pages 292–301, 2013.