EdgeScaler: Effective Elastic Scaling for Graph Stream Processing Systems

Daniel Presser
daniel.presser@posgrad.ufsc.br
Universidade Federal de Santa Catarina
Florianópolis, Brazil

Frank Siqueira
frank.siqueira@ufsc.br
Universidade Federal de Santa Catarina
Florianópolis, Brazil

Luís Rodrigues
ler@tecnico.ulisboa.pt
INESC-ID, Instituto Superior Técnico, ULisboa
Lisboa, Portugal

Paolo Romano
romano@inesc-id.pt
INESC-ID, Instituto Superior Técnico, ULisboa
Lisboa, Portugal

ABSTRACT

Existing solutions for elastic scaling perform poorly with *graph* stream processing for two key reasons. First, when the system is scaled, the graph must be dynamically re-partitioned among workers. This requires a partitioning algorithm that is fast and offers good locality, a task that is far from being trivial. Second, existing modelling techniques for distributed graph processing systems only consider hash partitioning, and do not leverage the semantic knowledge used by more efficient partitioners. In this paper we propose EdgeScaler, an elastic scaler for graph stream processing systems that tackles these challenges by employing, in a synergistic way, two innovative techniques: *MicroMacroSplitter* and *AccuLocal*. *MicroMacroSplitter* is a new edge-based graph partitioning strategy that is as fast as simple hash partinioners, while achieving quality comparable to the best state-of-the-art solutions. *AccuLocal* is a novel performance model that takes the partitioner features into account while avoiding expensive off-line training phases. An extensive experimental evaluation offers insights on the effectiveness of the proposed mechanisms and shows that EdgeScaler is able to significantly outperform existing solutions designed for generic stream processing systems.

CCS CONCEPTS

• **Information systems** → **Stream management**; *Parallel and distributed DBMSs*; • **Mathematics of computing** → *Graph algorithms.*

KEYWORDS

Graph Processing, Stream Processing, Elastic Scaling

ACM Reference Format:
Daniel Presser, Frank Siqueira, Luís Rodrigues, and Paolo Romano. 2020. EdgeScaler: Effective Elastic Scaling for Graph Stream Processing Systems. In *The 14th ACM International Conference on Distributed and Event-based Systems (DEBS '20), July 13–17, 2020, Virtual Event, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3401025.3401734

1 INTRODUCTION

Stream processing is a well established paradigm that allows users to make queries over continuous streams of data[5]. In this paper we focus on *graph stream processing systems*, where queries are made to a graph data structure that is continuously modified by a stream of updates that create, delete or change vertices and/or edges over time. Graph stream processing has important applications such as fraud detection [28], social trend analysis [15], advertising [8], among others.

Graph stream processing systems typically work by processing batches of updates, that are accumulated during a short *window* of time. Each batch is applied to the current graph, generating an updated graph that is processed while a new batch of updates is collected. This procedure is repeated while the phenomena that are being monitored persist. Running a graph stream processing system requires the continuous allocation of a significant amount of physical resources. This calls for systems that are able to optimize, and dynamically adjust, the right amount of resources allocated to the job. On the one hand, if the system is under-provisioned, it may not be able to process a given batch of updates before the next batch is available, decreasing or nullifying the value of the results. On the other hand, over-provisioning the system may lead to unnecessary economic costs.

Despite the large number of studies on elastic scaling in (generic) stream processing systems[14, 16], the existing approaches do not offer acceptable results for the relevant case of *graph* stream processing systems. This is illustrated in the blue line of Figure 1, labeled "Baseline", that shows the performance of a state-of-the-art elastic scaling technique [16] when applied to an evolving Twitter graph (see Section 4 for details). This Baseline technique [16] was chosen because it is an elastic scaling approach for data stream processing that can handle stateful operators and, therefore, can be easily adapted to graph stream processing.

The system considered in Figure 1 collects a batch of updates every 30 seconds, with a 33 seconds hard deadline, and aims at incorporating them in the graph before the arrival of a new batch. As can be noticed, the Baseline elastic scaler [16] detects that the system is under-provisioned based on the observed batch processing

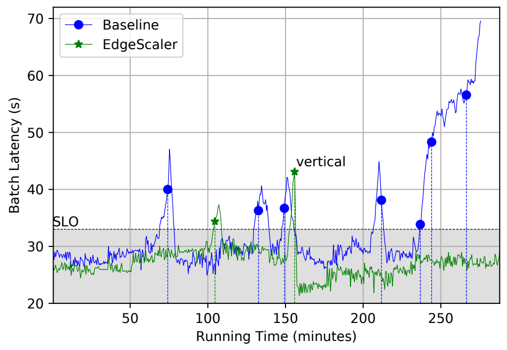


Figure 1: Comparing EdgeScaler vs a state-of-the-art technique[16], replaying 5 hours of Twitter traces[35] and running PageRank on Amazon EC2 with 30s windows and a 33s SLO.

latency (which reflects congestion) and adds new servers to the configuration (points marked with a blue dot on the chart). As the graph grows during the execution (around 50%, from 1M to 1.5M vertices), the elastic scaler performs this operation multiple times but, after the 5th reconfiguration, adding new servers is no longer effective. In fact, at that point, the overhead induced by communication among servers outweighs the benefits from adding additional computing power. Thus, the latency keeps increasing.

A closer analysis of this plot highlights the limitations of existing elastic solutions for generic stream processing, such as [16], when applied to graph stream processing. First, like most other previous techniques for dynamic reconfiguration of graph processing systems, [16] relies on consistent hashing to partition the graph. Consistent hashing is fast to execute (and, therefore, enables fast reconfigurations), but it produces low quality partitions. Thus, the system becomes increasingly inefficient as the graph size grows and new servers are added, causing reconfigurations to become more and more frequent. Second, existing elastic scalers for stream processing systems only address operator parallelism, through horizontal scaling, and lack an appropriate performance model that would enable selecting between horizontal *and* vertical scaling.

Based on these insights, we can identify two key challenges that must be addressed when designing an elastic scaling system for graph stream processing platforms.

First, contrary to most stream processing systems, where workers maintain little or no state, in a graph processing system, every time elastic scaling occurs, the graph needs to be re-partitioned and a large amount of data may need to be shuffled among workers. Also, if the resulting partition is of poor quality, adding new nodes to the system may actually have detrimental effects on performance, since the efficiency of graph processing is highly dependent on the partition quality[1]. Thus, *new re-partitioning techniques that are fast and produce results of good quality are needed.*

Second, elastic scaling should support both horizontal and vertical scaling. This requires the availability of a suitable performance model that can predict the effect of each potential adaptation. As already mentioned, though, the performance of a given configuration is highly dependent on the current graph size and on how the graph has been partitioned. Unfortunately, existing performance models for distributed graph processing platforms only consider simplistic/inefficient hash partitioning strategies [11, 23, 27]. Thus, *graph stream processing systems require performance models that can accurately predict the dynamics of less trivial/more efficient partitioning schemes.*

In this work we present EdgeScaler, an elastic scaling middleware for window-based graph stream processing systems that addresses the challenges above. Figure 1 also plots the performance of EdgeScaler when applied to the same run (green line). EdgeScaler performs less reconfigurations and more effective ones (points marked with a green star on the chart). In fact, while with the Baseline [16] the 33s hard deadline is violated over 25% of the time and eventually latency grows unbounded, EdgeScaler is able to run the system within the target service level objective (SLO), only violating it in less than 3% of the time. Also, in this scenario, the total operational cost of the system with EdgeScaler is 35% smaller than with the Baseline [16] (cost is not depicted in the figure, see Section 4 for additional details). EdgeScaler achieves these improvements by combining, in a synergistic way, two innovative techniques, which we named *MicroMacroSplitter* and *AccuLocal*.

MicroMacroSplitter is a novel *edge* graph partitioning algorithm that maps the original graph to a *micropartition graph*, where each micropartition clusters a subset of the edges of the original graph. Due to its smaller scale, the micropartition graph can be quickly re-partitioned (when an adaptation is required) using METIS[20], a high-quality partitioning algorithm. As a result, *MicroMacroSplitter* combines the speed of hashing, while achieving partitions' quality close to that of state-of-the-art partitioning algorithms. Further, the quality of the partitions produced by the *MicroMacroSplitter* (e.g., number of replicated vertices) for a target configuration can also be quickly, yet accurately, calculated, supporting the construction of accurate performance models.

AccuLocal is a new performance model for graph processing systems that builds on the key observation that, in realistic settings, even in the presence of rapid workload changes, the adaptations that are actually required tend to be localized, i.e., lead to configurations that are relatively close to the current one. We exploit this observation by purposely bounding the scope over which the performance model will be queried, so to include only the configurations in the vicinity of the current one. This led us to design a custom linear regression model that can be learnt in an online and lightweight fashion, by monitoring only a few performance metrics in the current configuration and exploiting the knowledge on the partitions produced by *MicroMacroSplitter* in the target configuration. As we will show, not only *AccuLocal* achieves high accuracy (less than 10% error) in the proximity of the current configuration, but can also guide the adaptation process in realistic scenarios effectively, i.e., identify the correct target configuration, whenever an adaptation is required.

We have built a prototype of EdgeScaler using two popular tools for stream processing and graph processing, namely Apache Spark Streaming [36] and GraphX [34]. This prototype was evaluated with real-world graphs, including static graphs and a dynamic graph extracted from Twitter. Our results show that EdgeScaler can reconfigure the system up to two orders of magnitude faster than existing sophisticated partitioning algorithms, and that it is able to reach stable configurations with a single adaptation in most cases.

The remainder of this paper is structured as follows. Section 2 provides background information on graph processing systems and reviews the related literature. Section 3 provides a detailed description of EdgeScaler, whose evaluation can be found in Section 4. Finally, Section 5 concludes this work and discusses future research directions.

2 BACKGROUND AND RELATED WORK

Although there is a relatively small number of works targeting elastic scaling for graph stream processing, the literature on distributed (static) graph processing and on stream processing (not necessarily for graphs) is quite extensive. Next, we highlight the key aspects of previous work that are relevant for the development of EdgeScaler.

2.1 Distributed Graph Processing

An influential system in the area of distributed graph processing is Google's Pregel [25], that spawned several open-source implementations, such as Apache Giraph [4], GPS [30], and other variants [17, 34]. EdgeScaler is built on top of GraphX [34], a graph processing system based on Spark [36], which also supports Pregel's graph processing model. Pregel uses a graph representation that consists of a list of vertices, and each vertex is composed by a user-defined value, a state (active or inactive), an adjacency list, and a message queue. The graph processing task is expressed as a function that takes as input a vertex from the graph and can change its value and state. The function is applied using a Bulk Synchronous Parallel (BSP) programming model [32], that works as a sequence of supersteps. In each superstep, the user-defined function is executed for every active vertex of the graph. When the graph is partitioned among multiple nodes, local updates must be propagated to other nodes at the end of each superstep.

2.2 Graph Partitioning

Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, partitioning consists in splitting G into k partitions P_i such that $P_1 \cup P_2 \cup \ldots \cup P_k = G$. Graph partitioning algorithms aim at fulfilling two key goals: ensuring good load balancing and minimizing the updates sent through the network at the end of each superstep. The two main approaches to perform graph partitioning – *vertex* and *edge* partitioning – are illustrated in Figure 2. The first approach operates on the vertex set V, attributing each vertex to a partition P_i. In this case, edges can span partitions (an edge between vertices (v, u) such that $v \in P_i$ and $u \in P_j$), also known as *edge-cut*. The second approach operates on the edge set E, attributing each edge to a partition P_i. In this case, vertices can span partitions (assume edge $(u, v) \in P_i$ and $(v, y) \in P_j$, such that v is in both partitions); vertices that span multiple partitions are said to have *replicas* in these partitions. Although both approaches are used, edge partitioning can be more efficient for scale-free graphs[17], which are common in many real-world applications, such as social networks. EdgeScaler is based on GraphX and uses edge partitioning.

Finding the optimal partitioning is known to be an NP-Hard problem[3], thus using heuristics that provide good results in short

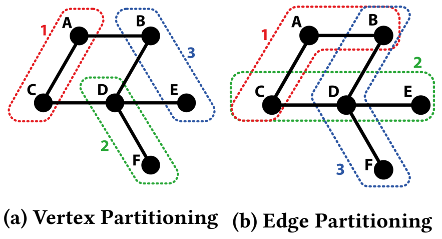


Figure 2: Vertex vs edge partitioning

time is extremely relevant. The simplest algorithm is *hash* partitioning, in which edges are randomly distributed among partitions. This algorithm is fast to execute, does not require any state to be maintained by the partitioner, and produces well balanced partitions. However, hash partitioning makes no attempt to reduce the number of replicas; thus it can introduce large network overheads at the end of each superstep. More sophisticated partitioning algorithms have been proposed, such as PowerGraph's Greedy[17], and HDRF[26]. HDRF has been shown to yield better results (i.e, less vertex replicas) than hash, but is slower. Further, although they are streaming partitioners (i.e., they can determine the target partition of each edge in a single pass), they introduce prohibitively high costs whenever the full graph has to be repartitioned due a change in the configuration of the computing platform. Incremental graph partitioners have also been studied [10], but lack the ability to handle recofigurations quickly. An in-depth analysis of partitioning algorithms for graph stream processing applications can be found in [1].

A few works have addressed the problem of finding partitioning strategies that are both fast and able to produce results of good quality. A notable example is the partitioning strategy proposed for Hourglass [19], that offline uses the slow METIS algorithm to partition the graph in a relatively small number of *micropartitions* (much smaller than the number of vertexes in the graph, but much larger than the number of workers) and, at runtime, places multiple micropartitions in each worker. Hourglass, though, is designed for vertex partitioning and for static graphs. EdgeScaler, conversely, is designed to use edge partitioning and support evolving graphs.

2.3 Graph Stream Processing (GSP)

A graph stream processing system is designed to run graph analytics over a flow of update messages that can change the graph topology and the properties of edges and vertices. Kineograph[7], which was one of the first GSP systems, produces periodically consistent snapshots of the graph, over which an user-defined algorithm can be executed. This programming model based on snapshots was also used by subsequent systems, such as Tornado[31] and GraphTau[18], as it is well suited for most graph processing tasks. GSP systems distinguish themselves from other stream processing systems because workers are required to maintain a large evolving state. It has been observed that, depending on the rate of events received, the graph can grow quickly, degrading performance[7]. Thus, the ability to perform elastic scaling on GSP systems is of the utmost importance.

2.4 Elastic Scaling in GSP

There are many proposals in the literature on elastic scaling for general-purpose stream processing systems[12–14, 16, 24]. However, these proposals are not suitable for GSP systems because they either do not address partitioning the state among workers [14] or assume stateful workers that, unlike graph processing, use simple data structures, such as key-value pairs [12]. Moreover, the graph partitioning strategy can have a significant impact on the overall performance of a GSP system. However, its effect is seldom considered by elastic scale managers, which typically assume a simplistic hash partitioning algorithm [12–14, 16, 24]. The performance dynamics of graph processing systems is also challenging, and often vertical scaling is required to satisfy timing constraints. Unfortunately, most auto-scaling proposals for stream processing systems only consider horizontal scaling.

2.5 Performance Modelling for GSP

Performance modelling of BigData systems has received significant attention in the literature, with proposals for generic systems such as CherryPick [2] or specific to graph processing models[11, 23, 27]. These models target applications for which an adequate configuration can be selected prior to their deployment, based on information gathered during an offline training phase. These approaches cannot be employed in GSP systems, which have to be reconfigured at runtime to adapt to evolving workloads. EdgeScaler, conversely, produces predictions based on information collected at runtime, thanks to its domain-specific nature. Moreover, previous graph performance prediction models [11, 23, 27] support only the hash partitioning algorithm. This work, on the other hand, adopts a sophisticated graph partitioning technique based on micropartitions. As a result, domain knowledge regarding the operation of the partitioner can be used to improve the accuracy of the model.

3 EDGESCALER

This section describes EdgeScaler, an elastic scaling middleware for graph stream processing systems. EdgeScaler is composed of two main components, namely a stream graph processing engine and an elastic scaling manager. In turn, these components use a novel graph partitioning system and a performance model that have been designed specifically to optimize the operation of EdgeScaler.

The stream graph processing engine is responsible for receiving update messages, aggregating them in batches, updating the graph, and computing new results. It is a distributed processing engine, based on Spark/GraphX, that can employ a (user-defined) number of workers to process the updates. EdgeScaler is based on the widely adopted Pregel programming model (other proposals, e.g. [7, 18], are still on experimental stage).The computation is distributed among a number of workers, which correspond to different (virtual) machines. Each worker handles one or more partitions of the graph (one per available CPU core on the worker).

To partition the graph among workers, Pregel originally adopts a vertex-based partitioning algorithm. EdgeScaler uses a more efficient[17] edge-based partitioning strategy, in which edges are assigned to workers and vertices can span partitions (and are replicated when this happens). In this scenario, the messages exchanged between partitions during a computation are bounded to the number of vertices replicated by the partitioning mechanism.

The EdgeScaler edge-based partitioner, called *MicroMacroSplitter*, is fast (i.e., can be executed efficiently) and generates high-quality partitions, with small vertex replication factors and good load balance among workers. Furthermore, the quality of the partitions produced by *MicroMacroSplitter* is predictable, i.e., it is possible to simulate its execution to determine the distribution of edges in a different configuration. This makes it possible to accurately predict its performance via modelling approaches, which are at the core of EdgeScaler's elastic scaling manager.

The elastic scale manager is responsible for reconfiguring the pool of workers. More precisely, EdgeScaler assumes a cloud environment where (virtual) machines with different hardware characteristics (such as the instance types of Amazon EC2) are available and can be launched by the system as needed. It aims at optimizing both the *type* and *number* of (virtual) machines in use. By considering different machine types, EdgeScaler can scale both horizontally (i.e., add more hosts of the same type) and vertically (i.e., replace the servers in the current worker pool with more powerful ones).

Reconfiguration may be needed more than once as the system evolves, because the size of the graph and the number of updates in each batch change over time. Typically, the larger the graph is, the larger the amount of resources that are needed to process a batch of updates. The elastic scale manager monitors the performance of the system (namely, how much time it takes to process a batch of updates using the current configuration) and evaluates the possibility of reconfiguring the system, whenever the latency for processing a batch approaches the SLO.

The choice of a new configuration is performed with the help of a *performance predictor* that we have developed specifically for this purpose. The predictor is based on a new performance model, named *AccuLocal*, that takes into account the specific features of EdgeScaler to estimate its performance in a set of relevant configurations (i.e., configurations in the vicinity of the current one), based on the current graph state. *AccuLocal* estimates the time needed by a configuration to process a batch of updates based on the number of the edges and vertices assigned to each worker and on the number of vertices replicated among different workers. *AccuLocal* uses a number of parameters and coefficients that can be learnt and calibrated quickly at run-time. This calibration is based on the performance of the current configuration and provides accurate estimations for other configurations that are close in the search space and that are likely to be used next. The use of *AccuLocal* avoids the need to run an expensive offline training phase, as typically required by general modelling techniques[2, 33].

3.1 *MicroMacroSplitter*

In an elastic graph processing system, when the system is reconfigured, the graph needs to be re-partitioned, as the number and/or the capacity of servers changes. Naturally, we would like the graph partitioning algorithm to be fast and to provide high-quality partitions. To tackle this quality vs. time dilemma, we have developed a novel edge partitioning system, called *MicroMacroSplitter*.

The key idea at the basis of *MicroMacroSplitter* is to logically map the original graph into a set of, so called, *micropartitions* $M =$

$\{\mu_1, \mu_2, ..., \mu_m\}$. The number of micropartitions $m = |M|$ is selected to be orders of magnitude smaller than the size of the graph — so that, upon reconfiguration, the resulting micropartitions can be efficiently mapped to a different set of processors using a high quality partitioner, like METIS or HDRF – while still being several times larger than the maximum number of servers expected to be used in the system – so that the partitions that are produced have a quality close to that achieved by a high-quality partitioner on the full graph. In case an initial graph exists, the mapping of the original graph to micropartitions can be computed in an offline phase, before an adaptation is needed, so that its latency does not have an impact on the reconfiguration time. It can, therefore, use an algorithm that derives micropartitions of good quality, even if the algorithm required to do so is slow. *MicroMacroSplitter* uses HDRF [26], which is a state-of-the-art edge partitioner that is known to provide results of good quality.

Whenever a reconfiguration is required (as well as when EdgeScaler is deployed in its initial configuration), *MicroMacroSplitter* maps the micropartitions into k larger partitions $P = \{P_1, P_2, ..., P_k\}$, where k is the total number of CPUs available in the target configuration, such that any $P_{i \in k} \subset M$ and $\cup P_{i \in k} = M$. The micropartition mapping problem is modeled as a vertex partitioning problem. We model each micropartition μ_k as vertices of a graph G_M, that contains an edge between (μ_i, μ_j) iff both micropartitions share vertices from the original graph (that is, they have replicas in common). The number of replicas shared between these partitions is defined as the weight of such edge. This graph G_M is then submitted to a vertex partitioning algorithm, namely METIS[20], with the objective of minimizing the sum of the weights of the edges cut and produce k partitions. Such minimization performed by METIS reduces the number of replicas between the actual partitions.

At run-time, once the set of partitions P for the current configuration is defined and new batches are received, any new edge is mapped to a partition in P via the following procedure:

(1) HDRF is used to first map a new edge to one of the micropartitions $\mu_i \in M$.
(2) At the end of a batch, every new edge is merged to the corresponding partition in P, based on the mapping of micropartitions ($M \longrightarrow P$) initially defined by the METIS algorithm.

MicroMacroSplitter contributes to the performance of EdgeScaler is several ways. First, when compared to hash partitioners, it makes the system faster at run-time, since it derives partitions of higher quality (via the joint use of HDRF at run-time, and of METIS upon reconfiguration). Second, it makes re-deployment faster when compared to conventional high-quality partitioners (e.g., HDRF), as it recomputes the new partitions only over G_M, instead of the whole graph. Further, by moving entire micropartitions, it is able to transfer data more efficiently than both Hash and HDRF, which operate at the granularity of individual edges (as will be shown in Section 4.3). Finally, since G_M is much smaller than the original graph, one can precisely, yet efficiently, predict the quality of the partitions produced by the *MicroMacroSplitter* in any target configuration by simulating its execution, i.e., computing the new partitions without actually shuffling data among machines. This property is key to derive accurate performance models, as will be described next.

Table 1: Model Variables and Coefficients

Symbol	Description
P	Set of k partitions
M	Set of m micropartitions
V_i	Number of vertices on partition P_i
E_i	Number of edges on partition P_i
R_{ij}	Replicas shared between P_i and P_j
R_i	All vertices in P_i that have replicas
T_i	Time required to process partition P_i
$\alpha, \beta, \gamma, \omega$	Coefficients estimated by the model

3.2 The *AccuLocal* Model

We model the performance of a graph stream processing system based on Pregel's model (see Section 2) via a multi-variate linear model that uses as inputs: the number of edges (E_i) assigned to each partition (P_i), the number of vertices assigned to the partition (V_i), and the number of replicas that these vertices have on other workers (R_i).

In Pregel's processing model, computation can be roughly divided in three steps: processing incoming messages, updating the vertex value, and (possibly) sending new messages. However, given the edge partitioning used by EdgeScaler, we have to distinguish between messages sent by vertices, that are related to the analysis algorithm being executed on the graph, and messages needed do synchronize the vertices replicas. The first kind of message is bound to the number of edges E_i present in the partition, and the operations performed over them are all local operations, as there are no edge-cuts in this partitioning scheme. Coefficient α aims to capture the contribution of this step in the overall execution time. Similarly, the time required to perform the vertex updates will be related to the number of vertices V_i present in the partition, and is captured by coefficient β. The messages required to synchronize the vertices replicas will be bound to R_i (see Eq. 2), and is captured by γ. Finally, coefficient ω aims to capture other additional fixed processing times.

More precisely, the estimated time it takes for each worker i to process partition P_i is given by:

$$T_i = \alpha E_i + \beta V_i + \gamma R_i + \omega \tag{1}$$

where R_i represents the number of vertices in the partition P_i that are replicated in other partitions $P_{j \neq i} \in P$, i.e.:

$$R_i = \sum_{j \neq i \in P} R_{ij} \tag{2}$$

The coefficients α, β, γ, and ω in Eq. 1 can be efficiently learnt at runtime using efficient fitting schemes [21] (see below). Table 1 offers a summary of the notation.

When predicting the performance of a target configuration, we estimate the time it takes to process a batch by considering the slowest partition, i.e.,

$$T = \max_{P_i \in P} T_i \tag{3}$$

Estimating the coefficients: Due to its simplicity, the *AccuLocal* model can be instantiated quickly, i.e., measurements from a few batches are sufficient to estimate the coefficients for the current configuration. EdgeScaler collects the time required to process each batch in the current configuration into a sliding window that maintains the most recent measurements. By monitoring each worker, EdgeScaler is able to accurately measure the time required to perform each of the previously described steps of a graph computation. Thus, the coefficients α, β, and γ can be estimated individually, by minimizing the model's error (MSE) in predicting the execution time of the corresponding computation steps over the set of measurements in the sliding window. Coefficient ω is estimated as a simple average related to the remaining time after accounting for the steps previously described.

Defining the space of candidate configurations: The E_i, V_i, R_i parameters on a target configuration can be efficiently and accurately computed by simulating the execution of *MicroMacroSplitter*, i.e., calculating the new partitions in the target configuration, without reshuffling the data in the system. However, the values of the model coefficients (α, β, γ and ω), which are estimated in the current configuration, are not guaranteed to be constant as the scale of the system or the type of servers varies. Interestingly, though, as we will show in the evaluation section, the coefficients learnt for a given configuration can be used to estimate remarkably well the performance of "close" configurations (i.e., configurations that do not differ much from the current one). Thus, instead of attempting to learn the coefficients for all possible configurations, we use the coefficient values computed in the current configuration (as described above) to estimate the performance of the system in a set of candidate configurations that are within a bounded distance, $\mathcal{D}$, and for which the coefficients learnt in the current configuration are still likely to be accurate.

This approach works well in practice because it is not reasonable to expect a live graph processing system to require redeployment on configurations drastically different from the current one. In fact, even in the presence of sudden changes in the rate of messages being processed by the system, the time taken for ingesting graph updates is typically negligible (less than 5%) compared to the time required to analyze the updated graph. Thus, the cost of the graph analysis phase is largely dependant on graph size, which varies slowly, in a realistic scenario, compared to the message arrival rate.

Let us now define more precisely how *AccuLocal* defines the set of candidate configurations to be considered as possible targets for a system reconfiguration. The search space is defined by a set of different machine types (belonging to the same family) and a maximum budget B_{max}, both defined by the user at configuration time. Each machine type $i \in \mathcal{I}$ is characterized by a tuple (i, v_i, p_i, m_i) where v_i is the cost of running an instance of that machine type, p_i is the processing power of the instance (number of CPUs), and m_i is the memory capacity of that machine (measured in terms of edges it can store and process). In short, *AccuLocal* considers machine types with different arrangements of CPUs and available memory. This is a common practice in cloud providers such as Amazon EC2, where machines are grouped into families with similar underlying architectures, and offered in different "sizes", varying the number of CPUs and memory for each type. We assume that machines are selected such that they always have enough disk space to store the graph partitions assigned to them. The maximum budget B_{max} captures the maximum amount of money the user is willing to pay to run the graph processing system. EdgeScaler only considers homogeneous deployments, i.e., deployments where all servers use the same machine type, such that every machine can progress on the computation at the same rate. Thus, a configuration is defined by a tuple $c = (n, i)$ where n is the number of instances used in the configuration and i is a machine type. The cost of a configuration is defined as $Cost(c) = n \cdot v_i$. A configuration is only considered a candidate if $Cost(c) < B_{max}$.

AccuLocal considers as set of candidate configurations for an adaptation, noted $C_{candidate}$, only the configurations within distance $\mathcal{D}$ from the current configuration. More precisely, we use Manhattan distance [6] computed after having mapped the tuple $(p_i, m_i) \forall i \in \mathcal{I}$ into a discrete two dimensional space defined by the tuple (P, M), where $P \in \{1, 2, \ldots, max_{p_i}\}$ represents the number of processors, $M \in \{1, 2, \ldots, max_{m_i}\}$ encodes the amount of available memory, and $max_{p_i} = max_{m_i} = \frac{B_{max}}{v_i}$ denote, respectively, the maximum number of processors and the maximum amount of memory that can be acquired for a machine of type i given the available budget B_{max} (in terms of hourly cost). The mapping between the values in each dimension of the original configuration space and the space over which the Manhattan distance is computed is obtained via a topological sort, that reflects an increasing "power".

3.3 The Elastic Scaling Manager

EdgeScaler's Elastic Scaling Manager (ESM) is responsible for managing the resources used by the graph processing engine such that a target SLO is not violated and the cost of operating the system is kept low[9]. In EdgeScaler we consider SLOs that define an upper bound on the time required to process a batch of updates.

The ESM operates in a control loop that: i) monitors the current performance; ii) checks if a reconfiguration is needed; iii) employs the *AccuLocal* model to choose the best configuration and; v) deploys the selected configuration.

Monitoring: ESM monitors the current system performance by collecting, at each window, the execution time statistics produced by each processor in each step of the computation.

Detecting the need for adaptations: Different policies can be used to determine whether to trigger a reconfiguration and these policies are somehow orthogonal to the main contributions of this paper. Our current prototype uses a classic threshold-based reactive policy, which is described next.

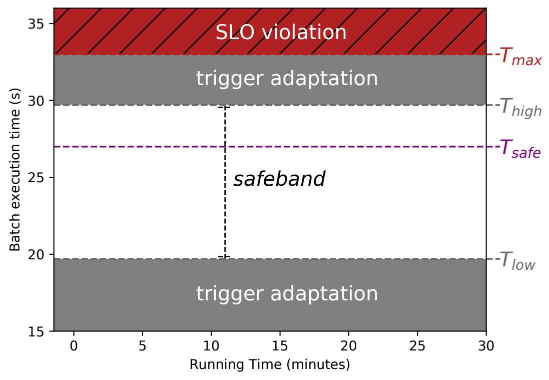


Figure 3: Illustration of the adaptation policy

We consider a SLO that represents the maximum allowed time for processing a batch window, named T_{max}. We also define two parameters, namely *slackfactor* and *safeband*, that are used to calculate a safety operation band $[T_{low}, T_{high}]$, in which the system is expected to operate. We set $T_{high} = T_{max} \cdot (1 - slackfactor)$ where $0 < slackfactor < 1$. As the name suggests, *slackfactor* represents a leeway to accommodate variations on the performance of the graph processing algorithm and any potential lack of accuracy in the performance predictions produced by *AccuLocal*, whose goal is to decrease the probability of violating the SLO. The *safeband* parameter, on the other hand, is used to calculate T_{low}, as $T_{low} = T_{high} - safeband$, where $T_{high} > safeband > 0$. The safety operation band also accommodates variations on the processing time and inaccuracies in the predictions. Unlike the slackfactor, though, the use of the safeband aims to prevent excessive allocation of resources, by defining a lower bound below which the system should perform an adaptation.

Additionally, when evaluating a configuration, *AccuLocal* considers a safe limit, defined as $T_{safe} = T_{high} - T_{max} \cdot slackfactor$. The purpose of T_{safe} is to serve as a target within the safety operation band for *AccuLocal* when evaluating configurations. By selecting a configuration that is not on the edges of the safety operation band, i.e., for which even small performance variations are likely to trigger a new reconfiguration, we aim at preventing reconfigurations from happening too often. Instead of targeting the middle of the *safeband*, we use the parameter *slackfactor* [1] to define T_{safe}, following the same rationale as with T_{high}: accommodate inaccuracies in the predictions while trying to keep the execution time as close as possible to the limit T_{high}, to prevent excessive allocation of resources. Further, in order to control the reactivity of the adaptation strategy, we trigger a reconfiguration if in the last k batches, at least ϵ batches took more than T_{high} time to be processed (in all experiments, we have used $k = 10$, $\epsilon = 5$).

In order to illustrate these parameters, we produced Figure 3. This figure presents the safety operation band, on which EdgeScaler aims to maintain the batch execution time, the areas that will trigger adaptations and the SLO. Within the safety operation band, we highlight the safe time T_{safe} that is used by *AccuLocal* when selecting new configurations during an adaptation. In this example we used the same parameters as in the experiment of Figure 1, namely $T_{max} = 33s$, $slackfactor = 0.1$ and $safeband = 10$. Note that the safety band $[T_{low}, T_{high}] = (29.7, 19.7)$, and that $T_{safe} = 26.4$. This means that adaptations will be triggered when the execution time gets above 29.7 seconds more than $\epsilon = 5$ times. When a reconfiguration happens, new configurations are selected aiming for an execution time of 26.4 seconds (that is, below T_{safe}).

Determining a target configuration.: Selecting the best configuration consists in determining the cheapest configuration that allows the system to operate in the safe band, that is:

$$\underset{c \in C_{candidate}}{arg\ min} \quad Cost(c) : T_{low} < T_{estimate}(c) < T_{safe}$$

EdgeScaler currently employs an exhaustive search algorithm to select the best configuration, i.e., the *AccuLocal* model is used to estimate the cost of all candidate configurations before the cheapest one is selected. As we will discuss in the evaluation section, *AccuLocal* can be queried fast enough for the search spaces that we have evaluated. However, classic search heuristics (e.g., hill climbing or simulated annealing) could be easily integrated to handle larger search spaces.

Reconfiguring the EdgeScaler: After *AccuLocal* has selected the new configuration, the ESM instantiates the new machines, if needed, and starts the migration process. In order to perform a migration, the system has to repartition and send the graph to the new machines. This is done using the Resilient Distributed Dataset (RDD) structure, provided by Spark/GraphX, according to the distribution of partitions determined by *MicroMacroSplitter*. Messages received during the migration are accumulated and included in the next snapshot, which is processed when the migration finishes.

4 EVALUATION

This section present an extensive experimental evaluation of EdgeScaler, based on a prototype that operates with GraphX[34], a distributed graph processing system based on Spark[36]. We have used this prototype with real graph datasets, so to consider a combination of micro-benchmarks and executions against real workload traces, and gather insights on the performance of EdgeScaler. Our evaluation aims at answering the following questions:

- How does *MicroMacroSplitter* perform during a reconfiguration of the worker pool, when compared with conventional hash partitioning and with the HDRF partitioner (§ 4.3)?
- What is the quality of the partitioning produced by *MicroMacroSplitter* graph partitioning scheme (§ 4.4)?
- How accurate is the *AccuLocal* model (§ 4.5, 4.6)?
- What is the relative importance of the different mechanisms embedded in EdgeScaler when faced with real workloads, and how they compare to the literature, namely Hourglass and Hash partitioner (§ 4.7)?

4.1 Experimental Platform

Despite being based on Spark, GraphX is not directly compatible with Spark Streaming. Therefore, to implement EdgeScaler and integrate it with both GraphX and Spark, we had to develop our own extensions for GraphX. Namely, we have implemented new operations that handle messages received from the streaming component and update the graph at each window, such that the user-defined analysis can be performed. This includes partitioning the incoming changes to the graph according to *MicroMacroSplitter*, collecting execution statistics that can be used by the prediction model and responding to adaptations to the execution environment as defined by the Execution Manager. EdgeScaler can run either with an initial graph, or build the entire graph from the messages received. The number of micropartitions that will be used by *MicroMacroSplitter* can be configured by the user. The user also has to provide an initial system configuration. Note that even if the initially specified configuration is not adequate, EdgeScaler can quickly adapt. We used Spark v2.3.3 and Hadoop v2.7.0 as basis for this prototype.

[1]We used *slackfactor* when computing T_{high} and T_{safe} for simplicity, but an additional slack parameter for T_{safe} could be easily included in our model

Table 2: Graphs used in the experiments

Graph	Abbrev.	#Vertices	#Edges	Type
Twitter	tw-1, tw-2	17M	164M	Dynamic
Livejournal	lj	4.8M	68M	Static
Orkut	orkut	3M	117M	Static
Hollywood	hlwood	1.1M	57M	Static

4.2 Experimental Setup

The experiments have been conducted both on a private cluster, owned by our laboratory, and on Amazon Web Services. On AWS we used machines of the m5 family, ranging from 2 virtual CPUs and 8 GiB of RAM (m5.large) to 16 vCPUs and 64 GiB of RAM (m5.4xlarge). In our laboratory, the cluster was configured to support different types of virtual machines, that mimic those available on AWS, considering their hardware configurations and prices, at the time of writing this paper. Specifically, the experiments were executed on machines similar to the m5 family previously described.

For most experiments, we have used a dynamic graph built from a dataset of over 476 million tweets extracted from Twitter [35]. To assess the performance of the proposed mechanisms with graphs that have different topological properties, we have also resorted to other well-known static graphs[22, 29]. The characteristics of all graphs that were used in the experiments are described in Table 2.

4.3 Repartitioning the Graph

The first experiments assess how *MicroMacroSplitter* performs during a dynamic reconfiguration of the worker pool, when the graph has to be repartitioned and then redistributed to a new pool of workers. We compare *MicroMacroSplitter* performance against HDRF[26] (a state-of-the-art edge partitioning algorithm that aims at achieving good partition quality at the expense of execution time) and against hash partitioning (that is used by most graph processing engines due to its fast execution).

A key factor for the performance of the system reconfiguration is the time it takes to migrate edges among workers. Edge migration is performed in two steps: first, the partitioning algorithm decides where to place each edge, and then the system actually moves edges among workers. Thus, ideally, besides achieving a good partitioning, a partitioner must also avoid moving too many edges and must decide quickly where to place them.

The results presented in Figures 4 and 5 show that the *MicroMacroSplitter* partitioner ("mms") performs much better than HDRF, and even outperforms Hash in this respect. We start with a deployment with just two workers and then double the number of workers at each reconfiguration. Therefore, the first bars on the left of each plot show the time required to reconfigure from a 2-worker to a 4-worker pool, next from a 4-worker to a 8-worker pool, an so on. Each bar includes both the time needed to perform the partitioning and the time spent moving edges between workers. The former corresponds to the dotted bottom part of each bar. *MicroMacroSplitter* was configured to use 256 micropartitions.

The partitioning time for hash algorithm is negligible and not visible in the chart. For *MicroMacroSplitter*, the partitioning time consists of first computing the the arrangement of micropartitions using METIS to decide where to place each micropartition; this introduces a small overhead when compared to hash, however, it

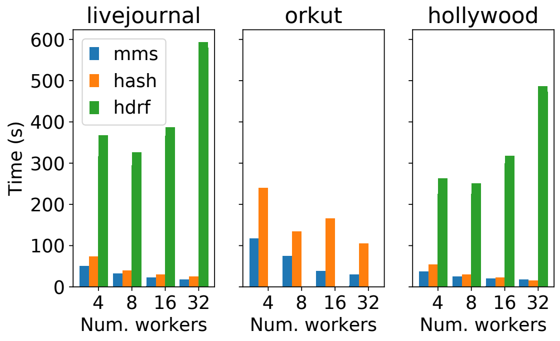


Figure 4: Time to repartition static graphs

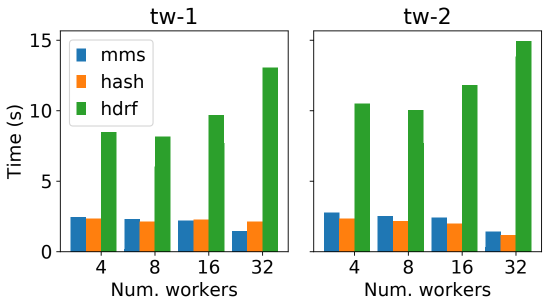


Figure 5: Time to repartition Twitter dynamic graph

is still not large enough to show in the chart for the simulated scenarios. Interestingly, unlike hash, which moves each edge at a time, *MicroMacroSplitter* moves whole micropartitions, so it can move data more efficiently between workers, which is reflected in smaller reconfiguration times when compared to hash. Because all three algorithms produce well balanced partitions (§ 4.4), the actual number of edges moved is similar in all scenarios. For both *MicroMacroSplitter* and hash, edges can be moved between workers in parallel, therefore they perform better for larger worker pools.

As for HDRF, by requiring a full graph repartition, the partitioning time is comparatively very long in our implementation. Since HDRF can only be executed on a single worker, it takes more time to repartition the graph as the worker pool increases. HDRF can also exhaust the worker resources if the graph is large. This is shown in the Orkut graph, for which it was impossible to produce an HDRF partitioning with the machines available. This experiment shows how algorithms such as HDRF, albeit producing high quality partitions, are not well suited for dynamic environments.

Figure 5 presents the same experiment on the dynamic Twitter graph. Given that Twitter is a dynamic graph, two points were selected in its evolution to perform this experiment. The points denoted "tw-1" and "tw-2" are snapshots of the dynamic graph when EdgeScaler has triggered adaptations in the experiment presented in Figure 1. The results are similar to the static graphs: *MicroMacroSplitter* performs similar to hash (albeit in this case the partitioning time becomes relatively larger, due to the smaller graph size), and much better than HDRF. Further, the reconfiguration time with *MicroMacroSplitter* ranges from 0.9 to 2.4 seconds, being fairly small in comparison to the 33 seconds deadline used in this experiment.

The time required to launch new machines during a reconfiguration was not considered in this experiment; only the time to redistribute the graph to the new machines was measured. Although launching machines usually incurs in a considerable overhead on cloud providers, this can be compensated by, after triggering an adaptation, continuing to process batches using the current configuration until the new machines are ready, and only then performing the migration. In our experiments, the time required to launch new machines is fairly small (around 40 seconds for both "tw-1" and "tw-2" scenarios, i.e., the reconfiguration is applied with a delay of two batches).

4.4 Quality of the Graph Partitioning

Although *MicroMacroSplitter* performs well during a dynamic reconfiguration, being as fast as hash partitioning in this regard, the quality of the partitions produced must also be evaluated, because low quality partitions can introduce large communication overheads during the execution of the system. This experiment assess the quality of the graph partitioning that results from using *MicroMacroSplitter*. We compare its performance against HDRF and hash partitioning, as in the last experiment. Two metrics are used to capture the quality of the graph partitioning, namely the *replication factor* (RF) and the *edge balance factor* (EBF).

The replication factor is defined as the number of replicas divided by the number of vertices in the graph (i.e., the average number of replicas per vertex). As described in Section 2.2, a smaller RF means that fewer edges are replicated in different workers and, therefore, a smaller communication overhead is incurred to synchronize the workers.

The edge balance factor is calculated as follows:

$$EBF = \frac{\textit{size of largest partition} \cdot \textit{number of partitions}}{\textit{total number of edges}}$$

Ideally, the EBF should be 1.0, as this means that every partition has exactly the same size. HDRF and other partitioning algorithms can deliver balance factors very close to this value. *MicroMacroSplitter* can approach this value depending on the number of micropartitions configured and the number of workers used.

Replication Factor: Figure 6a depicts the values of RF obtained when different techniques are used to partition several graphs into 32 partitions. The performance of *MicroMacroSplitter* ("mms") depends on the number of micropartitions that are used, thus we depict its performance for values in the range of 64 to 512 micropartitions. Three points were selected on the dynamic Twitter graph to perform this experiment. The points denoted "tw-1" and "tw-2" are snapshots of the dynamic graph taken right after the two reconfigurations triggered by EdgeScaler on the experiment described in Figure 1, i.e., after approximately 100 and 150 minutes; "tw-f" represents the state of the graph at the end of that experiment (at 300 minutes).

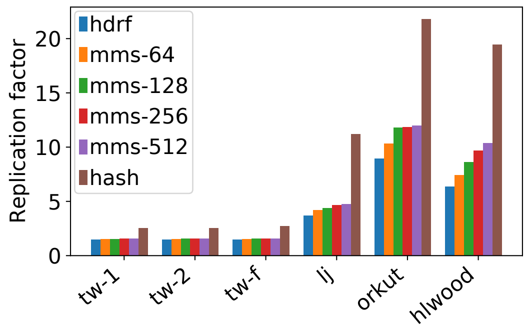
As expected, HDRF achieves the lowest RF for all static graphs (3.7 for livejournal, 9 for orkut, and 6.4 for hollywood) and hash partitioning has the worst results. (11.2 for livejournal, 21.8 for orkut, and 19.4 for hollywood). *MicroMacroSplitter*, on the other hand, is able to produce far better partitions than hash, with RFs very close to HDRF. The RF values for *MicroMacroSplitter* range from 4.1 to 4.7 for livejournal, 10.3 to 12 for orkut and 7.4 to 10.4 for hollywood. Experiments with the "twitter" graph follow the same trend, even if the differences are less expressive. *MicroMacroSplitter* produces RFs slightly higher than the direct use of HDRF, ranging from 1.53 to 1.57 for tw-1 and 1.54 to 1.59 for tw-2, compared to 1.47 and 1.48 for HDRF on tw-1 and tw-2, respectively. The RF presented by *MicroMacroSplitter* was, on average, 47% smaller than that of hash (compared to 54% smaller when using HDRF directly). Note that, although HDRF provides the best results, it is not suitable to be executed online, as required in systems that perform elastic scaling. In fact, the strategy of running HDRF every time the system needs to be reconfigured has two main drawbacks: it introduces latency, because running HDRF is computationally expensive (as shown in Section 4.3), and it may cause many edges to move, slowing down the reconfiguration process. *MicroMacroSplitter*, on the other hand, runs HDRF when the system is first started, and quickly assigns micropartitions to workers online. As shown in Figure 6a, it is able to do so without inducing a significant increase in the resulting RF.

The results for the "twitter" graph (tw-1, tw-2, tw-f) also show that *MicroMacroSplitter* is able to maintain the quality of the partitioning as the graph evolves. The three scenarios span the entire duration of the experiment depicted in Figure 1, being snapshots taken after approximately 100, 150 and 300 minutes, respectively. That is, the difference between tw-2 and tw-1 represents how *MicroMacroSplitter* performed after about 50 minutes of operation, and the difference between tw-2 and tw-f, after more than two hours of operation. The experiment shows that the quality of the partitions remains fairly constant during the whole experiment, with a maximum difference in RF of only 0.02 between snapshots.

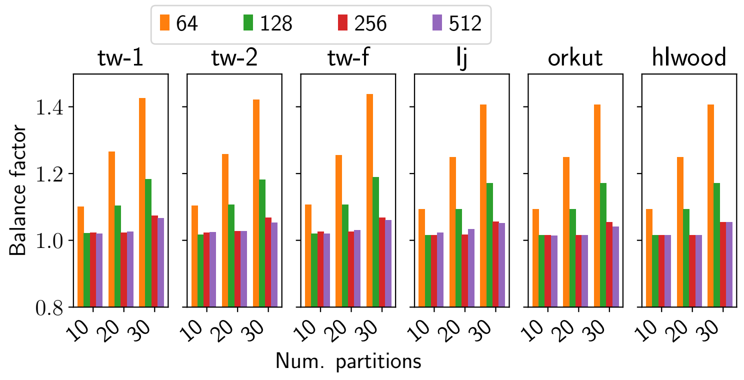
Edge Balance Factor: Figure 6b reports the EBF values obtained when applying *MicroMacroSplitter* to different numbers of partitions, namely for 10, 20 and 30 partitions, using different numbers of micropartitions computed offline. With only 64 micropartitions, *MicroMacroSplitter* does not show its full potential, achieving balance factors in the range from 1.2 to 1.4. However, with 256 micropartitions, *MicroMacroSplitter* already delivered balance factors very close to 1.0 (ranging from 1.01 to 1.07) for the three scenarios, with a small tradeoff in replication factor as depicted in Figure 6a. Thus, the use of a large number of micropartitions with *MicroMacroSplitter* has a small impact on RF, but allows to improve the EBF for a larger set of deployments.

In this scenario, *MicroMacroSplitter* also maintains the quality of the partitioning as the graph evolves. There is very little change in the EBF produced by *MicroMacroSplitter* from the first snapshot to the second. Similarly, the "tw-f" snapshot shows that the arrangement of micro-partitions remains balanced at the end of the experiment, after more than two hours without reconfigurations.

The general vs custom tradeoff: Figures 6a and 6b also expose an interesting tradeoff between the generality of a given partitioning approach and its quality. On the one hand, hash is the most general approach as it can be executed online, in an efficient manner, for any number of workers. HDRF, on the other hand, is the most specific: in this case it provides the best results for 32 partitions, and these partitions can be only used to generate balanced deployments



(a) Replication Factor (RF)



(b) Edge Balance Factor (EBF)

Figure 6: Quality of partition comparison

in worker pools of size 2,4,. . .,32. *MicroMacroSplitter*-256 provides slightly worse results for a pool of 32 workers, but with 256 micropartitions it allows to derive efficient deployments for a much larger set of different worker pools.

4.5 Accuracy of the *AccuLocal* model

A key feature of EdgeScaler is the use of a domain-specific model that allows to estimate accurately how different deployments will perform given a concrete graph that has been previously partitioned in micropartitions. The *AccuLocal* model leverages the fact that it is easy to compute the exact replication factor that will result from deploying the target graph on a given worker pool. The following experiments show how accurate the model used by EdgeScaler is.

This experiment considers the same snapshots of the dynamic twitter graph used in Section 4.4, i.e., the "tw-1" and "tw-2" snapshots obtained at the exact time a reconfiguration was performed during the experiment illustrated in Figure 1.

For this study, we have compared the performance predicted by the model with the actual performance. The set of configurations used in the experiments is composed by pools with from 1 to 10 machines of m5 family. Pools with m5.large (*l*), m5.xlarge (*xl*), m5.2xlarge (*2xl*) and m5.4xlarge (*4xl*) machines were created for this experiment. The number of machines in each pool is computed according to the definition of the space of candidate configurations, described in Section 3.2, considering $\mathcal{D} = 4$, that is, a maximum of 4 steps from the current configuration (although the results are presented only for the configurations that are predicted to achieve the safe limit T_{safe}). The base configuration for the first scenario is an initial configuration of two m5.large machines, while for the second scenario it is a set of three m5.large machines (i.e., the result of the first adaptation).

Figure 7 presents the results of this study. Each bar corresponds to a configuration and its respective cost in the format *num of machines.size*, with the cost below. The configurations are sorted by cost and actual execution time. We aim to assess if *AccuLocal* can choose the cheapest configuration that meets the user-defined SLO. As described in Section 3.2, we consider a 10% safe limit, so a configuration may be selected only if the predicted latency to process each batch is less than 27 seconds. For the first scenario ("tw-1"), EdgeScaler selected the cheapest configuration: 3 m5.large machines, with an error below 5% between the predicted and the actual execution time. Although the actual execution time for this configuration exceeded by a small margin 27 seconds, this configuration was able to meet the SLO for almost one hour in the original experiment (after which a new adaptation is performed).

As for the second scenario ("tw-2"), EdgeScaler's prediction was not as accurate. There were two configurations with the same cost: 1 m5.2xlarge machine and 2 m5.xlarge machines, with the former performing slightly better than the latter (21.07 vs 22.95 seconds). However, EdgeScaler predicted that the second configuration was faster (with an error just over 10%), so that configuration was chosen. This decision had no impact in the overall cost of the system or in its ability to meet the SLO, however, it illustrates the challenges of selecting the best configuration when considering both vertical and horizontal scaling. Overall, the average prediction error considering all configurations presented was around 10% for the first scenario and close to 12% for the second one.

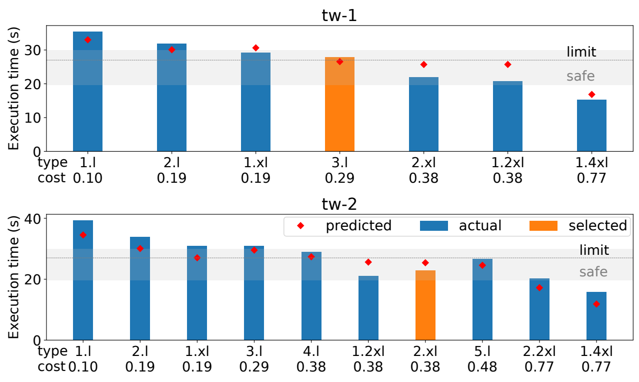
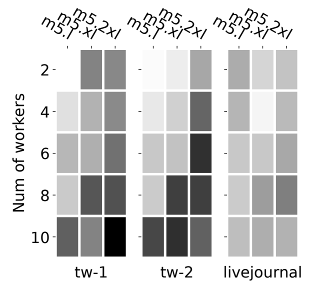
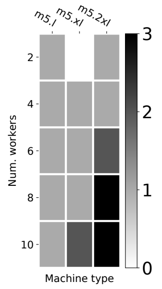


Figure 7: Precision of predictions on Twitter graph



(a) Precision



(b) Hops

Figure 8: Accuracy and number of hops to best configuration

4.6 Accuracy Horizon

Next, we aim at quantifying how large $\mathcal{D}$ can be set in order to produce predictions that are accurate enough to be useful in selecting the best configuration for a given scenario. We produced predictions for configurations ranging from 2 to 10 workers of m5.large, m5.xlarge and m5.2xlarge machines. That is, the largest configuration (10 m5.2xlarge machines) is 20x larger than the smallest one (2 m5.large machines). The prediction error was then calculated by comparing the predictions with the actual execution times.

Figure 8a presents the results of an experiment conduced on the Twitter graph, considering the two snapshots described in Figure 7, i.e., "tw-1" and "tw-2", and on the larger static graph, i.e., "livejournal". For the Twitter graphs, the initial configuration was composed of 2 m5.large machines (the top left point), and of 4 m5.xlarge machines for the livejournal graph. As can be seen, for both Twitter snapshots, the predictions are very precise within the vicinity of the initial configuration. If we consider a distance up to $\mathcal{D} = 4$ from the initial configuration, the average error is below 10%. As expected, when the configuration gets much larger than the initial one, the error increases (to a max of 24.1% for the largest configuration on "tw-1"). For the "livejournal" scenario the results present both up- and down-scaling possibilities. In this scenario, the average error was smaller, as the most distant configuration from the initial one is only 5x larger.

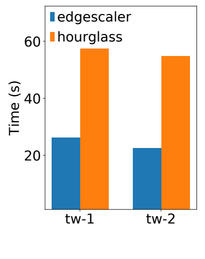
Note that although the prediction model is not as precise for configurations that are far from the current one, it can quickly adapt if the chosen configuration does not conform to the timing restrictions. Thus, a better configuration can be selected after only a few processing windows. To demonstrate this, we selected the "tw-2" scenario and counted how many different configurations (hops) the model chooses before reaching the best configuration for that scenario, for each of the initial configurations considered. Figure 8b presents the results of these experiments, and shows that for most of the initial configurations, a single hop is sufficient to reach the best configuration. Only when very different configurations are used as a starting point, more than one hop is required.

4.7 Overall Performance of EdgeScaler

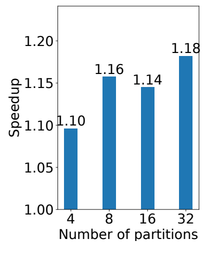
In order to handle elastic scaling for stream graph processing efficiently, EdgeScaler uses several techniques that have been described individually so far. This section shows how these components work in synergy, and how each component compares to the existing literature. To this end, the Twitter scenario presented in Section 1 is revisited, with experiments being conduced in several snapshots of the graph as it evolved during the period described in Figure 1. All experiments presented in this section were performed using pools of m5.large machines running PageRank, and the results are presented as the average time to process all batch windows (containing 30 seconds of updates), during 5 minutes of execution.

One important contribution of this work is *MicroMacroSplitter* that can be applied to evolving graphs, unlike previous work, such as Hourglass[19], which requires creating static snapshots for each processing window and, as such, incurs in large I/O and bootstrapping overheads. Figure 9a shows the overhead of Hourglass when applied to stream processing: it compares the average time to process a batch window for Hourglass and EdgeScaler, at the "tw-1" and "tw-2" reconfiguration snapshots, using in each snapshot the same worker configuration as in the experiment of Figure 1. In this experiment, we replaced the graph processing engine of our stream processing system with Hourglass. As such, for each window the graph had to be saved to a persistent storage (HDFS, in this experiment) as a static graph, Hourglass bootstrapped and executed, and the resulting graph had to be loaded again so that new updates could be applied to the graph. This results in the system being unable to process the graph within the 30 seconds window, taking in fact almost twice that time for each batch window.

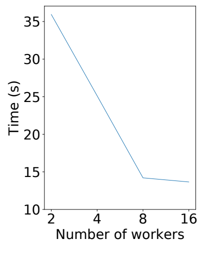
Comparing EdgeScaler to existing performance prediction models [11, 23, 27] is difficult because they are limited to the simple hash vertex partitioning algorithm, require a training phase, and are not designed for dynamic graphs. *AccuLocal*, on the other hand, supports sophisticated partitioning algorithms and can be used online on evolving graphs. Considering the precision of the predictions, *AccuLocal* is comparable to the models in the literature, with errors in the vicinity of 10% [23, 27]. However, it is clear that, by relying on *MicroMacroSplitter* to produce partitions with a small replication factor, EdgeScaler achieves faster processing times for each batch window when compared to the widely used hash partitioning that is supported by the existing performance models. To quantify this difference, snapshots of the Twitter graph were taken at the moment of the first and second reconfigurations. Then, both hash partitioning and *MicroMacroSplitter* were applied to create from 4 to 32 graph partitions, on which EdgeScaler was executed.



(a) Hourglass



(b) Speedups



(c) Horizontal scaling

Figure 9: Speedups using *MicroMacroSplitter* and horizontal scaling limitations of EdgeScaler

Figure 9b shows the average speedups obtained by *MicroMacroSplitter* when compared to hash, for the different configurations. *MicroMacroSplitter* achieves speedups that range from 10% to 18% over the hash partitioner. The speedup is smaller for 4 partitions and grows as the number of partitions increases. This happens because, for a small number of partitions, the difference in RF between *MicroMacroSplitter* and hash is also small (see Figure 6a). As the number of partitions grows, the RF for hash grows faster, while *MicroMacroSplitter* behaves more closely to HDRF and is able to deliver higher speedups.

It is also important to highlight the relevance of supporting both horizontal and vertical scaling in EdgeScaler. Figure 9c presents the batch window execution time for several configurations (ranging from 2 to 16 workers) at the same point in the Twitter graph described in the last experiment. These results show the limitation of performing only horizontal scaling: after 8 workers, adding additional workers does not yield gains in performance, because communication overheads start to dominate the execution time. By using the *AccuLocal* performance prediction model, EdgeScaler can detect when this happens and perform a vertical scaling, reducing the communication overheads.

Overall, the combination of techniques used by EdgeScaler results in a 30% reduction of the cloud costs in the scenario considered by Figure 1. Also, EdgeScaler avoids the flaws observed with our baseline, which fails to keep the system stable by the end of the experiment, missing almost all deadlines after its third reconfiguration. Even before that point, EdgeScaler misses 3 times fewer deadlines than the baseline.

5 CONCLUSIONS

This paper introduced EdgeScaler, a graph stream processing system that supports elastic scaling efficiently. An extensive experimental evaluation of EdgeScaler shows that, by combining a novel graph partitioning strategy and a performance model that takes the partitioning results into account, EdgeScaler is capable of satisfying user-defined timing restrictions, while minimizing the cost of running the system. As future work we plan to support the integration of incremental graph processing abstractions in our streaming system, such as those proposed by GraphTau [18], Kineograph [7] and others. We also plan to expand the performance prediction model using more sophisticated methods, such as Bayesian Optimisation.

Acknowledgements: This work was partially financed by CAPES and CNPq and supported by Portuguese funds through Fundação para a Ciência e a Tecnologia (FCT) via projects COSMOS (via the OE with ref. PTDC/EEI-COM/29271/2017 and via the "Programa Operacional Regional de Lisboa na sua componente FEDER" with ref. Lisboa-01-0145-FEDER-029271) and UIDB/50021/2020.

REFERENCES

[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming graph partitioning: an experimental study. *VLDN* 11, 11 (2018).

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics.. In *NSDI*.

[3] Konstantin Andreev and Harald Racke. 2006. Balanced graph partitioning. *TCS* 39, 6 (2006).

[4] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on Hadoop. In *Proceedings of Hadoop Summit*.

[5] Shivnath Babu and Jennifer Widom. 2001. Continuous queries over data streams. *ACM Sigmod Record* 30, 3 (2001).

[6] Paul E Black. 2019. Manhattan distance. https://www.nist.gov/dads/HTML/manhattanDistance.html

[7] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *Eurosys*. ACM.

[8] Marina Danilevsky and Eunyee Koh. 2013. Information graph model and application to online advertising. In *UEO*. ACM.

[9] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. 2012. Elastic virtual machine for fine-grained cloud resource provisioning. In *Global Trends in Computing and Communication Systems*. Springer.

[10] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. 2020. Incrementalization of graph partitioning algorithms. *VLDB* 13, 8.

[11] Kenrick Fernandes, Rami Melhem, and Mohammad Hammoud. 2018. Investigating and Modeling Performance Scalability for Distributed Graph Analytics. In *CloudCom*. IEEE.

[12] Raul Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*. ACM.

[13] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. *VLDB*.

[14] Tom Fu, Jianbing Ding, Richard Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2015. DRS: dynamic resource scheduling for real-time analytics over fast streams. In *ICDCS*. IEEE.

[15] Kiran Garimella, Gianmarco De Francisci Morales, Aristides Gionis, and Michael Mathioudakis. 2018. Quantifying controversy on social media. *ACM TSC* (2018).

[16] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic scaling for data stream processing. *IEEE TPDS* 25, 6 (2014).

[17] Joseph Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *OSDI*.

[18] Anand Iyer, Li Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *GRADES*. ACM.

[19] Pedro Joaquim, Manuel Bravo, Luís Rodrigues, and Miguel Matos. 2019. Hourglass: Leveraging Transient Resources for Time-Constrained Graph Processing in the Cloud. In *EuroSys*. ACM.

[20] George Karypis and Vipin Kumar. 1998. Multilevel k-way partitioning scheme for irregular graphs. *JPDC* 48, 1 (1998).

[21] Erich Lehmann and George Casella. 2006. *Theory of point estimation*. Springer Science & Business Media.

[22] Jure Leskovec and Andrej Krevl. 2017. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data

[23] Zengxiang Li, Bowen Zhang, Shen Ren, Yong Liu, Zheng Qin, Rick Siow Mong Goh, and Mohan Gurusamy. 2017. Performance modelling and cost effective execution for distributed graph processing on configurable VMs. *CCGRID*.

[24] Federico Lombardi, Leonardo Aniello, Silvia Bonomi, and Leonardo Querzoni. 2017. Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE TPDS* (2017).

[25] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*.

[26] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. HDRF: Stream-Based Partitioning for Power-Law Graphs. In *CIKM*.

[27] Daniel Presser, Frank Siqueira, and Fabio Reina. 2018. Performance Modeling and Task Scheduling in Distributed Graph Processing. In *BigData*. IEEE.

[28] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *VLDB* 11, 12 (2018).

[29] Ryan Rossi and Nesreen Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*.

[30] Semih Salihoglu and Jennifer Widom. 2013. GPS: A graph processing system. In *SSDBM*.

[31] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *SIGMOD*.

[32] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commu.of the ACM* 33, 8 (1990).

[33] Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics.. In *NSDI*.

[34] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. GraphX: A resilient distributed graph system on spark. In *GRADES*.

[35] Jaewon Yang and Jure Leskovec. 2011. Patterns of temporal variation in online media. In *WSDM*. ACM.

[36] Matei Zaharia et al. 2016. Apache Spark: A unified engine for big data processing. *Comm. of the ACM* 59, 11 (2016).