Planner: Cost-efficient Execution Plans Placement for Uniform Stream Analytics on Edge and Cloud

Laurent Prosperi supervised by Alexandru Costan

September 12, 2018

The general context

The age of offline only Big Data analytics is over, leaving room to online and interactive processing. Indeed, the proliferation of small sensors and devices that are capable of generating valuable information in the context of the Internet of Things (IoT) has exacerbated the amount of data flowing from all connected objects to cloud. The IoT applications raise specific challenges, as they typically have to handle small data arriving at high rates, from many geographical distributed sources, that need to be processed and acted upon with high reactivity.

Two axes are currently explored *separately* to achieve these goals: *cloud-based analytics* and *edge analytics*. The traditional approach of sending the data from edge devices to clouds for processing was largely adopted due to simplicity of use and the perception of unlimited resources. There exists a plethora of stream processing engines (SPEs) like Spark[37], Flink[20], Kafka[30], Storm[1]. In this case, the edge devices are used only to forward data to the cloud. However, pushing all the streams to clouds incurs significant latencies. On the other hand, since edge devices are getting more powerful, another vision is to perform a part of the analysis at the collection site. Such an approach allows to take local decisions and enables real-time analytics. Several edge analytics engines emerged lately (i.e. Apache Edgent [2], Apache Minifi [3]) enabling basic local stream processing IoT devices.

More details on the related work are presented in Section 8.

The research problem

More recently, a new *hybrid* approach tries to *combine both cloud and edge analytics* in order to offer better performance, flexibility and monetary costs for stream processing. First, processing live data sources can offer a potential solution that deals with the explosion of data sizes, as the data is filtered and aggregated locally, before it gets a chance to accumulate. Then, partial results are sent to the cloud for processing. Batch processing is still used to complement this online dimension with a machine / deep learning dimension and gain more insights based on historical data.

However, leveraging this dual approach in practice raises some challenges mainly due to the way in which stream processing engines organize the analytics workflow. Both edge and cloud engines create a *dataflow graph of operators* that are deployed on the distributed resources. In order to execute a request over such hybrid deployment, one needs a specific plan for the edge engines, another one for the cloud SPEs and to ensure the right interconnection between them thanks to an ingestion system (e.g. Kafka). Hence, the burden of connecting systems together and dividing the computation between them is left to users. Moreover, manually deploying this pipeline can lead to sub-optimal placement with respect to the network cost.

Your contribution

In this report, I argue that a *uniform approach* is needed to bridge the gap between cloud SPEs and edge analytics frameworks in order to leverage a single, transparent execution plan for stream

processing in both environments. First, I introduce a **system model** for stream processing and **a cost model** for the streams flowing from an edge operator to a cloud-based one (Section 3); then I **formulate the problem of operator placement** for an execution graph across distributed edge and cloud resources, with the objective of minimizing the makespan via the previous cost-model (Section 3.1); eventually I present **Planner**, a streaming middleware capable of automatically deploying fractions of the computations across edge and clouds, as a proof of concept of these models and principles (Section 4).

A paper summarising the research carried out during this internship and presented in this report has been submitted to the WORKS workshop, held in conjunction with IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis SC'18. Moreover, I have developped a generic testbed in order to do experiments over an edge-cloud environment on Grid5000. It eases and improves reproductibility of the deployment and the processing of experiments over grid computing.

Arguments supporting its validity

I implement a proof of concept of Planner with bindings with Flink for the cloud part and Edgent for the edge one. Then, I simulate an hybrid platform (i.e cloud and edge nodes) on the Grid'5000 platform and I run micro-benchmarks based on real worl data (New York City Taxi dataset [22]). This benchmarks show that Planner outperforms state-of-the-art solutions by more than 40%. More details are available in Section 6 and a comprehensive discussion of the approach is provided in Section 7.

Summary and future work

In this paper, I address the challenges of hybrid stream processing (which combine both Cloud computing and Edge analytics). With this paradigm, computation placement is usually done manually. Besides being a burden for users this can lead to sub-optimal computation placement with respect to network cost between Edge and Cloud.

I argue for a uniform approach in order to leverage a single, transparent and automatic execution on both environments. I provide a model of a hybrid infrastructure and a generic model of the network cost over Edge and Cloud links. From them, I define a plan placement problem in order to minimize the makespan and the network cost. I restrict this placement into a local one which processes (groups of) agents independently in order to improve scalability. Then I introduce a new raw-data locality-aware optimization which preserves the semantics of the computation and I derive a scheduler. As a proof of concept I implement Planner, a streaming middleware that automatically partitions the execution plans across Edge and Cloud. I evaluate our work by setting up an hybrid architecture on Grid'5000 where I deploy Planner with Apache Flink and Apache Edgent. By running real-world micro-benchmarks, I show that Planner reduces the network usage by more than 40% and the makespan by 15%

As future work, I plan to add optional workflow annotations and then enable support for heterogeneous sources. Moreover, I plan to introduce a new optimization (based on a weak equivalence of computation that guarantees not to introduce new behaviours) in order to export some stateful operators (e.g. reduce). Finally, I plan to switch from static placement to an adaptive one where metrics about operators (e.g selectivity) and infrastructure (e.g average throughput) are refined at runtime in order to increase the accuracy of the cost model and to periodically trigger the plan placement computation.

Notes and Acknowledgements

This report have been adapted from a workshop article [] therefore it is written in English.

I wish to thanks my supervisors Alexandru Costan and Gabriel Antoniu. I am gratefull to Pedro Silva for its proofreading. Thanks to Yacine Taleb, Nathanaël Cheriere and Luc Bougé for many helpful discussions. Experiments presented in this paper were carried out using the Grid'5000

testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

1 Context and motivation

This section provides the background for our work and introduces the problem statement.

1.1 Infrastructure

A common infrastructure for stream processing is split into two layers: the Edge, hosting the devices which generate data, and the Cloud, used for ingestion (i.e., gathering data from devices and aggregating it into streams) and processing (i.e., the analytics on the incoming streams). While the Edge devices are becoming more and more resourceful and energy-efficient, the Cloud has (order of magnitude) more computing power. In this paper, I assume that the Cloud has enough resources to process the whole dataset after data collection. What remains prohibitive, however, is the high latency of the wide area networks connecting the Edge and the Cloud, leading to significant network usage and costs, and increased end-to-end processing latencies.

1.2 Data streaming

A data *stream* is an unbounded collection of atomic items. Processing *operators* consume streams and produce values (e.g., reduce, aggregate) or new streams (e.g., map, filter) using some User Defined Functions (UDFs). For instance, a *map* operator can transform a stream of temperature measurements into a stream of heat wave alerts). An operator source only produces streams (e.g reads items from files, devices) and, therefore, it irrigates the computation pipeline with data. An operator data sink only consumes data (e.g, writing data in a file).

Operators can be split into two categories: *stateless* and *stateful*. A stateless operator (e.g., map) processes items independently, one at a time, and consequently doesn't need to save its "state" in case of failures. In contrast, a stateful operator (e.g., reduce) processes items according to its local state (e.g., a rolling sum) or aggregates items and processes them by bucket (e.g., windows[15]).

1.3 Stream processing graphs

A common abstraction for modeling stream computations are the the *stream graphs*. They are directed acyclic graphs composed of operators (the vertices) interconnected by data streams (the edges).

I refine the notion of stream graph into a *weighted* DAG in order to model the network usage induced by streams and their sources (i.e, the average rate of events flowing through a stream). More formally $G_{sp} = (V_{sp}, E_{sp}, \mathscr{W}_{sp})$ denotes a stream graph where V_{sp} is the set of operators, E_{sp} is the set of streams and $\mathscr{W}_{sp} : E_{sp} \cup Sources \longrightarrow \mathbb{R}^+$ is the network usage. An operator o is composed of an UDF f_o and of a type τ_o (e.g., map, reduce) that describes an input and an output contract [17]. An input contract describes how the input items are organized into subsets that can be processed independently (e.g, by parallel instances) whereas an output contract denotes additional semantics information of the UDF (e.g, indicates that a UDF is stateless). The output contract can also give bounds for the selectivity s_o of an operator o. The selectivity[26] is the ratio of the output items rate over the input one of an operator (e.g, an operator issuing two items for one input has a selectivity of 2). For the sake of clarity, I summarize operator characteristics in Table 1.

1.4 Problem statement

In order to run a computation, one needs to deploy the stream graph on the underlying infrastructure, i.e, to place operators on nodes. This mapping is called the *execution plan*. SPEs today can

Type	Selectivy s_o	Locally-replicable $\mathscr{R}o$	Combination a_{τ_o}
Map	1	1	Id
FlatMap	≥ 0	1	Id
Filter	≤ 1	1	Id
Split	1	1	Id
Select	1	1	Id
Fold	1	0	Id
Reduce	1	0	Id
Union	1	1	$\sum$
Connect	1	0	min
Window	parametric[a]	0	Id

Table 1: Operators overview. *Map:* takes one item and produces one item. *FlatMap:* takes one item and produces zero, one, or more items. *Filter:* takes one item and produces zero or one items. *Split:* splits a stream into two or more streams. *Select:* selects one or more streams from a split stream. *Fold:* combines the current item with the last folded value and emits the new value. *Reduce:* combines the current item with the last reduced value and emits the new value with an initial value. *Union:* union of two or more data streams. *Connect:* connects two data streams retaining their types. *Windows:* groups the data according to some characteristic.

do such schedules either for the Cloud or for the Edge, *separately* (e.g., Spark deploys its execution plan on the Cloud, Minifi on the Edge).

In the case of complex hybrid infrastructures mixing both Edge and Cloud, however, the burden to define the partial computations, i.e., subgraphs, to be executed on each infrastructure, is delegated to the user. In most cases, this leads to sub-optimal performance.

2 Models

In this section, I present the abstractions I leverage to model the resource graph on which the stream computation relies, as well as the network cost model that our approach aims to minimize. In Table 2, I summarize the notations used throughout the paper.

2.1 Resources model

The computing and network resources used to execute a computation can be represented as a directed graph:

$$G_{sys} = (Devices \cup \varsigma, E_{res}, \mathscr{W}_{res}) \tag{1}$$

where $Devices = \{d_i\}_{i \leq f}$ represent the set of Edge devices and ς represent the Cloud computing units. I aggregate the Cloud nodes in one logical node ς since I consider the Cloud powerful enough to run the full workflow (after collecting the data) and because I delegate the inner placement to the SPE (i.e., an engine like Spark or Flink will map the subgraph identified by our approach for cloud execution on the actual cloud nodes). E_{res} denotes the physical links between Edge devices and Cloud: $Devices \times \varsigma \subseteq E_{res}$. I do not model other links since I focus on the bottlenecks of the network between Edge and Cloud. Finally, $\mathscr{W}_{res} : E_{res} \to \mathbb{R}^+$ represents the cost of the transmission of an item through a link. I use a per item approach since the size of an item can arbitrary vary according to the nature of the UDF and there is no generic introspection mechanism to distinguish the shape of an item (e.g, items are arbitrary Java objects in Flink).

Special care should be taken when modeling sources, as they can produce data coming from multiple physical nodes. For instance, let us take some workflow monitoring crops where a data source aggregates (thanks to an ingestion queue) the temperature coming from several connected thermometers in order to increase reliability. I model such data dependencies by defining for each

source $s \in Sources$ the group $g(s)$ of Edge devices that host the raw data used by s. Reciprocally, I define $g^{-1}(u)$ the group of sources using raw data hosted in the node u.

With this resource model, I can use different cost functions depending on the metric I want to optimize (e.g, I can use an energetic cost per item or the latency of the link).

2.2 Network cost model

Due to the black-box nature of UDFs, I approximate the network usage of the streams over the links between Edge and Cloud. The network usage of a stream $(i, j) \in E_{sp}$ depends on the input rate of operator i, the selectivity of i and its type of τ_i. This is formally expressed as follows:

$$\mathscr{W}_{sp}(i,j) = \begin{cases} s_i * a_{\tau_i}\left(\mathscr{W}_{sp}(\xi_{i_1}), \ldots, \mathscr{W}_{sp}(\xi_{i_l})\right) & \text{if } i \notin Sources \\ \mathscr{W}_{sp}(i) & \text{otherwise} \end{cases} \tag{2}$$

where $(\xi_{i_1}, \ldots, \xi_{i_l})$ are the input streams of i in G_{sp}, $a_{\tau_i} : \mathbb{R}^k \to \mathbb{R}$ is the weight aggregation function for an operator of type τ_i and where k is the input arity of i. a_{τ_i} describes the combination of network usage of incoming streams (cf. Table 1). Furthermore, $\mathscr{W}_{sp}(i)$ denotes the average event rate produced by a source i, which should be estimated using static analysis or by probing the source at runtime. Finally, $\mathscr{C}_{\xi}^{l} = \mathscr{W}_{res}(l) * \mathscr{W}_{sp}(\xi)$ denotes the communication cost of a stream $\xi \in E_{sp}$ flowing through a link $l \in E_{res}$.

3 Uniform Stream Graph Placement

Our key idea for finding the ideal cut (between Cloud and Edge) of the stream graph is to solve an optimisation problem for placement, while trying to minimise the network cost. I formulate this problem and its optimisations in this section.

3.1 The placement problem

Not all operators can be executed on Edge devices due to their limited computing power, memory, battery life or simply because some operators are not supported by the Edge analytics frameworks. Therefore, for each Edge device d I encode such a restriction in a constraint C_d. A stream graph H can be placed to the device d if and only if it satisfies the constraint C_d, denoted by $H \models C_d$.

The placement problem aims at minimizing the global communication cost (and, consequently, the stream processing makespan) by executing some computations on the Edge devices instead of moving all the data to the Cloud. This is formally expressed as follows:

$$\min_{\mathscr{P}} \sum_{\{V_{sp}^d\} \in \mathscr{P}} \sum_{\xi \in E_{sp} \cap (V_{sp}^d \times (V_{sp} \setminus V_{sp}^d))} \mathscr{C}_{\xi}^{(d,\varsigma)} \tag{3}$$

subject to:

$$V_{sp}^d \models C_d$$

where ξ is a stream flowing over a link between Edge and Cloud and $\mathscr{P}$ denotes the set of operators that should be executed on each Edge device. A placement $\mathscr{P}$ is defined as follows:

$$\mathscr{P} = \bigcup_{d \in Devices} \{V_{sp}^d\} \tag{4}$$

where V_{sp}^d is the subgraph of G_{sp} mapped to the device d. The remaining part of the workflow will be executed in the Cloud.

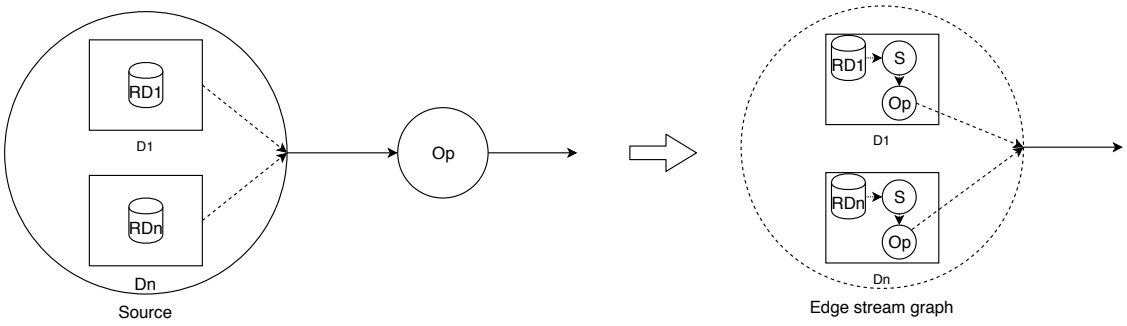

Figure 1: Raw-data locally-aware optimization where Op is an operator, D_i denotes an edge device and RD_i is the raw data (hosted by D_i) used by the source.

I can define a placement problem as a conjunction of independent placement problems for each device by restricting the constraint placement C_d for each device d. A *local placement problem* for a device d is then formally stated as follows:

$$\min_{V_{sp}^d} \sum_{\xi \in E_{sp} \cap (V_{sp}^d \times (V_{sp} \setminus V_{sp}^d))} \mathscr{C}_\xi^{(d,\varsigma)} \tag{5}$$

subject to:

$$V_{sp}^d \models \tilde{C}_d$$

where the restricted constraint is as follows:

$$\tilde{C}_d = C_d \wedge Source(X) \subset g^{-1}(d) \tag{6}$$

$Source(X)$ is the set of sources of the candidate subgraph and C_d is the previous constraint for the device d.

3.2 Locality-awareness optimization

I introduce a locality-aware optimization in order to address the *local placement problem*. This optimization aims at deploying operators near the raw data in the Edge by allocating one version of the operator per device and to further collect the results in one stream inside the ingestion system (Figure 1).

An operator o is *locally-replicable* (and denoted by the indicator function $\mathscr{R} : V_{sp} \longrightarrow \{0,1\}$) if and only if the former optimization applied to o preserves the equivalence of computation defined using the following equivalence relation.

Computation equivalence. In order to compare computations done by two stream graphs composed of deterministic[b] operators, I define a notion of equivalence based on the outputted items. I state that two stream graphs G_1 and G_2 are equivalent if for any trace $\mathscr{T}$ (an ordered sequence of input items for each input streams) $G_1(\mathscr{T}) == G_2(\mathscr{T})$, where $G_1(\mathscr{T})$ denotes the content of the output streams produced by G_1 when applied to $\mathscr{T}$. Two streams are equal if they coincide for any finite prefix.

Theorem 1. *A stateless deterministic operator is locally-replicable.*

The idea of the proof for operator o is as follows. Let us split the trace in sub-traces: $\mathscr{T} = \bigcup_i \mathscr{T}_i$ (one for each device involved). Now, by combining the local results with the same interleaving

[b]The notion of equivalence is not defined in the non-deterministic case. Some weak-equivalence can be defined by defining $G_1(\mathscr{T})$ as the set of possible output traces.

$\bigcup_i o(\mathscr{T}_i)$ I obtain $o(\mathscr{T})$ since the output of o only depends of the current input item (because o is stateless and deterministic). An overview of locally-replicable operators is available in Table 1.

Nothing can be said for stateful operators due to the unknown behaviour of the UDFs. Indeed, a stateful operator can be locally-replicable (e.g, the identity map can be simulated with a reduce operator by ignoring its state). In turn, I can exhibit the following situation where an operator is not locally-replicable. Let us take two devices one providing the odd numbers and the other the even ones, and a source which outputs the data for both devices. Eventually, the produced stream is consumed by a reduce operator computing a rolling sum (i.e, the sum of the current item with the last reduced value). If I take a trace that alternates even and odd value, then before the optimization the output stream is composed of odd numbers and after it is composed of even numbers.

4 Planner Overview

I implement this approach into a proof of concept, called Planner - a streaming middleware unifying Edge and Cloud analytics. Planner automatically and transparently delegates a light part of the computation to Edge devices (e.g., running embedded edge processing engines) in order to minimize the network cost and the end-to-end processing time of the Cloud based stream processing. It does so as a thin extension of a traditional cloud-based SPE (e.g., Apache Flink in our case) to support hybrid deployments, as shown in Figure 2.

In this section, I first introduce the design principles backing such an approach, then I provide an architectural overview of Planner. I particularly zoom on its scheduler, which is responsible for finding cost-efficient cuts of execution plans between Edge and Cloud.

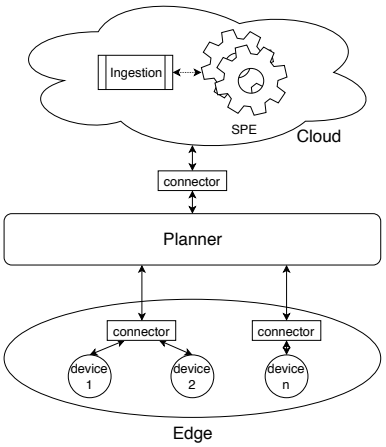

Figure 2: A hybrid infrastructure where Planner is used to deploy parts of the computation on edge devices. The connectors are small pieces of software used to plug Planner with other cloud-based analytics systems (e.g., Apache Flink).

4.1 Design principles

Planner has been designed according to the following three principles:

4.1.1 A transparent top-down approach

Streaming applications are submitted by users unchanged to the SPEs. The latter translate them into streaming graphs (execution plans) that Planner intercepts and divides between Cloud and Edge. Therefore, Planner is fully transparent for users. As a side effect, this top-down approach is well suited especially for Cloud plans, which tend to be more expressive and complex than the Edge ones, as they leverage many global stateful operators (e.g., window based operators).

4.1.2 Support for semantic homogeneity

Edge devices are considered to be homogeneous in terms of semantics of the computation, i.e., each device provides data to the same group of sources ($\exists A \subset Sources, \forall d \in Devices, g^{-1}(d) = A$). This restriction is a drawback of the transparency. Indeed, the graphs submitted to Planner are limited by the expressivity of the SPE. And most SPEs are not designed to track the provenance

of the data produced by sources. However, this limitation does not apply to the former resource model which supports heterogeneous types of devices.

4.1.3 Support for interaction with SPEs

Planner is system agnostic: the core of the middleware is not aware of the details of the cloud or edge SPE but only of some abstract representation. This allows any backend to be easily plugged to Planner thanks to specific external connectors. A connector is responsible of the deep interaction with external systems (e.g exporting plans to Planner and importing and executing instructions from Planner).

4.2 Architecture overview

Planner takes as input a stream graph expressed in the "dialect" of an SPE and outputs multiple stream graphs: one for the Cloud SPE and one per (groups of) Edge device(s). To enable this behavior, Planner is structured in three layers (Figure 3): an Abstraction Layer, a Cost Estimator, and a Scheduler.

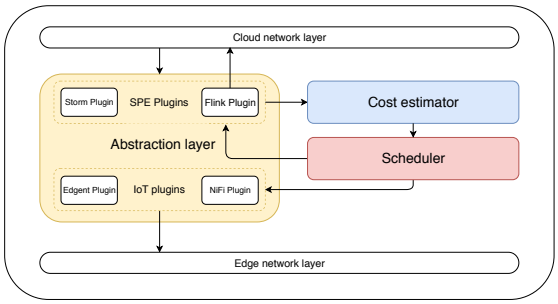

Figure 3: The Planner architecture.

4.2.1 The Abstraction Layer

translates input stream graph (coming from the cloud SPE connectors) to an abstract graph which is an instance of the one presented in Section 1.3. However, some relevant information usually lacks from the cloud SPE streaming plan (e.g. operator selectivity is not present in Apache Flink plan). In such a case, I provide default values based on the operator type (using Table 1). Conversely, this layer also translate an inner abstract representation to cloud or edge SPE "dialects".

4.2.2 The Cost Estimator

enhances the abstract stream graph with the network usage of the streams. This is computed by applying the model represented in the Equation 2 in a topological order starting from the sources. It is also in charge of maintaining the transmission costs of the links between Edge and Cloud (in the current implementation I assume a constant transmission cost).

4.2.3 The Scheduler

selects which part of the stream graph should be deployed to the Edge in order to minimize the overall cost of the placement by applying the optimization presented in Section 3.2. Moreover, it restricts the placement constraints (see Equation 5) in order to process each Edge device independently.

I use a two-phase approach, as shown in Algorithm 1. Firstly (lines 4-14), I do a traversal of the graph and extract the maximal subgraph that can be placed on an Edge device with respect to the constraint satisfaction. $Current$ denotes the set of operators that will be moved to the device d. Note that if a data source has no successor in $Current$, then it will remain in the Cloud. Moreover, $Current$ verifies:

$$Pred(Current) \subset Current \wedge H \models \tilde{C}_d \wedge Current \cap Sinks = \varnothing \tag{7}$$

where $Pred(o)$ is the set of the predecessors of o in G_{sp} and H is the subgraph of G_{sp} induced by $Current$. $Opened$ denotes the operators to process such that $\forall x \in Opened, Pred(x) \subset Current$. $Closed$ denotes the operators that have been processed.

Algorithm 1 Placement algorithm for an Edge device d

Require: $S = g^{-1}(d)$
1: $Current \leftarrow S$
2: $Opened \leftarrow \bigcup_{s \in S} \{x \in Succ(s) \mid Pred(x) \subset Current\}$
3: $Closed \leftarrow S$
4: **while** $Opened \neq \varnothing$ **do**
5: Pick u in $Opened$
6: Let H the subgraph of G_{sp} induced by $Current \cup \{u\}$
7: **if** $H \models \tilde{C}_d$ **then**
8: $Current \leftarrow Current \cup \{u\}$
9: $Applicants \leftarrow \{x \in Succ(u) \mid Pred(x) \subset Current\} \setminus Sinks$
10: $Opened \leftarrow Opened \cup Applicants \setminus Closed$
11: **end if**
12: $Opened \leftarrow Opened \setminus \{u\}$
13: $Closed \leftarrow Closed \cup \{u\}$
14: **end while**
15: $Border \leftarrow N(Current) \setminus Current$
16: F is the subgraph of G_{sp} induced by $Border \cup Current$
17: **return** a minimum (S, $Border$)-cut in F

Secondly, I compute a minimum (S, $Border$)-cut (using the Stoer–Wagner algorithm [35]) where $Border$ denotes the external neighbours of $Current$ operators. In the implementation (unlike the model) I do not limit the computing power of an edge device since refining the resources needed to run an operator would require to analyze arbitrary UDFs using static analysis or online profiling.

Let us discuss the optimality of the previous algorithm with respect to the *local placement problem* (see Equation 5), depending on the nature of the constraint. If $\tilde{C}_d$ does not encode any notion of capacity of the Edge devices (for instance the constraint could be: "the operator is locally-replicable" or "the Edge SPE cannot run this kind of computation") then this algorithm gives an optimal placement by definition of min-cut. Otherwise, if some capacity constraint is encoded in the constraint (e.g., maximun memory consumption), there is an underlying knapsack problem. One way to improve this algorithm is to refine the selection of u (line 4).

The complexity $T(n)$ of the algorithm does not depend on the number of Edge devices and it is defined as $O(n^2 \log n + n\alpha(n) + nm)$ where n is the number of operators, m is the number of streams and $\alpha(n)$ denotes the complexity of the constraint satisfaction[c]. In practice, the number of operators is small (from tens to a few hundreds) and stream graphs are commonly sparse because most of the operators have an input (and output) arity of one or two.

In order to process all the devices, I simply apply the former algorithm for each device. This naive approach leads to a complexity of $O(|Devices| * T(n))$. Therefore, it is linear on the number of devices (which can reach several millions). However, I can refine the previous algorithm, in order to scale more, by grouping devices and by applying the Algorithm 1 for each group. Grouping should be done according to device nature, i.e, a group of connected cars and a group of connected thermometers. More formally, a group is the set of devices that share the same $g^{-1}(d)$). Finally, if the characteristics (e.g., computer power) of the devices in the same group are not close enough, I can use a hierarchical structure. I thus create subgroups based on characteristic similarities; the input graph of a subgroup is the output stream graph of its parent group.

[c]We can obtain $\alpha(n) = O(n)$ with simple constraints expressing computing power or memory limitations.

5 Flink

Apache Flink is a distributed processing processing engine, it can support event processing, stream processing and batch processing. It is one of the mostly used open source SPE. Moreover, it run on any kind of clusters. In this section, I present the Flink architecture and how it can be plugged with Planner.

5.1 Architecture overview

A Flink deployment (see. Figure 13) is split in two parts: the *client part* in charge of launching/monitoring an application and the *runtime part* in charge of running applications (potentially coming from multiple users). Most commonly, the runtime is running on a cluster and the client is running by the user machine (e.g. its laptop).

An user writes its application in Java (see. Figure 14) then he submits it to the *Flink client* which will translate the application into a program dataflow, optimize it and then submit it to the *Flink runtime* thanks to the *jobmanager*. The runtime is in charge of the durability of data during the time of computation, of the failure mitigation and of course of processing the program dataflow. A Flink runtime is composed of a jobmanager that supervise the computation (namely deploy operators and monitor fault-tolerance) and of several taskmanagers that do computations. Commonly taskmanagers are deployed on multiple machines (to hundred or thousand of them).

5.2 Plan execution chain

Now, let us dive into a common execution (see Figure. 15) of a *Flink application* (the one targeted by Planner). There is plethora of slight variations of execution (e.g. stream processing, batch processing, interactive mode for both). I focus on the default non-interactive stream processing execution. In this case, the application is compiled to a JAR which is submitted to a Flink client (thank to a command-line interface or by an other JAVA program).

Then, the client loads the JAR, extracts the main class (according to some parameter) and runs the "main" method of this class. This triggers the *execute* function of the *StreamExecutionEnvironment* which basically enriches operators with some configurations (e.g. operators parallelism) and outputs a set of *StreamTransformation*. A *StreamTransformation* is a line stream graph (for instance a source linked to its next map), there are a *transformation* object per *DataStream* in the application code (see Figure 14).

This set of simple dataflow graphs is aggregated by the *StreamGraphGenerator* in a *stream graph*. Moreover, a first step of optimization is done here, namely grouping virtual operators (for instance split or select one) with a concrete one (e.g. map, filter). By doing so, every remaining nodes are UDF-based operators that will transform streams and not just manage them (e.g. redistribute data across node). Afterwards, comes the major plan optimization pass during which the stream graph is transformed into a *JobGraph*. During this transformation operators are grouped in logical ones in order to enhance data-locality (for instance filter and map operators are always groups with there neighbours). Moreover *slot sharing* behaviour is configured, i.e, operators are marks in order to run on the same slot (a taskmanager thread) if possible or on contrary not to run on the same taskmanager. This job graph is then submitted by the client to the jobmanager.

Eventually, this jobgraph is transformed in an *execution graph* in order to fit the Flink cluster characteristics (e.f configuring the correct checkpointing, matching rules on slot with cluster slots). Then, the execution graph is schedule on taskmanagers, i.e, operators are maps to taks slots.

5.3 Modifying Flink

In order to bind Flink with Planner, I have first modified the flink client and then implemented a dedicated connector. Most of this work have been slow down due to a bad design of Flink dependencies, namely they are exchanging JAR internally in order to by pass circle module references

(n.b. this issue has been known since several releases but they do not want to refactor the million and half lines of code).

I have modified the Flink client in order to export the *stream graph* in a serialized format (we use JSON since Flink have basic JSON output) and not the jobgraph since the first one contains less Flink specific information (as slot and checkpoint) but offer an optimize graph compare to the raw dataflow. Moreover, I also implement the reverse: modifying an actual stream graph according Planner instructions (removing nodes and adding Kafka connectors). However, part of the reverse (namely creating external connectors) must be done in an outside connector because of the Flink circle dependencies. The connector creates operators needed and send them to the client in a serialized format.

6 Validation

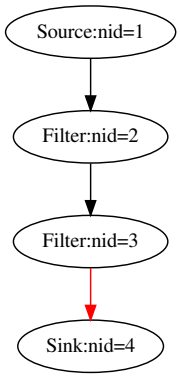

Figure 4: The source produces taxi ride items then they are filtered in order to get statistics on the rides in New York city (e.g., rides that have not started, rides taking too much time) and eventually stored in a Kafka topic.

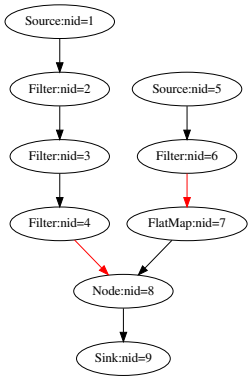

Figure 5: The source 1 produces taxi ride items and the source 5 taxi fare ones. This workflow computes the set of well-formed night rides in NY city and each ride is joined with its fare (by operator 8).

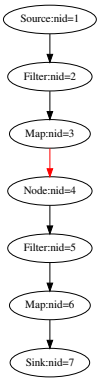

Figure 6: This benchmark calculates every 5 minutes popular areas where many taxis arrived or departed in the last 15 minutes.

6.1 Experimental setup

I emulate a hybrid Cloud and Edge platform on the Grid'5000 testbed[d]. Cloud nodes are located on the *paravance* cluster in Rennes and Edge devices on the *graphene* cluster in Nancy. Cloud nodes are composed of 2 x E5-2630v3 (8 cores/CPU) with 128 GB of memory and they are connected to the network by 2 x 10 Gbps links. Edge devices run on one core of an Intel Xeon X3440 with 16 GB of memory and connected to network by 1 Gbps links. For our experiments I have used up to 10 nodes and a total of 80 cores.

To emulate the high latency WANs connecting Edge and Clouds I use *tc* [4] to vary latency (with *netem*) and the available bandwidth (with *tbf*) until reaching the desired quality of service. Edge nodes on steroids (i.e., the least powerful nodes in the Grid'5000, yet quite powerful for an average Edge device) should not impact the validation since the network is constrained with *tc* and I ignore performance capacity during placement (we only target expressiveness constraints).

For the validation I use Apache Flink as the Cloud SPE and Apache Edgent as the Edge SPE. I have chosen this Edgent-Flink duo for simplicity of integration since both are written

[d]Grid'5000 is supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

in Java and are based on arbitrary UDF-operators. I use one cloud node for Flink (with one *jobmanager* and one *taskmanager*), one cloud node for the ingestion system (one Apache Kafka broker). I deploy five Edgent devices on distinct Edge nodes and I collocate Planner with the *jobmanager*. Interconnection with Planner is done via a dedicated Flink connector collocated with the *jobmanager* and five Edgent connectors hosted in each Edge node.

6.2 Experimental protocol

I ran two real-life application dataflows presented in Figure 4 and Figure 5 where the red arrows represent the cut found by applying Algorithm 1. They rely on the [e] of the New York City Taxi dataset [22] composed of data containing fares (1.5M entries) and rides (3M entries) description for 15K distinct taxis, with a total size of about 10GB. The rides dataset contains especially the start location, the stop location, the start time, the end time and the fares dataset contains in particular the tip, the toll and the total fare of a ride. For each dataflows, I compare two deployment scenarios where raw data are hosted in the Edge devices. For the first one, the whole computation is processed on Cloud with Flink (with data served from the Edge using Kafka). For the other one, Planner is used to deploy part of the computation to the Edge devices.

6.3 Results

In our first series of experiments, I measured the reduction in terms of transferred data with our approach. As seen in Figure 7, Planner is able to reduce the network usage over links by 51% for the workflow $w1$ (Figure 4) and by 43% for the workflow $w2$ (Figure 5). Our Cost Estimator is mainly based on the selectivity and filter operators have the lowest selectivity (sinks excepted). Therefore, the scheduler will place as much filter operators as possible on the Edge. However, even for one of the worst cases for Planner (the workflow $w3$ in Figure 6, where there is global stateful operator - here a time window - near the sources and a very light preprocessing - here a light clean of data), our approach is still able to reduce the network usage compared to vanilla Flink.

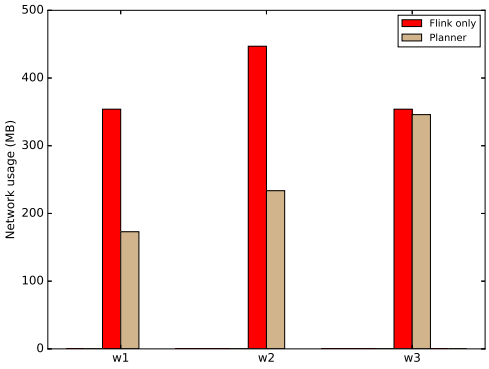

Figure 7: Network usage over links between Edge and Cloud induced by application execution where $w1$ denotes the workflow in Figure 4, $w2$ denotes the workflow in Figure 5 and $w3$ the one of Figure 6. The red bar corresponds to the whole computation in Cloud and the tan one corresponds to the usage of Planner.

In the second series of experiments, I measured the reduction of the end-to-end processing latency (Figure 8) and of the makespan (Figure 9) with our approach when the bandwidth between Edge devices and Cloud varies. As seen in both plots, Planner gains over vanilla Flink (all execution in the Cloud) is smaller than for the network usage because of the lack of inner optimizations in Edgent. For instance in Flink, operators are grouped in the same logical operator (and then in the same thread) in order to optimize computation. Moreover, I can observe that the gain brought by Planner is better for $w2$ than $w1$ for the latency and, inversely, better for $w1$ than $w2$ for the makespan. This is due the fact that there is a connect operator linking fares and rides according to the *taxi id*. Minimizing the network usage between Edge and Cloud reduces the congestion of this operator and, consequently, single rides (or fare) stalls less and eventually the latency gain is greater. Conversely, Planner performs better on $w1$ than $w2$ for the makespan because the network usage decreases more for the first one. Overall, I notice an average of 15% improvement for the makespan and the latency, proportional with the bandwidth between the Edge and Cloud.

[e] `http://training.data-artisans.com/exercises/taxiData.html`

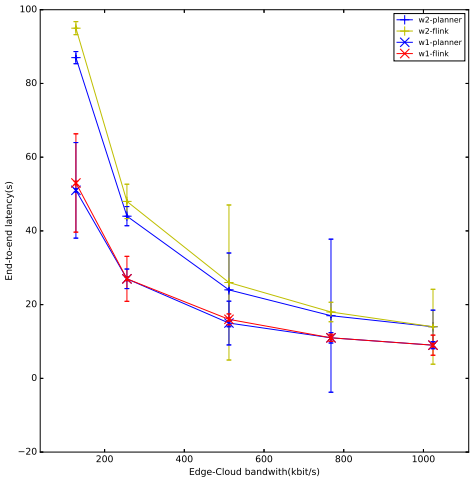

Figure 8: End-to-end processing latency: the green and red lines correspond to the whole computation in the Cloud and the blue one corresponds to the Planner approach.

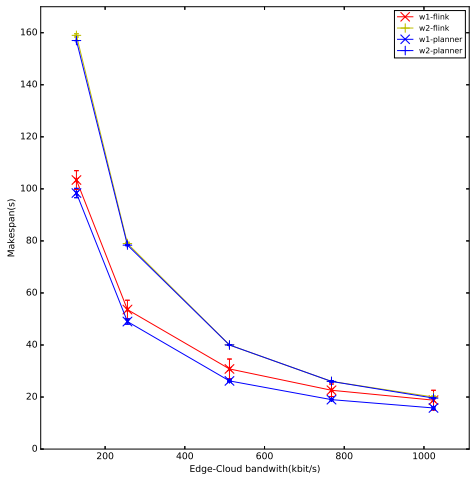

Figure 9: Makespan: the green and red line correspond to the whole computation in the Cloud and the blue one corresponds to the Planner approach.

7 Discussion

7.1 Assumptions

7.1.1 Communication bottleneck

In this work, I focus on the network as the main bottleneck of stream processing over Edge and Cloud systems, starting from the time skew observed in the literature [15] between the generation time and the processing time of events. Nevertheless, I also consider other limitations of the overall performance (e.g., power consumption, memory, computing power of an Edge device). This is the role of the constraints introduced in Section 3.1.

7.1.2 Cloud power

I consider that systems deployed on the Cloud are able to run the whole computation. For common use cases, this assumption holds. However, there are some situations where one datacenter can not run the whole process (e.g. more than 80 datacenters are involved in the processing of the data produced by MonALISA[31] monitoring ALICE[13] one of the four experiments of the LHC). In this case, one can use a geo-distributed cloud middleware for stream processing (like SpanEdge [33]) as a cloud SPE for Planner.

7.1.3 Additional metrics

In this work, Planner has been tailored to focus on the optimization of network usage and, consequently, of the makespan. However, it can also optimize other metrics (e.g. the throughput of the application) by careful selection of a convenient definition of the transmission cost $\mathscr{W}_{res}$ (defined in Section 2.1) for an item.

7.2 How to overcome limitations

7.2.1 Homogeneity of IoT devices

Each IoT device, independently of its intrinsic characteristics, will run the same sub-graph - with distinct datasets) due to the *semantic homogeneity* (see Section 4.1.2). This can be mitigated by letting the user add optional annotations to the cloud SPE workflow in order to specify which group (i.e., $g(o)$) of devices provides raw data to a source operator. Moreover, annotations can also be used to distinguish non Edge sources (e.g., a DB connector). Finally, the transparency can be preserved with some SPEs by encoding annotations in operator names or uids (if available).

7.2.2 Graph optimization

Planner does not do any execution plan optimization (e.g. [27][26]), contrary to Apache Flink for instance. A plan optimization is a rewriting of the stream graph in order to improve its shape for further processing (for instance, by pushing filter operators near the sources). Moreover, stream graph rewriting is mainly a matter of finding operators that commute using static analysis (distinct for each SPE). Therefore, the static analysis should be done before the Abstraction Layer and the rewriting should be combined with the Cost Estimator.

7.2.3 Increased model accuracy

One point of improvement is the accuracy of the cost model and particularly the measure of operators metrics (e.g., selectivity or record rate of sources). This is due to the unknown behaviour of operators. However, I can enhance the accuracy of the record rate of a source by refining the measurements at runtime with an embedded probe. The same approach can be applied in order to improve the accuracy of the selectivity of operators.

7.3 Take-aways

7.3.1 What Planner is

Planner is a streaming middleware capable of automatically and transparently deploying parts of the computation across Edge and Cloud resources. Furthermore, it is modular enough to be plugged with any combination of Cloud and Edge SPEs via external connectors.

7.3.2 On the generality of the Planner approach

The combination of Cloud and Edge SPEs on which Planner works is only limited by their expressiveness. For instance, an UDF-based Cloud SPE cannot be plugged with a non UDF Edge SPE since the framework will not be able to run the exported operators. But slight discrepancy of expressiveness (e.g., some small subset of non-supported operators) can be tolerated thanks to the constraints introduced in Section 3.1.

7.3.3 What Planner is not

Planner does not deploy streaming systems on Edge device or in the Cloud (as it expects the SPEs to be already deployed) and it is neither intended to place operators at the granularity of physical nodes (this is delegated to the SPEs, which schedule the received sub-graphs). Besides, Planner is not yet preserving consistency. However, with the right combination of Cloud SPEs, Edge SPEs and ingestion systems, some levels of consistency could be guaranteed (for instance, *exactly once* using Kafka and Flink). Also, Planner does not ensure any data persistence. Indeed, by moving computation to Edge devices parts of the data will never reach the ingestion system and therefore will not be persistently stored.

7.4 Lessons learned

In this subsection, I discuss the things I have learned on the experimental part of my work and what should be done to ease them.

7.4.1 Testbed tools are not powerful enough

Several tools (e.g. Ansible, Chef, Puppet, EasyBuild) exists to deploy software stack however performing a correct setup, running experiments, detecting termination and collecting results is commonly do in an empirical way in a per situation bases. There is a need for high level tools managing the whole lifetime of an experimentation. I am currently porting my own testbed platform developed for my experimentations to a generic one. It should help structuring the experimentation and it reuses when possible external tools for specific points (e.g. Ansible for the software stack deployment, Execo for running job in others nodes). More details on the testbed are presented in Section **??**.

7.4.2 Flee deep interaction with state of the art solution

One way to reduce the complexity of the designing and the implementation of the proof of concept could have been to avoid the directed interaction with Flink by writing plans in our own dialect then processing them by Planner and exporting a Java plan for Flink, compile it and feed Flink with it. Such an approach cloud have avoided all the issue and delays due to the gap between Flink specification and real implementation.

8 Related Work

I divide the state-of-the-art into three categories: 1) classical stream processing systems, 2) systems for hybrid processing and 3) works on general stream processing optimizations.

8.1 Stream Processing Engines

A stream processing system is used to run a stream workflow, it translates the application source code into a workflow, deploys it to physical resources and processes it. This engines are mainly used in two distinct situations: on Cloud and on Edge with specific needs and goals. I discuss both situation in the remaining of this subsection.

8.1.1 Edge Analytics Framework

Edge analytics frameworks, like Apache Edgent [2], Apache MiNiFi [3], are used to execute a data stream application on an Edge device. They are optimized to do local light-weight stream processing. Such frameworks commonly export results to an ingestion system, like Apache Kafka[30] or RabbitMQ [36].

8.1.2 Cloud SPEs

A common approach to execute a stream graph is to use SPEs that will take care of the deployment of the computation to many physical nodes (of some cluster), their management and their fault-tolerance. Several SPEs exists (e.g, Apache Flink [20], Apache Spark [37], Amazon Kinesis [5], Google Dataflow [6]). They are mainly designed and optimized in order to run in the Cloud and particularly in a single data-center[33].

8.2 Hybrid approaches

A new paradigm has emerged which combines Cloud-based and Edge analytics in order to do real-time processing at the Edge (for some timely but inaccurate results) and offline processing in Cloud (for late but accurate results) inside the same application.

Several companies are providing solutions (e.g, Azure Stream [7], IBM Watson IoT [8], Cisco Kinetic [9]) that should ease the deployment of the stream processing on Edge devices and to interconnect with their own Cloud-oriented SPEs. However, they are provided "as a-service" and the user is dependant of the companies platforms.

SpanEdge [33] focuses on unifying stream processing over geo-distributed data-centers (between a main datacenter and several near-the-edge ones) in order to take advantage of user and data locality to reduce the network cost and the latency. They are placing computations on distinct data-centers whereas I am targeting locality-aware placement on edge devices.

Echo [34] generalizes data stream processing on top of an Edge environment. It maps operators onto Cloud nodes and Edge devices in order to take advantage of the unused computing power available in the Edge. Unlike Echo, I am using a locality-aware placement approach in order to minimize communication cost. Furthermore, the placement of Echo works at the granularity of nodes (e.g., it bypasses the placement strategies of the SPEs and their potential optimisations) whereas Planner places stream sub-graphs to systems (leveraging the SPEs strategies to place operators onto nodes and benefit of their inner optimizations).

8.3 Optimizing stream processing

There are two orthogonal approaches (commonly applied in sequence[16]) that try to optimize the processing of a stream graph: graph rewriting and graph scheduling.

8.3.1 Graph rewriting

It rewrites the input plan in an equivalent one that should improve the performance of the computation (e.g. [27], [26]). The main approach are operator permutation (e.g. move light weight operator first), operator grouping (i.e. several operators are merge in a logical one in order to increase data locality) and operator replication (i.e. duplicates operators in order to improve the parallelism).

8.3.2 Placement optimization

This focuses on the mapping of the operators to physical nodes in order to optimize some metrics. For instance, [32] and [18] tries to reduce the network cost during the execution of an application however they do not take cares of specific characteristics of an Edge-Cloud deployment. The authors of [21] propose a new approach by grouping the operator replication and the operator placement in order to improve arbitrary metrics. Placement optimization is done at the granularity of a node and therefore it is not performed (and neither intended to be) by Planner, which in turn delegates this fine grained placement to other systems.

9 A generic testbed

In this section, I give a quick overview of the generic testbed I am developing for easily running HPC and Big Data experiments at scale in order to address the limitation of the current tools I have tried during the validation phase. This tool handle the whole life of an experiment from the hardware reservation (e.g. Cloud reservation or Grid'5000 one) to the collection of the logs and results. This tool is written in Python3 ($\sim$ 4000 lines) for ease of use and for interaction with Ansible [10] (a software deployment tool) and Execo [28] (use for executing process on remote host). I use a centralized architecture for simplicity where a central node (called the orchestrator) supervises the whole experiment.

9.1 Architecture overview

An experiment is composed of multiple optional steps (see Figure. 10) combined according the needs. Each step is a Python class and the whole code used by a step is contained in a specific Python module. Thanks to this modular architecture the testbed can be tailored to a specific case by configuring the needed steps thanks to external configuration files (written in YAML for the user part). Moreover, extending the tool can be easily done by subclassing Python class. The steps are grouped in two parts: the deployment and the running one.

9.2 Deployment part

During deployment nodes are configured (hardware and software) in order to meet the requirements for the tests. First, the *hardware reservation* step takes place, nodes are reserved on the cloud or on an experimental platform (for now only Grid'5000 is supported). Then, there is the *labelling* step when nodes are labelled. This step aims to do the translation between the IP address of the node and its label which denotes the role played by the node during the experiment (e.g. some nodes are tagged as *flink* for Flink nodes); a node can have several labels. The remaining steps use label instead of address.

The next step is the *software deployment* one where software stake is deployed on nodes according to the previously defined label (for instance Java and Flink is deployed on nodes tagged with *flink*). The deployment is done thanks to Ansible roles. Then a full factorial experimental test [29] is performed according some parameters, i.e, an instance of the experiment is run for each item of the cartesian product of all parameter values.

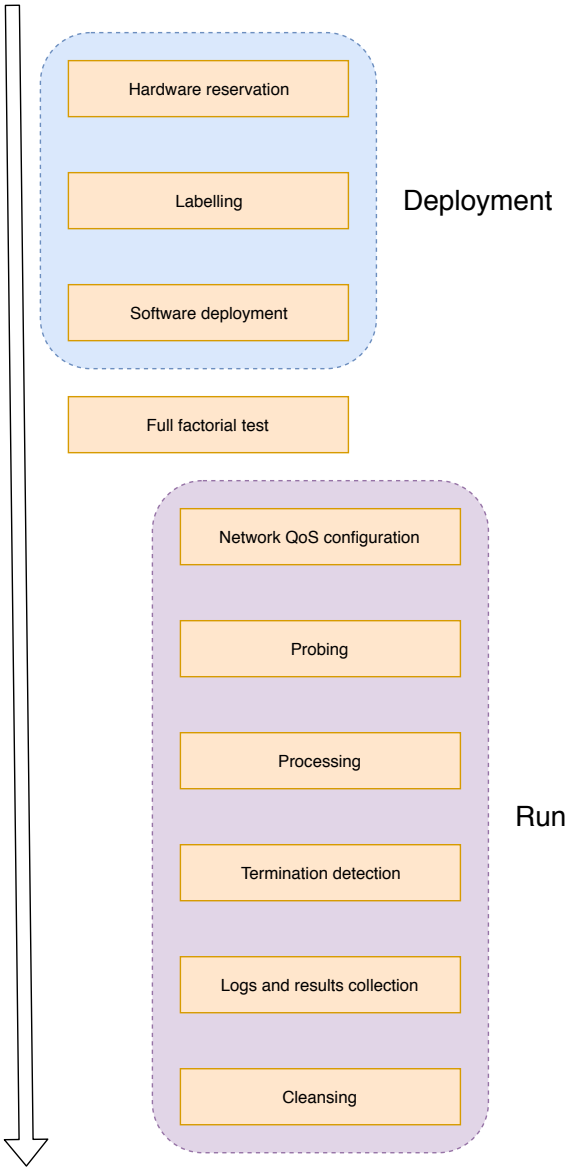

Figure 10: Steps of an experiment

9.3 Running part

The running part describes all the computation done by an instance of the experiment. First the network between nodes is configured thanks to *tc* to reach the desired quality of service (e.g. bandwidth, latency) in a per link bases. Moreover, the performances (e.g. average latency for relevant links) are evaluated during the *probing* step. Then, the datasets needed for the experiment are generated and deployed. Afterward, the core of the experiment is run and the termination can be detected by timeout, by reading log or by reading from a distributed queue (e.g. a Kafka topic). Then, logs and results are collected on a custom localisation. Eventually, all the hosts are cleaned in order to remove all the modifications done during a run.

9.4 Examples

Now, I present a fraction of the configuration of the experiment I use for the validation (Section 6). I focus on the *running part*, the parameters are defined in the *params.yml* file (see. Fig. 11). And an instance of the experiment is run by the *orchestrator* (see. Fig. 12). I use context managers (i.e. *with* statement) whenever possible in order to ensure a correct cleansing.

Adding a new step (e.g. limiting memory consumption) can easily done by creating a module with a context manager, adding one line in *run_experiment* and adding an entry with some values into *params.yml* if needed.

```
---
experiment: # python class used for a run of an experiment
    - NYCTaxi
use_planner: # use planner or not
    - True
    - False
sample: # which dataflow should be executed
    - jar: 'sample.jar' # where is the JAR
      class: 'basics.RideCleansing' # the class to run
    - jar: 'sample.jar'
      class: 'state.JoinRidesWithFares'
network: # network QoS
    - "EdgeCloudRateConfig(rate=128)" # up to 128kbit/s
    - "EdgeCloudRateConfig(rate=256)"
dataset: # which dataset to use
    - NYCTaxiDataset
```

Figure 11: Values of parameters used for the full factorial test

```
class Orchestrator(Engine):
    '''Orchestrate a set of experiments'''
    def run_experiment(self, **kwargs):
        '''Perform a run of experiment based on the values of parameters kwargs'''
        Exp = kwargs['experiment'] # the instance of the experiment to run
        del kwargs['experiment']

        with kwargs['network'](self.testbed): # specific network config
            with kwargs['dataset'](self.testbed) as dataset: # dataset deployment
                kwargs['dataset'] = dataset

                # here the context manager is used to correctly collect
                # logs and results and remove them from nodes.
                with Exp(self.testbed, **kwargs) as exp:
                    exp.start()
                    exp.join()
```

Figure 12: Orchestrator

10 Conclusion

I will not repeat the conclusion of this report but refer the reader to the one available in page 2 for a summary of contributions and a discussion of further work.

References

[1] http://storm.apache.org/. [Online; accessed 9-August-2018].

[2] http://edgent.apache.org/. [Online; accessed 13-July-2018].

[3] https://nifi.apache.org/minifi/. [Online; accessed 13-July-2018].

[4] https://github.com/shemminger/iproute2. [Online; accessed 9-August-2018].

[5] https://aws.amazon.com/kinesis/. [Online; accessed 13-July-2018].

[6] https://cloud.google.com/dataflow/. [Online; accessed 13-July-2018].

[7] https://azure.microsoft.com/en-us/services/stream-analytics/. [Online; accessed 13-July-2018].

[8] https://www.ibm.com/internet-of-things. [Online; accessed 13-July-2018].

[9] https://www.cisco.com/c/fr_fr/solutions/internet-of-things/iot-kinetic.html. [Online; accessed 13-July-2018].

[10] https://www.ansible.com/. [Online; accessed 19-August-2018].

[11] https://pulsar.incubator.apache.org/. [Online; accessed 9-August-2018].

[12] http://samza.apache.org/. [Online; accessed 9-August-2018].

[13] Kenneth Aamodt, A Abrahantes Quintana, R Achenbach, S Acounis, D Adamová, C Adler, M Aggarwal, F Agnese, G Aglieri Rinella, Z Ahammed, et al. The alice experiment at the cern lhc. *Journal of Instrumentation*, 3(08):S08002, 2008.

[14] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *the VLDB Journal*, 12(2):120–139, 2003.

[15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[16] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal—The International Journal on Very Large Data Bases*, 23(6):939–964, 2014.

[17] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130. ACM, 2010.

[18] Benjamin Billet and Valérie Issarny. From task graphs to concrete actions: a new task mapping algorithm for the future internet of things. In *MASS-11th IEEE International Conference on Mobile Ad hoc and Sensor Systems*, 2014.

[19] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[21] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator replication and placement for distributed stream processing systems. *ACM SIGMETRICS Performance Evaluation Review*, 44(4):11–22, 2017.

[22] B Donovan and DB Work. New york city taxi trip data (2010–2013), 2014.

[23] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, 2015.

[24] Rajrup Ghosh and Yogesh Simmhan. Distributed scheduling of event analytics across edge and cloud. *arXiv preprint arXiv:1608.01537*, 2016.

[25] Nithyashri Govindarajan, Yogesh Simmhan, Nitin Jamadagni, and Prasant Misra. Event processing across edge and the cloud for internet of things applications. In *Proceedings of the 20th International Conference on Management of Data*, pages 101–104. Computer Society of India, 2014.

[26] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[27] Fabian Hueske, Mathias Peters, Matthias J Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the black boxes in data flow optimization. *Proceedings of the VLDB Endowment*, 5(11):1256–1267, 2012.

[28] Matthieu Imbert, Laurent Pouilloux, Jonathan Rouzaud-Cornabas, Adrien Lèbre, and Takahiro Hirofuchi. Using the execo toolkit to perform automatic and reproducible cloud experiments. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 2, pages 158–163. IEEE, 2013.

[29] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1990.

[30] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[31] Iosif Legrand, C Cirstoiu, C Grigoras, R Voicu, M Toarta, C Dobre, and H Newman. Monalisa: An agent based, dynamic service system to monitor, control and optimize grid based applications. 2005.

[32] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 49–49. IEEE, 2006.

[33] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *Edge Computing (SEC), IEEE/ACM Symposium on*, pages 168–178. IEEE, 2016.

[34] Sarthak Sharma, Prateeksha Varshney, and Yogesh Simmhan. Echo: An adaptive orchestration platform for hybrid dataflows across cloud and edge. In *Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, Spain, November 13–16, 2017, Proceedings*, volume 10601, page 395. Springer, 2017.

[35] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM (JACM)*, 44(4):585–591, 1997.

[36] Alvaro Videla and Jason JW Williams. *RabbitMQ in action: distributed messaging for everyone*. Manning, 2012.

[37] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

Appendices

List of Figures

List of Tables

Symbol	Description	Units
G_{sp}	Stream graph representing the application	
V_{sp}	Set of vertices (operators) of G_{sp}	
V_{sp}^{d}	Set of vertices run by the device $d \in Devices$	
E_{sp}	Set of edges (streams) of G_{sp}	
$\mathscr{W}_{sp}(i,j)$	Communication usage induced by the stream $(i,j) \in E_{sp}$	$item.s^{-1}$
f_o	UDF executed by operator $o \in V_{sp}$	
τ_o	Type of operator $o \in V_{sp}$	
s_o	Selectivity of $o \in V_{sp}$	N/A
G_{sys}	Graph representing computing and network resources	
$Devices$	Subset of vertices (computing Edge devices) of G_{sys}	
E_{res}	Set of edges (links between Edge and Cloud nodes) of G_{sys}	
$\mathscr{W}_{res}$	Transmission cost of an item through a link $l \in E_{res}$	$\beta.s.item^{-1}$
d_i	An Edge device $d_i \in Devices$	
ς	Represents all the cloud nodes	
$g(o)$	Set of nodes ($g(o) \subset Devices$) that hold part of the raw data used by $o \in Sources$	
$g^{-1}(u)$	Set of sources ($g^{-1}(u) \subset Sources$) that used part of the raw data of $u \in Devices \cup \{\varsigma\}$	
a_τ	Aggregation function for operator of type τ	$item.s^{-1}$
$\mathscr{C}_l^s$	Communication cost induced by the stream $(i,j)E_{sp}$ on link $l \in E_{res}$	β
C_{d_1}	Placement constraint for Edge device $d_i \in Devices$	
$\mathscr{P}$	Operator placement	
$\mathscr{T}$	Trace of items	
$\mathscr{R}$	Local-replication indicator function	

Table 2: Main notations. We list the measurement units in the Units column, β denotes the measurement unit of the communication cost and N/A denotes a dimensionless variable.

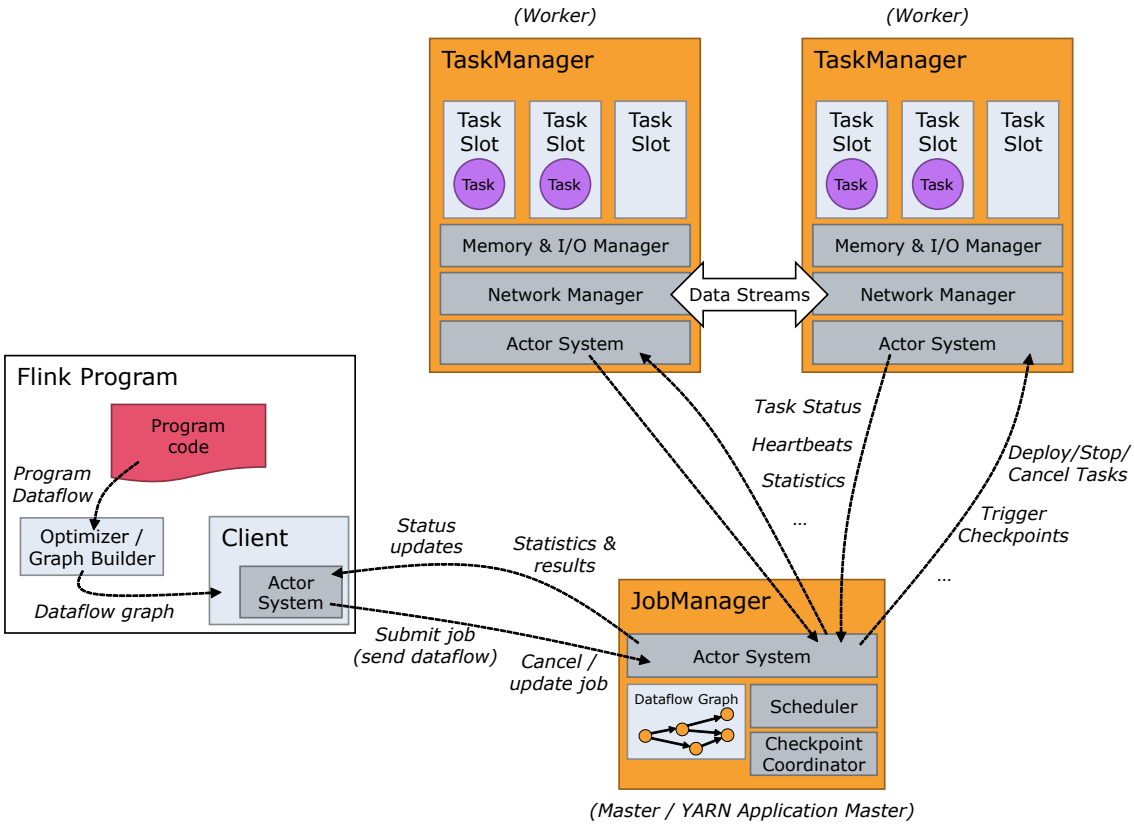

Figure 13: Flink architecture (image from `https://ci.apache.org/projects/flink/flink-docs-release-1.5/concepts/runtime.html`)

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();

    // get default input data
    DataStream<String> text;
    text = env.fromElements(WordCountData.WORDS);

    DataStream<Tuple2<String, Integer>> counts =
        // split up the lines in pairs (2-tuples) containing: (word,1)
        text.flatMap( new FlatMapFunction<String, Tuple2<String, Integer>> {
            @Override
            public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
                // normalize and split the line
                String[] tokens = value.toLowerCase().split("\\W+");

                // emit the pairs
                for (String token : tokens)
                    if (token.length() > 0)
                        out.collect(new Tuple2<>(token, 1));
            }
        })
        // group by the tuple field "0" and sum up tuple field "1"
        .keyBy(0).sum(1);

    // emit result
    counts.print();

    // execute program
    env.execute("Streaming WordCount");
}
```

Figure 14: Flink application

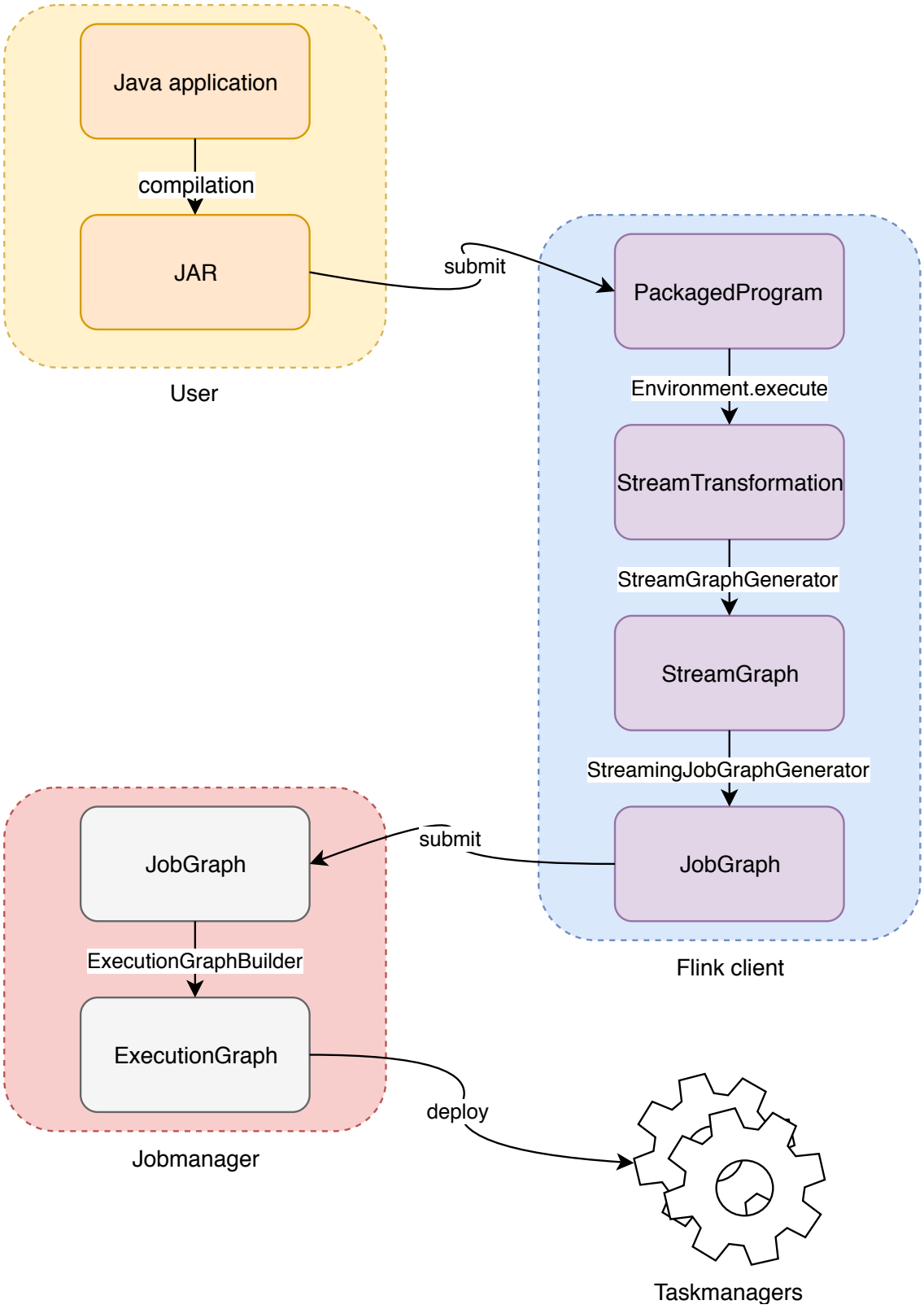

Figure 15: Application execution in Flink