Cluster Computing (2020) 23:241–263
https://doi.org/10.1007/s10586-019-02908-2

Efficient deadline-aware scheduling for the analysis of Big Data streams in public Cloud

Mahmood Mortazavi-Dehkordi[1] · Kamran Zamanifar[1]

Received: 8 February 2018 / Revised: 20 November 2018 / Accepted: 8 January 2019 / Published online: 14 February 2019

Abstract
The emergence of Big Data has had a profound impact on how data are analyzed. Open source distributed stream processing platforms have gained popularity for analyzing streaming Big Data as they provide low latency required for streaming Big Data applications using Cloud resources. However, existing resource schedulers are still lacking the efficiency and deadline meeting that Big Data analytical applications require. Recent works have already considered streaming Big Data characteristics to improve the efficiency and the likelihood of deadline meeting for scheduling in the platforms. Nevertheless, they have not taken into account the specific attributes of analytical application, public Cloud utilization cost and delays caused by performance degradation of leasing public Cloud resources. This study, therefore, presents BCframework, an efficient deadline-aware scheduling framework used by streaming Big Data analysis applications based on public Cloud resources. BCframework proposes a scheduling model which considers public Cloud utilization cost, performance variation, deadline meeting and latency reduction requirements of streaming Big Data analytical applications. Furthermore, it introduces two operator scheduling algorithms based on both a novel partitioning algorithm and an operator replication method. BCframework is highly adaptable to the fluctuation of streaming Big Data and the performance degradation of public Cloud resources. Experiments with the benchmark and real-world queries show that BCframework can significantly reduce the latency and utilization cost and also minimize deadline violations and provisioned virtual machine instances.

Keywords Streaming Big Data analysis query · Deadline-aware scheduling · Cloud-based stream processing

1 Introduction

The emergence of data-intensive applications has rapidly increased the volume, variety and velocity of the generated data during its lifecycle which represents a major challenge for many organizations and is known as the Big Data problem [1]. Streaming Big Data is related to the velocity dimension of Big Data and refers both to how fast data are generated and how fast they need to be analyzed [2]. Analysis of streaming Big Data is the last and most important stage of the streaming Big Data lifecycle in analytical applications [3]. The real-world examples of the applications can be in the form of analytical queries to analyze the network traffic, healthcare data, financial transactions or web-clicks [4]. As the structure of the queries is usually complex and the input streams of the queries have high intensity of data, they can benefit from large-scale infrastructures [5].

Among such infrastructures, public Cloud is an appropriate infrastructure to host the queries because it can operate as pay-per-use model and is able to provide dynamic resource scaling in response to the fluctuating resource demand of the queries. This Cloud utilization model is called *Infrastructure as a Service* (IaaS) and can be used by the Cloud-based stream computing platforms to schedule accepted *soft deadline-constrained* queries using the leased resources. For a *soft deadline-constrained* query, the maximal value of the computation is not achieved if the assigned deadline is violated and additional deadline

✉ Kamran Zamanifar
zamanifar@eng.ui.ac.ir

Mahmood Mortazavi-Dehkordi
mortazavi@shbu.ac.ir

[1] Software Department, Computer Engineering Faculty, University of Isfahan, Isfahan, Iran

violations can occur when the required resources have been leased from public Cloud. As reported by [6], public Cloud does not provide regular performance in terms of execution and data transfer time. The performance fluctuation of public Cloud resources can vary up to 30% for execution and up to 65% for data transfer. The performance fluctuation can lead to delayed execution of the query operators and if the operators be part of the query critical path, a deadline violation can occur.

The efficient scheduling of the provisioned Cloud resources can increase the likelihood of the deadline meeting and reduce the average tuple latency and utilization cost of queries accepted by the distributed stream computing platforms. However, current platforms do not adequately address the problem of efficient scheduling of soft deadline-constrained streaming Big Data analytic queries in public Cloud. Recent works have proposed solutions to reduce the average tuple latency of streaming Big Data queries in private clouds. However, to the best of our knowledge, the soft-deadline meeting of the analytical queries, the reduction of the public Cloud utilization cost of the analytical queries and performance variation of public Cloud resources are not jointly investigated.

To address these limitations, we developed a scheduler, BCframework, with low tuple latency and utilization cost suitable for soft deadline-constrained streaming Big Data analytic queries by considering the poor performance of public Cloud resources. Our approach is inspired by the idea that using the structure of soft deadline-constrained streaming Big Data analytic query to partition the query critical path operators and applying a distinct scheduling strategy for each partition can reduce both the average tuple latency and the utilization cost of the soft deadline-constrained query. More precisely, as a sample query shown in Fig. 1, the weight of the outgoing edge of the operator O_{Join} (J_1) is considerably higher than the operator $O_{Project}(\Pi_2)$ outgoing edge; using this edge weight difference can result in multi partitions of operators and therefore, a specific scheduling strategy can be applied for each partition. Furthermore, we can replicate the query operators and schedule them using the specific scheduling strategy to the provisioned resource to mitigate the performance violation of public Cloud resources.

Key contributions of this study can be summarized as follows:

- Formal definition of scheduling model for the deadline-constrained Big Data stream analytic query in public Cloud.
- Automated provisioning of the public Cloud resources required by a deadline-constrained streaming Big Data analytic query.
- Scheduling of the deadline-constrained streaming Big Data analytic query operators in public Cloud resources.
- Automated replication and scheduling of the streaming Big Data analytic query replicated operators to the provisioned public Cloud resources.
- Simulation and performance evaluation of BCframework.

The remainder of this paper is organized as follows. Section 2 reviews the related works on Big Data and Cloud-based stream processing schedulers. In Sect. 3, the formal definition of the scheduling model for the deadline-constrained Big Data stream analytic query in public Cloud is presented. Section 4 presents the conceptual architecture of BCframework. Furthermore, it focuses on the partitioning, provisioning, scheduling and replication algorithms of BCframework. Section 5 shows the experimental environment, parameters setup, and the experimental results of BCframework. Finally, conclusions and future works are given in Sect. 6.

2 Related works

This section presents two broad categories of related works: scheduling and provisioning in cloud-based distributed stream processing (DSP) open source platforms and the scheduling and provisioning for stream Big Data queries or tasks.

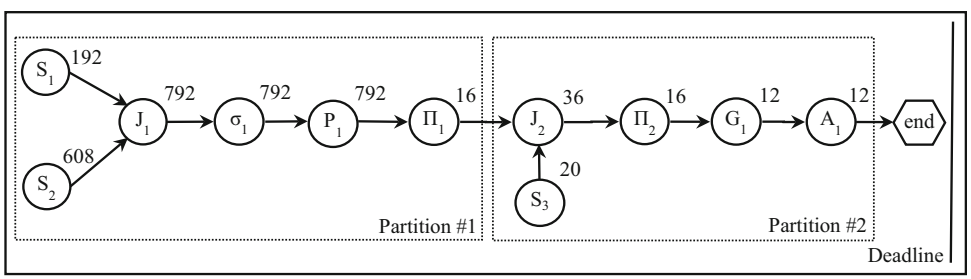

Fig. 1 Bq_{ij}, a sample soft deadline-constrained streaming Big Data analytic query instance

2.1 Cloud-based DSP scheduling

In this section, BCframework as a DSP scheduler is compared with the recent state-of-the-art studies in terms of the proposed task, resource and scheduling models.

Xu et al. designed and implemented a stream processing system developed based on Storm platform [7] whose traffic-aware scheduler assigns and reassigns accepted tasks at runtime [8]. Furthermore, the scheduler aims to minimize the inter-node and inter-process traffic loads and also monitors the overload of machines. In comparison with BCframework, this study is neither resource-aware nor task-aware scheduler because its task model only considers the tasks as a set of given threads and models resource as a set of given processes. Furthermore, its scheduling model only takes into account the inter-node traffic optimization.

In [9], Storm platform is extended to operate in geographically distributed and highly variable environments. Authors present a distributed scheduler which is QoS-aware and operates at runtime. It takes into account the latency, resource utilization and resource availability as QoS attributes. The scheduling model aims to solve two placement problems: assignment of the task operators to virtual machines and assignment of the virtual machines to the physical machines where in comparison with our work, BCframework only considers the operator to virtual machine assignment. Furthermore, its scheduler can be distributed among multiple worker nodes which BCframework is not capable of. In addition, no special strategies are proposed by the authors while BCframework has proposed efficient strategies for virtual machine assignment. And also, our task model models tasks as a set of the operators which is modelled as directed graph.

Li et al. proposed a scheduling framework which predicts the processing time of the application based on the structure of the task [10]. They also present a scheduling algorithm which assigns threads to machines based on the prediction result. In comparison with BCframework, its task model considers tasks as a set of threads and processes; its resource model considers resources as a set of machines, however their load and capacity is not considered. Furthermore, its only objective is to minimize the inter-operator traffic load of a task.

In [11, 12], the placement problem for DSP is studied and a scheduler for Storm platform is presented based on Integer Linear Programming. Similar to BCframework, they model tasks as a graph. Their resource model considers machines with respect to QoS metrics like resource availability. Furthermore, their scheduling objectives model the two QoS metrics for resource availability and response time. In addition, their solutions take into account the network delay while the capacity of the machines is not considered.

De Matteis et al. aim to provision resources needed for data stream applications in the presence of workload fluctuation [13]. They use a control-theoretic method to find an optimal stream application configuration to address latency constraints. In comparison with BCframework, they mainly focus on the resource provisioning problem where the focus of BCframework is on both resource provisioning and scheduling problems.

The Flink platform [14] includes a scheduling framework which is able to schedule a path of operators of an accepted task or multiple paths of multiple tasks in a machine. As the framework does not take into account the QoS requirements, its scheduler is not QoS-aware. In [14], a scheduler for Flink, S-Flink, is proposed using SDN controllers to place tasks to the underlying resources while considering the QoS requirements of the tasks. In comparison with BCframework, the resource model of S-Flink is defined based on software defined network infrastructures while BCframework's resource model relies on traditional network infrastructures. In [15], Heron, a new distributed system for managing stream processing application, is developed to overcome the limitations of Storm. In this system, Aurora [16] is used as a generic resource scheduling framework where the specific definition of scheduling and provisioning strategies is done by the users of Aurora while BCframework already includes predefined scheduling and provisioning strategies.

In [17] authors propose a cloud-based scalable and elastic stream processing engine which is able to process large data stream volumes using a novel parallelization technique. The engine aims to minimize the computational resources used while balancing the traffic load. In comparison with BCframework, they mainly focus on the query parallelization while the focus of BCframework is on the scheduling of queries which are already paralleled.

Li et al. proposed a task optimization and scheduling algorithms which their scheduling algorithm takes into account the structure and traffic load of task and resources characteristics [18]. The task model is a Directed Acyclic Graph (DAG) and the scheduling algorithm assigns operators to machines. The objective of algorithms is to minimize the inter-node network traffic. In comparison with BCframework, their task, resource and scheduling models have close assumption with BCframework but their algorithms are not specially designed for analytic queries and their solution has some limitations.

In summary, most optimization strategies for improving the cloud-based DSP scheduling and provisioning consider the task structure and inter-machine traffic or machine load aspects for their algorithms. However, our proposed algorithms could improve DSP scheduling and provisioning by

jointly considering the task structure, inter-machine network traffic and workload of machines while taking into account the public Cloud resources characteristics.

2.2 Streaming big data scheduling

In this section, BCframework as a streaming Big Data scheduler is compared with the recent state-of-the-art studies in terms of the proposed task, resource and scheduling models.

Sun et al. proposed a real-time scheduling framework which aims to meet the low response time and high energy efficiency of the query [19]. Similar to BCframework, it models tasks as graph which encompasses operators and their inputs and outputs. Furthermore, it models resources by considering resource capacity and the aim of its scheduling model is to minimize response time and maximize the energy efficiency of the machines. However, the variety dimension of streaming Big Data is not considered.

In [20], a fault tolerance framework is presented which allocates tasks to fault tolerant machines before the execution of tasks and also reassigns the tasks to the machines with lower response time. The task model considers task as graph and its resource model is similar to BCframework and models the capacity of the resources. Its scheduling model aims to minimize the response time and maximize the system reliability where in comparison with BCframework the 3Vs model of streaming Big Data is not considered completely.

Kaur et al. predict the data characteristics (5Vs) of streaming Big Data and are expressed in CoBA value which is used to determine the required machines of the tasks [21]. In comparison with BCframework, It just considers the streaming Big Data and there is no model for task and resources.

In [22], a stable scheduling is presented which allocates tasks to the stable machines before the execution of tasks and reassigns the task to the machines with lower response time. Similar to BCframework, it models tasks as graph which encompasses operators with their input and outputs and the capacity is included in the resource model. Its scheduling model aims to minimize the response time and maximize the stability of the machines. However, the 3Vs model of Big Data is not considered completely.

Kaur et al. estimated data characteristics of streaming Big Data based on the 4Vs model as a value named Characteristics of Data (CoD) [23]. Furthermore, Self-Organizing Maps (SOM) is used to allocate cloud resources to streaming Big Data using its estimated CoD. In comparison with BCframework, this work only considers the streaming Big Data, the streaming Big Data queries are not considered.

In [24], authors derived effective optimization rules which are used to reconfigure the structure of an accepted task at runtime using real-time performance statistics. In comparison with BCframework, the result of this work can be used as an optimized input query to BCframework.

Tolosana-Calasanz et al. developed a system architecture which accepts concurrent data streams and are able to regulate and control the received streams. Furthermore, their architecture provisions the required resources for processing the streams while maximizing the Cloud provider revenue [25]. In comparison with BCframework, their work does not take into account the query model and scheduling model for Big Data stream.

In [26], authors propose a static hybrid provisioning mechanism which prioritizes and manages the grouped VM requests to determine the required VM instances. In comparison with BCframework, their mechanism solely focuses on managing VM requests while there are no query, data or scheduling models for Big Data stream.

Zhang et al. propose a cloud scheduling model to overcome the speed and size difficulties of streaming Big Data analysis based on Markov chain prediction model [27]. In comparison with BCframework, the authors propose the data model for streaming Big Data analysis but the query model is not considered.

In [28] authors propose a VM allocation and placement strategy which consider the type of application and the capacity and dynamic load of the physical machines to assign them VM instances. In comparison with BCframework, there is no data or query model proposed in their work.

Baughman et al. propose a novel forecasting and predictive model which can be used for the analysis of Big Data streams to cover the velocity dimension of Big Data stream [29]. In comparison with BCframework, there is no data, query and scheduling model proposed in their research.

In [30] an elastic online scheduling framework is proposed which encompasses features such as online scheduling of Big Data stream applications, profiling the mathematical relationship between system response time, multiple application fairness and high-velocity continuous stream. In comparison with BCframework, its resource model does not take into account the analytical characteristics of the streaming Big Data queries.

In summary, the current optimization strategies for improving the scheduling and provisioning of streaming Big Data only consider some aspects of the Big Data characteristics. However, BCframework's scheduling and provisioning algorithms which not only considers all of the mentioned aspects but it is also suited for analytical queries of streaming Big Data.

3 Problem statement

In the context of stream Big Data computing, a set of deadline-constrained streaming Big Data analytic queries is hosted by a public Cloud provider and one or multiple streams of Big Data flow continuously through the instances of the accepted queries.

Efficient execution of each query (instance) is mainly achieved by the efficient provisioning and scheduling decisions made about public Cloud resources. To incorporate these factors into the provisioning and scheduling decisions, we bring some related definitions below using notations addressed in Table 1.

3.1 Data model

In BCframework, a data stream *DS* is a sequence of *ds* data sets, where each data set Ds_q is a set of *ql* number of t_{qr} tuple. A continuous data stream is an infinite sequence of data sets, and parallel continuous data streams are multiple streams that can be processed at the same time. Big data stream *Bs* is a continuous data stream with high velocity, high volume and a wide variety of data sets. We applied the structured, semi-structured and unstructured definitions of Big Data [1] to formally define *Bs* using Definition 1.

Definition 1 (*Big Data stream*) The Big Data stream *Bs* is a continuous stream. The volume of its data sets is defined by Bs^{Vo}, velocity of its data sets is defined by Bs^{Ve} and the variety of its data sets is defined by Bs^{Va}, where data type of each Bs^{Va} can be structured, semi-structured or unstructured. In addition, let *Bf*, Big Data flow, be defined as edge flows *Bs*.

3.2 Query model

In BCframework, a deadline-constrained Big Data stream analytic query can be defined as an aperiodic continuous query with a set of operators which analyzes one or multiple input Big Data streams $Bs(s)$ and produces most of its output Big Data streams $Bs(s)$ within associated soft deadline. We formally define the query using Definition 2.

Definition 2 (*Deadline-constrained Big Data stream analytic query*) Bq_{ij} can be expressed as a direct acyclic graph and characterized by four-tuple $Bq_{ij} = \left(Bq_i^O, Bq_i^E, Bq_{ij}^{DL}, Bq_i^W\right)$. The operator O_t of Bq_i is characterized by the triplet $O_t = \left(O_t^{Inp}, O_t^{out}, O_t^{tpe}\right)$. The in-degree of operator O_t is the number of incoming edges, and the out-degree of operator O_t is the number of the outgoing edges. The source operator is the operator whose in-degree is zero, and the end operator is the operator whose out-degree is zero. A Bq_i has at least one source and one end operator.

Figure 1 shows the *j*-th execution instance of the deadline-constrained Big Data stream analytic query Bq_i whose Bq_i^O includes 12 operators and a soft deadline Bq_{ij}^{DL} is associated to Bq_{ij}. Source operators are O_{S1}, O_{S2},O_{S3} and the end operator is O_{end}. The operator O_{J1} has two edges E_{S1J1} and E_{S2J1} where E_{S1J1} flows Big Data flow *Bf* as input Big Data stream *Bs*, O_{J1}^{Inp}, and E_{S2J1} flows another O_{J1}^{Inp}. The volume of Bs, Bs^{Vo}, of the E_{S1J1} is estimated to be 192 bytes while its recently aggregated tuples Bq_i^W are not considered. As the O_{J1} is defined by user, its type O_{J1}^{tpe} is considered to be an original operator and it is not replicated by BCframework. Bq_{ij} is partitioned into two partitions *Partition*#1 and *Partition*#2 based on the output *Bf* of an operator O_t^{out}. This enables BCframework to apply different provisioning and scheduling strategies for each partition.

3.3 Scheduling model

In BCframework, the scheduler accepts a set of independent queries, provisions their required public Cloud resources and schedules all operators of the accepted queries on the provisioned resources to achieve the lowest average query latency and Cloud resource utilization cost with subject to the soft deadline constraint associated with the query instances. The following model reflects these factors.

Let *BQ* denote a set of independent *bq* queries accepted by the scheduler $BQ = \left\{Bq_1, \ldots Bq_i, \ldots, Bq_{bq}\right\}$, Furthermore, let *VM* denote a set of *vm* virtual machines $VM = \{Vm_1, \ldots, Vm_k, \ldots, Vm_{vm}\}$. Each Vm_k is charged per amount of time by public Cloud provider. In addition, for the execution of the query, there is no limit imposed on the number, type and provisioning time of Vm_k.

The runtime matrix $Runtime^j_{(bq \times on) \times vm}$ indicates the estimated runtime of the *j*-th instance of all accepted queries operators $Runtime^j_{(bq \times on) \times vm} = \begin{pmatrix} R^j_{11} & R^j_{bq \times on \times 1} \\ R^j_{1vm} & R^j_{bq \times on \times vm} \end{pmatrix}$, where R^j_{tk} specifies the estimated runtime of the *j*-th instance of the operator O_t on a VM type Vm_k.

The schedule matrix $Schedule^j_{(bq \times on) \times vm}$ represents the assignment of the *j*-th instance of all operators of all accepted queries to the virtual machines $Schedule^j_{(bq \times on) \times vm} = \begin{pmatrix} S^j_{11} & S^j_{bq \times on \times 1} \\ S^j_{1vm} & S^j_{bq \times on \times vm} \end{pmatrix}$, where S^j_{tk} shows that the *j*-th instance of operator O_t is assigned to the

Table 1 The notation used in the problem statement

Notion	Definition
Data model	
DS	Data stream, $DS = \{Ds_1, \ldots, Ds_q, \ldots, Ds_{ds}\}$
Ds_q	Data set, $Ds_q = \{t_{q1}, \ldots, t_{qr}, \ldots, t_{ql}\}$
t_{qr}	r-th Tuple of Ds_q
Bs	Big data stream
Bs^{Vo}	Volume of Bs,$Bs^{Vo} = \left\{Ds_1^{Vo}, \ldots, Ds_q^{Vo}, \ldots, Ds_{ds}^{Vo}\right\}$
Bs^{Ve}	Velocity of $Bs, Bs^{Ve} = \left\{Ds_1^{Ve}, \ldots, Ds_q^{Ve}, \ldots, Ds_{ds}^{Ve}\right\}$
Bs^{Va}	Variety of $Bs, Bs^{Va} = \left\{Ds_1^{Va}, \ldots, Ds_q^{Va}, \ldots, Ds_{ds}^{Va}\right\}$
Ds_q^{Vo}	Volume of data set Ds_q (measured in byte)
Ds_q^{Ve}	Velocity of data set Ds_q (measured in byte/second)
Ds_q^{Va}	Variety of data set Ds_q
Bf	Big Data flow
Query model	
Bq_i	Deadline-constrained Big Data stream analytic query i
Bq_{ij}	Instance j of a deadline-constrained Big Data stream analytic query i, $Bq_{ij} = \left(Bq_i^O, Bq_i^E, Bq_{ij}^{DL}, Bq_i^W\right)$
Bq_i^O	Finite set of on number of Bq_i operators $O = \{O_1, \ldots, O_t, \ldots, O_{on}\}$
Bq_i^E	Finite set of Bq_i directed edges $E = \{E_{11}, \ldots, E_{pq}, \ldots E_{onon}\}$
Bq_{ij}^{DL}	Soft deadline associated to the $j-th$ instance of query Bq_i
Bq_i^W	Window size of query Bq_i
E_{pq}	Execution precedence between two operators p and q which is member of Bf
O_t	Query Operator, $O_t = \left(O_t^{Inp}, O_t^{out}, O_t^{tpe}\right)$
O_t^{Inp}	Set of input Big Data flows $Bf(s)$ to operator O_t
O_t^{out}	Set of output Big Data flow(s) $Bf(s)$ from operator O_t
O_t^{tpe}	Type of O_t as an original or replicated operator
Scheduling model	
BQ	Set of independent bq queries accepted by the scheduler $BQ = \{Bq_1, \ldots Bq_i, \ldots, Bq_{bq}\}$
VM	Set of vm virtual machines, $VM = \{Vm_1, \ldots, Vm_k, \ldots, Vm_{vm}\}$
Vm_k	Virtual machine k, $Vm_k = \left(Vm_k^{Cu}, Vm_k^{Rm}, Vm_k^{Ct}\right)$
Vm_k^{Cu}	CPU capacity of Vm_k
Vm_k^{Rm}	Memory capacity of Vm_k
Vm_k^{Ct}	Utilization cost of Vm_k

virtual machine Vm_k. To meet the soft deadline, we should meet the constraint defined by Eq. (1), so that the assignment of j-th instance of the end operator S_{endk}^j to the any Vm_k would not exceed the deadline at which operators can be executed.

$$S_{endk}^j \leq Bq_{ij}^{DL} \quad \forall k \in \{1 \ldots vm\} \tag{1}$$

In BCframework, the schedule model used by its scheduler is defined by Definition 3.

Definition 3 (*scheduling model Sm*) Sm is the scheduling model used in BCframework and is represented by four-tuple $Sm = \{BQ, VM, CO, OB\}$, where $BQ = \{Bq_1, \ldots Bq_i, \ldots, Bq_{bq}\}$ is the set of bq accepted queries and VM is the set of provisioned virtual machines as the underlying public Cloud resources of BCframework. CO is the constraint defined by Eq. (1) and OB is the objective function of Sm which is defined to minimize the average tuple latency and the Cloud resource utilization cost of each Bq_{ij} instance of the accepted queries BQ according to Eq. (2)

$$OB(Bq_{ij}) = \text{Min}\ \big(\text{Avg}(\text{Latency}(Bq_{ij})) \ and \text{Avg}(\text{Cost}(Bq_{ij}))\big), \quad (2)$$

where $\text{Latency}(Bq_{ij})$ represents the tuple latency of Bq_{ij} and it is defined according to Eq. (3)

$$\text{Latency}(Bq_{ij}) = \text{Time}(Bq_{ij} \cdot O^{Out}_{end}) - \text{Time}(Bq_{ij} \cdot O^{Inp}_{source}), \quad (3)$$

where $\text{Time}(Bq_{ij}.O^{Out}_{end})$ is the timestamp of the earliest Bs of the output Bf in $Bq'_{ij}s$ end operator and $\text{Time}(Bq_{ij}.O^{Inp}_{source})$ is the timestamp of the earliest Bs of the $Bq'_{ij}s$ source operators. Furthermore, $\text{Cost}(Bq_{ij})$ is defined by the Amazon AWS EC2 pricing model [31].

4 BCframework

The goal of BCframework is to minimize the average tuple latency and public Cloud resource utilization cost of the accepted queries and increases the likelihood of completing the execution of the accepted queries (instances) before or on their deadlines. To reach this goal, two related sub problems have to be solved: *provisioning* and *scheduling*. In BCframework, the provisioning problem consists of the determination on number and type of VM to be used for the query instance Bq_{ij} execution and the scheduling problem includes the determination of the placement and ordering of the Bq_{ij} operator instances on the provisioned VMs.

The BCframework's idea for solving the problems is to use more than one provisioning and scheduling strategy for the query Bq_i. The rationale of this idea is related to an attribute of analytical queries where the output size of analytic query end operators is usually smaller than the input size of its source operators [32]. BCframework makes use of this characteristic to realize its idea by profiling the output size of Bq_i operators and partitioning the operators using the profiled output size.

To support the idea, the objectives of BCframework provisioning and scheduling algorithms are established based on the characteristics of the identified partitions. For the first partition, the objective of the algorithms is to minimize the average tuple latency of this partition operators where the objective of algorithms for the second partition is to minimize the public Cloud resource utilization cost of this partition operators. The algorithms apply their own provisioning and scheduling strategies to achieve the objectives of each partition.

To mitigate the delay affected by the poor performance of public Cloud resources, BCframework replicates a set of selected Bq_{ij} operators on the idle slots of the provisioned VMs. In addition, BCframework reacts to missed deadlines of previous executed query instances and incrementally provisions additional *Vm* instances to increase the deadline meeting probability.

Furthermore, BCframework adapts the algorithm presented in [33] which proposed a deadline and cost aware algorithms for provisioning and scheduling using critical path idea for applications which are developed using workflow paradigm and deployed on cloud environment. Our algorithms extend the critical path idea to develop deadline and cost aware scheduling and provisioning algorithms for Big Data stream analytic queries which are developed based on the dataflow paradigm. In addition, our algorithms take into account the performance degradation of public Cloud resources.

4.1 BCframework algorithms

This section presents the detail of the algorithms used in the BCframework. First, it is described the *Bpartitioner* algorithm, which is used to partition the selected query Bq_i. Bq_i is selected among the accepted queries based on the EDF strategy [2] where prioritizes earliest deadline query first. Then the *partial critical path* (PCP) algorithm is described, which is applied to determine the critical paths of Bq_i. Subsequently the *Oscheduler* and *Rscheduler* algorithms are described that are used for the provisioning and scheduling of Bq_i original and replicated operator respectively. Finally, Deadline Controller algorithm to decrease the deadline violation of the query instances is explained.

Figure 2 demonstrates the architecture of BCframework. *Bpartitioner* module takes two parameters *Vm Capacity* determined by the *monitor* module and Query Bq_i accepted by BCframework to implement the Bpartitioner algorithm and to annotate the partitioned query Bq'_i. *PCP* module uses *Load* and *Traffic* of *Vm* received form monitor module to implement the Partial Critical Path algorithm and determines the *PCP* of Bq'_i. *Oscheduler* and *Rscheduler* modules implement the Oscheduler and Rscheduler algorithms respectively where Oscheduler uses *PCP* to generate schedule *Schedule* and Rscheduler uses *Schedule* to generate schedule with replicated operators *Schedule*$'$. Deadline Controller module receives the execution status of Bq_{ij} and implements the Deadline Controller algorithm to provision additional *Vm* instances.

4.1.1 Query partitioning algorithm

The aim of partitioning algorithm is to partition the operators of Bq_i in way that the different scheduling strategies can be applied for each partition. This algorithm uses the defined *outBfsize* metric to partition Bq_i into at most two

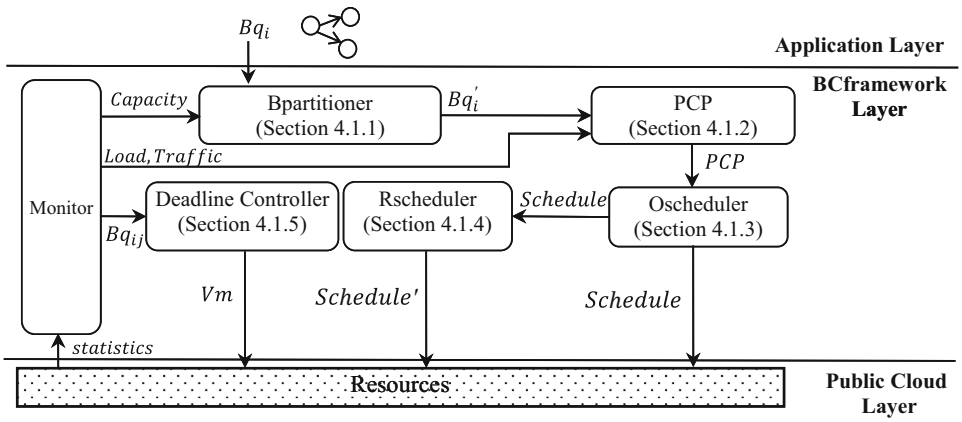

Fig. 2 The conceptual architecture of BCframework

distinct operator partitions named Bq_i^{p1}, Bq_i^{p2}. The algorithm examines the query structure to estimate the value of a *Bf* metric named *outBfsize,* which indicates the estimated output *Bf* size of an operator. As a result, the partition Bq_i^{p1} contains operators whose average size of *outBfsize* is greater than that *outBfsize* for the operators of partition Bq_i^{p2}. The value of *outBfsize* metric is estimated according to Eq. (4).

$$outBfsize(O_t) = \sum_{p=0}^{out-degree(O_t)} \sum_{q=1}^{q=ds} Size\left(O_t^{out_p} \cdot \mathrm{Ds}_q^{Vo}\right) \quad (4)$$

In Eq. (4) $\sum_{q=1}^{q=ds} Size\left(O_t^{out_p}.\mathrm{Ds}_q^{\mathrm{Vo}}\right)$ denotes the sum of the size of the recent data sets of the $p-th$ output *Bf* of O_t. The recent data sets are a set of t_{ql} tuples of the data set Ds_i and a set of ds number of data sets DS. The ds is determined by the query window size Bq_i^W and Bs^{Vo} includes DS. To estimate the value of Ds_k^{Vo} initially, Bq_i is executed by setting the number of the input *Bf* tuples of Bq_i to 1 and deploying on a provisioned *Vm* instance. Algorithm 1 describes how this metric is used to partition the Bq_i operators.

The algorithm operates in two main steps:

- Lines 1–4: Calculates the outBfsize metric of the source and end operators of Bq_i. As there may be more than one source operator, the average value of outBfsize is calculated.
- Lines 5–12: Calculates the outBfsize value of all Bq_i operators to determine the corresponding partition of each operator. The algorithm computes the distance of the outBfsize of the *operator* with both average outBfsize of the source operators and outBfsize of the end operator. The partition Bq_i^{p1} includes the operators of Bq_i which has the minimum distance with the source operators. Whereas, the partition Bq_i^{p2} encompasses the Bq_i operators which has the minimum distance with the end operator.

Algorithm 1: Bq_i partitioning

Input: Bq_i, query

Output: Bq'_i, partitioned query

1. $sourceoperators \leftarrow Bq_i.sources$
2. $endoperator \leftarrow Bq_i.end$
3. $sdf \leftarrow$ Avg(OutBfsize ($sourceoperators^{Inp}$))
4. $ddf \leftarrow$ OutBfsize ($endoperator^{Out}$)
5. **for** ($operator$: TopologicalOrder(Bq_i))
6. $sdistance \leftarrow |\text{OutBfsize } (operator^{Inp}) - sdf|$
7. $ddistance \leftarrow |\text{OutBfsize } (operator^{Inp}) - ddf|$
8. **if** ($sdistance \leq ddistance$)
9. $Bq'_i.operator^p = 'p1'$
10. **if** ($sdistance > ddistance$)
11. $Bq'_i.operator^p = 'p2'$
12. **end for**
13. $return\ Bq'_i$

Algorithm 1. Bpartitioner, Bq_iQuery partitioning algorithm

4.1.2 Partial critical path algorithm

A partial critical path (PCP) of query instance Bq_{ij} is a longest execution path ended to Bq_{ij} end operator and started from the one of the Bq_{ij} operators where the deadline of Bq_{ij} is not missed [33]. BCframework efficiently provisions and schedules each PCP operators to increase the likelihood of Bq_{ij} deadline meeting. Each PCP is calculated based on *earliest start time* (est), *latest finish time* (lft) and the minimal runtime of each Bq_{ij} operator instance demonstrated in Bq_{ij} runtime matrix.

The $est(o_{it})$ of an operator o_{it}, represents the earliest time that operator o_{it} can start its execution and it occurs when all parents of such operator finish their executions as early as possible. The *lft* of an operator o_{it}, $lft(o_{it})$, indicates the latest time an operator o_{it} can finish its execution without violating the deadline of Bq_{ij} and it happens when all children of o_{it} are executed as late as possible. The $est(o_{it})$ and $lft(o_{it})$ are formally defined by Eqs. (5) and (6) respectively,

Algorithm 2: Partial critical path determination
Input: Bq'_{ij}, partitioned query instance
Output: $Bq'_{ij}.pcps$, partial critical paths of the partitioned query instance

$pcp \leftarrow \emptyset$; $parent \leftarrow -1$; $Bp \leftarrow \emptyset$
$toassign \leftarrow Bq_{ij}.end$
$check \leftarrow Bq_{ij}.end$
$parents \leftarrow$ Unassignedparents$(Bq_{ij}.end)$
while $(Top(toassign)! = Bq_{ij}.end)$
  while $(parents\ ! = null)$
  for $(p: parents)$
    if $(p^{\text{est}} + p^{\text{runtime}^{\min}} + E^{\text{Trf}}_{pcheck} > parent)$
      $parent \leftarrow p^{\text{est}} + p^{\text{runtime}^{\min}} + E^{\text{Trf}}_{pcheck}$
      $Bp \leftarrow p$
    end if
  end for
  $pcp \leftarrow Bp \cup pcp$
  $toassign \leftarrow parents - Bp$
  $check \leftarrow$ Top(pcp)
  $parents \leftarrow$ Unassignedparents$(check)$
end while
$Bq_{ij}.pcps += pcp$
$pcp \leftarrow \emptyset$
$check \leftarrow$ Top$(toassign)$
Remove Top$(toassign)$ from the $toassign$
$pcp \leftarrow check$
$parents \leftarrow$ Unassignedparents$(check)$
end while
return $Bq_{ij}.pcps$

Algorithm 2. Partial Critical Path algorithm

$$est(O_{it}) = \begin{cases} 0, & fO_{it} = \mathrm{O}_{source} \\ max_{t_a \in parents(O_{it})}\left(est(t_a) + Min\left(R^{j}_{t_a k}\right) + E^{Trf}_{t_a O_{it}}\right), & Otherwise \end{cases} \quad (5)$$

$$lft(O_{it}) = \begin{cases} Bq^{DL}_{ij}, & IfO_{it} = \mathrm{O}_{end} \\ \max_{t_s \in childs(O_{it})}\left(lft(t_s) - Min\left(R^{j}_{t_s k}\right) - E^{Trf}_{O_{it} t_s}\right), & Otherwise \end{cases} \quad (6)$$

As shown in Algorithm 2, the PCP initially includes one of the parents of O_{end} or, if it was already assigned, the operator with latest *lft* that has not been yet assigned to a PCP. The next element of PCP is the parent of the included operator with latest finish time which is calculated based on the earliest start time of the parent operator p^{est}, the minimum runtime of the parent operator $p^{\text{runtime}^{\min}}$ and the network traffic delay between the parent operator p and candidate parent operator *check*, E^{Trf}_{pcheck}. The process is iterated for each parent of O_{end}.

4.1.3 Provisioning and scheduling for original operator

Oscheduler algorithm aims to decrease the number of provisioned public Cloud virtual machines and assign them to the original operator instances of the accepted query instance Bq_{ij} to decrease the average tuple latency, the public Cloud resource utilization cost and the deadline misses of Bq_{ij}.

Algorithm3: Oscheduler: Bq_i original operator instances scheduling
Input: $Bq'_{ij}.pcps$, partial critical paths of the partitioned query instance
Output: $Schedule$, Schedule

$p1 \leftarrow 0,\ p2 \leftarrow 0$ // for counting the virtual slots of the first and second partitions respectively.
for $(pcp: Bq_{ij}.pcps)$
  if $(pcp^{\mathrm{p}} = p1)$
    $vslot =$ Vmtype (Fastest)
  else
    $vslot =$ Vmtype (Cheapest)
  end if
  $vslot^{\mathrm{st}} = \mathrm{Start}(vslot)$
  $pcp^{\mathrm{st}} = \mathrm{Max}(pcp^{\mathrm{est}}, vslot^{\mathrm{st}})$
  $schedule[\mathrm{Vm}\ (vslot), pcp.operators] \leftarrow 1$
  $pcp^{\mathrm{lag}} = \mathrm{Lag}(pcp)$
  $pcp^{\mathrm{Runtime}} = \mathrm{Runtime}(pcp)$
  if $(\mathrm{pcp}^{\mathrm{p}} = p1)$
    $vslot^{\mathrm{p1}}_{\mathrm{p1++}} = vslot$
    Determine freeslots and add them to fvslo
  else
    $vslot^{\mathrm{p2}}_{\mathrm{p2++}} = vslot$
    Determine freeslots and add them to fvslo
  end if
end for

Algorithm 3. Oscheduler algorithm

To provision public Cloud *VMs*, the algorithm applies two provisioning strategies for the original operator instances of Bq_{ij} identified partitions. For the first partition, the algorithm provisions the fastest *VM* type of the underling public Cloud based on the highest Vm^{Cu}_{k} and Vm^{Rm}_{k} to improve the processing speed of the first partition operators. This strategy reduces the average tuple latency of these operators and provides the input *Bf* of their succeeding operators faster. For the second partition, the algorithm provisions the cheapest *VM* type of the underling

public Cloud based on the lowest Vm_k^{Ct} to reduce the cost of the Cloud *VM* instances utilization (Lines 1–7).

To assign Bq_{ij} original operator instances, the algorithm assigns all operators of each PCP in each partition to the corresponding provisioned *Vm* instance. This strategy can reduce the inter-operator traffic load of the operators and lead to reduce the average tuple latency of Bq_{ij}. Furthermore, the algorithm adjusts the start time of the first operator of each assigned PCP to the maximum between the assigned operator *est* and assigned slot *starttime*. This adjustment can decrease the likelihood of Bq_{ij} deadline violations (Lines 8–10).

In addition, the algorithm determines the initial unassigned virtual machine slots, free *Vm* slots, which will be used by Rscheduler to replicate the operator instances. Therefore, the algorithm considers the runtime of the assigned PCPs to calculate the available free *Vm* slots. Furthermore, the free *Vm* slots are partitioned into two partitions by their VM types. More precisely, the free *Vm* slots of the first partition include unassigned fastest virtual slots and the second partition free *Vm* slots encompass unassigned cheapest virtual slots (Lines 13–19).

4.1.4 Provisioning and scheduling for replicated operator

Rscheduler algorithm aims to mitigate the performance variation of public Cloud virtual machines through the efficient replication of Bq_{ij} operator instances. To approach the goal, the algorithm efficiently determines the candidate Bq_{ij} operators which can be replicated, $O_t^{tpe=replicated}$, and assigned to the efficient candidate free *Vm* slots.

To efficiently determine the candidate operators for replication, the algorithm takes into account the structure and lag time of the Bq_{ij} operators to determine the operators whose position in the query are early and their lag time is minimum. As delay in the early operators execution causes delays of the bigger number of succeeding operators, prioritizing the early operators for replication may reduce the average tuple latency of Bq_{ij}. Furthermore, prioritizing the minimal lag time operators can decrease the likelihood of Bq_{ij} deadline misses.

The candidate free *Vm* slots are determined by the following conditions (Lines 4–13).

- The candidate slot does not violate the earliest start time of PCP.
- PCP can complete its execution before the end of the candidate slot.
- The candidate slot is available after the precedent operators of PCP and before the succeeding operators of PCP.
- The *Vm* of the candidate slot does not host the corresponding original operators of PCP.

Algorithm 4: Rscheduler: Bq_i replicated operator instances scheduling
Input: Bq_{ij}, query instance
Output: $Schedule'$, Schedule

1. $tpcp \leftarrow \emptyset$
2. $Bq_{ij}.pcps = \mathrm{Sort}(Bq_{ij}.pcps, \mathrm{Lag}, \mathrm{Asc})$
3. $fvslot = \mathrm{Sort}(fvslot, \mathrm{size}, \mathrm{Desc})$
4. **for** $(pcp: Bq_{ij}.pcps)$
5. **if** $\left(\mathrm{Min}\left(\bigcup_{i1=1}^{|Bq_{ij}.pcps|} Bq_{ij}.pcp_{i1}^{\mathrm{runtime}}\right) > \mathrm{Max}(\sum_{k1=1}^{|fvslot|} fvslot_{k1})\right)$ break
6. **for** $(slot: fvslot)$
7. **if** $(pcp^{\mathrm{p=p1}}$ **and** $slot^{\mathrm{p\neq p2}})$
8. **if** $(slot^{\mathrm{st}} + pcp^{\mathrm{runtime}} < pcp^{\mathrm{lft}})$
9. **if** $(VM(slot).operators \cap pcp.operators == \mathrm{null})$
10. **if** $\left(slot^{\mathrm{st}} \geq \left(\mathrm{prec}(pcp)\right)^{\mathrm{ft}}$ **and** $slot^{\mathrm{ft}} \leq \left(\mathrm{succ}(pcp)\right)^{\mathrm{st}}\right)$
11. **if** $(slot^{\mathrm{time}} \geq pcp^{\mathrm{runtime}}$ **and** $pcp^{\mathrm{est}} \geq slot^{\mathrm{st}}))$
12. $candidateslots += slot$
13. **end for**
14. **if** $(candidateslots == \mathrm{null}$ and $pcp! = \mathrm{null})$
15. $\mathrm{Bq}_{ij}.pcps -= pcp$
16. $pcp -= \mathrm{Last}(pcp.operators)$
17. $\mathrm{Bq}_{ij}.pcps += pcp$
18. $tpcp += \mathrm{Last}(pcp.operators)$
19. $\mathrm{Bq}_{ij}.pcps += tpcp$
20. **else if** $(candidateslots! = \mathrm{null})$
21. $Bslot = \mathrm{Min}(\bigcup_{k2=1}^{j} |candidateslots_{k2}^{start} - pcp^{start}|)$
22. $schedule'[Vm\,(Bslot), pcp.operators] \leftarrow 1$
23. $Bslot -= pcp^{\mathrm{runtime}}$
24. $\mathrm{Bq}_{ij}.pcps' += pcp$
25. **if** $(Bslot \neq \mathrm{null})$
26. *Add Bslot into fvslot*
27. **else**
28. *Remove Bslot from fvslot*
29. **end if**
30. **end if**
31. **end for**

Algorithm 4. Rscheduler algorithm

The algorithm efficiently selects the candidate free *Vm* slots based on the strategy which reserves the fastest candidate slots for the replicated operators of first partition and reserves the largest candidate slots for the replicated

operators of both partitions. Using this strategy can improve the processing speed of the operators of PCP in the first partition and provide the largest remained free *Vm* slots, not assigned slots with maximum free slots, which may be assigned to the most operators of an unassigned PCP.

The efficient assignment of the determined free *Vm* slots to the candidate operators is achieved through obeying a defined rule which state that the PCP whose operators are the member of the first partition of Bq_{ij}, first partition PCP, must be assigned to *Vm* slots which are fastest. For a second partition PCP, it can be assigned to both first or second free *Vm* slots. In a situation where it is possible to assign the first partition free *Vm* slots to both first and second partition PCPs, the priority of the first partition PCP is higher (Lines 20–24). This rule aims to minimize the delay of the PCP start time.

The assignment may create new smaller free slots when the assigned PCP size is smaller than the assigned free slot. The algorithm adds these new slots to the list of the available free slots (Lines 24–29). Furthermore, for a PCP which there is no free *Vm* slots, the algorithm removes at least operators of the PCP to find a free slot to fit the modified PCP (Lines 14–19). In addition, the algorithm inserts the new created PCP into the list of accepted PCPs.

4.1.5 Deadline controller

The aim of the *deadline controller* algorithm is to reduce the likelihood of missing the Bq_{ij} deadline, Bq_{ij}^{DL}, through incrementally provisioning *Vm* instances required for Bq_{ij}. To reach the goal, the algorithm evaluates the missed deadline ratio of the previously executed query instances to provision an appropriate virtual machine for the next execution instance of Bq_{ij}.

The algorithm calculates the ratio of the proportional deadline miss using the Eq. (7) [2] which prioritizes deadline missed query instances whose missed deadline tuples of their window are high.

Using the *PDMR* value of Bq_i, the algorithm provisions the appropriate new type of *Vm* when the *PDMR* value is greater than the predefined threshold. To determine the appropriate *Vm* type, the algorithm examines the PCP of the last instance of Bq_i whose deadline violation is maximum to determine its *Vm* type. The last instance of Bq_i and maximum deadline violations are determined by First function and descending (Desc) parameter in Sort function respectively. This strategy can lead to provide the new free *Vm* slots which will be used by Rscheduler to schedule the PCPs like the identified PCP to the new added virtual slots.

Algorithm5: Deadline controller
Input: Bq_{ij}, query instance
Output: Vm, virtual machine

1. $pdmr \leftarrow PDMR(\mathrm{Bq}_{ij})$
2. **if** $(pdmr > threshold)$
3. **for** $(pcp' : Bq_i.pcps')$
4. $pcp'^{violation} = pcp'^{finishtime-lft}$
5. **end for**
6. $Bq_{ij}.pcps' = \mathrm{Sort}(Bq_{ij}.pcps', \mathrm{violation}, \mathrm{Desc})$
7. $type = \mathrm{Vmtype}\ (\mathrm{First}(Bq_{ij}.pcps'))$
8. $vslot = \mathrm{Vmtype}\ (\mathrm{type})$

Algorithm 5. Deadline controller algorithm

$$PDMR(Bq_{ij}) = \frac{\sum_{j1=1}^{j1=|missedinstnces|}\left(Bq_{ij1}^{w} - \sum_{q=1}^{q=windowsize}\sum_{r=1}^{r=|Ds_p|}!deadlinemiss(Ds_{qr})\right)\Big/ Bq_i^w}{\sum_{j=1}^{j=|executedinstnces|} Bq_{ij}} \tag{7}$$

Figure 3 shows the simplified schedule *Schedule′* for instance *j* of Bq_{ij} where its deadline is twenty. As shown in Fig. 3a, there are four PCPs with a simplified estimated runtime which are identified based on the output *Bf* size of the operators and partitioned into two partitions P1 and P2. As shown in Fig. 3b, two *Vm* types VmF, fastest *Vm*, and VmC, cheapest *Vm*, and four *Vm* instances are provisioned

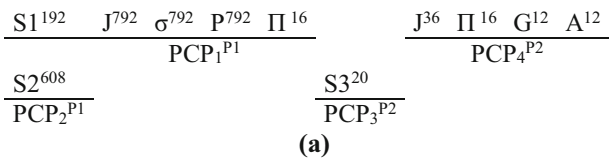

Fig. 3 Scheduling and provisioning of Bq_{ij}. **a** Instance *j* of Bq_i' and its PCPs. **b** *Schedule′* of Bq_{ij}

and their possible free *Vm* slots are used to replicate the operators of three PCPs including PCP1'P1, PCP3'P2,PCP4'P2.

4.2 Complexity analysis

The main cost of the partitioning algorithm comes from the sorting and traversing Bq_i operators. Therefore the time complexity of the algorithm is $O(on \log on + on)$, where on represents the number of Bq_i operators. In partial critical path algorithm, the computation load is primarily made up of traversing Bq_{ij} operators. As a result the time complexity of the algorithm for each query instance is $O(on)$.

For Oscheduler, the main step is to provision the *Vm* s for all available PCPs. So, the time complexity of the examining PCPS is $O(|PCPs|)$. Rscheduler sorts both PCPs and free *Vm* slots and examines all virtual slots to determine the possible and efficient free *Vm* slots for a PCP. Therefore, the time complexity of this algorithm is $O(|\text{PCPs}| \log|\text{PCPs}| + |fvslot| \log|fvslot| + |\text{PCPs}| \times |fvslot|)$. In addition, as the deadline controller algorithm evaluates and sorts the replicated PCPs, the time complexity of this algorithm is $O(|\text{PCP's}| \log|\text{PCP's}| + |\text{PCP's}|)$.

5 Evaluation

This section describes the experiments that have been performed to validate and analyze the performance of BCframework. First, the query partitioning algorithm of BCframework is validated by evaluating *outBfsize* of identified partitions during execution of the benchmark and real-world queries. Then, BCframework performance is assessed by analyzing the average tuple latency, the utilization cost, the number of deadline misses and the number of provisioned *Vm* instances for the benchmark and real-world queries in presence of different *Bs* fluctuation scenarios.

The performance of BCframework is compared with Storm default scheduler under simple and complex queries because it is one of the most popular Big Data stream computing platforms both in academia and industry [30]. Furthermore, the performance of the BCframework is also compared under real-world queries with a state-of-the-art scheduler [30] because of its similarity. Similar to the problem statement of BCframework, the state-of-the-art scheduler addresses the problem of provisioning and scheduling Big Data stream applications and its framework is based on a cost and deadline aware scheduler [34].

BCframework is evaluated on the simulated Storm platform [7] and Amazon EC2 [35] public Cloud while not limited to these infrastructures. As the data and query models of BCframework are developed based on Direct Acyclic Graph (DAG), BCframework is compatible to other DAG-based distributed stream processing platforms like Apache Spark Streaming [36] and schedule model of BCframework can be deployed on these systems. BCframework is also portable to other cloud providers as there is no assumption about the specification of a particular Cloud provider except that the Cloud provider pricing model should be imported to BCframework.

5.1 Queries

The benchmark queries used in the experiments have been applied from the BigBench [37] and Linear Road Benchmark [38] benchmarks. In addition, two real-world queries WordCount [39] and Top_N [40] are also used in the experiments.

BigBench includes the complex queries which cover the declarative and procedural and also the structured, semi-structured and unstructured aspects of the Big Data analytic queries. The characteristics of the BigBench queries are described in Table 2.

The volume of BigBench queries is supported by a parallel data generator which is able to generate data in scale, the velocity of BigBench queries is covered with data clicks and sensor data at an increasing rate and the variety of BigBench is supported by the structural relational tables, semi-structured key-value web-clicks and unstructured social data text. The workload of BigBench is covered with 30 queries which covers five major area of Big Data analytics. Each structured, semi-structured and unstructured data types and their combination are covered in the queries [42].

As BigBench queries include SQL-like and the user-defined operators and also, the structured, semi-structured and the unstructured data types, it is chosen to evaluate BCframework as a scheduler for Big Data scenarios. The

Table 2 The characteristics of BigBench queries [37]

Query type	Queries	Data types	Queries
Declarative	6,7,9,13,14,16,17, 19,21,22, 23,24	Structured	1,6,7,9,13, 14,15,16,17,19,20,21, 22,23,24, 25,26,29
Mixed	1,4,5,8,11,12,15,18,20,25,26,29,30	Semi-structured	2,3,4,5,8, 12,30
Procedural	2,3,10,27,28	Unstructured	10,11,18, 27,28

fluctuating scenarios of our experiments are conducted by controlling the data generation rate of BigBench. Furthermore, as BigBench is targeted Cloud online services [46] we used it to evaluate the diversity of analytical queries in public Cloud.

Linear Road Benchmark queries encompass seventeen simple stream queries whose operators are SQL-like and their data types are structured. As Linear Road Benchmark has been used as benchmark for the evaluation of the distributed stream management systems, it is chosen to evaluate BCframework as a scheduling framework on distributed stream processing systems.

Real-world query, WordCount counts the number of words in a sentence and TOP_N, other real-world query, does a continuous computation of the top N words that the query has seen in terms of cardinality. These queries are used to evaluate BCframework performance in comparison with the state-of-the-art scheduler.

5.2 Environment

Table 3 describes the simulation test bed used in the experiments to run the queries. The test bed consists of a data center containing 20 hosts. Each host has 256 GB of RAM and 8 cores. Based on the pricing model of Amazon AWS EC2, there is no a utilization cost for the inter virtual machine network traffic but the utilization cost of the incoming and outgoing traffic loads for the zone of the leased virtual machines is calculated.

The our simulation conducted using CEPSim [43] which is an extension to CloudSim [44] and is used for the simulation of Complex Event Processing and Stream processing systems on different deployment models including private, public, hybrid and multi-clouds. It can be used to analyze the performance and scalability of user-defined queries and to evaluate the effects of various query processing strategies. CEPSim can simulate the systems in large Big Data scenarios with accuracy and precision [41].

To implement the queries, a software stack is used which includes Windows 7 64-bit, Jdk 1.8 and CEPSim simulator [45] and is executed on an Intel Dual-Core CPU 3.0 GHz 64-bit with 4 GB RAM. Implemented queries are also available in the repository.[1]

Table 3 Test bed specifications

VM type	Core speed (ECU)	Memory (GB)	Cores	Cost ($)
m1.small	1	1.7	1	0.06
m1.medium	2	3.75	1	0.12
m1.large	2	7.5	2	0.24
m1.xlarge	2	15	4	0.48
m3.xlarge	3.25	15	4	0.50
m3.xxlarge	3.25	30	8	1.00

5.3 Set-up

Table 4 describes the parameters used by BCframework to execute the queries.

Performance degradation model of public Cloud presented by [6] is used to conduct the experiments. Loss in performance observed in *Vm* for a specific scheduling period is sampled from a normal distribution with the average of 15% loss and standard deviation of 10% loss. Similarly, loss in each data transfer is modeled with uniform distribution with average of 30% loss and standard deviation of 10% loss.

For each query instance Bq_{ij} and each level of Bs^{Ve}, a soft deadline was generated as follows (adapted from [33]). A *base* runtime is defined as the execution time obtained with a provisioning and scheduling strategy which assigns each operator of the query to an instance of the most powerful *Vm* instance and assumes there is no inter node traffic cost. The *base* value obtained with such strategy is then multiplied by the number of operators. In addition, a deadline factor α is defined which is set by our conducted experiments. Therefore, the deadline of query Bq_{ij} is set by $\alpha.base$.

The same random number generation seeds were used for each strategy, which ensures that the condition faced by each algorithm is exactly same regarding infrastructure behavior. Furthermore, the average of output metrics is presented in the following experimental result tables and figures.

The range of values for Bs^{Ve}, is set to evaluate BCframework in face of the fixed low rate and high fluctuating rate of the incoming Big Data streams. Furthermore, the value of *ds* is also set to reflect the behavior of the windows-based operators of the benchmark and real-world queries. The range of values for Bs^{Va} is set using the structure of benchmark queries to evaluate BCframework algorithms in face of the different tuple sizes. The value of *Threshold* and α parameters are determined by the conducted experiments. Each query is run for 15 min so Execution and Period parameters are set to 15 min.

5.4 BCframework validation

In order to validate BCframework, we validate the functionality of the query partitioning algorithm under the simple, complex benchmark and real-world queries. The validation is conducted by evaluating the identified partitions during their executions. In order to compare the

[1] shbu.ac.ir/mortazavi/Cepsimqueries/.

Table 4 Parameters used in BCframework

Parameter	Definition	Value
Bs^{Ve}	*Bf* fixed average rate *Bf* fluctuating average rate level	10^3 tuples/second [10^4, 15×10^3, 30×10^3] tuples/second
ds	Number of data sets based on window size $\left(Bq_i^W\right)$	30 s
Bs^{Va}	Tuple size	[4B–10 KB]
Execution	Estimated execution period	15 min
Period	Billing period of *VM*	15 min
α	Deadline factor of query	[2–6]
Threshold	Deadline miss threshold	50%

Table 5 Partitioning of Linear Road Benchmark queries

Q	R (%)	P	Q	R (%)	P	Q	R (%)	P	Q	R (%)	P
1	1.3	2	6	49.8	2	11	31.1	2	16	10.1	2
2	51.7	2	7	27.1	2	12	44.2	2	17	17.7	2
3	56.1	2	8	26.2	2	13	57.8	2			
4	0	1	9	49.9	2	14	18.1	2			
5	4.8	2	10	39.2	2	15	26.9	2			

identified partitions, the ratio between the average *outBfsize* of the first partition and the average *outBfsize* of the second one is calculated and shown in the R column of the result tables. Furthermore, the number of identified partitions is reported in the P column of the following tables.

5.4.1 Simple queries

Table 5 shows the result of partitioning Linear Road Benchmark queries during execution of the queries. To execute these experiments, the configuration parameters Bs^{Ve} and Bs^{Va} are set to [10,000–30,000 tuples/s] and [4B–45B] respectively, which later is derived from the structure of the Linear Road Benchmark queries.

The Table 5 demonstrates that the most Linear Road Benchmark queries are partitioned in two partitions but the average rate for these identified partitions is low. Since the distance of the tuple size of the first partition operators from the second one is small, the rate remains low. In addition, the table notes that, for the query Q4, the number of identified partitions is 1 which is affected by the simple structure of the query so all operators of Q4 are partitioned in the first partition.

5.4.2 Complex queries

Table 6 shows the result of partitioning BigBench queries during the execution of the queries. To execute these experiments, the configuration parameters Bs^{Ve} and Bs^{Va} are set to [10,000–30,000 tuples/s] and [500B–10 KB] respectively, which later is derived from the structure of the BigBench queries.

Table 6 Partitioning of BigBench queries

Q	R (%)	P	Q	R (%)	P	Q	R (%)	P
(a) Queries 1–15								
1	97.1	2	6	85.4	2	11	97.4	2
2	61.1	2	7	98.1	2	12	83.1	2
3	94.3	2	8	68.1	2	13	61.9	2
4	45.7	2	9	97.1	2	14	97.2	2
5	91.1	2	10	0	1	15	94.1	2
(b) Queries 16–30								
16	94.1	2	21	37.4	2	26	90.3	2
17	98.8	2	22	88.6	2	27	0	2
18	80.9	2	23	82.8	2	28	74.1	2
19	91.3	2	24	92.1	2	29	95.9	2
20	84.1	2	25	90.2		30	89.9	2

As shown in Table 6, for the most BigBench queries, they are partitioned into two partitions and their rates are high. The high rate of the identified partitions is due to this fact that the distance of the tuple size of the first partition operators from the second one is large. In particular, Q17 has the highest rate which is a caused by the complexity of its structure and the number of its operators. Meanwhile, for queries like Q4, the rate is low which is affected by its linear structure. In addition, for the queries Q10, Q27 the number of identified partitions is 1 which is affected by the simple structure of the query so all operators of each query are partitioned in the first partition.

5.4.3 Real-world queries

The result of the partitioning of the WordCount and TOP_N queries is plotted in Fig. 4. The needed configuration parameters Bs^{Ve} and Bs^{Va} are set to [10,000–30,000 tuples/s] and [4B–800B] respectively, which later are derived from the structure of the queries.

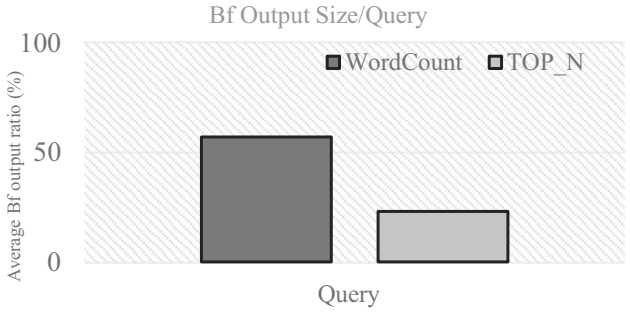

Fig. 4 Result of partitioning two real-world queries

As shown in Fig. 4, the average outBfsize ratio between the first partition and second one for the WordCount query is higher than the TOP_N query because the structure of WordCount includes more complex operators than Top_N query.

5.5 BCframework performance

In order to evaluate BCframework performance, we compare the result of the average tuple latency, the utilization cost, the number of the deadline misses and the number of provisioned *Vm* instances achieved by Storm default scheduler (the default scheduler) with BCframework under the simple and complex benchmark queries. In addition, BCframework performance is compared with the state-of-the-art scheduler [30] under the real-world queries. All these experiments have been performed in the presence of fixed and fluctuating *Bs* scenarios. Therefore, the configuration parameter Bs^{Ve} is set to 1000 tuples/s and [10,000–30,000 tuples/s] for fixed and fluctuating scenarios respectively.

The following tables and figures show the experimental results in presence of fixed and fluctuating rate in their FX and FL symbols respectively by using BCframework (BC) algorithms, Storm default scheduler (DF) or the state-of-the-art scheduler (ST). The observed output metrics tuple latency, utilization costs, total number of deadline violations and the number of provisioned *Vm* instances are reported for each conducted experiment.

5.5.1 Simple queries

Figure 5 shows the average tuple latency of a sample Linear Road Benchmark query during the execution time of fifteen minutes. The average tuple latency reduction of all Linear Road Benchmark queries is presented in Fig. 6. The experiments are performed by the configuration parameter Bs^{Va} set to [4B–45B] based on the each Linear Road Benchmark query structure. In addition, the deadline factor α is set to [2, 3].

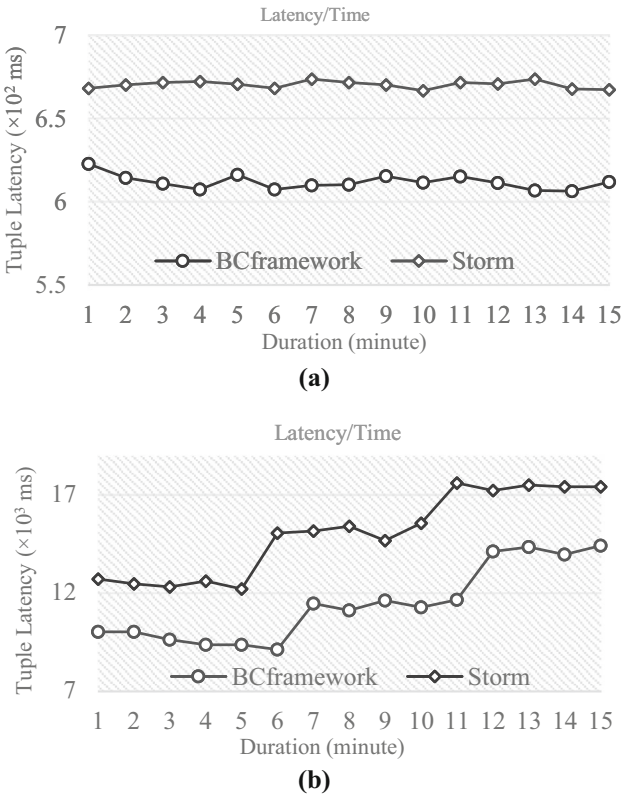

Fig. 5 Average tuple latency reduction of Linear Road Benchmark Query Q8. **a** Fixed rate. **b** Fluctuated rate

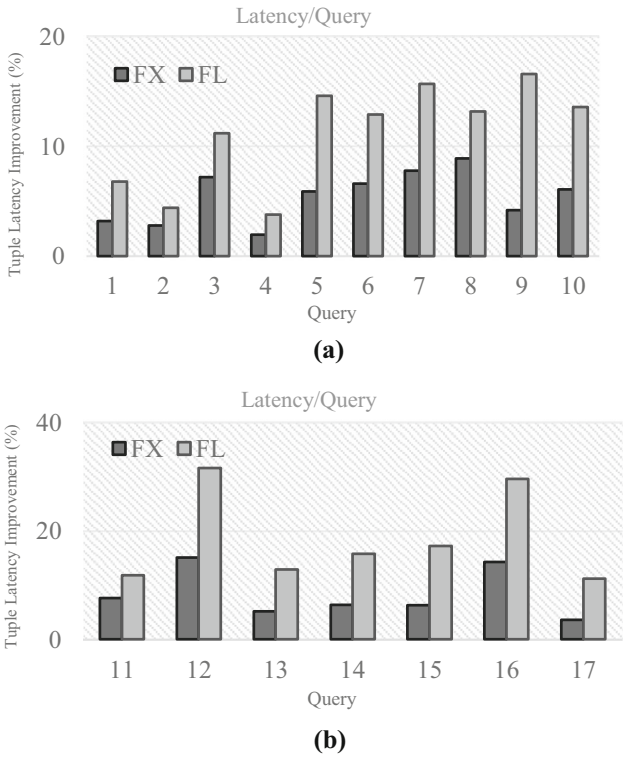

Fig. 6 Average tuple latency improvement under Linear Road Benchmark by BCframework. **a** Queries 1–10. **b** Queries 11–17

As shown in Fig. 5a and b, for query Q8 an average tuple latency reduction of 8.9% and 13.2% is achieved respectively. The improvement in Q8 is related to its structure where two PCPs are identified by BCframework. These enable Oscheduler to provision one instance of the fastest VM type and one instance of the cheapest VM type to assign them to the operators of PCPs. Whereas, the default scheduler does not take into account Q8's structure and the type of VM and simply distributes Q8 operators.

Figure 6 shows that BCframework only yields latency improvement of about 4% for the most Linear Road Benchmark queries like Q2 and Q4. This can be caused by the linear structure of the queries, the small numbers of identified PCPs and the short length of identified PCPs.

Figure 7 shows the average cost reduction of the Linear Road Benchmark queries. For these queries, the length of PCPs in the second partition is short and as a result BCframework's VM utilization strategy can only be applied for a small number of operations in the second partition. Whereas, for a given set of the cheapest VM instances, the default scheduler is able to assign the VM instances to all operators of the queries.

Table 7 shows that the BCframework either completely eliminates deadline violations of the simple queries or reduces them significantly. The BCframework's deadline control mechanism monitors the deadline violation of simple queries continuously whereas the default scheduler does not take into account the deadline of the queries.

Figure 8 shows that BCframework outperforms the default scheduler in *Vm* instance provisioning under simple queries. The default scheduler assigns each operator of a query to a provisioned *Vm* instance while BCframework provisions *Vm* instances by considering structure of the queries and applies different provisioning strategies for a query. For query Q8, BCframework has identified two PCPs in the partitions and provisions two instances where the default scheduler uses six provisioned instances for six operators of Q8. For query Q16, BCframework and the default scheduler assign the same number of *Vm* instances because BCframework provisions five *Vm* instances for five identified PCPs and the default scheduler distributes nine operators of the query to all five provisioned instances.

5.5.2 Complex queries

Figure 9 shows the average tuple latency of the sample BigBench query during their 15 min execution. The average tuple latency reduction of all BigBench queries is presented in the Fig. 10. These experiments are performed

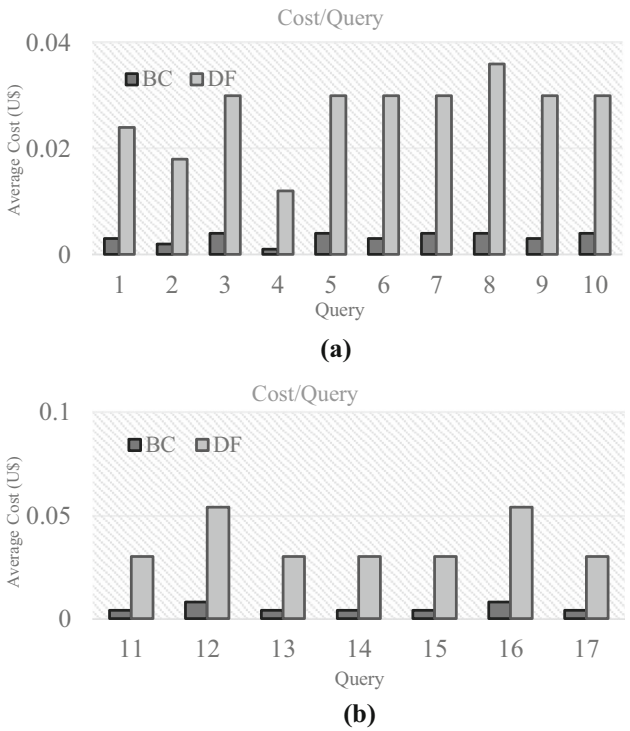

Fig. 7 The average cost (U$) under Linear Road Benchmark **a** Queries 1–10. **b** Queries 11–17

Table 7 Total number of deadline violations under Linear Road Benchmark queries

Bs^{Ve}	Q	α	BC	DF	Bs^{Ve}	Q	α	BC	DF
(a) Queries 1–10									
FX	1	3	0	0	FX	6	3	0	0
FL		3	2	8	FL		3	1	7
FX	2	3	0	0	FX	7	3	0	0
FL		3	1	8	FL		3	1	8
FX	3	3	0	0	FX	8	3	0	2
FL		3	1	9	FL		3	3	10
FX	4	3	0	0	FX	9	3	0	0
FL		3	3	9	FL		3	2	8
FX	5	3	0	0	FX	10	3	0	0
FL		3	2	9	FL		3	3	9
(b) Queries 11–17									
FX	11	3	1	3	FX	16	2	2	6
FL		3	2	10	FL		3	2	14
FX	12	3	0	0	FX	17	2	0	0
FL		3	1	12	FL		3	1	7
FX	13	3	0	0					
FL		3	2	9					
FX	14	3	0	0					
FL		3	2	8					
FX	15	3	0	0					
FL		3	3	9					

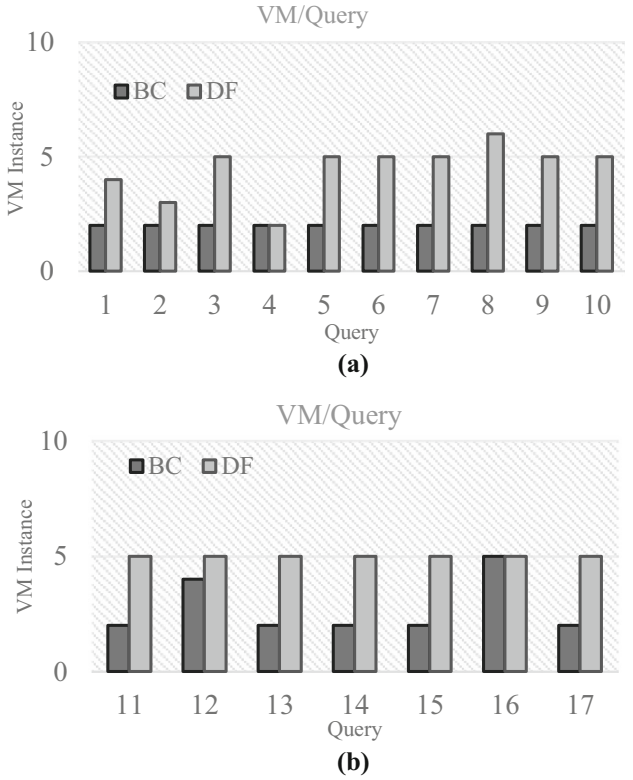

Fig. 8 The number of provisioned *Vm* instances under Linear Road Benchmark **a** Queries 1–10. **b** Queries 11–17

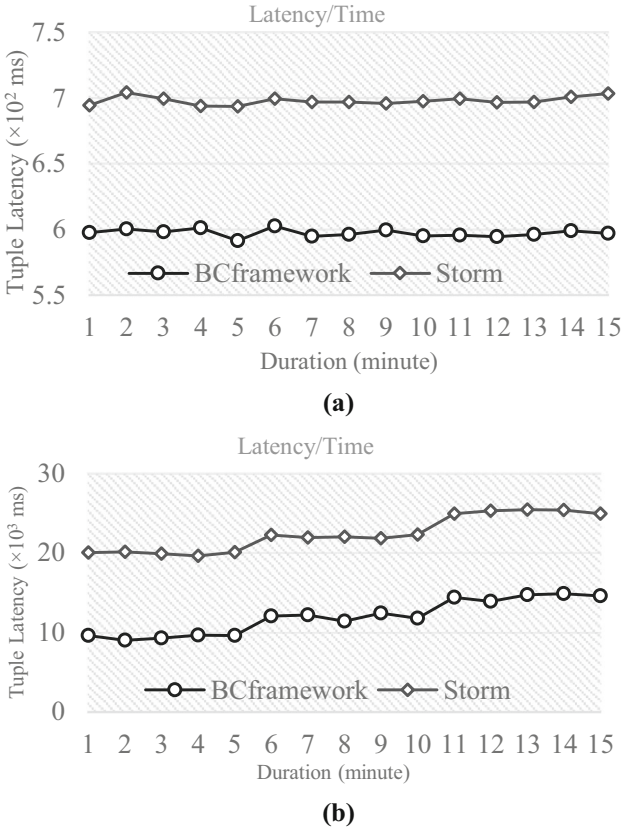

Fig. 9 Average tuple latency reduction of BigBench query Q1

with the configuration parameter Bs^{Va} set to [500B–10 KB] based on the structure of BigBench queries.

Figure 9a and b show that BCframework yields latency improvement of 16.2% and 46.8% respectively for the query Q1. The improvement of Q1 is related to its complex structure where BCframework identified more than one PCP in each partition of the query Q1. Oscheduler assigns all original operators of the identified PCP to a *Vm* instance and Rscheduler assigns the provided free *Vm* slots to the Q1 selected replicated operators. Whereas, the default scheduler does not take into account the Q1 structure, VM types and does not provide any scheduling strategy for replicated operators.

As shown in Fig. 10, the achieved improvement for the BigBench queries varies between 7.2 and 65.8% which might be caused by three main reasons: First, the structure of these queries encompasses the different styles (i.e., simple linear style or more complex star and diamond patterns or even a mix of them).

For simple linear queries like Q2 the most BCframework optimizations are not applicable while BCframework optimizations can be completely applied to complex queries like Q13. Second, the number of query operators is significantly different. For some queries like Q19, there are more than 30 operators which lead to the better performance improvement achieved by the framework. Third, the variance tuple size Bs^{Va} of these queries can lead to their variable tuple latency improvement. For example, the tuple size of Q6 is about 2 KB larger than that of Q5 which increases traffic load incurred by Q6. Therefore, traffic load reduction techniques of BCframework lead to better tuple latency reduction.

Figure 11 demonstrates that the average cost reduction of the most BigBench queries like Q2 and Q4 is about 90% which is mainly caused by the length of the identified PCPs. The longer length of PCPs in the second partition, in comparison with the first partition PCPs, allows BCframework to utilize its cheapest cost VM utilization strategy for the most numbers of operators of queries like Q2 and Q4. Whereas, the default scheduler assigns the cheapest and the most expensive VM type instances equally.

Table 8 shows that the BCframework reduces deadline violations of complex queries significantly. The deadline control mechanism of BCframework monitors the deadline violation of complex queries continuously whereas the default scheduler does not take into account the deadline of the queries.

Figure 12 shows that BCframework outperforms the default scheduler in *Vm* instance provisioning under

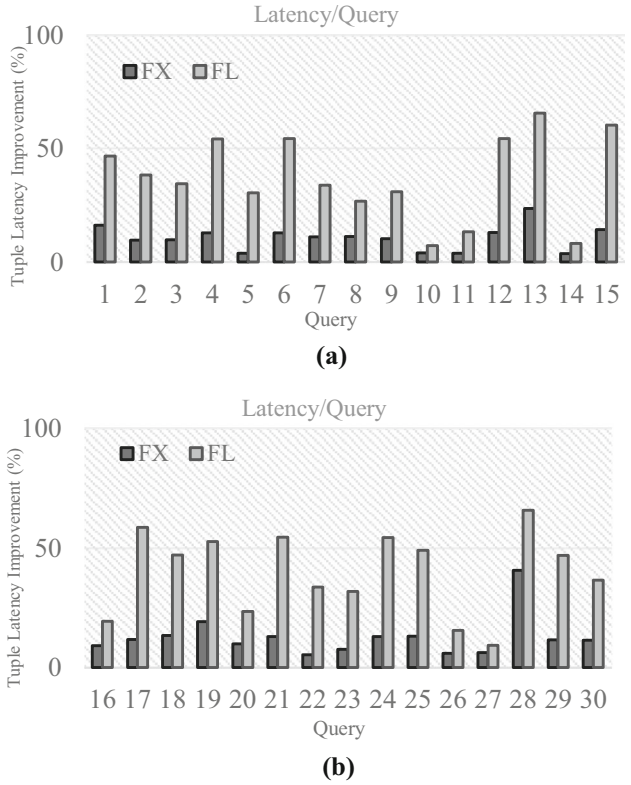

Fig. 10 Average tuple latency improvement under BigBench by BCframework. **a** Queries 1–15. **b** Queries 16–30

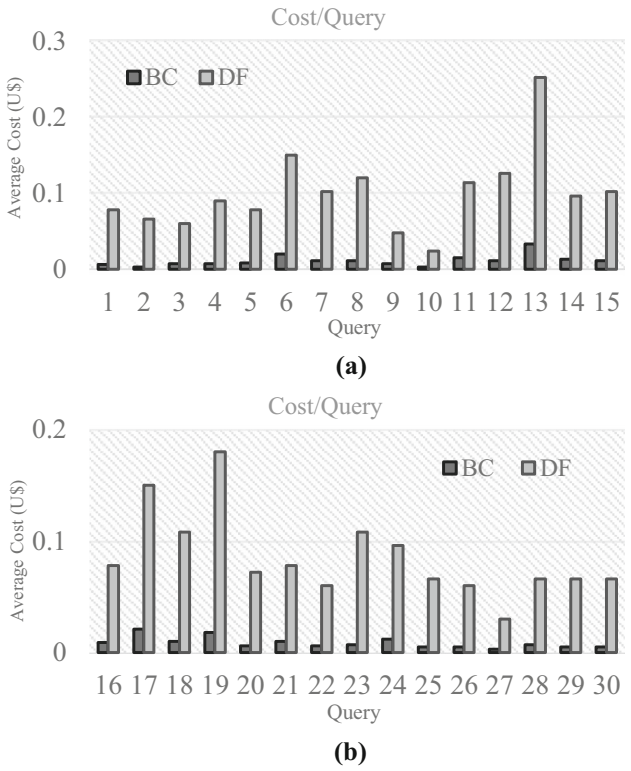

Fig. 11 The average cost (U$) under BigBench. **a** Queries 1–15. **b** Queries 16–30

Table 8 Total number of deadline violations under BigBench queries

Bs^{Ve}	Q	α	BC	DF	Bs^{Ve}	Q	α	BC	DF
Queries 1–15									
FX	1	2	1	12	FX	9	2	2	7
FL		4	0	10	FL		3	6	10
FX	2	2	2	12	FX	10	2	0	0
FL		3	5	5	FL		3	0	0
FX	3	2	2	9	FX	11	3	0	0
FL		3	4	10	FL		3	5	7
FX	4	2	3	11	FX	12	2	3	11
FL		3	3	15	FL		3	4	14
FX	5	3	0	0	FX	13	4	1	12
FL		3	4	10	FL		4	3	15
FX	6	2	0	9	FX	14	5	2	7
FL		3	4	15	FL		6	3	8
FX	7	2	3	14	FX	15	2	0	8
FL		3	4	10	FL		3	5	15
FX	8	2	1	6	FX	16	5	3	13
FL		3	3	12	FL		6	5	11
(b) Queries 16–30									
FX	17	5	3	12	FX	25	5	1	11
FL		6	3	13	FL		6	3	13
FX	18	2	3	13	FX	26	2	3	12
FL		3	4	14	FL		3	5	10
FX	19	2	2	11	FX	27	2	0	0
FL		3	1	11	FL		3	0	0
FX	20	2	2	11	FX	28	2	0	0
FL		3	6	10	FL		3	0	0
FX	21	2	5	15	FX	29	2	3	10
FL		3	8	15	FL		3	6	11
FX	22	2	6	11	FX	30	2	3	10
FL		3	6	11	FL		3	6	10
FX	23	2	3	8					
FL		3	3	9					
FX	24	2	0	10					
FL		3	3	15					

complex queries. The default scheduler assigns each operator of a query to a *Vm* instance while BCframework provisions *Vm* instances by considering structure of the queries and applies the different strategies for a query. For query Q11, BCframework has identified six PCPs and six instances are provisioned while the default scheduler assigns the provisioned eighteen instances to the nineteen operators. For query Q13, both BCframework and the default scheduler assign the same number of *Vm* instances because BCframework provisions eighteen *Vm* instances for eighteen identified PCPs and the default scheduler

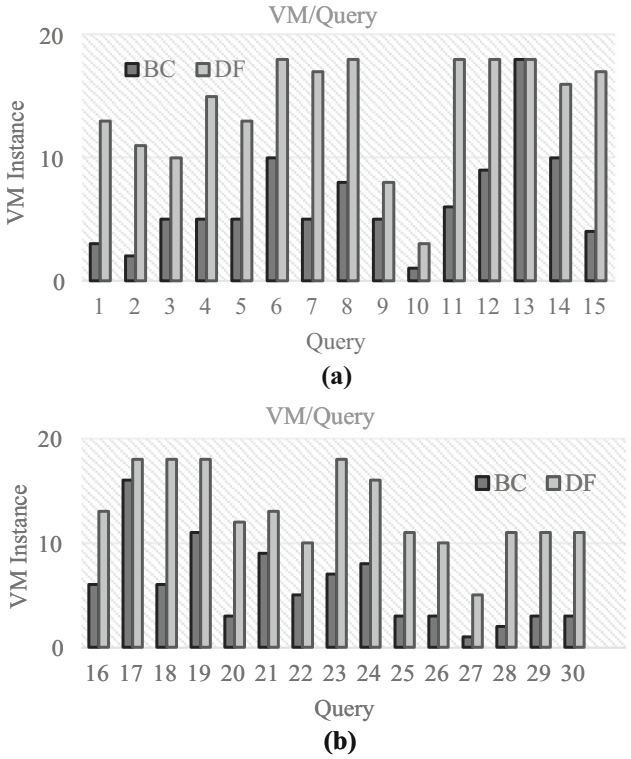

Fig. 12 The number of provisioned *Vm* instances under BigBench queries. **a** Queries 1–15. **b** Queries 16–30

distributes forty-two operators of the query to all eighteen provisioned instances.

5.6 Real-world queries

As shown in Fig. 13a, BCframework reduces the average tuple latency of WordCount and Top_N queries up to 36% and 41% respectively. The state-of-the-art scheduler prioritizes WordCount's operators according to their position in the query to order the operators with highest computing time first. Furthermore, the scheduler's policy for scheduling the operators on virtual machines is based on the minimum total time of available time and computing time which leads to schedule the operators to three *Vm* instances and increase the traffic load between these instances. Whereas, BCframework partitions WordCount's operators based on the *Bf* size and schedules them on two *Vm* instances based on the available time, computing power and computing cost of the operators which reduce the traffic load of virtual machines. To schedule Top_N, the

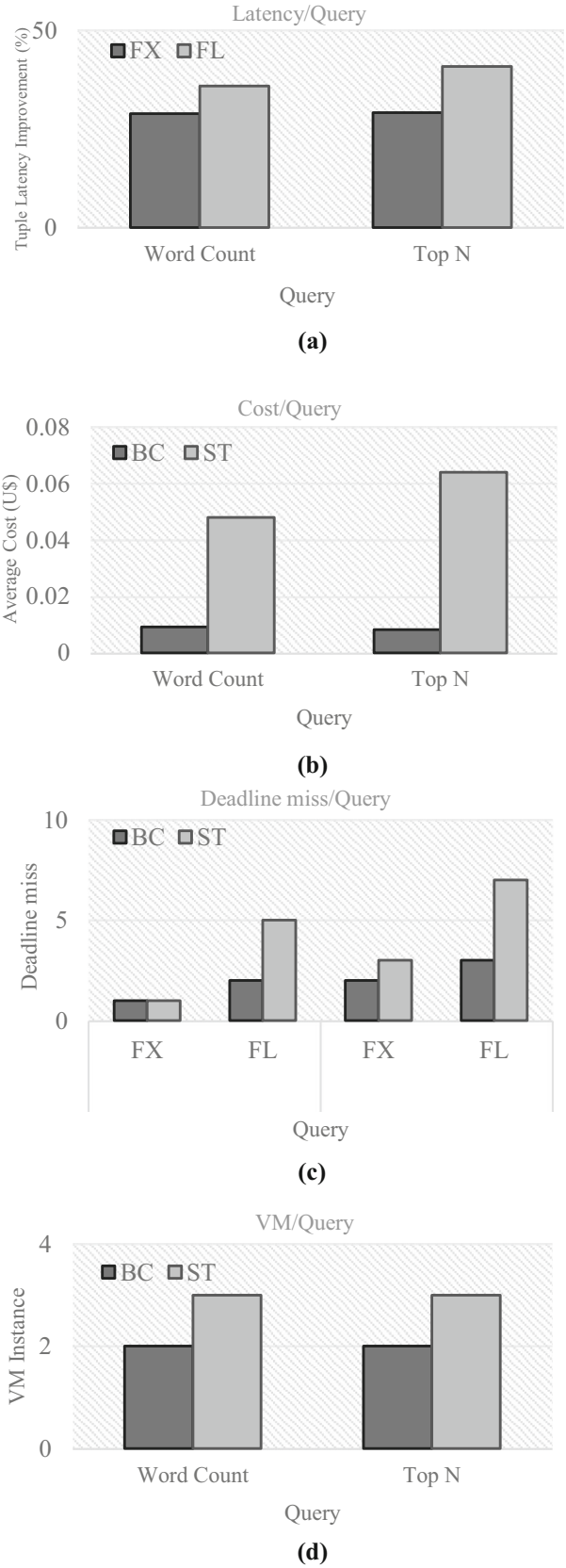

Fig. 13 Performance evaluation of BCframework and the state-of-the-art scheduler under WordCount and Top_N queries. **a** Average tuple latency improvement **b** Average cost. **c** Total number of deadline violations. **d** The number of provisioned *Vm* instances

state-of-the-art scheduler also considers three *Vm* instances while BCframework only considers two instances.

Figure 13b shows the cost reduction of the WordCount and Top_N queries. For these queries, the state-of-the-art scheduler takes into account virtual machines whose computing time is minimized which leads to an increase of utilization cost of the provisioned *Vm* instances while BCframework considers the highest computing time for first partition operators and the lowest computing utilization cost for the second partition operators which leads to reduce the computation utilization cost of a query.

As shown in Fig. 13c, in the face of fixed and fluctuating scenarios, BCframework either completely eliminates deadline violations of the WordCount and Top_N queries or reduces them significantly. The BCframework's deadline control mechanism monitors the deadline violation of the queries continuously and reschedules operators using a rescheduling strategy which provisions additional *Vm* instances. Whereas the state-of-the-art scheduler applies its primary scheduling strategy to reschedule the operators of a query.

Figure 13d shows that BCframework outperforms the state-of-the-art scheduler in *Vm* instance provisioning under real-world queries. The state-of-the-art scheduler's policy for provisioning *Vm* instances is based on minimum total time of available time and computing time of *Vm* instances which leads to the provision of three *Vm* instances Whereas, BCframework partitions WordCount's operators based on the *Bf* size and schedule them on two virtual machines based on the available time, computing power and computing utilization cost of the operators. To provision *Vm* instances for Top_N, the state-of-the-art scheduler also choices three virtual machine instances while BCframework just considers two instances.

5.7 Parameter sensitivity

To determine the parameters *Threshold* and α, two group of experiments has conducted. For the first experiments, the deadline misses of the queries are evaluated to determine the deadline factor α and for the second experiments, the deadline misses and utilization costs of the queries are evaluated to determine the *Threshold* parameter. For both experiments, the queries are executed using a range of different values for α and *Threshold* while respecting the fixed scenario $Bs^{Ve} = 1000\,\text{tuples/s}$, and fluctuated scenario $Bs^{Ve} = 15,000\,\text{tuples/s}$. Figure 14 illustrates the result of the parameter sensitivity of the simple query Q1 of Linear Road Benchmark and complex query Q1 of BigBench.

As shown in Fig. 14a and b for the simple query Q1 and complex query Q1 under fix and fluctuating scenarios, the

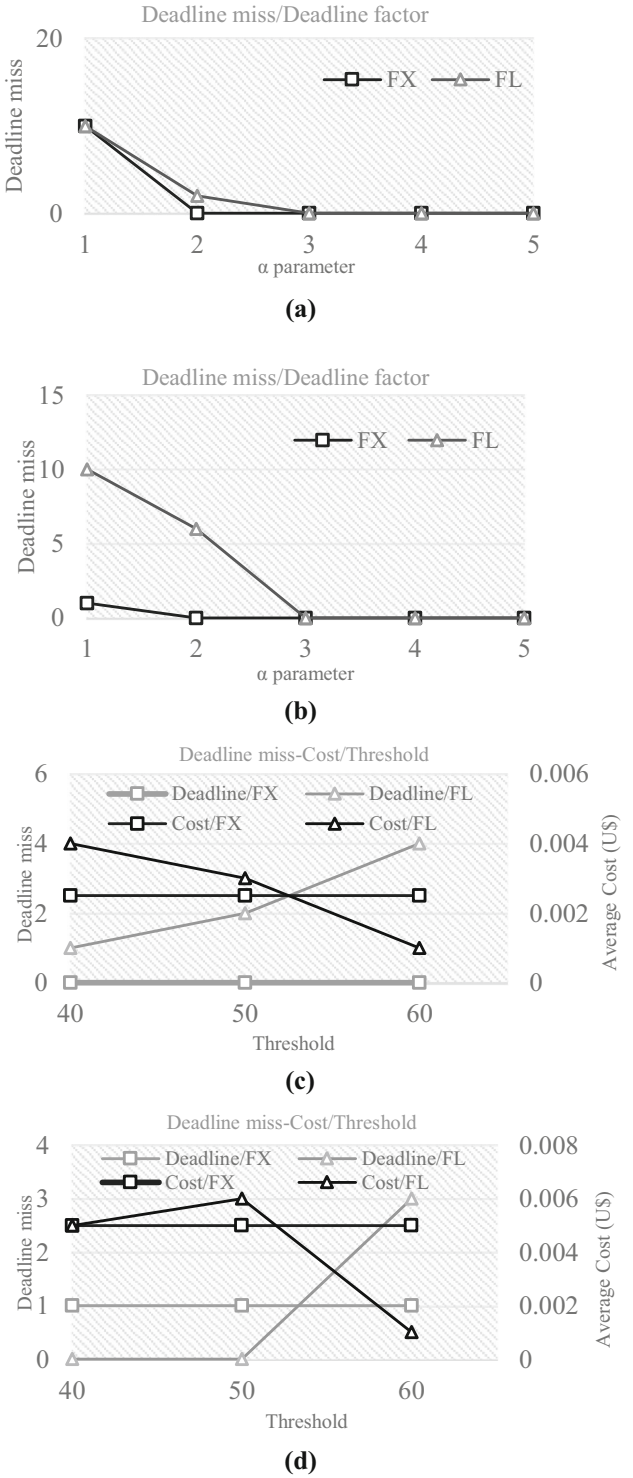

Fig. 14 Evaluation of BCframework's parameter sensitivity. **a** Effect of α on the deadline miss under simple query Q1. **b** Effect of α on the deadline miss under complex query Q1. **c** Effect of *Threshold* on the deadline miss and the utilization cost under the simple query Q1. **d** Effect of the *Threshold* on the deadline miss and the utilization cost under the complex query Q1

deadline miss will be increase by decreasing deadline factor α. For the query Q1 under both scenarios, by the most restricted value of α, e.g., $\alpha = 2$, BCframework cannot guarantee the execution of the query before its deadline. Whereas, for a relaxed value of α, e.g., $\alpha = 4$, the query can be executed without deadline violation. Therefore, for the simple query Q1, the α parameter is adjusted to 3. In a similar way, for the complex query Q1, the α parameter is adjusted to two and four for fixed and fluctuated scenarios respectively. The adjusted value of other α parameters of all simple and complex queries are illustrated in Tables 7 and 8 respectively.

As shown in Fig. 14c and d for the simple query Q1 and complex query Q1 under fix and fluctuating scenarios, decreasing the value of *Threshold* leads to a decrease of the deadline violations and to an increase of utilization cost of the query. By decreasing the *Threshold* value, deadline controller of BCframework has to react more quickly and must provision more *VM* instances to decrease the deadline violations that result in increasing the utilization cost. On the other hand, by increasing *Threshold* value, deadline controller just monitors the execution of the queries.

5.8 Discussion

According to the conducted experiments and their analysis, we can conclude that BCframework is able to correctly partition the streaming Big Data analytic queries based on its input Big Data stream. Experimental results show that BCframework is able to correctly partition simple and complex analytic queries of Big Data stream.

Furthermore, BCframework outperforms the Storm default scheduler in the average tuple latency reduction under the simple and complex streaming Big Data analytic queries up to 29.6% and 65.8% respectively. In addition, BCframework outperforms the state-of-the-art scheduler in the average tuple latency reduction under the real-world queries up to 41%. Experimental result have shown that BCframework is able to reduce the average tuple latency of simple queries mainly through the appropriate scheduling of virtual machines. For complex queries, BCframework can also be used to efficiently reduce the average tuple latency through the efficient scheduling and provisioning of VM instances.

In addition, BCframework outperforms Storm default scheduler in the reduction of public Cloud resource utilization cost and the deadline violations of streaming Big Data analytic queries. Experimental results have proven that BCframework can reduce the utilization cost of the simple and complex Big Data stream analytical queries through the appropriate VM utilization policy for the operators of the query. Furthermore, experiments have shown that BCframework reduces the deadline violation using the deadline monitoring mechanism and efficient VM provisioning policy.

Finally, BCframework performs best in complex fluctuating streaming Big Data analytic queries. Experimental results prove that most BCframework optimization techniques are completely utilized by queries with complex structures and fluctuating input Big Data streams.

6 Conclusion

The demand for streaming Big Data analysis applications is increasing. As the demand is growing, the efficient execution of the applications using suitable Cloud platforms become more important. However, current platform schedulers are not suitable for these applications in an effective manner. This study presents BCframework, a framework designed to efficiently schedule the accepted streaming Big Data analytic queries in the public Cloud. The BCframework proposes partitioning, critical path determination, scheduling and provisioning algorithms for the original and replicated operators. BCframework is developed based on the idea of using more than one scheduling strategy for streaming Big Data analytic query original operator and replicated operator which is used by the proposed algorithms to reduce the utilization cost, the deadline miss ratio of queries and mitigate public Cloud performance fluctuation. BCframework partitions a query based on its Big Data stream characteristics and applies a set of efficient provisioning and scheduling strategies for each partition during the execution of query.

Experimental results show that, compared to Storm's default scheduler and the state-of-the-art scheduler, BCframework is able to efficiently reduce tuple latency of streaming Big Data analytic queries up to 65% and 41% respectively. Furthermore, BCframework significantly decreases the likelihood of the deadline misses and utilization cost of streaming Big Data analytic queries.

In the future work, it is planned to extend BCframework on multiple cloud resources. A more efficient resource provisioning mechanism using streaming Big Data fluctuation prediction model will be considered in our following work. Moreover, experiments that execute Health Big Data queries will also be included.

References

1. Chen, M., Mao, S., Zhang, Y., Leung, V.C.M.: Big Data: Related Technologies, Challenges and Future Prospects. Springer, New York (2014)
2. Safaei, A.A.: Real-time processing of streaming big data. Real-Time Syst. **53**(1), 1–44 (2017)
3. Barlow, M.: Real-Time Big Data Analytics Emerging Architecture. O'Reilly Media, Inc, Newton (2013)
4. Berman, J.J.: Principles of Big Data: Preparing, Sharing, and Analyzing Complex Information. Elsevier, Amsterdam (2013)
5. Pedrycz, W., Chen, S.-M.: Information Granularity, Big Data, and Computational Intelligence. Springer, New York (2014)
6. Jackson, K.R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., et al. (eds.): Performance analysis of high performance computing applications on the amazon web services cloud. In: Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), IEEE (2010)
7. Apache Storm [Online]. http://storm.apache.org/index.html. Accessed 4 Feb 2016
8. Xu, J., Chen, Z., Tang, J., Su, S., (eds.): T-storm: Traffic-aware online scheduling in storm. In: Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS), IEEE (2014)
9. Cardellini, V., Grassi, V., Lo Presti, F., Nardelli M., (eds.): Distributed QoS-aware scheduling in Storm. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, ACM (2015)
10. Li, T., Tang, J., Xu, J., (eds.): A predictive scheduling framework for fast and distributed stream data processing. In: Proceedings of the 2015 IEEE International Conference on Big Data (Big Data), IEEE (2015)
11. Cardellini, V., Grassi, V., Lo Presti, F., Nardelli, M.: Optimal operator replication and placement for distributed stream processing systems. ACM SIGMETRICS Perf. Eval. Rev. **44**(4), 11–22 (2017)
12. Cardellini, V., Grassi, V., Lo Presti, F., Nardelli, M., (eds.): Optimal operator placement for distributed stream processing applications. In: Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, ACM (2016)
13. De Matteis, T., Mencagli, G., (eds.): Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM (2016)
14. Dwarakanathan, S, (ed.): S-Flink: schedule for QoS in Flink Using SDN. In: Proceedings of the IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), 2016, IEEE (2016)
15. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., et al. (eds.): Twitter heron: stream processing at scale. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM (2015)
16. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., et al.: Aurora a new model and architecture for data stream management. VLDB J. **12**(2), 120–139 (2003)
17. Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., Valduriez, P.: Streamcloud: an elastic and scalable data streaming system. IEEE Trans. Parallel Distrib. Syst. **23**(12), 2351–2365 (2012)
18. Li, C., Zhang, J., Luo, Y.: Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm. J. Netw. Comput. Appl. **87**, 100–115 (2017)
19. Sun, D., Zhang, G., Yang, S., Zheng, W., Khan, S.U., Li, K.: Re-Stream: real-time and energy-efficient resource scheduling in big data stream computing environments. Inf. Sci. **319**, 92–112 (2015)
20. Sun, D., Zhang, G., Wu, C., Li, K., Zheng, W.: Building a fault tolerant framework with deadline guarantee in big data stream computing environments. J. Comput. Syst. Sci. **89**, 4–23 (2017)
21. Kaur, N., Sood, S.K.: Dynamic resource allocation for big data streams based on data characteristics (5Vs). Int. J. Netw. Manag. **27**, e1978 (2017)
22. Sun, D., Huang, R.: A stable online scheduling strategy for real-time stream computing over fluctuating big data streams. IEEE Access **4**, 8593–8607 (2016)
23. Kaur, N., Sood, S.K.: Efficient resource management system based on 4Vs of big data streams. Big Data Res. (2017). https://doi.org/10.1016/j.bdr.2017.02.002
24. Huang, R., Sun, D.: Analysing and evaluating topology structure of online application in Big Data stream computing environment. Int. J. Wireless Mobile Comput. **10**(4), 317–324 (2016)
25. Tolosana-Calasanz, R., Bañares, J.Á., Pham, C., Rana, O.F.: Resource management for bursty streams on multi-tenancy cloud environments. Future Gener. Comput. Syst. **55**, 444–459 (2016)
26. Rahman, M.M., Graham, P.: Responsive and efficient provisioning for multimedia applications. Comput. Electr. Eng. **53**, 458–468 (2016)
27. Zhang, Q., Chen, Z., Yang, L.T.: A nodes scheduling model based on Markov chain prediction for big streaming data analysis. Int. J. Commun. Syst. **28**(9), 1610–1619 (2015)
28. Peng, J-j, Zhi, X-f, Xie, X-l: Application type based resource allocation strategy in cloud environment. Microprocess. Microsyst. **47**, 385–391 (2016)
29. Baughman, A.K., Bogdany, R.J., McAvoy, C., Locke, R., O'Connell, B., Upton, C.: Predictive cloud computing with big data: professional golf and tennis forecasting [application notes]. IEEE Comput. Intell. Mag. **10**(3), 62–76 (2015)
30. Sun, D., et al.: Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams. J. Supercomput. **74**(2), 615–636 (2018)
31. Amazon AWS EC2 pricing model [Online]. www.amazon.com/EC2/pricing/pricingmodel.html. Accessed 7 June 2016
32. Fegaras, L.: Incremental query processing on Big Data streams. IEEE Trans. Knowl. Data Eng. **28**(11), 2998–3012 (2016)
33. Byun, E.-K., Kee, Y.-S., Kim, J.-S., Maeng, S.: Cost optimized provisioning of elastic resources for application workflows. Future Gener. Comput. Syst. **27**(8), 1011–1026 (2011)
34. Shi, J., et al.: Elastic resource provisioning for scientific workflow scheduling in cloud under budget and deadline constraints. Clust. Comput. **19**(1), 167–182 (2016)
35. Amazon EC2 [Online]. https://aws.amazon.com/ec2/. Accessed 10 May 2016
36. Apache Spark - Unified Analytics Engine for Big Data [Online]. https://spark.apache.org/ Accessed 10 Apr 2018
37. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., et al. (eds.): BigBench: towards an industry standard benchmark for big data analytics. In: Proceedings of the 2013 ACM SIGMOD international conference on Management of data, ACM (2013)
38. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., et al. (eds.): Linear road: a stream data management benchmark. In: Proceedings of the Thirtieth international conference on Very large data bases, vol. 30, VLDB Endowment (2004)
39. Tutorial [Online]. http://storm.apache.org/releases/1.1.2/Tutorial.html. Accessed 10 Apr 2018

40. Common Toplogy Patterns [Online]. http://storm.apache.org/releases/1.2.1/Common-patterns.html. Accessed 10 Apr 2018
41. Byrne, J., et al.: A review of cloud computing simulation platforms and related environments. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science. SCITEPRESS-Science and Technology Publications, Lda Portugal (2017)
42. Rabl, T., et al.: BigBench Specification V0. 1, in Specifying Big Data Benchmarks, pp. 164–201. Springer, New York pp (2014)
43. Higashino, W.A., Capretz, M.A., Bittencourt, L.F.: CEPSim: modelling and simulation of Complex Event Processing systems in cloud environments. Fut. Gener. Comput. Syst. **65**, 122–139 (2016)
44. Calheiros, R.N., et al.: CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software **41**(1), 23–50 (2011)
45. CEPSim - Simulator for CEP/SP systems. https://github.com/virsox/cepsim. Accessed 10 Apr 2018
46. Han, R., Lu, X., Xu, J.: On Big Data benchmarking. In: Zhan, J., Han, R., Weng, C. (eds.) Big Data Benchmarks, Performance Optimization, and Emerging Hardware. BPOE 2014. Lecture Notes in Computer Science. Springer, Cham (2014)

Mahmood Mortazavi Dehkordi obtained his M.Sc. in software engineering from University of Isfahan. He is faculty member of Sheikhbahaee University. Presently he is pursuing his Ph.D. in Big Data and cloud computing at University of Isfahan. His research areas are Big Data, Cloud Computing and Software Engineering.

Kamran zamanifar received his B.Sc. and M.Sc. from the University of Tahran in electronic engineering. He got his Ph.D. in parallel and distributed systems form University of Leeds in UK. He is associate professor at the software engineering department at university of Isfahan. His research interests are in parallel and distributed computing, cloud computing, soft computing and pervasive computing.