Future Generation Computer Systems 136 (2022) 270–281

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Network-aware worker placement for wide-area streaming analytics☆

Habib Mostafaei [a,*], Shafi Afridi [b], Jemal Abawajy [c]

[a] Eindhoven University of Technology, The Netherlands
[b] Technische Universität Berlin, Germany
[c] Deakin University, Australia

ARTICLE INFO

Article history:
Received 18 May 2021
Received in revised form 2 June 2022
Accepted 10 June 2022
Available online 18 June 2022

Keywords:
Internet of Things (IoT)
Worker node placement
Wide-area stream analytics
Stream processing
Simple additive weighting
Wide Area Network (WAN)

ABSTRACT

Many organizations leverage Distributed Stream processing systems (DPSs) to get insights from the data generated by different users/devices, e.g., the Internet of Things (IoT) devices or user clicks on a website, on geographically distributed datacenters. The worker nodes in such environments are connected through Wide Area Network (WAN) links with various delays and bandwidth. Therefore, minimizing the execution latency of a task on the worker nodes while using the links with enough bandwidth and lower cost to steer the traffic of the applications is a challenging task. In this paper, we formulate the worker node placement for a geo-distributed DSPs network as a multi-criteria decision-making problem. Then, we propose an additive weighting-based approach to solve it. The users can prioritize the worker node placement according to the network-relevant parameters. We also propose a framework that can be integrated with the current DPSs to execute the tasks. We test our placement approach on three widely used stream processing systems, i.e., Apache Spark, Apache Storm, and Apache Flink, on three custom graphs adopted from the real cloud providers. We run the streaming query of the Yahoo! streaming benchmark on these three DPSs. The experimental results show that our approach improves the performance of Spark up to 2.2x–7.2x, Storm up to 1.2x–3.4x, and Flink up to 1.4x–3.3x compared with other placement approaches, which makes our framework useful for use in practical environments.

1. Introduction

Emerging applications running on the Internet of Things (IoT) devices, social networks, or user-clicks on websites typically generate a massive amount of continuous stream of raw data from distributed geographical locations. Organizations need to analyze and mine these data to obtain insights and valuable intelligence such as trend detection in social networks in real-time. These applications normally have stringent requirements ranging from low-latency to high bandwidth. Therefore, processing this massive amount of streaming data in a single location within a limited timeframe is not practically possible. As a result, distributed stream processing systems (DSPs) composed of geographically distributed (geo-distributed) networks that communicate via WAN have recently become a popular choice to run these applications [2]. The main idea of geo-distributed stream data processing systems is to push the computations to the edge of the network close to the sources of the streaming data. This approach provides several benefits that include privacy preservation and cost-saving due to the minimization of data transfer overhead.

Although geo-distributed cluster networks have the capacity to handle stream data processing, stringent QoS requirements of the stream data processing applications raise fundamental resource management and scheduling challenge. Currently, various systems such as Apache Spark [3], Apache Storm [4], and Apache Flink [5] are used for processing streaming data. However, these systems are designed to process the queries at a single location on a cluster rather than on geo-distributed cluster networks.

There are several attempts to address the stream data processing in geo-distributed cluster networks [2,6,7]. The techniques in such solutions are mostly applied for batch scenarios in which the input data is available prior to query execution. Although the approaches proposed in existing works [8–11] do not assume that the data size and the rate are not known in advance, these solutions focus on different aspects such as links delay, bandwidth, and cost of running a task in different datacenters. For example, the work in [12] claims that the link cost should have higher priority than the delay and bandwidth in a multi datacenters

☆ The preliminary version of this paper, titled "SNR: Network-aware Geo-Distributed Stream Analytics", authored by H. Mostafaei, S. Afridi, and J. H. Abawajy, was published in the IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (NEAC workshop) (Mostafaei et al. 2021) [1]. Part of this work has been conducted while Habib Mostafaei was with Technische Universität Berlin.

* Corresponding author.
E-mail addresses: h.mostafaei@tue.nl (H. Mostafaei), safridi@inet.tu-berlin.de (S. Afridi), jemal.abawajy@deakin.edu.au (J. Abawajy).

https://doi.org/10.1016/j.future.2022.06.009

environment. The reason for such a claim is due to the amount of data transferred using high-cost links. Nevertheless, none of these solutions offer the flexibility in prioritizing various network parameters in placing the workers on a geo-distributed cluster.

In this paper we extend our work in [1] and address the placement of worker nodes on geo-distributed cluster networks. Specifically, we attempt to answer the following questions: (i) How can we prioritize network metrics to efficiently run a task on geo-distributed cluster networks?; and (ii) What is the trade-off among different network parameters in placing worker nodes in a multi-cloud environment? To answer the above questions, we first formulate the problem of placement of worker nodes on geo-distributed cluster network as a multi-criteria decision-making problem and use the Simple Additive Weighting (SAW) [13] method to solve it. We call the proposed approach a SAW-based Node Ranking (SNR) algorithm. The main goal of SNR is to find the best placement of the worker nodes on geo-distributed cluster networks by considering multiple network criteria. We check the different impacts of SNR on the placing nodes, and the obtained results show that the internode delays of our algorithm on average are 2.4x less than the current default algorithm. To perform the real-world experiments, we use the Yahoo! streaming benchmark [14] on a cluster of 11 VMs running Apache Spark, Apache Storm, and Apache Flink. Our results show the significant performance improvement of SNR compared with the default placement approach.

The contribution of our work can be summarized as follows:

- We develop SNR (worker node placement on geo-distributed cluster network) that considers the most significant network relevant parameters when placing the workers in a geo-distributed cluster network;
- We study the trade-off among the relevant parameters of worker selection;
- We perform real-world experiments to evaluate the performance of SNR on a set of custom and all networks taken from TopologyZoo [15].
- We report that SNR improves the execution latency of Apache up to 2.2x–7.2x, Apache Storm up to 1.2x–3.4x, and Apache Flink up to 1.4x–3.3x compared with the default placement approach.

The rest of the paper is organized as follows. Section 2 states the system model and problem formulation. The detail of the SNR algorithm comes in Section 3. Section 4 details the evaluation of the SNR and reports the obtained results. The related works come in Sections 5 and 6 concludes the paper.

2. System model and problem statement

2.1. System model

We consider scenarios in which several geographically distributed datacenters (DCs) form a network. Fig. 1 shows an example scenario with five DCs, i.e., DC1 to DC5, connected through several network devices such as routers or switches using WAN links. Each DC in Fig. 1 consists of a set of compute slots, i.e., CPU cores and memory, with diverse uplink and downlink for data transmission depending on the amount of investment to build. We rely on virtualization techniques to create Virtual Machines (VMs) to run DSPs on each DC. We now explain how wide-area stream processing can be executed on a set of DCs in our model using DSPs.

The DSPs such as Apache Storm [4] and Apache Flink [5] have a master–slave architecture in which the master node is in charge of executing user-submitted tasks on a set of slave nodes. We need to deploy the slave nodes of DSPs on different DCs to run

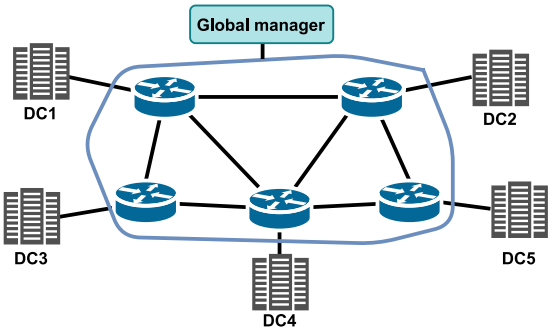


Fig. 1. A geo-distributed stream-processing system spanning over five datacenters (DCs).

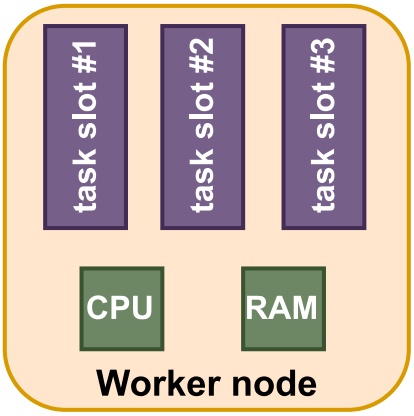


Fig. 2. A worker node with three task slots.

a streaming task in a geo-distributed setting. We leverage the available VMs on each DC to execute the functionality of the slave nodes of each DSP — we call these available VMs as the worker nodes. Fig. 2 shows an example worker node with three task slots. For instance, *task slot*#1 can run on a data stream to identify the security issues in real-time, while *task slot*#2 can execute another task such as top k event detection. Therefore, assigning tasks to each task slot depends on the application use cases. Other example use cases of such platforms are fraud detection by the credit card companies and ad-hoc analysis of live data in the transportation automobile. E-commerce industries such as Alibaba and Zalando use streaming platforms for real-time analysis of user data to optimize search and end-user suggestions [16]. We call the network of worker nodes a cluster for the scope of our work.

Modern DSPs such as Apache Flink manages the compute resources of each worker node using their dedicated resource managers. Each worker node is a Java Virtual Machine (JVM) process and can execute a set of tasks using a set of separate threads. We can control the number of executable tasks on a worker node by task slots. Each task slot represents a fixed amount of resources. The worker node shares its memory among all its task slots. For example, the worker node in Fig. 2 can share 1/3 of its memory among *task slot*#1, *task slot*#2, and *task slot*#3. Each task slot can also run a separate instance of a streaming task without interfering with the others.

Processing model. We now explain how data processing happens in a streaming scenario. The data processing in streaming scenarios runs for an infinite time and requires the following steps:

- Ingesting a sequence of data,

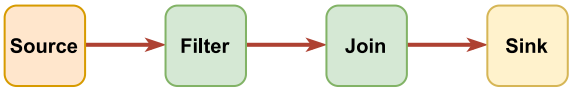


Fig. 3. An example DAG of a streaming query.

- Updating metrics and reports, and
- Summarizing statistics in response to each arriving data record.

This kind of processing is better suited for real-time monitoring and response functions. The input data comes as a stream, i.e., continuous data records in small sizes, from different sources. Examples of data sources are sensors, user clicks in a website, or mobile users. The size of incoming data is not known in advance since it depends on the application scenarios and the number of sources generating data streams. Data analytics in stream processing includes tasks like finding correlations or aggregations among the data or filtering data according to the application needs to state a few examples.

Herein, we provide an example streaming task in the form of a DAG to explain our streaming query. Fig. 3 shows the DAG of streaming query in which it receives the incoming data from a source. The query filters the incoming data according to the user-defined values and joins the obtained results. The final results are then pushed to a sink for other purposes.

One of the goals of wide-area stream processing is to process the data close to the source for several reasons such as saving the bandwidth and cost or privacy of data. An example constraint on data privacy is the European Union General Data Protection Regulation (GDPR) [17]. It obligates the user's consent for data processing or exchanging with third-party systems. The same rule applies if the data should be processed or stored in the infrastructure of other countries. In such scenarios, data privacy cannot be guaranteed.

In our model, the global manager runs the tasks across different DCs depending on the enterprise requirements. It has a view of the available resources and decides where a task should run. The global manager is located in the central datacenter that interacts with the data managers located at each edge datacenter [7]. This is a realistic assumption since with the adoption of Software-Defined Networking (SDN) [18], the controller has the global view of the network and can provide us enough information for decision making. Having a single global manager can be a point of failure in this model, and leveraging multiple global managers can mitigate this problem. However, the global manager is not a performance bottleneck since the worker nodes do data processing.

The global manager receives the streaming tasks from the users and executes them on the DCs. We assume that the global manager runs the same streaming task on all DCs. However, the number of required task slots to process the input data can be different, since it depends on the amount of incoming streaming data. The global manager can decide to assign the task slots depending on the processing power of each task slot and the amount of input data.

The task slots of each DC are connected through WAN links that have different properties such as bandwidth, delay, and cost. For example, the uplink and downlink bandwidths of a link can be different because different applications share the same links [6]. The main advantage of having a global manager is that it can periodically query the underlying network infrastructure to obtain updated information on the status of links and update the tasks execution plan accordingly. This way of running the tasks leads to minimizing query execution time. However, using a global manager can be a single point of failure for the whole task execution since it is in charge of running tasks. But, other architectures such as having multiple global managers or even a manager for each WAN can be considered.

In this model, the worker nodes in each DC process the incoming input data and produce the intermediate results. The intermediate results are aggregated in the central DC that receives data through WAN links. We now provide an example word-count scenario to explain how this works. The worker nodes in each DC count the number of unique words independently and transfer the results of the counts to the central DC. The central DC aggregates all the word counts over a specific time window and reports them as the final results to the end-user.

2.2. Problem statement

The main objective is to minimize the execution latency and cost of streaming tasks while obeying the bandwidth constraints in placing the workers of a cluster in a geo-distributed environment. The streaming tasks run for an infinite time. We consider the network relevant parameters such as bandwidth, delay, and cost when placing the tasks for execution. The main reason to consider the network-related parameters is that the WAN links properties dictate the overall query execution in the streaming scenarios. Furthermore, in contrast to bath processing tasks that have to process a large amount of data, in streaming tasks, the amount of data in a specific time window plays a determinant role in the execution time. For example, consider a windowing task that runs over several DCs, and the input data rate is very few. This amount of data can be processed without restrictions on the VMs since the incoming data rate is too low. However, the overall query execution time is the time that the last input data on the slowest link is finished. Therefore, WAN links dictate the overall query execution, and we consider this factor as our goal.

We consider a network topology of $G = (V, E)$ in which V is the set of worker nodes, and E is the set of edges (links) connecting the worker nodes. Each worker has a set of available task slots to execute tasks. We assume that the workers already exist and ready for task execution. Each link in G has three different parameters, namely, bandwidth, delay, and cost. The main goal is to enable the users to select a set of worker nodes in the network according to the given priority to the network-related parameters.[1]

Delay. The internode delays in a geo-distributed cluster are different from those in a single cluster. The delay can vary from 10 s to 100 s of milliseconds depending on the locations of the compute slots [19]. Therefore, link delay plays a determinant role in task execution, especially for delay-sensitive applications. For such kinds of applications, every millisecond of latency can have a huge economical impact on enterprises. For example, Akamai in 2017 reported that every 100 ms of delay have a determinant impact in dropping the customers of online businesses [20]. Therefore, one of our goals is to select a subset of computing slots in such a way that the sum of internode delays is minimized. Mathematically,

$$\text{minimize} \quad \sum_{x=1}^{n} T_x^y \tag{1}$$

$$\text{subject to} \quad T_x^y > 0, \quad \forall (x, y) \in G' \quad \text{and} \quad x \neq y,$$

where T_x^y is the link delay in milliseconds (ms) crossing from node x to y, and n is the number of chosen worker nodes from graph G.

[1] We use the available task slots on a VM as the same meaning of the worker nodes to avoid any confusion.

We assume that G' is the sub-graph of the chosen nodes in graph G.

Cost. We model the cost as follows. Let C_x^y be the data transmission cost from worker x to y through the link among them. The total transmission cost can be computed as follows.

$$\text{minimize} \quad \sum_{x=1}^{n} C_x^y \tag{2}$$
$$\text{subject to} \quad C_x^y > 0, \quad \forall (x, y) \in G' \quad \text{and} \quad x \neq y.$$

We consider the cost as the cost of steering 1 GB of data traffic over a link in USD ($). The goal is to minimize the sum of the cost of G'.

Bandwidth. We model the network traffic as follows. Let B_x^y be the available bandwidth of a link in Mbps from worker x and y. The total generated traffic on link $(x, y) \in E$ can be computed as follows.

$$\sum B_x^y, \quad \forall (x, y) \in E' \quad \text{and} \quad x \neq y, \tag{3}$$

where E' is the set of edges in G'. We assume that multiple tasks can be executed on the available task slots on a worker. Therefore, each task generates data traffic and consumes a portion of the available bandwidth between nodes x and y. The goal is to maximize the minimum available bandwidth among the selected nodes in G'. Consider a scenario in which we have several worker nodes distributed on a geo-distributed network. In this case, there are multiple hops between two nodes in the graph with diverse parameters. Therefore, the link with the lowest bandwidth determines the amount of traffic that can be sent through that path.

3. SNR algorithm

In this section, we state how SNR selects the task slots by considering multiple network-based criteria. We also explain a running example of SNR in placing worker nodes in the cluster network.

3.1. SNR algorithm

We use different criteria in the node selection step of the SNR algorithm. This problem is known as the multi-objective problem, and we use the Simple Additive Weighting (SAW) method to transform the problem into a single objective one [21]. The SAW method is a simple and popular technique to make a decision on a set of attributes for each alternative. We explain the detail of SNR as follows.

Let E be a set of links and m be the set of decision criteria for each link in G. Here, we consider the available bandwidth, delay, and cost as the decision criteria, i.e., $|m| = 3$. We prefer the highest value for the available bandwidth. While for the delay, and cost the lowest values are better. Let assume that w_k is the weight of importance for each criterion.

Algorithm 1 presents the pseudo-code of the SNR algorithm. In this algorithm, we first normalize the network-relevant parameters of each connected link to a node in the graph. Then, we compute the rank of each node to select a node in each step of the algorithm. The algorithm is general-purpose and can be used to select a subset of task slots available on the worker nodes.

First node selection. The SNR algorithm starts by selecting the first node on a graph. Therefore, we first normalize the criteria based on the network-related parameters of neighbors of each node using Eq. (4). To do so, we take the average of the available bandwidth of all connected links to a node and divide it over the maximum available bandwidth. Similar to the bandwidth, SNR computes the average link delays and costs. Then, it divides the averaged values into their minimum values. This procedure executes once when SNR selects the first node. We normalize the values of attributes of each node x in graph G as follows.

$$\psi_x = \begin{cases} \dfrac{\frac{\sum_{e=1}^{n} e_{xy}}{n}}{max(e_{xy})} & \text{, bandwidth} \\ \dfrac{min(e_{xy})}{\frac{\sum_{e=1}^{n} e_{xy}}{n}} & \text{, delay and cost,} \end{cases} \tag{4}$$

where n is the number of neighbors of a node and e_{xy} is the edge from node x to y. This equation returns the normalized value ψ_x for the entire nodes of the graph. Now, we assign the weight for each criterion of a link connected to node x according to the user needs. Mathematically,

$$R_{node} = \sum \psi_x w_k, \quad \text{for} \quad x = 1, \ldots, n \quad \text{and} \quad k = 1, 2, 3, \tag{5}$$

where R_{node} is the rank of each node in the graph G, n is the number of neighbors of node x, and w_k is the priority weight of each attribute, i.e., bandwidth, latency, and cost. Each w_k has a value in the range (0,1), and the sum of them is equal to 1. We select the node with the highest rank as the first node. We

Algorithm 1: The SNR algorithm

input : Network graph G, links bandwidth, delay, cost , number of worker nodes, priority of each parameter
output: A set of nodes W

```
W = ∅                          /* The set of worker nodes */
/* We compute the average value of each
   parameter and normalize them. */
for each x ∈ G do
    N=G.getNeighbors(x)
    L_sum = 0, BW_sum = 0, C_sum = 0
    for y ∈ N do
        BW_sum += B_x^y
        L_sum += T_x^y
        C_sum += C_x^y
    end
    normalize the parameters of each node using Eq. (4)
end
x = firstNode(G)
W = W ⋃ x
while |W| < t do
    /* We look for the neighbors of x to add the
       next node to W */
    for y ∈ x.Neighbors do
        find maximum value of each parameter
    end
    for y ∈ N do
        apply Eq. (6)
    end
    find rank using Eq. (5)
    β=get the neighbor with the highest rank
    W = W ⋃ β
    x = β
end
```

colocate the master and the first worker node of the cluster on the first node. We use the master node to submit the query to a set of workers/clients. Algorithm 2 shows the pseudo-code of the first node selection of the SNR approach.

Algorithm 2: Find the first node

input : Network graph G , bandwidth, delay, and cost of each link
output: A node with highest rank (x_{best})

Function `firstNode(G)`:
  tmp=∅
  for ***each*** $x \in G$ **do**
    for $y \in N$ **do**
      BW_{max} = maxBandwidth(x,y)
      L_{min} = minLatency(x,y)
      C_{min} = minCost(x,y)
    end
    for $y \in N$ **do**
      $E_{BW} = w_1 \cdot \frac{B_x^y}{BW_{max}}$
      $E_L = w_2 \cdot \frac{L_{min}}{T_x^y}$
      $E_C = w_3 \cdot \frac{C_{min}}{C_x^y}$
    end
    find rank using Eq. (5)
    tmp= tmp $\bigcup$ x
  end
  $x_{best} = \varnothing$
  $R_{tmp} = 0$
  for $x \in tmp$ **do**
    `/* R_x is the rank node x. */`
    if $R_x > R_{tmp}$ **then**
      $R_{tmp} = R_x$
      $x_{best} = x$
    end
  end
  return x_{best}
End Function

Node selection. After selecting the first node, we use the neighbor nodes of this node to add the next node to our subset of worker nodes. To do so, we normalize the available bandwidth, delay, and cost of the links that are connected to the current node using Eq. (6). Then, we apply Eq. (5) by replacing the ψ_x with π_x to rank the neighbors of the current node. The neighbor with the highest rank will be selected as the next node, and this procedure continues until the desired number of worker nodes has been chosen.

$$\pi_x = \begin{cases} \frac{\sum_{e=1}^{n} e_{xy}}{max(e_{xy})} & \text{, bandwidth} \\ \\ \frac{min(e_{xy})}{\sum_{e=1}^{n} e_{xy}} & \text{, delay and cost.} \end{cases} \tag{6}$$

Note that we remove the selected node from the available nodes list while using Eq. (5). This mechanism speeds up the node selection step of the SNR algorithm due to the reduction in the search space.

Time complexity. We analyze the time complexity of SNR in more detail. First, the algorithm needs to find the rank of each node for the first node selection. This function needs $k \times N$ iteration for the execution where k is a constant and equals to the number of neighbors for each node and N is the number of nodes in the graph. Therefore, this function requires $O(N)$ to complete the execution. Lines 14–25 of Algorithm 1 needs $t \times N$ to finish in which t is the number of worker nodes and N is the number of nodes in the graph. All in all, Algorithm 1 has the time complexity of $O(N) + O(N)$.

Connectivity of the chosen worker nodes. After selecting a subset of worker nodes by SNR from the network graph, we have a connected sub-graph. The reason to have the connectivity among the chosen nodes comes from the nature of the SNR node selection phase. We prove the connectivity of the chosen nodes in Theorem 1 as follows.

Theorem 1. *Let G' be a sub-graph of graph G, then G' is a connected graph.*

Proof. To prove the connectivity of G', we explain how the nodes are selected. Suppose that SNR selects node x from graph G in the first node selection phase. In the node selection phase, SNR adds a node to the sub-graph G' by using the neighbors of the first node. Then, the subsequent nodes are selected using the neighbors of currently selected nodes. This property makes the final sub-graph connected. □

Measuring the impact of node selection. To measure the impact of the node selection phase, we use the chosen nodes in the sub-graph. To do so, we take the shortest path from one node to other nodes to calculate the average inter-node delays, the maximum available bandwidth, and the sum of the data transmission cost. We perform the computations for the cases that there are multiple hops among the nodes as follows.

- **Average inter-node delays:** if a node such as x is in a few hops distance from another node like y in the chosen sub-graph, we take the sum of delays of different hops to use the delay of node x to node y. We take the average inter-node delay among all the nodes in the chosen sub-graph after computing the inter-node delay for each pair of nodes.
- **Maximum available inter-node bandwidth:** if there multiple hops between node x and node y in the chosen sub-graph, we take the minimum available bandwidth of different hops to use the maximum available bandwidth from node x to node y. We do this procedure for all the nodes in the chosen sub-graph and take the minimum available bandwidth as the maximum available bandwidth of SNR.
- **Sum of inter-node cost:** if there multiple hops between node x and node y in the chosen sub-graph, we take the sum of different hops as the cost from node x to node y. We do this procedure for all the nodes in the chosen sub-graph and take the sum of all paths among different nodes as the sum of cost of SNR.

3.2. SNR orchestrator

We develop an orchestrator to integrate SNR with the current DSPs, i.e., Storm, Spark, and Flink. Fig. 4 shows the internal architecture of SNR. To apply to the worker node selection algorithm, we need to have the network topology information such as the number of nodes, links, and the properties of each link. This information provides the required input for the worker node selection step. Then, the orchestrator receives the desired weight of each networking parameter, i.e., bandwidth, latency, and cost, from the user to apply the coefficient of the parameters as well as the required number of worker nodes. SNR applies the node selection phase to select the required number of worker nodes. At the end of this phase, we have a list of worker nodes from the network topology that should have been started to run the desired streaming query.

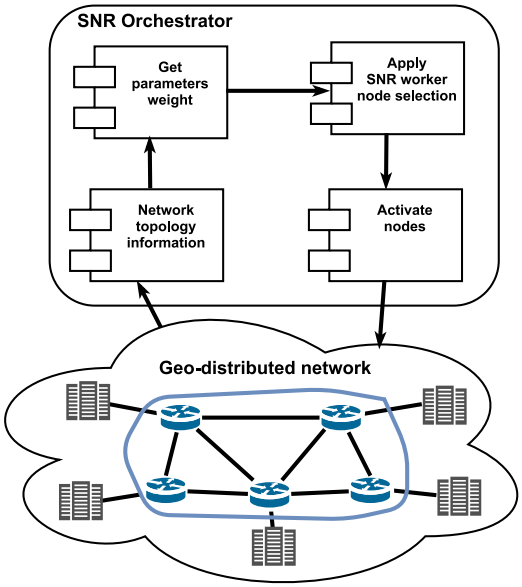


Fig. 4. The internal architecture of SNR orchestrator.

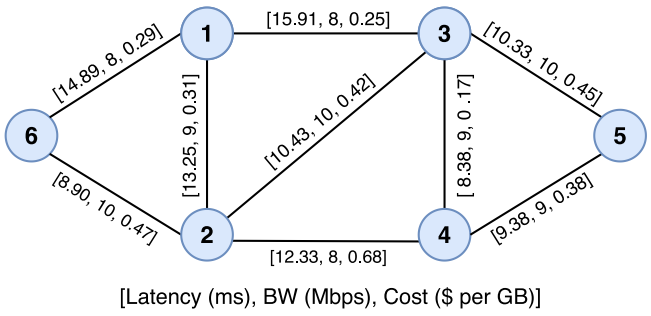


Fig. 5. An example graph with six nodes.

Table 1
The normalized values for each node and their corresponding rank in first node selection step of SNR.

Node	Latency	BW	Cost	E_x
Node 1	0.90	1.00	0.68	2.57
Node 2	1.00	0.60	0.88	2.48
Node 3	1.00	0.88	0.87	**2.75**
Node 4	0.90	0.69	0.98	2.57
Node 5	0.97	0.68	1.00	2.65
Node 6	0.97	0.75	0.83	2.55

Table 2
The normalized values for each node with their neighbors rank in SNR.

Node	Normalized values			E_x	To
	BW	Latency	Cost		
Node 1	1.00	1.00	0.81	2.81	Node 2
	0.89	0.83	1.00	2.72	Node 3
	0.89	0.89	0.86	2.64	Node 6
Node 2	0.90	0.67	1.00	2.57	Node 1
	1.00	0.85	0.74	2.59	Node 3
	0.80	0.72	0.46	1.98	Node 4
	1.00	1.00	0.66	2.66	Node 6
Node 3	0.80	0.53	0.68	2.01	Node 1
	1.00	0.80	0.40	2.21	Node 2
	0.90	1.00	1.00	**2.90**	**Node 4**
	1.00	0.81	0.38	2.19	Node 5
Node 4	0.89	0.68	0.25	1.82	Node 2
	1.00	1.00	1.00	3.00	Node 3
	1.00	0.89	0.45	**2.34**	**Node 5**
Node 5	1.00	1.00	1.00	3.00	Node 3
	0.89	0.52	0.59	1.99	Node 4
Node 6	0.80	0.60	1.00	2.40	Node 1
	1.00	1.00	0.62	2.62	Node 2

After the worker node selection procedure, we have a list of worker nodes to be started by SNR for the query execution but we need to install a set of forwarding rules on top of the network devices to steer the network traffic among them. Therefore, SNR generates the required IPv4 forwarding rules to install on the network devices among the worker nodes.

To execute the given streaming tasks on the chosen worker nodes, SNR needs the directory of each DSP on all worker nodes. This information is mandatory since the only way to start the worker nodes is to have access to the right directory that the DPS resides on each system. The orchestrator uses the daemons provided by each DSP to start either the master or the worker nodes. It also sets all the desired parameters on each DPS using the predefined tunable parameter file for each DPS. For example, if the worker nodes of Apache Flink have the capacity to run multiple instances of the same task, we set the `parallelism.default` parameter in `flink.yaml` file to the desired value. There is an equivalent parameter in Apache Storm and Apache Spark for the same purpose. Then, the SNR orchestrator submits the received streaming tasks to the chosen worker nodes for execution.

3.3. Running example for SNR

In this section, we explain our SNR method in selecting the worker nodes by using an example. Fig. 5 depicts a network graph of 6 nodes. Each edge in this graph has three network-related parameters, namely, the link bandwidth in Mbps, the link latency in milliseconds, and the link cost as the cost of transferring 1 GB of data over that link in $. We consider the Round-Trip Time (RTT) delay as the link delay in this graph.

First node selection. To select the first node from the graph in SNR, we compute the normalized value of all nodes in the graph. In this example, we have 6 nodes, and Table 1 indicates the normalized value for the bandwidth, latency, and cost of each edge connected to that node. We compute these values using the same parameters on each edge in Eq. (4). SNR computes the weight of each edge in the graph using the normalized values. We apply Eq. (5) and select the maximum value as the highest rank. SNR selects the corresponding node to that highest rank link as the first node. According to the example graph in Fig. 5, the SNR selects node number 3 as the first node.

Node selection. We assume that our goal is to select three nodes for the cluster and all three networking parameters have the same priority for the placement. After choosing the first node, we apply a similar mechanism for the remaining nodes in the graph until we reach the desired number of nodes for the cluster. According to our example, SNR selects node number 3 as the first node.

Table 2 shows the rank of each node in the example graph. Now, we check the neighbors of this node and their rank to choose the next node. Among the neighbors of node number 3, node number 4 has the highest rank value, and we add this node as the next node to our list of selected nodes. We need to pick another node according to our needs, and we check the neighbors of node number 4 to choose the next node because this is the last selected node in the cluster. Among the neighbors of this node, SNR picks node number 5. Therefore, the final selected nodes are 3, 4, and 5.

Table 3
The trade-off among the metrics in example graph of SNR.

Prioritized metric	Nodes	BW	Latency	Cost
Bandwidth	{3,2,6}	10,10,10	10.43,8.9,19.33	0.42,.47,0.89
Latency	{5,4,3}	8,10,9	9.38,10.33,8.38	0.38,0.45,0.17
Cost	{1,3,4}	8,9,8	15.91,8.38,24.29	0.25,0.17,0.42

Table 4
Custom graph properties.

Graph	Nodes	Links	BW [Mbps]	Delay [ms]	Cost [$ per GB]
Small	20	35	[7, 14]	[1, 25]	[0.02, 0.25]
Medium	30	65	[7, 14]	[26, 75]	[0.02, 0.25]
Large	50	140	[7, 14]	[76, 125]	[0.02, 0.25]

Tradeoff among the parameters. We now show the tradeoff among different parameters. Table 3 shows the impact of giving a higher priority to the bandwidth, delay, and cost. The nodes column in this table shows the selected nodes by SNR for the corresponding prioritized metric. When we give the highest priority to the bandwidth, SNR picks the nodes with the highest available bandwidth. For example, by prioritizing the bandwidth, SNR selects nodes 3, 2, and 6. There are 3 links among these nodes, and thus, we have three values for each row in Table 3 that shows the chosen parameter value for that link. While for the latency and cost, SNR selects the nodes with the lowest latency and cost. The corresponding values for each prioritized parameter are highlighted in red in Table 3. The results show that the priority of each network parameter dictates the selection of the nodes. The obtained results confirm that SNR selects the worker nodes based on the applications' demand.

4. Evaluations

In this section, we first study the tradeoff among different network-related parameters of the worker node placement by assigning priorities to bandwidth, latency, and cost of WAN links. Then, we measure the performance of the SNR algorithm on three custom and all topologies of TopologyZoo [15]. The main reason to choose three custom topologies to run our experiments relies on the fact that the networks of TopologyZoo lack information regarding the link latency, bandwidth, and cost. The goal is to assess the impact of placement on various network-related performance metrics such as average delay among the nodes and the number of hops. Finally, we measure the impact of placement on the execution latency of streaming queries using Yahoo! streaming benchmark [22].

We use the real networks' delay such as the ones in [19,23], cost [24–26], and bandwidth [27] among the worker nodes and set the locations of worker nodes using the real-world datacenter locations [28–30]. We create three custom random graphs, namely small, medium, and large, to simulate diverse networks in terms of size and other parameters. Table 4 presents the corresponding network parameters with their values in each graph. The link delay information for TopologyZoo networks is obtained using the coordination information.

4.1. Illustrating tradeoff

We check the tradeoff among different network-relevant parameters by assigning different priorities among them for scenarios with 8 worker nodes. In this experiment, we first give the highest priority to the available bandwidth while keeping the priority among the delay and cost fixed and the same in Eq. (5). Then, we do the same measurements for the delay and cost in all three custom topologies. The values for the bandwidth are in [Mbps], latency in [ms], and cost [$ per GB].

Table 5
The trade-off among the metrics in small topology.

Prioritized metric	min(BW)	$\overline{\text{Latency}}$	$\sum$Cost
Bandwidth	9.4	34.29	17.12
Latency	7.3	20.55	22.58
Cost	7.3	26.29	15.47

Table 6
The trade-off among the metrics in medium topology.

Prioritized metric	min(BW)	$\overline{\text{Latency}}$	$\sum$Cost
Bandwidth	10.1	107.35	16.44
Latency	7.5	92.18	24.03
Cost	8.3	96.16	11.28

Table 7
The trade-off among the metrics in large topology.

Prioritized metric	min(BW)	$\overline{\text{Latency}}$	$\sum$Cost
Bandwidth	8.7	223.89	16.98
Latency	7.0	132.95	11.53
Cost	7.2	155.95	10.4

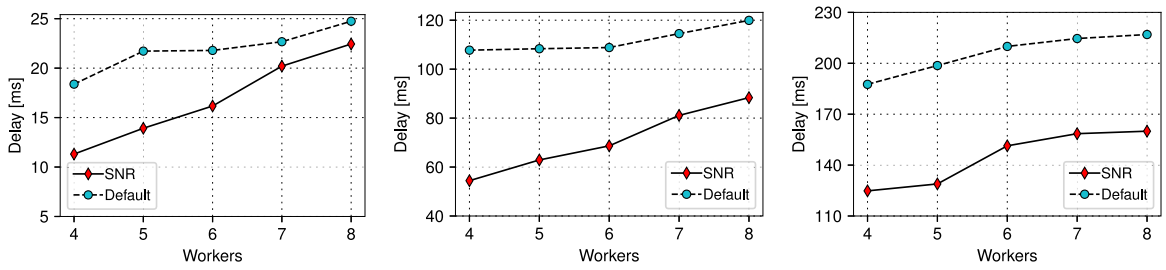
Tables 5, 6, and 7 reports the tradeoff among bandwidth, delay, and cost of running SNR on small, medium, and large topologies for the links among the chosen nodes. In these tables, we report the minimum available bandwidth (min(BW)) among the chosen nodes because it will impact the data transfer rate. We put the average delays ($\overline{latency}$) among the nodes, while for the cost, we sum the cost ($\sum$cost) of each among the selected nodes. We highlight the corresponding values from the output of SNR for each prioritized parameter in red. The obtained results confirm that SNR can select the worker nodes according to their priority. For example, it selects worker nodes with minimum delays among them when link delay has the highest priority in Tables 5, 6, and 7.

4.2. Topology-aware results

In this section, we report the results of our experiments on different network topologies. We measure the effectiveness of the SNR in placing the worker nodes on all network topologies of TopologyZoo.

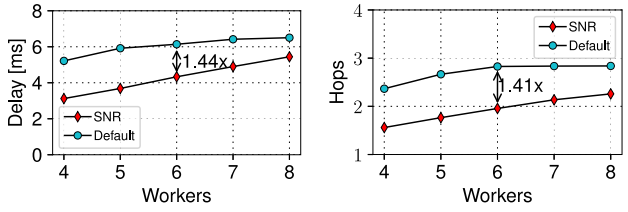
Custom topology. First, we use the same priority among the parameters and measure the average link delay among the chosen worker nodes by the SNR and default algorithms. In the default approach, each DSP selects the worker nodes in an ordered fashion starting from node number 1. We run the SNR and default methods to choose 4 to 8 worker nodes by assuming that all the worker nodes have the same number of available task slots to execute the streaming tasks.

Fig. 6 shows that by increasing the number of worker nodes in a cluster, the average delay among them also increases in all three topologies. We summarize the main reason for such an increment in the average link latency among the chosen worker nodes. When the DSP cluster has more worker nodes, the number of links connecting them increases because we need more links to connect the worker nodes. We may need to cross several links to reach from a worker node to another one. In such scenario, the link latency between two nodes is the sum of latency of the intermediate links latency. Consequently, we need to find the average values of more links possibly with higher latency. Therefore, the overall average links latency of the algorithms increases. The obtained results show that the average delays in SNR are 1.35x, 1.42x, and 1.61x less than the default method in



(a) Small (b) Medium (c) Large

Fig. 6. The average delay among the worker nodes on custom topologies for different number of workers in a cluster network.



(a) (b)

Fig. 7. SNR vs. default. (a) Average delay among the chosen nodes in TopologyZoo networks (b) Average hops among the chosen nodes in TopologyZoo networks.

small, medium, and large topologies, respectively. However, SNR starts the node selection procedure by using the links with lower latency so that the overall average link latency is less than the default approach.

TopologyZoo results. We check the impact of placement in SNR on all network graphs of TopologyZoo. We use the coordination information of the nodes to compute the link delay among the nodes for the networks with such data. Fig. 7(a) shows that the SNR selects the worker nodes on average 1.44x less link delay than the default approach. Furthermore, Fig. 7(b) presents that the traffic among the nodes should cross 1.41x times more hops in the default approach than SNR. Using a fewer number of hops decreases the routing overhead among the nodes, and it also better utilizes the available capacity of the links.

4.3. Evaluation on real systems

In this section, we report the testbed used to run the Yahoo! streaming benchmark. The Yahoo! streaming benchmark [14] is a popular streaming benchmark that has been used in other research studies related to the performance evaluation of big data analytics platforms [31]. The Yahoo! streaming benchmark measures the performance of DSPs, e.g., Apache Storm, Apache Spark, and Apache Flink. The benchmark emulates an advertisement analytics pipeline in the DSPs and measures the performance of the various systems. There is a number of advertising campaigns in the query in which each one gets a set of advertisements. The producer of the benchmark generates events with a timestamp. Then, it truncates them to a specific digit that determines the campaign it belongs to. Each event also carries the last update timestamp along with the event information. The benchmark uses Apache Kafka [32] for event generation. After processing the event by each DSP, the benchmark calculates the event latency by deducting the window timestamp and duration from the last updated timestamp. The benchmark uses Redis [33] as the sink of query and it was the bottleneck for the benchmark. The bottleneck has been removed from the benchmark in [22].

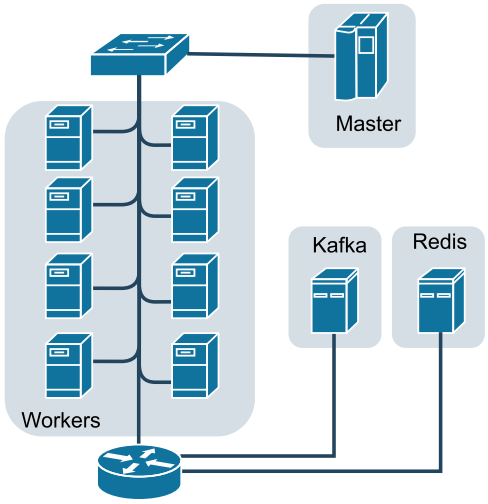
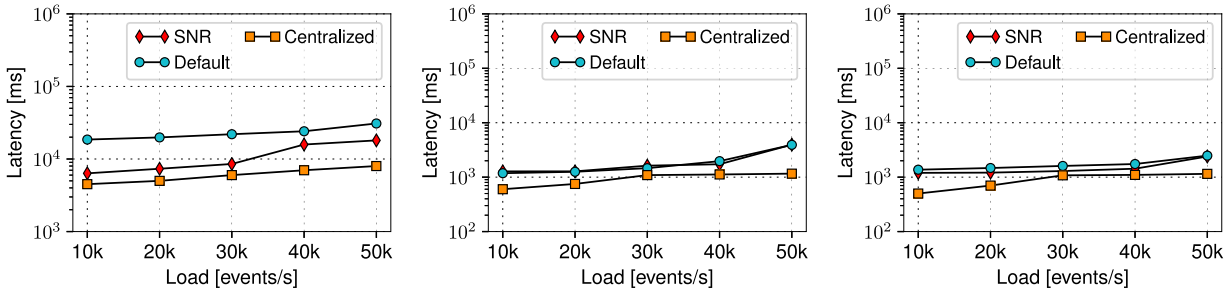


Fig. 8. The topology of testbed used for our emulation.

Finally, the obtained latency value along with the number of processed events are written into appropriate files. More detailed information on the benchmark could be found in [14].

The testbed has 11 VMs with 16 CPU cores and 8 GB of RAM running Debian 10. Each DSP has a server–client architecture, where the master node executes the tasks on the set of slaves or worker nodes. We assign a VM for the master node, and 8 VMs for the worker nodes in each system. Therefore, the cluster executes the query with 8 worker nodes emulating 8 different locations in the network. Each worker node has 6 task slots to execute the streaming query and we use 6700 MB of memory in each worker to use for task execution. We dedicate a VM for Kafka and a VM for the Redis database. We use Kafka VMs to generate the required input rate. Each Kafka producer can generate up to 25k events/second without adding delay to the events. One of the main goals for geo-distributed stream processing is to process the events close to the source of data. To emulate such an environment, we connect Kafka VM via dedicated links to the worker nodes resulting in zero milliseconds of delay among them. We add more Kafka VMs to the cluster to generate the desired input data. Furthermore, we connect the Redis VM to all workers via a dedicated collision domain to have zero milliseconds of delay among the workers and sink. Fig. 8 shows the topology of our testbed.

We measure the impact of the worker node placement on three custom topologies on Spark, Storm, and Flink. To do so, we apply SNR on the network graphs, i.e., small, medium, and large, to select the place of the worker nodes and obtain the network-related parameter values. Then, we use the `tc` tool to set artificial



(a) Spark (b) Storm (c) Flink

Fig. 9. The execution latency of Spark, Storm, and Flink on small topology by varying the input data rate.

delays among the worker nodes given from running SNR. Each node is connected to other nodes via a link avoiding the routing overhead.

We also apply the recommended settings of the Yahoo! streaming benchmark to set the configuration parameters in each DSP. We set the Spark batch interval to 10k ms, and also the event acknowledgment of Storm to 0. We connect the Kafka and Redis VMs directly to the worker nodes. This setting allows us to simulate the scenarios in which the query execution of the system is performed close to the source. Furthermore, we use the parallelism parameter of Flink, i.e., `parallelism.default`, to 48 and the settings for Spark by assigning `spark.default.parallelism` to 48. We also use 6 task executors per worker in Storm. Finally, we assume that latency has the highest priority in selecting worker nodes and assign $w_2 = 0.9$ in Eq. (5).

The $ cost of executing a task on a worker plays a role when the DCs are geographically distributed around the globe and more than a continent. We have checked the $ cost of steering traffic of different regions of the public cloud providers such as Google [25] and Amazon [26] and found that it remains fixed and constant on the same region/continent. However, this factor plays a determinant role when the DCs are on different continents. Therefore, we exclude this parameter in placing the workers.

We compare the performance of SNR with those of the default approach and the centralized ones. The latter one indicates the performance of DSPs when running the experiments in a rack on a DC. The main goal to include the results of the centralized approach can be summarized as follows. First, we study the effect of running experiments on WANs compared with those of a single DC. Second, we show the impact of different algorithms in selecting different places for the worker nodes in the streaming scenarios.

4.3.1. Small topology

In the *small topology*, the nodes are close to each other, which results in having low link delay among them. The worker node placement on the *small topology* can give us insight regarding the performance of DSPs when running them across a small continent around the globe such as Europe since the inter-node latency is mostly less than 30 ms. Therefore, the obtained results from this set of experiments on the *small topology* can help us to better design the network for geo-distributed systems in Europe size. We vary the input rate in the range of 10k to 50k events per second.

Figs. 9(a), 9(b), 9(c) show the 99-percentile execution latency for each DSP. The general trend in these three figures can be summarized as follows. By increasing the input rate of each system, the execution latency of the events also increases. However, the performance of Storm and Flink are similar, while Spark has the highest execution latency. Furthermore, the SNR improves the execution latency of Spark up to 1.5x–2.9x, Storm up to 1.1x–2.5x, and Flink up to 1.1x–1.45x compared with the default approach, respectively. The reason for such improvements is due to a low inter-node latency among the worker nodes in the DSPs. We also checked 95th- and 90th percentile latency of the task execution for Spark, Storm, and Flink, and found similar behavior. The obtained results confirm that Spark, Storm, and Flink can process the events irrespective of WAN delays when they are in the range of a few 10 s of milliseconds.

We also report slight performance degradation of all DSPs in small topology compared with a centralized approach. We summarize our observation as follows. All DSPs can tolerate some milliseconds of links delay. However, this observation is limited due to the availability of Kafka VMs in our testbed, but our goal is to show that even a small amount of links delay can degrade the performance of stream processing systems.

4.3.2. Medium topology

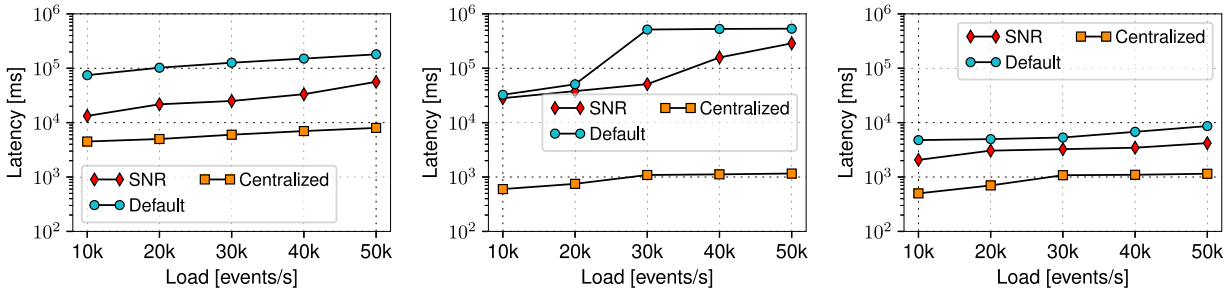
We do the same experiments on *medium topology* that has higher inter-node delays compared with those of small topology. The obtained results from the *medium topology* can give us insights into the performance of DSPs when running them on a US-size network since the inter-node latency is less than 75 ms. However, crossing several links can result in higher latency between the source and the destination of the connection.

Fig. 10 shows the execution latency in the medium topology is longer than the small topology since the internode delays are longer. However, SNR improves the execution latency of Spark up to 2.2x–7.2x, Storm up to 1.2x–3.4x, and Flink 1.4x–3.3x compared with the default approach. One of the main reasons for such results lies in Transmission Control Protocol (TCP) that suffers from high RTT among the worker nodes. The second reason for the difference among DSPs comes from the different architectures of each one. Spark streaming is not a pure stream processing system and processes the incoming stream of events in a micro-batch fashion. Therefore, the batch interval of Spark plays a determinant role here. Furthermore, Storm topology uses a different number of task executors using Spout and Bolts operators. Therefore, running a Storm topology in a geo-distributed environment needs further attention. Even though if all DSPs use the Netty framework [34] for internode communications, but the way they exchange data is different.

We report the performance of the default approach on Spark and Storm degrades more than an order of magnitude on medium topology compared with a centralized one while having less effect on Flink.

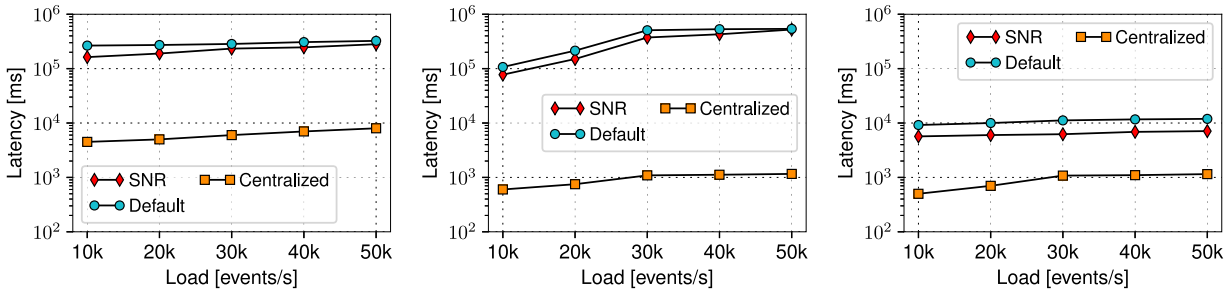
4.3.3. Large topology

We also obtain close results like the medium topology for Spark, Storm in the large custom topology. The obtained results from the *Large topology* can give us insights into the performance



(a) Spark (b) Storm (c) Flink

Fig. 10. The 99-percentile execution latency of Spark, Storm, and Flink on medium topology by varying the input data rate.



(a) Spark (b) Storm (c) Flink

Fig. 11. The 99-percentile execution latency of Spark, Storm, and Flink on large topology by varying the input data rate.

of DSPs when running them on a network that is scattered around the globe since the inter-node latency is less than 125 ms. However, crossing several links can result in higher latency between the source and the destination of the connection. For example, the link between Tokyo and New York on a network topology such as WonderNet [19] can result in more than 200 ms of delay.

Fig. 11 shows the execution latency of Spark, Storm, and Flink on the large topology. The general trend in the results of large topology is that all the considered DSPs have higher execution latency compared with the small and medium topology. The main reason for such behavior relies on the inter-node latency among the worker nodes in the large topology. Since the links have longer latency the execution latency is also higher regardless of the worker node selection algorithm. While Storm and Spark have similar execution latency, Flink can process the incoming events faster. Additionally, SNR improves the execution latency of Spark up to 1.15x–1.62x, Storm up to 1.1x–1.44x, and Flink 1.47x–2.1x compared with the default approach.

We report the performance of the default approach on Spark, Storm, and Flink degrades more than order of magnitude on large topology compared with a centralized one. The main reason for such results relies on the nature of links in large topology since some links have a latency of more than 150 ms. The links with higher latency dictate the query execution time.

4.3.4. Throughput

We now evaluate the throughput of the DSPs by varying the input load on the small topology using the SNR algorithm. The input data rate varies in the range of 10k to 50k events per second. The goal of this experiment is to understand how different DSPs react to the input rate. We take the total sum of processed events and report them for the throughput analysis with million unit (M). Table 8 shows that Spark, Storm, and Flink have similar throughput for the different scenarios with various input rates. Furthermore, this trend exists for both SNR and Default methods.

We now report the throughput of DSPs for the medium topology in Table 9. Spark can process a slightly less number of events compared with the small topology for both SNR and Default approaches. Storm performs completely differently when applying SNR and Default mechanisms compared with the small topology. In the case of using SNR, Storm has similar throughput compared with the small topology except for scenarios with 50k load. In contrast, for the case of Storm Default, the throughput increases by increasing the input rate up to 40k load but it decreases when the load is 50k. Flink still has the same throughput as the small topology.

Table 10 shows the total throughput of Spark, Storm, and Flink for different scenarios with various input rates on the large topology. The throughput of Spark and Storm is significantly decreased compared with the medium topology since the worker nodes have longer inter-node delays. In contrast, Flink process a similar

Table 8
The throughput of Spark, Storm, and Flink on small topology for SNR and Default.

Load	Spark		Storm		Flink	
	SNR	Default	SNR	Default	SNR	Default
10k	1.94M	1.94M	1.98M	1.98M	1.98M	1.98M
20k	3.88M	3.88M	3.97M	3.97M	3.97M	3.97M
30k	5.82M	5.82M	5.95M	5.95M	5.86M	5.85M
40k	7.88M	7.88M	7.81M	7.81M	7.81M	7.84M
50k	9.7M5	9.75M	9.93M	9.93M	9.86M	9.93M

Table 9
The throughput of Spark, Storm, and Flink on medium topology for SNR and Default.

Load	Spark		Storm		Flink	
	SNR	Default	SNR	Default	SNR	Default
10k	1.78M	1.78M	1.98M	1.98M	1.98M	1.98M
20k	3.88M	3.88M	3.97M	3.97M	3.97M	3.97M
30k	5.82M	5.82M	5.95M	4.35M	5.96M	5.95M
40k	7.88M	7.88M	7.94M	6.95M	7.94M	7.91M
50k	9.75M	9.75M	9.1M	5.44M	9.93M	9.93M

Table 10
The throughput of Spark, Storm, and Flink on large topology for SNR and Default.

Load	Spark		Storm		Flink	
	SNR	Default	SNR	Default	SNR	Default
10k	1.48M	1.14M	1.98M	1.98M	1.98M	1.98M
20k	1.78M	1.31M	3.97M	3.89M	3.97M	3.96M
30k	4.78M	4.22M	5.71M	2.82M	5.95M	5.95M
40k	5.42M	4.46M	5.44M	2.91M	7.97M	7.93M
50k	5.46M	4.76M	4.71M	3.02M	9.93M	9.92M

number of events. SNR improves 1.14x–1.36x the throughput of Spark, while this improvement is up to 2.0x for Storm. Flink can process a similar number of events in both algorithms. Note that we already reported the differences of DSPs and algorithms in terms of execution latency.

5. Related work

This section briefly reports the current state-of-the-art placement in the geo-distributed analytics systems.

Wide-area Data Analytics. Several works have been proposed to execute queries in wide-area scenarios [2,6,7,12,35,36]. These works consider different aspects of job execution in a wide-area such as minimizing the bandwidth usage or handling the WAN delays. For example, Kimchi [12] studies the impact of the network cost on the task placement by giving the priority to the link cost rather than bandwidth and delay. Additionally, the paper considers the dynamic of link cost in the selection procedure by applying the proposed approach on Apache Spark. The work in [36] consider the communication costs of the WAN link in the geo-distributed scenarios to reduce the latency of query execution. RTSATD [37] minimizes the completion time and monetary cost of processing big data workflows in clouds without delaying the completion of workflows. These solutions mostly tackle the problems in scenarios in which the input data size known prior to the query execution.

Wide-area Stream Analytics. There are numerous works that have been considered the applications for the streaming scenarios [8–10,38,39]. In this case, there is no ending for the job execution and the input data rate can change due to several reasons like the number of users generating the data for the system. A WAN-aware scheduling algorithms for Apache Flink in multi-query scenarios have been proposed in [39,40]. The system checks the common parts of the input queries and schedules them to run once. The work in [38] considers WAN bandwidth limitations of a geo-distributed streaming cluster to schedule the streaming task in Apache Spark streaming using Amazon EC2. CONA [41] addresses the congestion problem in the inter-datacenter transfer methods that use the bandwidth allocation for high utilization. There are some attempts that find the tradeoff between two network-relevant parameters and performance in wide-area analytics. For example, the works [6,12,35] consider the tradeoff between the query execution and WAN usage. However, the contributions of these works consider some of the network-relevant parameters for the task execution in wide-area scenarios.

Third-party resource managers. There are several third-party resource management daemons such as Apache Yarn [42], Mesos [43], or Omega [44] that are integrated with the big data analytics frameworks. Yarn and Mesos do not consider the network bandwidth in managing the system resources. While Oktopus [45] and Orchestra [46] consider the availability of the network resources. Nevertheless, both system do not focus on big data analytics.

6. Conclusion

We propose a worker node placement framework using a simple-additive weighting approach for geo-distributed datacenters running stream processing tasks. The framework does the placement according to the preferred network-relevant parameters, i.e., bandwidth, latency, and data transmission cost. It also finds the tradeoff among the parameters when placing the worker nodes. We applied the worker node selection algorithm of our framework on all the networks of TopologyZoo and a set of custom topologies to show the effectiveness of our algorithm. Furthermore, we performed a set of experiments to emulate the real-world streaming use-cases using the Yahoo! streaming benchmark. The results show that our worker node selection algorithm significantly improves the performance of the current DSPs. Additionally, the framework can be easily integrated with the current DSPs. The contribution of this work is limited to the network-relevant parameters for the placement. We plan to extend our framework to use the concept of Software-Defined Networking (SDN) for network monitoring and placement. We also intend to include the available physical resource information of the different worker nodes in the network for the placement purpose. We also plan to include the heterogeneity of task slots in placing the worker nodes in the geo-distributed streaming scenarios.

CRediT authorship contribution statement

Habib Mostafaei: Conceptualization, Investigation, Project administration, Software, Supervision, Validation, Visualization, Writing—original draft, Writing–review and editing. **Shafi Afridi:** Software, Validation, Writing–review and editing. **Jemal Abawajy:** Writing—original draft, Writing–review and editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was partially funded by the German Ministry for Education and Research as BIFOLD — Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A).

References

[1] H. Mostafaei, S. Afridi, J.H. Abawajy, SNR: NEtwork-aware geo-distributed stream analytics, in: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2021, pp. 820–827, http://dx.doi.org/10.1109/CCGrid51090.2021.00100.
[2] R. Viswanathan, G. Ananthanarayanan, A. Akella, CLARINET: WAN-aware optimization for analytics queries, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 435–450.
[3] Apache spark, 2020, https://spark.apache.org/.
[4] Apache storm, 2020, https://storm.apache.org/.
[5] Apache flink, 2020, https://flink.apache.org/.
[6] Q. Pu, G. Ananthanarayanan, P. Bodík, S. Kandula, A. Akella, P. Bahl, I. Stoica, Low latency geo-distributed data analytics, in: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015, 2015, pp. 421–434.
[7] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, G. Varghese, WANalytics: ANalytics for a geo-distributed data-intensive world, in: CIDR 2015, 2015.
[8] D. Kumar, J. Li, A. Chandra, R. Sitaraman, A TTL-based approach for data aggregation in geo-distributed streaming analytics, Proc. ACM Meas. Anal. Comput. Syst. 3 (2) (2019).

[9] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, E.A. Lee, AWStream: ADaptive wide-area streaming analytics, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, in: SIGCOMM '18, 2018, pp. 236–252.
[10] F. Lai, J. You, X. Zhu, H.V. Madhyastha, M. Chowdhury, Sol: Fast distributed computation over slow networks, in: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 2020, pp. 273–288.
[11] B. Heintz, A. Chandra, R.K. Sitaraman, Optimizing timeliness and cost in geo-distributed streaming analytics, IEEE Trans. Cloud Comput. (2017) 1.
[12] K. Oh, A. Chandra, J. Weissman, A network cost-aware geo-distributed data analytics system, in: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), 2020, pp. 649–658.
[13] T. Evangelos, Multi-Criteria Decision Making Methods: A Comparative Study, Vol. 4, Kluwer Academic Publication, Netherland, 2000.
[14] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B.J. Peng, P. Poulosky, Benchmarking streaming computation engines: Storm, flink and spark streaming, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 1789–1792.
[15] The internet topology zoo, 2020, http://www.topology-zoo.org/dataset.html.
[16] Apache flink use-cases, 2021, https://flink.apache.org/usecases.html.
[17] Data protection in the EU, the general data protection regulation (GDPR); regulation (EU) 2016/679, 2016, http://bit.ly/3qdVUVo.
[18] N. Feamster, J. Rexford, E. Zegura, The road to SDN: An intellectual history of programmable networks, SIGCOMM Comput. Commun. Rev. 44 (2) (2014) 87–98, http://dx.doi.org/10.1145/2602204.2602219.
[19] G. p. s. p. times between wondernetwork servers, 2020, https://wondernetwork.com/pings.
[20] J. Young, T. Barth, Web performance analytics show even 100-millisecond delays can impact customer engagement and online revenue, Akamai Online Retail Performance Report, 2017.
[21] E. Triantaphyllou, Multi-criteria decision making methods, in: Multi-Criteria Decision Making Methods: A Comparative Study, Springer, 2000, pp. 5–21.
[22] Extending the yahoo streaming benchmarks, 2020, https://github.com/dataArtisans/yahoo-streaming-benchmark.
[23] ATT Network delay, 2020, [link]. URL http://soc.att.com/30cKc2m.
[24] Microsoft azure: bandwidth pricing details, 2020, http://bit.ly/3e5PkxF.
[25] Google cloud: pricing, 2020, https://cloud.google.com/pubsub/pricing.
[26] Amazon EC2 on-demand pricing, 2020, http://amzn.to/3beAFOJ.
[27] Akamai's [state of the internet]: Q1 2017 report, 2017, https://www.bit.ly/3jPSKEP.
[28] Microsoft azure, 2020, https://bit.ly/3qdWimV.
[29] Google datacenters, 2020, https://about.google/locations/.
[30] Amazon, 2020, https://amzn.to/38zsFq4.
[31] S. Zeuch, B.D. M., J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, V. Markl, Analyzing efficient stream processing on modern hardware, VLDB 12 (5) (2019).
[32] Apache kafka, 2020, https://kafka.apache.org/.
[33] Redis, 2020, https://redis.io/.
[34] Netty framework, 2020, https://netty.io/.
[35] C.-C. Hung, G. Ananthanarayanan, L. Golubchik, M. Yu, M. Zhang, Wide-area analytics with multiple resources, in: Proceedings of the Thirteenth EuroSys Conference, in: EuroSys '18, 2018.
[36] W. Xiao, W. Bao, X. Zhu, L. Liu, Cost-aware big data processing across geo-distributed datacenters, IEEE Trans. Parallel Distrib. Syst. (ISSN: 2161-9883) 28 (11) (2017) 3114–3127.
[37] H. Chen, J. Wen, W. Pedrycz, G. Wu, Big data processing workflows oriented real-time scheduling algorithm using task-duplication in geo-distributed clouds, IEEE Trans. Big Data 6 (1) (2020) 131–144, http://dx.doi.org/10.1109/TBDATA.2018.2874469.
[38] W. Li, D. Niu, Y. Liu, S. Liu, B. Li, Wide-area spark streaming: Automated routing and batch sizing, IEEE Trans. Parallel Distrib. Syst. 30 (6) (2019) 1434–1448.
[39] A. Jonathan, A. Chandra, J. Weissman, Multi-query optimization in wide-area streaming analytics, in: Proceedings of the ACM Symposium on Cloud Computing, in: SoCC '18, 2018, pp. 412–425.
[40] A. Jonathan, A. Chandra, J. Weissman, WASP: WIde-area adaptive stream processing, in: Proceedings of the 21st International Middleware Conference, in: Middleware '20, 2020, pp. 221–235, http://dx.doi.org/10.1145/3423211.3425668.
[41] X. Tao, K. Ota, M. Dong, W. Borjigin, H. Qi, K. Li, Congestion-aware traffic allocation for geo-distributed data centers, IEEE Trans. Cloud Comput. (2020) 1.
[42] Apache hadoop YARN, 2020, http://bit.ly/3kTErjX.
[43] Apache mesos, 2020, https://mesos.apache.org/.
[44] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, J. Wilkes, Omega: Flexible, scalable schedulers for large compute clusters, in: Proceedings of the 8th ACM European Conference on Computer Systems, in: EuroSys '13, 2013, pp. 351–364, http://dx.doi.org/10.1145/2465351.2465386.
[45] H. Ballani, P. Costa, T. Karagiannis, A. Rowstron, Towards predictable datacenter networks, SIGCOMM Comput. Commun. Rev. 41 (4) (2011) 242–253, http://dx.doi.org/10.1145/2043164.2018465.
[46] M. Chowdhury, M. Zaharia, J. Ma, M.I. Jordan, I. Stoica, Managing data transfers in computer clusters with orchestra, in: Proceedings of the ACM SIGCOMM 2011 Conference, in: SIGCOMM '11, 2011, pp. 98–109, http://dx.doi.org/10.1145/2018436.2018448.

Habib Mostafaei received the Ph.D. in Computer Science and Engineering from Roma Tre University in 2019. He is currently an Assistant Professor of Computer Science at the Eindhoven University of Technology. Before, he was a postdoctoral researcher at Technische Universität Berlin where he was involved in the BIFOLD-BBDC project from 2019 to 2022. He is a member of ACM and IEEE. His main research interests include networked systems, network measurements, and distributed systems. For additional information: https://mostafaei.bitbucket.io/.

Shafi Afridi received a dual degree EIT Digital masters program at KTH Royal Institute of Technology, Sweden, and Technische Universität, Berlin in 2022. He is affiliated as a research assistant with the research group Internet Network Architectures (INET) of TU-Berlin. He received a B.E degree in Electrical (Telecom) Engineering from the National University of Sciences and Technology, Islamabad, Pakistan, in 2016. His research topics include big data, internet measurements, and programmable networks.

Jemal Abawajy is currently a Full Professor with the Faculty of Science, Engineering and Built Environment, Deakin University, Australia. He has authored/coauthored over 250 refereed articles and supervised numerous Ph.D. students to completion. He has delivered over 50 keynote and seminars worldwide and has been involved in the organization of over international conferences in various capacity, including chair and general co-chair. He has also served on the editorial board of numerous international journals, including the IEEE Transactions on Cloud Computing.