This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2018.2876242, IEEE Transactions on Cloud Computing

Cloud Resource Scaling for Time-Bounded and Unbounded Big Data Streaming Applications

Olubisi Runsewe, *Student Member, IEEE,* and Nancy Samaan, *Member, IEEE,*

Abstract—Recent advancements in technology have led to a deluge of big data streams that require real-time analysis with strict latency constraints. A major challenge, however, is determining the amount of resources required by applications processing these streams given their high volume, velocity and variety. The majority of research efforts on resource scaling in the cloud are investigated from the cloud provider's perspective with little consideration for multiple resource bottlenecks. We aim at analyzing the resource scaling problem from an application provider's point of view such that efficient scaling decisions can be made. This paper provides two contributions to the study of resource scaling for big data streaming applications in the cloud. First, we present a Layered Multi-dimensional Hidden Markov Model (LMD-HMM) for managing time-bounded streaming applications. Second, to cater to unbounded streaming applications, we propose a framework based on a Layered Multi-dimensional Hidden Semi-Markov Model (LMD-HSMM). The parameters in our models are evaluated using modified Forward and Backward algorithms. Our detailed experimental evaluation results show that LMD-HMM is very effective with respect to cloud resource prediction for bounded streaming applications running for shorter periods while the LMD-HSMM accurately predicts the resource usage for streaming applications running for longer periods.

Index Terms—Big Data; Cloud Computing; Stream Processing; Resource Prediction; Resource Scaling; Layered Hidden Markov Model; Layered Hidden Semi-Markov Model.

1 Introduction

Recent years have witnessed a new class of real-time big data streams generated from social media, web clicks, IP logs, and sensing devices [1]. Processing these big data streams in real-time to extract value has become a crucial requirement for many time-bounded applications, running for a short period of time or unbounded big data stream processing applications that execute for a long period of time. With the real-time stream processing requirements for big data, traditional data management systems are not suitable and are prohibitively expensive when dealing with very large amounts of data [2]. To handle fast and massive streaming data, cloud services are increasingly leveraged. Cloud computing is a powerful paradigm that allows users to quickly deploy their applications in an elastic setting through on-demand acquisition/release of resources at a low cost [3]. Oftentimes, the cloud hosts many of the stream processing engines (SPEs) (e.g., Storm [4], Spark [5], and S4 [6]), which users can make use of depending on their requirements. These SPEs allow for the continuous processing of data as it is generated [2]. They also alleviate the performance problems faced by the traditional store-and-process model of data management since they process data directly in-memory.

Nevertheless, it is naturally challenging for big data application service providers to determine the right amount of resources needed to meet the QoS requirements of streaming applications, especially when multiple applications with varying resource bottlenecks (e.g., CPU, memory) running on a shared cluster of hosts are involved. A statically provisioned SPE (i.e., an SPE deployed on a fixed number of processing nodes) can lead to overprovisioning of resources for peak loads or to underprovisioning, which can have a huge impact on the performance of the streaming applications. Thus, to efficiently process dynamic streams, SPEs need to be able to scale up/down the used cloud resources accordingly. Different scaling techniques are used to reduce the cost of utilizing cloud resources. These involve i) adding virtual machine instances, i.e., horizontal scaling, ii) changing the amount of resources allocated to a virtual machine, i.e., vertical scaling, or iii) distributing workload among available virtual machines.

Prior work on resource scaling for streaming applications can be classified as either reactive or proactive. Although, reactive techniques are computationally attractive, i) resource sensitive applications may suffer from a degraded performance or even terminate before the need for resource up scaling is detected [7], ii) the experienced delay resulting from the scaling operations (e.g., adding a new virtual machine (VM), searching for a spot at a datacenter, allocating IP address, configuring and booting the OS [8]) may not be tolerated by most streaming applications [9] and, iii) they perform poorly when multiple resources (e.g., CPU and memory) must be adjusted simultaneously [10].

Proactive techniques [11], [12] overcome the aforementioned limitations by predicting future applications resource needs before resource bottlenecks occur. However, existing predictive approaches proposed for streaming applications can be cumbersome and may lead to inaccurate results. This problem is further complicated by the stringent delay constraints, increasing parallel computation requirements, dynamically changing data volumes and arrival rates of the data streams. Overall, these techniques often address the

• *The authors are with the School of Computer and Electrical Engineering, University of Ottawa, Canada.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2018.2876242, IEEE Transactions on Cloud Computing

auto-scaling problem from a cloud provider's point of view and mostly focus on web applications.

In our previous work [13], we proposed the use of a layered multi-dimensional HMM (LMD-HMM) framework to facilitate the automatic scaling up/down of resources (i.e., add or decommission virtual machine) for time-bounded streaming applications. The proposed design provides an approach for modelling each application individually at the lower-layer while the upper-layer models interactions between group of streaming applications running on a cluster. However, our experimental results showed that the LMD-HMM may not be sufficiently accurate for predicting the resource needs of unbounded streaming applications that run over very long periods. This can be attributed to its implicit assumption of constant or geometric distribution of resource consumption state durations that do not change over time.

In this paper, we overcome the aforementioned problem by proposing a novel Layered Multi-dimensional Hidden Semi-Markov Model (LMD-HSMM) that can accurately capture the resource demands of unbounded big data streaming applications. More precisely, the main contributions of this model are two fold. First, we introduce a novel mathematical model for predicting the resource usage patterns of multiple unbounded streaming applications running simultaneously on a cluster of cloud nodes. Second, we present a modified forward-backward algorithm [14], commonly used in calculating the predicted parameters of the Markov model that takes into consideration the prediction of multiple resource bottlenecks at each layer of the model. The proposed layered approach simplifies resource prediction for a large cluster of nodes by modelling the behavior of each streaming application individually at the lower layer of the model.

Furthermore, the multi-dimensional Markov process is used for each of these streams to separately capture the individual resource (e.g., memory, CPU and I/O) needs. The model also captures at a finer grain the frequency of resource consumption state changes per resource for each stream, which are then fed to the upper-layer in the model. The upper-layer then models the dependency and collective resource usage for the unbounded streaming applications in the cluster. This layered approach not only allows for the usage of different Markov models simultaneously but also accurately captures varying application needs while avoiding the need to use a larger dataset to train since each streaming application model is trained individually. The frequency of retraining these models can be adjusted for each application.

The rest of this paper is organized as follows: Section 2 briefly discusses stream processing applications and key technologies used for big data stream processing. It also presents the challenges encountered in determining resources for big data stream processing applications and the existing prediction approaches. Section 3 provides an overview of our system architecture and introduces the prediction system. Section 4 presents the learning and inference model for the resource scaling solution proposed based on a layered multi-dimensional HMM/HSMM prediction technique. Section 5 reports our experimental results, and Section 6 discusses future work and concludes this article.

2 Background and Related Works

This section provides a background on stream processing applications, key technologies used for their processing, and the challenges encountered. Also, it discusses some existing resource scaling approaches used in cloud environments.

2.1 Stream Processing Applications

In the past few years, a new class of stream processing applications have emerged in domains such as financial analysis, traffic monitoring, anomaly detection, healthcare, sensor data acquisition and multimedia [2]. These applications allow data produced from heterogeneous, autonomous and a large number of globally-distributed data sources to be aggregated and analyzed dynamically to generate results in real-time. They are usually designed as Data-, Task- or Pipeline-Parallel applications containing structured directed acyclic graphs (DAG), where the vertices represent the operators (i.e., sources or sinks) and the edges are represented as data, task or pipeline streams [15], respectively.

On the one hand, the Data-Parallel application distributes computations involving the data streams to multiple nodes to reduce the computation latency of the application. An example of this type of application is Twitter stream processing application for sentiment analysis. On the other hand, a Task-Parallel application, which requires long running time to complete, is one that distributes computation tasks (processes) to multiple nodes. Task pairs in a Task-Parallel application are executed on different parallel branches of the original stream graph with the output of each task never reaching the input of the other. In comparison to the aforementioned types, the Pipeline-Parallel applications offer low latency and the ability to run stateful operators in parallel by mapping chains of producers and consumers in a stream graph to different nodes.

In this paper, we focus on Data-Parallel stream processing application but the proposed approach can also be employed in other stream processing applications or combination of stream processing application types.

2.2 Key Cloud Technologies for Big Data Stream Processing

Stream processing has been an active area of research for many years. In traditional relational systems, Borealis [16] and STREAM [17] are examples of systems that are designed to provide stream processing middleware and languages for writing streaming applications. Also, System S [18] is used for large-scale, distributed stream processing middleware and application development. These systems are evidence of the past success of the stream processing paradigm. In recent years, new innovative open source, off-the-shelf SPEs such as Spark [5], Storm [4], and S4 [6] have been introduced to support high-volume, low latency stream processing applications and to address needs that are different from the analytical or extract, transform & load (ETL) domain typical of MapReduce [19] jobs.

Apache Storm [4] is a free and open-source distributed real-time computational system commonly used by companies such as Twitter and Spotify. It can process a high volume of data with low latency. Stream processing using

Storm is achieved by distributing data streams across multiple machines as an unbounded sequence of tuples (key-value pairs). A Storm program is defined in terms of spouts and bolts that run on cluster together as a directed acyclic graph (DAG) topologies. A spout is a source of a stream that pulls data from a messaging system such as Kafka [20] and a bolt processes the incoming data stream, which can be passed on to another set of bolts. Apache Storm [4] is known to perform Task-Parallel computations.

Apache Spark [5] is an open-source distributed framework for data analysis. From the onset, Spark supported both batch and streaming workloads, which makes it different from other streaming engines that are designed for only stream processing. Spark performs in-memory analytics on streams in a distributed and real-time manner by discretizing the streaming data into micro-batches before computation. Spark [5] is known to perform Data-Parallel computations. In addition, Yahoo S4 [6] has been proposed as an alternative framework for processing big data stream requiring real-time processing. S4 is a decentralized, scalable and fault-tolerant framework that provides a programming interface for processing data streams on top of a highly available cluster in a distributed computing environment.

SPEs allow for dynamic scaling of cluster resources allocated to applications, however, the required resources must be provisioned at run-time. Also, they do not provide the option of adding or removing VM from clusters when they are no longer needed.

2.3 Challenges of Big Data Stream Processing Applications

For real-time analysis, the majority of aggregated data sources are very dynamic and extremely heterogeneous, i.e., continuously changing data streams with highly unpredictable data arrival patterns, which can differ in accuracy and timeliness. In general, response time is of utmost importance when processing data streams from these sources. This may involve time-unbounded applications that require data to be processed within very short time intervals with low latency requirements, e.g., financial streams, fraud detection. Therefore, low response time is critical and any missed deadline can lead to results that are unusable. In contrast, the applications may involve the execution of complex simulations, often executed as time-bounded processes with medium to high latency requirements, e.g., surveillance and monitoring, smart-traffic management.

In most cases, deadlines of streaming applications can range from milliseconds to hours depending on the particular application constraints. In the face of dynamic workloads, SPEs need to be able to automatically handle the fluctuating demands and scale accordingly without disrupting existing requests. Most existing streaming applications are only allocated a fixed amount of resources at deployment time. Therefore, for streaming applications that rely on real-time stream processing, this may entail a significant service interruption [21] during shutdown or reconfiguration to scale. Additionally, the application state may not be preserved during scaling operations when the system is terminated, and thus, work may be lost, thereby leading to erroneous results [21]. These characteristics coupled with the need to adapt to high throughput efficiency, parallel data processing and dynamic data-driven topology graphs of tasks introduce additional challenges that make resource scaling in the cloud a tedious task for stream processing applications. Therefore, correctly profiling each streaming application and its associated characteristics to forecast the resource demands is crucial when scaling resources for big data streaming applications in the cloud.

2.4 Resource Scaling in the Cloud

Reactive scaling schemes in the cloud are known to allocate resources based on the workload changes detected using threshold-based rules. This approach requires that the application providers be aware of the resource usage patterns in order to set triggers for scaling activities. In practice, most cloud service providers, e.g., Amazon autoscale [22] and Azure [23], describe a set of service provider-defined rules that govern how the services scale up or down at run-time in reaction to changes in traffic. This approach can lead to a large delay before resource is provisioned. To address this limitation, proactive schemes are used whereby resource utilization patterns predicted through forecasting are provided in advance of workload changes and allocations are adjusted accordingly prior to any change.

Various predictive techniques [11], [12], [24], [25], [26], [27] have been proposed for the cloud based on approaches such as time series analysis (e.g., linear regression, autoregression, Neural Networks, and ARIMA), control theory, reinforcement learning and queuing theory. The time series approach [28] incorporates temporal and spatial correlations in order to provide prediction results. An approach based on time series forecasting methodology was proposed by Baldan et. al. [29] to forecast cloud workloads by combining statistical methods, cost function and visual analysis with focus on a short-time horizon. However, no seasonal pattern was found since the analyzed series were non-linear. Control theory [28] has been applied to automate management of resources in various engineering fields, such as storage systems, data centers and cloud computing platforms. The main objective of a controller is to maintain the output of the target system (e.g., performance of a cloud environment) to a desired level by adjusting the control input (e.g., number of VMs). The suitability of control theory approaches depends on the type of controller and the dynamics of the target system.

Queuing theory [28] is used to find the relationship between jobs arriving and leaving a system. It is mostly used for Internet applications, however, it is not an efficient approach when used with elastic applications that have to deal with changing workloads and a varying pool of resources. While reinforcement learning [30] automates scaling tasks without prior knowledge of the application workload, it can lead to scalability problems when the number of states grow. Also, it yields bad performance since the policy has to be readapted as the environment conditions (e.g., workload pattern) change. Vijayakumar et. al. [31] developed an algorithm that can handle dynamic patterns in data arrival to identify overflow or underflow conditions. Different from our work, this is a reactive approach that focuses mainly on CPU adaptation whereby streaming applications are

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2018.2876242, IEEE Transactions on Cloud Computing

implemented as a set of pipelined stages. Here, each stage acts as a server for the next.

Nikravesh et. al. [32] proposed an auto-scaling system based on Hidden Markov Model (HMM). In their research, the states are associated with system conditions and the observation are the sequence of generated scaling actions. Their experiments targeted e-commerce websites, i.e., multi-tier applications, which typically includes a business-logic tier, containing the application logic as well as a database tier. While this approach makes use of HMM, its applicability directly to big data stream processing applications is limited because it does not take into consideration multiple streaming applications, which can lead to ample retraining of the model in response to changes in active streaming jobs, and overfitting of data. In contrast, our approach makes use of a two-layered multi-dimensional Hidden Markov/semi-Markov model; the lower-layer is used for training individual streaming jobs without altering the upper-layer, and whenever a job is not running, the model does not need to be recalculated while the upper-layer is used for training group applications running on clusters. By so doing, the outcome of the upper-layer will be less sensitive because it is based on the collective behaviour of the streaming jobs from the lower-layer that are expected to be well trained. Finally, our framework can accommodate different streaming applications and it is application independent.

Several predictive models have been developed in the past for short-term (bounded) and long-term (unbounded) predictions of performance metrics for VM clusters using machine learning techniques. However, few works exist for big data streaming applications. PLAStiCC [33] uses short term workload and infrastructure performance predictions to actively manage resource mapping for continuous dataflows to meet QoS goals while limiting resource costs. For time-unbounded streaming application, the work in [34] proposed a resource scheduler which dynamically allocates processors to operators to ensure a real-time response. This work focuses on streaming operator scaling while our research is aimed at scaling resources. The authors in [35] proposed Elysium, a fine-grained model for estimating the resource utilization of a stream processing application, which enables the independent scaling of operators and resources using the reactive and proactive approaches. However, they did not consider multiple applications with varying resource bottlenecks.

The use of containers has been exploited recently for scaling stream processing operators. Pahl & Lee [36] reviewed the suitability of container and cluster technology as a means to address elasticity in highly distributed environments. While the use of containers can provide for faster provisioning during scale in/out as compared to VMs, streaming applications will need to be designed as lightweight SOA-style independently deployable microservices to gain this benefit. Moreover, some containers run on VMs instead of bare metal machines and are limited to the amount of VMs provisioned. Hence, VM scaling is required. To the best of the author's knowledge, the application of HSMM to unbounded big data streaming applications is novel.

2.5 Hidden Markov and Hidden Semi-Markov Models

Hidden Markov Model (HMM) [14] is a widely employed and powerful statistical tool for modeling and analyzing sequential data. HMMs have been extensively used in many applications, such as speech recognition, video activity recognition, and gesture tracking.

In the single-layer, one-dimension HMM model, an observation O_t at time t is produced by a stochastic process, but the state $q_t = S_i$ of this process cannot be directly observed, i.e., it is hidden. This hidden process is assumed to satisfy the Markov property, where state q_t at time t depends only on the previous state, S_{t-1} at time $t-1$.

The HMM is described as a process evolving over discrete time intervals. Let $S = \{S_t, \quad t = 1, ..., T\}$ denote a sequence of hidden states, drawn from a finite state space of size N and let $O = \{O_t, \quad t = 1, ...T\}$ denote a corresponding set of observable states with M discrete possible observation symbols $O = O_1, O_2, ...O_M$. The standard HMM has the form:

$$P(S, O) = P(S_o) \prod_{t=1}^{T} P(S_t|S_{t-1}) P(O_t|S_t) \tag{1}$$

where $P(S_o)$ is the initial probability distribution denoted as π, $P(S_t|S_{t-1})$ is the transition probability distribution of moving from state $S_{t-1} = i$ to state $S_t = j$, where $\{i, j \in 1, .., N\}$, denoted as A, i.e., $N \times N$ matrix and $P(O_t|S_t)$ is the observation probability distribution denoted by B, i.e., $N \times M$ matrix. Hence the model $\lambda = (\pi, A, B)$.

State duration in HMM is usually characterized by a probability distribution function $P_i(d)$, e.g., in state i, the value of $P_i(d)$ is the probability of remaining in state i for d time units. Given the Markov assumption and from probability theory, $P_i(d)$ is the product of all the probabilities:

$$P_i(d) = a_{ii}^{d-1}(1 - a_{ii}) \tag{2}$$

where a_{ii} is the self-loop probability of state i. $P_i(d)$ is a geometrically decaying function of d and when applied directly to resource usage prediction of big data streaming applications, the probability that the underlying process remains in a resource usage state i for d time units is a geometrically decaying function of d.

HSMM [37] is an extension of HMM designed to remove the constant or geometric distribution of state durations assumed in HMM by introducing the concept of explicit state durations. The HSMM is similar to HMM in which an embedded Markov chain represents the transitions between the hidden states. However, it incorporates a discrete state occupancy distribution $D = P_i(d)$, representing the duration in non-absorbing states, i.e., transient states. The HSMM model is specified as $\lambda = (\pi, A, B, D)$, where D models the time spent in a particular state.

In the HSMM, each state variable has variable time intervals and a sequence of observations $O_{t_1}, ..., O_{t_2}$ are generated while in the state as opposed to a single observation O_t in the HMM. The length of the observation sequence for each state is determined by the duration spent in each state.

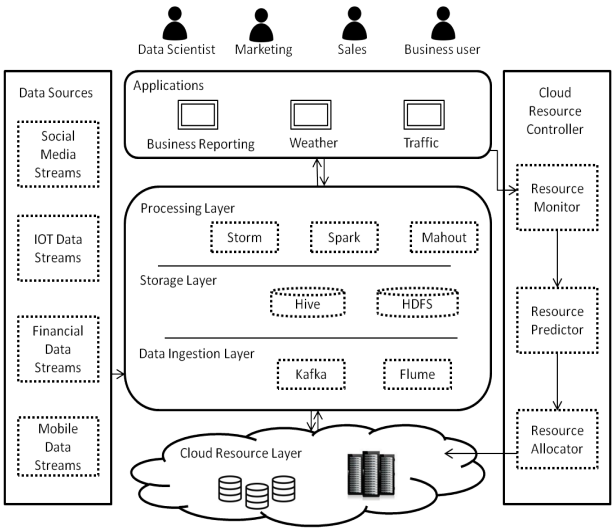

Fig. 1. Proposed Architecture.

3 System Architecture

We first introduce our system architecture of the streaming environment as shown in Fig. 1. We consider a big data application service provider (ASP) that provides big data stream processing services for cloud users in a scalable distributed environment. The applications collect data from a wide variety of sources, and based on user requirements, integrate them to detect interesting events and patterns. Our architecture includes a data ingestion layer, a processing layer and a storage layer. It also incorporates a prediction system for inferring scaling decisions. Fig. 1 shows the overall system architecture.

The ingestion layer, which provides services such as Flume [38] and Kafka [20], facilitates access to heterogeneous data sources for each streaming event while concealing the technical facets of the data streams. The ingestion layer is capable of integrating data streams from multiple sources as required by each streaming event. The processing layer, which contains the streaming engines such as Spark [5] and Storm [4], supports the analysis of the streamed data in order to identify meaningful patterns in the data streams and trigger actions. The storage layer supports record ordering and strong consistency to enable fast reads and writes of large streams of data whenever required. Storage services, e.g., HDFS [39], in this layer may serve as an intermediate or final storage point for stream analysis results.

To support resource scaling, present within our architecture is also a cloud resource controller that is made up of a resource monitor, a predictor and a resource allocator. The resource monitor is responsible for real-time monitoring of streaming applications at each time interval. It provides system-wide visibility into streaming application characteristics and workload changes. Several tools can be used to monitor applications running on a cluster, e.g., Ganglia [40]. Information gathered from these tools are passed on to the resource predictor to create a model, and evaluation result from the predictor can be used by the resource allocator to perform scaling operations for running applications. For big data stream processing applications running on a cluster of virtual machines, the resource prediction model needs to be robust enough to represent the typical variations in the underlying streaming jobs. Section 3.1 provides details pertaining to the resource predictor.

3.1 Prediction System

In this section, we present our frameworks for handling the resource prediction of big data streaming applications, whereby resource needs are forecasted in advance by monitoring each streaming job and its associated characteristics. Our solution combines knowledge of stream processing with statistical Markov models. The key idea of our approach is to decouple the problem space into layers, whereby each streaming job is independently modeled as a multi-dimensional HMM/HSMM at the lower-layer and the generated observations are concatenated afterwards at the upper-layer.

We hereby extend the standard HMM and HSMM to a layered multi-dimensional HMM/HSMM for the resource predictor. The layered approach allows for the decomposition of the parameter space into distinct layers, whereby each layer is connected to the next by inferential results. In our case, we make use of a two-layered model (upper-layer and lower-layer) as shown in Fig. 2 where each streaming application can be modeled as either a multi-dimensional HMM or multi-dimensional HSMM at the lower layer. The lower-layer allows for the abstraction of individual streaming jobs' observations at particular times and also ensures independence from others while the upper-layer is used to predict the resource usage pattern for the group streaming applications running on a cluster.

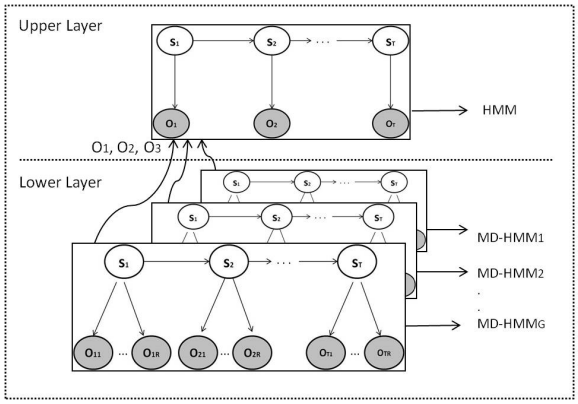

Fig. 2. A Layered Multi-dimensional HMM.

4 Proposed Prediction Model

4.1 The Lower-Layer

The lower-layer denotes the resource utilization for each individual streaming job (e.g., traffic analysis, weather analysis, and anomaly detection), where each streaming job's HMM/HSMM has more than one observable symbols at each time t. We make use of the multi-dimensional HMM (MD-HMM) or multi-dimensional semi-HMM (MD-HSMM) at the lower-layer to represent individual streaming job's resource usage pattern, whereby the system bottlenecks (e.g., CPU, memory) being considered form the

TABLE 1
Notations used in the LMD-HMM & LMD-HSMM models

Variables	Definitions
N	Number of hidden states in the model representing the resource usage patterns
R	Number of observable dimensions for each streaming job
M	Number of distinct observation symbols, which describes the set of states for individual streaming job's dimension
A	Hidden state transition probability distribution for each streaming job
B	Set of observable probability distribution matrices
O	$O = \{O_1, O_2, ..., O_T\}$ Observation sequence
S	$S = \{q_1, q_2, ..., q_T\}$ State sequence
D	Time spent in a particular state
$b_j(O_k^r)$	Probability of emitting observation symbol O_k^r in state j for each dimension r
π	Initial probability of distribution
$\alpha_t(i)$	Forward variable that denotes the probability of generating $O_{1:T}$ and ending in state S_i at time t
$\beta_t(i)$	Backward variable that denotes the probability of generating $O_{t+1:T}$ and begins in state S_i at time t
$\gamma_t(i)$	Probability of being in S_i at time t, given the observation sequence $O_{1:T}$
$\xi_t(i,j)$	Conditional the probability of being in state S_i at time t and in state S_j at time $t+1$, given the observation sequence $O_{1:T}$
$\bar{\lambda_g}$	Re-estimated parameters of the HMM/HSMM model for each streaming job

dimensions to the process. The observations from the submodel MD-HMM/MD-HSMM corresponding to each streaming job from the lower layer are then concatenated afterwards into a new single layer HMM at the upper-layer. Next, we discuss estimation and learning algorithms for the models. We list in Tbl. 1 important notations used in this paper.

4.2 MD-HMM Inference & Learning Mechanisms

The parameters in our multi-dimensional HMM can be defined as follows:

- N: Number of states associated with the resource usage patterns for each streaming job. This is denoted as $S = \{S_1, S_2, ..., S_N\}$, and the state at time t as q_t.
- R: Number of observable dimensions for each streaming job. In our model, system bottlenecks such as CPU and memory utilization are dimensions in the process.
- M: Number of distinct observation symbols in each state for individual streaming job's dimension. Hence, for a given streaming job, in state $q_t = S_i$, there will be $M \times R$ distinct observations, where $O^r = \{O_1^r, O_2^r, ..., O_M^r\}$ are individual observation sequences for each dimension $1 \leq r \leq R$. For simplicity, we assume that the number of states for each dimension is the same.
- A: Hidden state transition probability distribution for each streaming job $A = \{a_{ij}\}$, where $a_{ij} = P\{S_{t+1} = S_j \mid q_t = S_i\}, \quad S_i, S_j \in S, \quad 1 \leq i, j \leq N$, described by the characteristics of individual streaming job at time t.
- **B**: A set of observable probability distribution matrices, where $b_j(O_k^r)$ is the probability of emitting observation symbol O_k^r in state j for each dimension r, i.e., $b_j(O_k^r) = P[O_t = O_k^r | q_t = S_j]$ for $1 \leq k \leq M$ and $1 \leq r \leq R$. In the multi-dimensional HMM, there will be R B-matrices, i.e., **B** = $\{B_1, B_2, ..., B_R\}$, where each B_r characterizes the stochastic distribution for each dimension of the individual streaming job.
- π: Initial probability of distribution $\pi_i = P(q_1 = S_i), \quad 1 \leq i \leq N$.

Given these parameters, a multi-dimensional HMM (MD-HMM) can be used to derive the sequence of observed symbols $O = \{O_1, O_t, ..., O_T\}$, where $O_t = \{O_t^1, O_t^2, ..., O_t^R\}$.

The complete set of the model is represented as $\lambda = (N, R, M, A, B, \pi)$. The MD-HMM model for each streaming job $\lambda_g = (N_g, R_g, M_g, A_g, B_g, \pi_g)$, where $g = 1, ..., G$, can be used to solve three basic problems (i.e., evaluation, decoding and training) that arise:

- How to compute the probability $P(O|\lambda_g)$ of the observation sequence $O = \{O_1, O_2, ..., O_T\}$ given the model λ_g. This problem corresponds to evaluation, which can be solved using the Forward algorithm.
- How to compute the corresponding state sequence that best explains the observation. This is a decoding problem in which the Viterbi algorithm [14], a dynamic algorithm that computes both the maximum probability and the state sequence given the observation sequence, is used.
- Given the observation O, how to find the model λ_g that maximizes $P(O|\lambda_g)$. This problem corresponds to learning to determine the model that best fits the training data. The Baum-Welch algorithm [41], which can be extended to reflect the multi-dimensional properties based on the independence assumption, can be used for solving this problem. The learning problem adjusts the model parameter λ_g, such that $P(O|\lambda_g)$ is maximized.

For simplicity, we will drop the subscript g thereafter since we are focusing on individual streaming job at the lower layer.

Given a model λ, an observation sequence $O = \{O_1, O_2, ..., O_T\}$, and a state sequence $Q = \{q_1, q_2, ..., q_T\}$ where $q_t = S_i$ at time t, all the parameters for solving the problems depend on the forward-backward variables. The forward variable $\alpha_t(i)$ is the joint probability of the partial observation sequence $O_1, O_2, ..., O_t$, given the hidden state at time t is $q_t = S_i$. The backward variable $\beta_t(i)$ is the joint probability of the partial observation sequence $O_{t+1}, O_{t+2}...O_T$, given the hidden state at time t is $q_t = S_i$.

The forward procedure is a recursive algorithm for calculating $\alpha_t(i)$ for the observation sequence of increasing length t. In our framework, since each observed dimension is assumed to be independent, the forward variable $\alpha_t(i)$ can be modified and the output for each state i recomputed

as the Cartesian product of the observation probabilities of each dimension as follows:

$$\alpha_t(i) = P(O_1, O_2...O_t, q_t = S_i|\lambda) \tag{3}$$

$$\alpha_1(i) = \pi_i \prod_{r=1}^{R} b_i(O_1^r), \quad 1 \le i \le N \tag{4}$$

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i)a_{ij}\right] \prod_{r=1}^{R} b_j(O_{t+1}^r), \quad 1 \le t \le T-1, \quad 1 \le j \le N \tag{5}$$

where $\alpha_t(i)$ is the probability of the partial observation sequence $O_1, O_2, ..., O_t$ with state $q_t = S_i$, given the model λ and $\alpha_{t+1}(j)$ is the probability of the partial observation sequence $O_1, O_2, ..., O_t, O_{t+1}$ at state S_j at time $t+1$.

Likewise, recursion described in the forward algorithm can as well be used backwards in time in the backward algorithm. The backward variable can be recomputed as:

$$\beta_t(i) = P(O_{t+1}, O_{t+2}...O_T|q_t = S_i, \lambda) \tag{6}$$

$$\beta_T(i) = 1, \quad 1 \le i \le N \tag{7}$$

$$\beta_t(i) = \left[\sum_{j=1}^{N} a_{ij}\beta_{t+1}(j)\right] \prod_{r=1}^{R} b_j(O_{t+1}^r), \quad 1 \le t \le T-1, 1 \le i \le N \tag{8}$$

where $\beta_t(i)$ is the probability of the partial observation sequence $O_1, O_2, ..., O_T$ from $t+1$ to the end, given the state $q_t = S_i$ at time t and the model λ. The state transition probability distribution from S_i to S_j is a_{ij}, and $b_j(O_{t+1}^r)$ is the observation symbol probability distribution of the occurrence of observable value (O_{t+1}^r), given state S_i at time t. b_j is normalized based on the outputs of each dimension. The observation evaluation is then:

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_t(i)\beta_t(i), \quad \forall t. \tag{9}$$

especially the forward variable as defined in Equation 3,

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i) \tag{10}$$

To address the training problem of finding the optimum model parameter vector $\lambda \in \Lambda$ that maximizes $P(O|\lambda)$, given an observation sequence O, the Baum-Welch algorithm [41] can be extended for the MD-HMM in terms of the joint events and state variables, respectively:

$$\begin{aligned}\xi_t(i,j) &= P(q_t = S_i, q_{t+1} = S_j|O, \lambda) \\ &= \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{P(O|\lambda)} \\ &= \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^{N} \alpha_T(i)}\end{aligned} \tag{11}$$

$$\begin{aligned}\gamma_t(i) &= P(q_t = S_i|O, \lambda) \\ &= \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N} \alpha_T(i)}\end{aligned} \tag{12}$$

$\xi_t(i,j)$ is the probability of being in state S_i at time t and in state S_j at time $t+1$, given the observation sequence $O_1, O_2, ..., O_T$ and $\gamma_t(i)$ is the probability of being in S_i at time t.

The variables $\alpha_t(i)$, $\beta_t(i)$, $\xi_t(i,j)$, $\gamma_t(i)$ are found for every training sample and then, re-estimation is performed on the accumulated values. To generate the new better model $\bar{\lambda}$, the state transition, observation symbol and initial state probabilities can be updated as follows:

$$\bar{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}, 1 \le i \le N, 1 \le j \le N \tag{13}$$

$$\bar{b}_j^r(k) = \frac{\sum_{t=1, O_t^r = O_k^r}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}, \quad 1 \le r \le R, 1 \le j \le N, 1 \le k \le M \tag{14}$$

$$\bar{\pi}_i = \gamma_1(i), 1 \le i \le N \tag{15}$$

A set of new model parameters $\bar{\lambda} = (\bar{A}, \bar{B}, \bar{\pi})$, which are the average of the result from each individual training sequence, is obtained after the update and the process is repeated until changes in the parameters are smaller than a predefined threshold, i.e., optimal is reached.

In the testing phase, a test sequence is feed to each of the MD-HMM and the maximum likelihood given the observation is evaluated using the Viterbi algorithm [14]. The Viterbi algorithm is used to obtain the probable sequences using the forward item at time t in state i based on the output of the learned parameters from training.

4.3 MD-HSMM Inference & Learning Mechanisms

In this section, we extend the standard HSMM in order to address the need to model multiple resource bottlenecks. HSMM is very useful in applications that run unbounded for longer periods to accommodate shifts in behaviour from the HMM-based model. This model relaxes the duration assumption on resource usage states for big data streaming applications. To define the parameters for the MD-HSMM, duration is included in addition to the parameters defined for the MD-HMM in Section 4.2 as:

- D: The probability distribution matrix of remaining in state i for d time intervals, i.e., $P_i(d)$.

We defined the lower-layer model MD-HSMM as $\lambda = (\pi, N, R, M, A, B, D)$ to reflect the duration of resource usage states for each streaming application, i.e., each streaming application is modeled as a multi-dimensional hidden semi-markov (MD-HSMM) at the lower-layer. Therefore, given the observable sequence in the past T duration, we need to find its most likely state sequence S^* and the most likely state model MD-HSMM λ^* for each streaming application. This involves solving the basic problems of evaluation, recognition and training as described next.

The process of training and recognition involves redefining the forward and backward variables. The addition of duration in the model makes the algorithm more complex than the HMM-based model. We define a forward variable

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2018.2876242, IEEE Transactions on Cloud Computing

$\alpha_t(i)$ as the probability of generating $O_1, O_2, ..., O_t$ and ending in state i as defined in Eqn 3. Assuming that the first state begins at time 1 and the last state ends at time T, the initial conditions for the forward recursion formula become:

$$\alpha_1(i) = 1, \quad (16)$$

$$\alpha_t(j) = \sum_{i=1}^{N}\sum_{d=1}^{D}\alpha_{t-d}(i)a_{ij}P_j(d)\prod_{r=1}^{R}\prod_{s=t-d+1}^{t}b_j(O_s^r) \quad (17)$$

Assuming $\alpha_{t-d}(i)$ has been determined where the duration in state j is d, a_{ij} is the state transition probability from state i to state j, $b_j(O_s^r)$ is the output probability of the observation O_s^r for each dimension from state i and $P_j(d)$ is the state duration probability of state j. Also, N is the number of states in the MD-HSMM and D is the maximum duration in state i.

Likewise, the backward variable can be written as Eqn 6 and the value for the initial conditions of the backward recursion formula $t = T$ is as written in Eqn 7. Summing over all the states N and all possible states duration d, we have the recurrence relations:

$$\beta_t(i) = \sum_{i=1}^{N}\sum_{d=1}^{D}a_{ij}P_j(d)\beta_{t+d}(j)\prod_{r=1}^{R}\prod_{s=t+1}^{t}b_j(O_s^r) \quad (18)$$

We then define three variables to obtain the re-estimation formulae for the MD-HSMM as follows:

$$\alpha_{t,t'}(i,j) = P(O_1^{t'}, q_t = S_i, q_{t'} = S_j|\lambda) \quad (19)$$

$$\Omega_{t,t'}(i,j) = \sum_{d=1}^{D}[P(d = t' - t|i) \cdot P(O_{t+1}^{t'}|q_t = S_i, q_{t'} = S_j, \lambda)] \quad (20)$$

$$\xi_{t,t'}(i,j) = P(q_t = i, q_t' = S_j|O_1^T, \lambda) \quad (21)$$

where $\alpha_{t,t'}(i,j)$ is the probability of the partial observation sequence and state i at time t and state j at time t', $(t' = t+d)$, $\Omega_{t,t'}(i,j)$ is the mean value of the probability of being in state i for d time intervals and moving to state j and $\xi_{t,t'}(i,j)$ is the probability of being in state i for d time intervals and moving to state j given $O = O_1, O_2, .., O_T$.

$\alpha_{t,t'}$ can be described in terms of $\Omega_{t,t'}(i,j)$ as:

$$\alpha_{t,t'}(i,j) = P(O_1^{t'}, q_t = S_i|\lambda) \cdot P(O_{t+1}^{t'}, q_t = i, q_{t'} = j|O_1^t, q_t = i, \lambda) \quad (22)$$

$$= \alpha_t(i)a_{ij}\Omega_{t,t'}(i,j) \quad (23)$$

and the relationship between $\alpha_t(i)$ and $\alpha_{t,t'}(i,j)$ is given as:

$$\alpha_{t'}(j) = P(O_1^{t'}, q_{t'} = S_j|\lambda) \quad (24)$$

$$= \sum_{i=1}^{N}\sum_{d=1}^{D}[P(d = t' - t|S_j)\alpha_{t,t'}(i,j) \quad (25)$$

The $\xi_{t,t'}(i,j)$ can be derived as follows:

$$\xi_{t,t'}(i,j) = \frac{\sum_{d=1}^{D}\alpha_t(i)a_{ij}\Omega_{t,t'}(i,j)\beta_{t'}(j)}{\beta_0(i = s_0)} \quad (26)$$

where s_0 is the initial resource usage state. Given the initial value of the model λ^0, the Expected-Maximization (EM) algorithm is used to find the best model λ^* using the recursive computing procedure. The re-estimation formulas for the MD-HSMM can be written as follows:

- The re-estimation for the initial distribution is the probability that state i was the first state given O.

$$\bar{\pi}_i = \frac{\pi_i[\sum_{d=1}^{D}\beta_d(i)P(d|i)b_j(O_1^d)]}{P(O_1^T|\lambda)} \quad (27)$$

- The re-estimation of state transition probabilities is the ratio of the expected number of transitions from state i to state j to the expected number of transitions from state i.

$$\bar{a}_{ij} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{j=1}^{N}\sum_{t=1}^{T}\xi_{t,t'}(i,j)} \quad (28)$$

- The re-estimation formula for the observation distribution is the expected number of times that observation $O_t = v_k$ occurred in state i, normalized by the expected number of times that any observation occurred in state i.

$$\bar{b}_i(k) = \frac{\sum_{t=1}^{T}\alpha_t(i)\left[\frac{\Omega_{t,t'}(i,j)}{\sum_{d=1}^{D}P(d=t'-t|i)}\right]\beta_t(i)}{\sum_{t=1}^{T}\alpha_t(i)\left[\frac{\Omega_{t,t'}(i,j)}{\sum_{d=1}^{D}P(d=t'-t|i)}\right]\beta_t(i)} \quad (29)$$

The optimal states S^* and model λ^* are then continuously updated by replacing λ with λ^* if it gives a higher probability, until a limit is reached. The limit represents the number of iterations and signifies when to stop the re-estimation or find the probability of observing the sequence O cannot be improved beyond a certain threshold.

A problem with duration of the HSMM-based model is the large number of parameters associated with each state. This makes the re-estimation problem more difficult than the HMM-based model. To alleviate this, we make use of a parametric state duration density instead of the non-parametric duration density that require less training. We adopt the Gaussian distribution method. The theorem states that the mean of any set of variates with any distribution having a finite mean and variance tends to the Gaussian distribution. The mean $\mu(j)$ and the variance $\sigma^2(j)$ of the duration of state j can be determined as follows:

$$\mu(j) = \frac{\sum_{d=1}^{D}\{\sum_{t=1}^{T-d}(\sum_{t=1}^{N}\alpha_t(i)a_{ij})P_j(d|j)b_j(O_{t+1}^{t+d})\beta_{t+d}(j)\}d}{\sum_{d=1}^{D}\sum_{t=1}^{T-d}(\sum_{t=1}^{N}\alpha_t(i)a_{ij})P_j(d|j)b_j(O_{t+1}^{t+d})\beta_{t+d}(j)} \quad (30)$$

$$\sigma^2(j) = \frac{\sum_{d=1}^{D}\{\sum_{t=1}^{T-d}(\sum_{t=1}^{N}\alpha_t(i)a_{ij})P_j(d|j)b_j(O_{t+1}^{t+d})\beta_{t+d}(j)\}d^2}{\sum_{d=1}^{D}\sum_{t=1}^{T-d}(\sum_{t=1}^{N}\alpha_t(i)a_{ij})P_j(d|j)b_j(O_{t+1}^{t+d})\beta_{t+d}(j)} - \mu^2(j) \quad (31)$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2018.2876242, IEEE Transactions on Cloud Computing

4.4 The Upper-Layer

Since the lower-layer does not model the relationships between the streaming jobs, the outputs from the lower layer, which are expected to be well-trained are used as the new observation O' sequence for the upper-layer. Therefore, devising the right mechanism for determining how the outputs from the lower-layer will be used as input features into the upper-layer becomes one of the issues. We hereby discuss our approach to this problem.

Suppose G MD-HMMs or MD-HSMMs are trained at the lower-layer, with $g = \{1, .., G\}$. To transform the lower layer MD-HMM/MD-HSMM into a single HMM/HSMM at the upper layer, the likelihood of observing the g^{th} MD-HMM/MD-HSMM in state i is used in the recursion formula of the upper layer HMM/HSMM to pass the probability distribution up. More precisely, the predicted states from the trained model λ^* from the lower layer MD-HMM/MD-HSMM becomes the input observations for the upper layer HMM/HSMM.

We define the observation sequence from the lower-layer as $O' = O'_{1:T}$, where T is the length of the observable sequence. We also define the probability $\rho(i,t) = \alpha(i,t)$ of being in state i at time t for each model, where $\alpha(i,t)$ is the forward variable of the Baum-Welch algorithm [41] for the observed sequence $O'_{1:T}$, with $\alpha(i,t) \triangleq P(O'_{1:T}, q_t = i)$. We normalize the probability $\rho(i,t)$ for all states of all the models, i.e.,

$$\sum_{j=1}^{N_s} P(q_t = j) = 1 \tag{32}$$

where N_s is the number of all states for all models. Then, the probability $P(q_t = i|O'_{1:T})$ of state i given a sequence $O'_{1:T}$ is:

$$P(q_t = i|O'_{1:T}) = \frac{P(q_t = i, O'_{1:T})}{P(O'_{1:T})} \tag{33}$$

$$= \frac{P(q_t = i, O'_{1:T})}{\sum_{j=1}^{N_s} P(q_t = j, O'_{1:T})} \tag{34}$$

$$= \frac{\rho(i,t)}{\sum_{j=1}^{N_s} \rho(j,t)} \tag{35}$$

where i is the state in the model, which is a subset of the states of all models, and N_s is the total number of states.

The observations at the upper-layer is the probability distribution, $\{P(1:G)_1, .., P(1:G)_T\}$, for each of the lower-layer model λ_g for the time interval, i.e., a vector of G reals for each time interval, will be used as input to the upper-layer. The upper-layer HMM/HSMM is then used to complete the probability of a certain state as a function of time given the observation sequence produced by the lower layer MD-HMMs/MD-HSMMs.

At the upper- layer, a single HMM/HSMM can then be used to model the resource usage of group applications running on the cluster, where each state in the HMM/HSMM corresponds to a resource scaling action. For instance, the HMM for the upper-layer is formally defined as $\lambda' = (N', M', A', B', \pi')$, where N' describes the resource scaling actions (which can be scaling up, down or no scaling), M' is the set of observation symbols for the resource demand states learned from the lower-layer observations, π' is the initial state distribution, A' is the state transition probability distribution matrix and B' is observable probability distribution matrix. With this new model, training can be done using the same learning and inference mechanisms used for HMM. In the next section, we implement our proposed models, making use of the modified variables and parameters.

5 Experiments

The focus of our experiment is to validate the LMD-HMM and LMD-HSMM models. We first describe the experimental setup including the environment and datasets. To verify the flexibility and general applicability of the proposed models on predicting the resource usage for big data streaming applications, the framework was tested on applications developed to access Twitter API due to lack of big data streaming benchmarks and data repository that fits into our problem. Apache Spark was chosen as the stream processing engine due to its flexibility of deployment over available machines.

5.1 Experimental Setup

The Spark cluster was setup on a physical machine with 8 core i7-3770 CPU @ 3.40 GHz Intel processors and 16GB of RAM. Two VM instance types were considered with configurations of 4GB and 8GB RAM with 2VCPUs each. Each node was installed with Hadoop 2.6, Spark 1.6, Scala 2.11 along with JDK/JRE v1.8. One node was configured as the master while the other nodes were configured as slaves. The master node was equipped with a 195GB SSD drive and the slave nodes with 50GB SSD drives each.

Hadoop was configured to make use of HDFS for providing storage for the streaming data and Apache Spark was configured to run on top of Hadoop since it does not provide an independent storage system. The streaming data was collected using Apache Flume, a distributed system for collecting, aggregating data and transporting the events generated. At each interval, an amount of data is downloaded and persisted in HDFS for immediate processing by the Spark engine. Two streaming application programs (Tophashtags and Sentiment Analysis) were used in evaluating the prediction model. The developed applications were launched at different time intervals on the master node and monitored. Fig. 3 shows a snapshot of application submission to the Spark Master VM. The workload arrival rates, number of streams and a mix of system performance metrics (CPU, memory) for each streaming applications were collected and stored for further analysis. A snapshot of the multi-dimensional resource usage (CPU utilization & Memory) for one of the applications recorded at 1 min intervals is presented in Fig. 4.

5.2 LMD-HMM Implementation

To implement the LMD-HMM model, our goal is to first evaluate the lower-layer MD-HMMs with respect to each streaming job. The collected datasets were partitioned into

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2018.2876242, IEEE Transactions on Cloud Computing

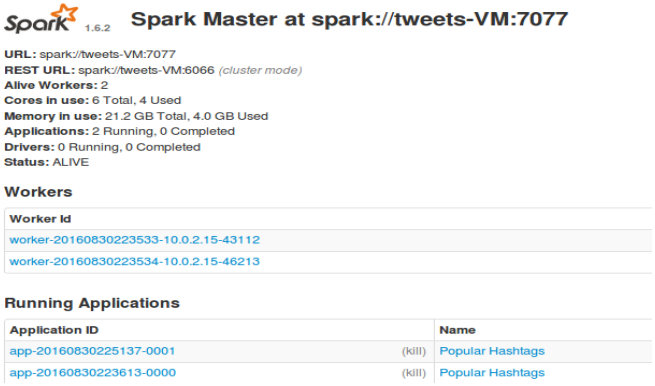

Fig. 3. Big Data Streaming Applications

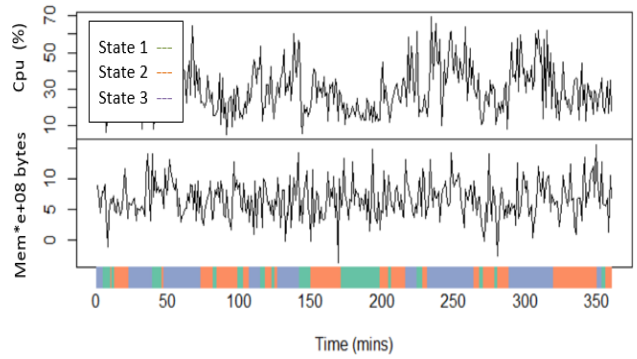

Fig. 4. Multi-dimensional (CPU & Memory) Observation for a stream

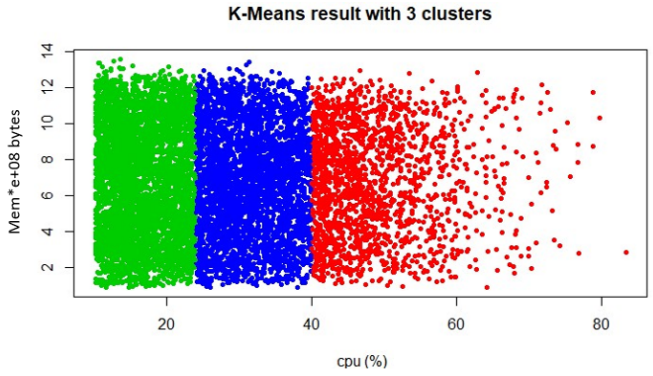

Fig. 5. K-Means using CPU & Memory Values

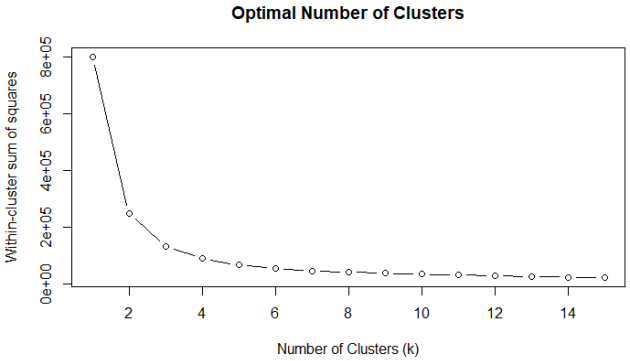

Fig. 6. K-Means Optimal Number of Clusters

two sets: a training set and a prediction set. After the models have been processed using the training set, we tested them against the prediction sets to determine their accuracy. The results from the MD-HMM were then used as input features to the upper-layer HMM to make the final resource scaling decision. For training and testing, we make use of RStudio, a statistical software for R.

5.2.1 *Training the MD-HMMs*

We employ our modified Baum-Welch expectation maximization algorithm to fit the model when the dataset is not trained. When the prediction model is trained, the Viterbi algorithm can be applied to determine the hidden states. The objective of the training algorithm is to optimize the MD-HMM parameters such that the overall training sequence likelihood is maximized or until convergence is reached. We observed the resource usage for the streaming applications for a period ranging from a few minutes up to $60hrs$.

The optimal number of states for the MD-HMMs is determined using the K-means clustering algorithm to find the initial parameters of the distributions belonging to each state. It is worth noting that our choice to employ the K-means algorithm was due to its simplicity, however, any clustering algorithm can be used. The K-means algorithm grouped the multi-dimensional data into k clusters during the clustering process using the data points as shown in Fig. 5, where each colored cluster represents the resource usage states. From the elbow graph in Fig. 6, it can be seen that the location of a bend in the graph is between 2 to 4 points, which is an indicator of the appropriate number of clusters. Considering the within-cluster sum of squares (SS), we selected 3 states from the result of the clusters as determined by the K-means algorithm. Each cluster contains a pair of values representing the mean and data points that belong in the cluster.

We use the k cluster value derived from the K-means clustering as input to the training algorithm and set the initial parameters as shown in Tbl. 2. Since it starts from a randomly initialized MD-HMM, the algorithm is run several times with different initializations and the best model is kept for prediction.

- We set states in the initial probability distribution matrix as $\pi = [0 \quad 0 \quad 1]$.
- In our 3-states model, there will be nine transition probabilities A.
- For the observation probabilities, we use Gaussian mixture distribution. We define the mean μ and variance σ^2 for each dimension of the model.

With the help of the defined variables and the re-estimation formula, we then find the best MD-HMM model λ^*. The modified Baum-Welch algorithm is used to train the model to produce the state probabilities, transition state probabilities, and observation mean and variances. The Baum-welch estimates the model such that the log-likelihood probability of generating the observation sequence O is maximized. The model is trained by varying the initial parameters (λ) with multiple runs to obtain the

TABLE 2
Probabilities Matrix

Transition States (A)		
0.858	0.142	0.000
0.053	0.926	0.021
0.013	0.019	0.968
Observation (μ_1) - cpu (%)		
16.877	26.351	38.981
Observation (μ_2) - mem $(b) * e + 08$		
6.795	6.889	6.986
Variance (σ_1^2) - cpu (%)		
18.883	76.428	214.096
Variance (σ_2^2) - mem $(b) * e + 08$		
9.657	8.961	8.716

updated parameters ($\bar{\lambda}$). Fig. 7 shows that the log-likelihood achieves its maximum at about 3 iterations.

Determining the best model is a model selection problem tackled to prevent overfitting of data. Since there is no simple, theoretically correct way of making model choices, techniques like the Bayes Information Criterion (BIC) can be used for determining the best model. The BIC uses the likelihood as a measure of fit and penalizes complexity (increased number of states) as a means of mitigating the risk of overfitting. Also, BIC takes the number of parameters into account when determining the best model. The graph of BIC values obtained with different choices of k states will show increase with k to a certain point before reaching a plateau. The smallest k for which the BIC no longer increases signals a better model. To select the best model, we trained a series of candidate MD-HMM with an increasing number of states (k).

5.2.2 Predicting the MD-HMMs

This step involves computing the current state of the system and using this to predict the future state. Here, we try to generate the sequence of states responsible for producing the observation sequence. After training the MD-HMMs for each streaming job, the Viterbi algorithm [14] is used to compute the most likely sequence of hidden variables which could have generated our observations. We use as inputs values (initial, transition and observation) from the defined MD-HMM model and expect the list of states as outputs. In order to determine the effectiveness of our approach in predicting workloads of varying sizes, we varied the training set period and the prediction period as shown in Tbl. 3 to determine the effect that changes in training and prediction periods will have on the accuracy of the result. The MAPE and RSME values indicate the level of correlation between predicted and referenced durations for the test data. As shown in the table, the accuracy of the result decreases as the prediction period increases, which indicates greater accuracy for shorter prediction intervals. Therefore, the MD-HMM model is ideal for predicting accurately the resource usage patterns for time-bounded streaming applications with shorter durations of up to 4hrs.

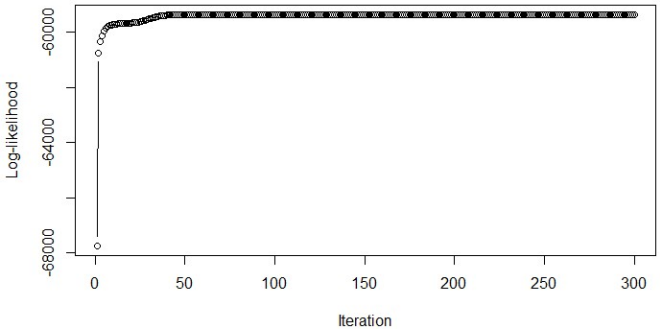

Fig. 7. Log-likelihood of the MD-HMM Model

TABLE 3
Prediction Set Size Variation & Accuracy of the MD-HMM

Experiment Period	Prediction Period	RSME (%)	MAPE (%)
24hrs	4hrs	88.7	91.2
24hrs	6hrs	88.2	89.0
24hrs	8hrs	86.7	86.0
48hrs	8hrs	88.6	89.0
48hrs	12hrs	86.3	86.5

5.2.3 Validating the MD-HMMs

The accuracy of our model is determined by how well the model performs on new datasets that were not used when fitting the model. We evaluated the prediction accuracy of the MD-HMM models based on two performance indexes, which are the Mean Absolute Percentage Error (MAPE), a commonly used scale-independent measure, and the Root Mean Square Error (RSME), a scale-dependent measure to determine the % error in our model. MAPE is defined as the sum of the absolute deviation by the total of the predictions. The calculation is written as:

$$MAPE = \frac{1}{n}\sum\left(\frac{|Actual - Forecast|}{|Actual|} * 100\right) \quad (36)$$

This was calculated for the next forecast interval in time divided by the number of interval.

The RSME is calculated as:

$$RSME = \sqrt{\sum_{t=1}^{n}\frac{(Actual - Forecast)^2}{n}} * 100 \quad (37)$$

5.2.4 Upper-layer HMM

The upper-layer HMM is trained on the outputs from the lower-layer MD-HMMs using the normalized probabilities of the states of the MD-HMMs as input. We compute the A' and B' matrices, which are also trained using the Baum-Welch algorithm [41] and the most probable states determined using the Viterbi algorithm [14]. The graph in Fig. 8 shows a prediction scenario where the x-axis indicates an interval of 1 min for a 6hr prediction period depicting the accuracy of the prediction system for highly variable workload and the y-axis shows the resource scaling states, (i.e., 1-down, 2-no scaling, and 3-up).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2018.2876242, IEEE Transactions on Cloud Computing

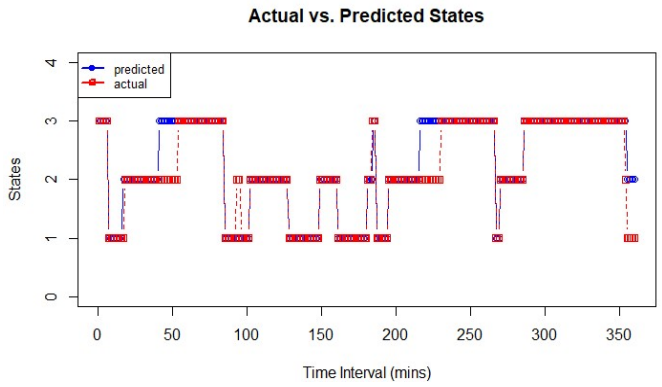

Fig. 8. Actual vs. Predicted States

5.3 LMD-HSMM Implementation

In big data streaming applications, measurements of real-time streaming workloads often indicate a significant amount of workload variability is present over time. The variability is as reflected in the off-peak and peak periods that often characterizes the workload arrival. The exhibited temporal patterns can have significant impact on the resources required by streaming applications, hence, demand resources be scaled up or down accordingly. A major advantage of using the HSMM-based model is in its capability of making predictions for longer periods. The conditional independence in the HSMM-based model is ensured when the process moves from one distinct state to another. In this section, we use the LMD-HSMM to evaluate the resource usage states of big data streaming applications that run for a longer period of time.

5.3.1 Training the MD-HSMM

In this model, we also classify the resource prediction process into 3 states. We ensure the initial value of D is sufficiently large enough to cover the maximum duration of any state while being small enough to provide efficient computation, we use $D = 60hrs$. The order of transition A from each state signifies the change in time and the duration of a state D represents the time spent in each state. The observation sequence of each state throughout its duration represents the constraints being modeled, e.g., CPU, memory. The sample variables used as inputs to the MD-HSMM are as listed in Tbl. 4. We set the initial values for the initial state probabilities π (0.33 0.33, 0.34), the state transition probabilities a_{ij}, the state duration distributions $P_i(d)$ and the observation $b_i(O^r)$. A Gaussian mixture model is used to describe the observation, where (μ_1, μ_2) indicate the mean values of the observation dimensions and (σ_1^2, σ_2^2) represent the variances.

Given these initial states for the MD-HSMM, we then use the modified Forward-Backward algorithm to re-estimate the model parameters and obtain results. The MD-HSMM training model is as shown in Fig. 9 indicating the convergence of the parameters in the MD-HSMM. Working with the log-likelihood makes it more convenient to decrease the value returned by likelihood function. The x-axis shows the training steps and the y-axis represents the likelihood probability of different states. The progression of the states

TABLE 4
Probabilities Matrix

Transition States (A)		
0.00	0.50	0.500
0.50	0.00	0.500
0.47	0.53	0.000
Observation (μ_1) - cpu (%)		
16.607	25.672	38.355
Observation (μ_2) - mem$(b) * e + 08$		
6.728	6.806	6.892
Variance (σ_1^2) - cpu (%)		
19.228	64.884	158.598
Variance (σ_2^2) - mem $(b) * e + 08$		
9.709	9.269	8.793
Duration		
Shape	0.732	1.663	1.139
Scale	6.826	3.693	5.709

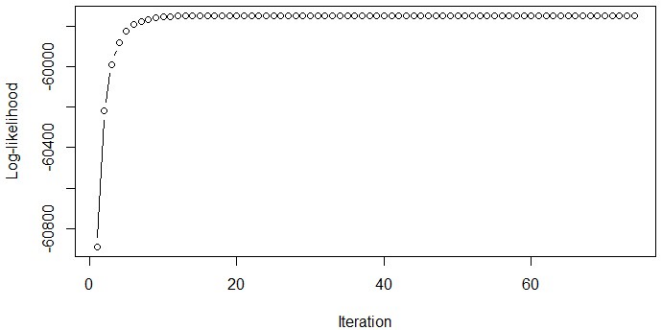

Fig. 9. Log-likelihood of the MD-HSMM model

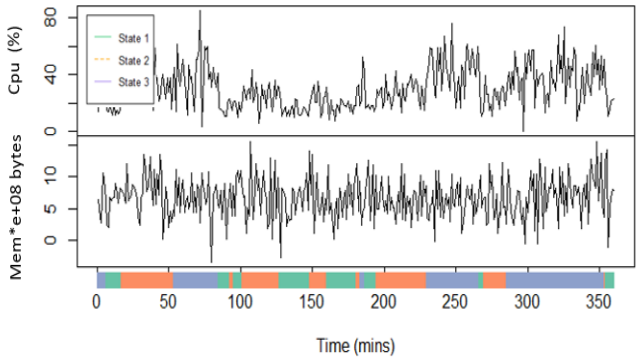

Fig. 10. Simulated data and states for resource usage observations

reaches the set error in less than 3 steps. This indicates that an appropriate MD-HSMM model is generated within 3 k iterations.

5.3.2 Predicting and Validating the MD-HSMM

Using the observations and the MD-HSMM model, we executed the Viterbi algorithm to determine the states that must have given the observations. After obtaining the parameters of the model, we use a simulation approach to produce our own sequence of observations. Fig. 10 is a snapshot of the CPU usage and memory data. The curve shows the values

TABLE 5
Prediction Period Variation & Accuracy of the MD-HSMM

Experiment Period	Prediction Period	RSME (%)	MAPE (%)
24hrs	4hrs	91.2	91.5
24hrs	6hrs	89.6	90.5
24hrs	8hrs	90.8	92.1
48hrs	8hrs	91.6	91.7
48hrs	12hrs	91.1	91.4

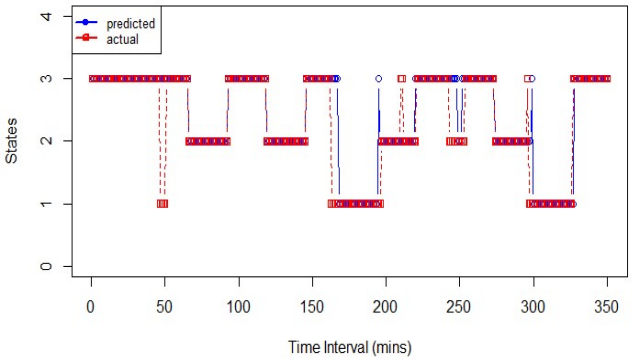

Fig. 11. Actual vs. Predicted States

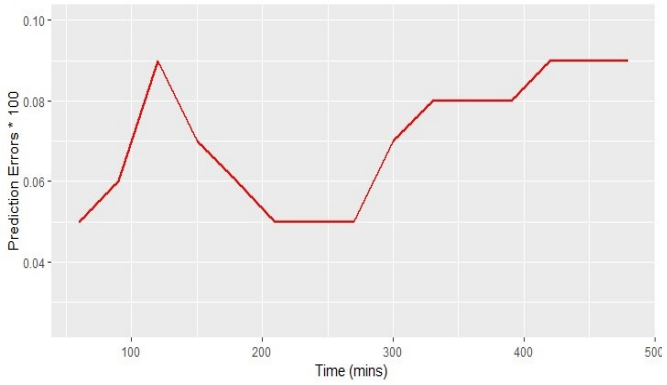

Fig. 12. Predictive Accuracy of the MD-HSMM

from the observation distribution while the horizontal bar indicates the different states. We then compute the means of the model and the prediction data to validate the model. Tbl. 5 presents the results of the training and accuracy of prediction. Fig. 11 shows the predictive accuracy of the model over a 6hr period. Fig. 12 depicts the accuracy of the MD-HSMM in predicting the resource usage states of streaming applications, highlighting its consistency in retaining its predictive properties for a longer period of time.

6 Conclusion

In this paper, we have investigated the challenges of provisioning resources for big data streaming applications. Based on our findings, we presented a layered multi-dimensional HMM (LMD-HMM) and multi-dimensional HSMM (LMD-HSMM) approaches for resource usage prediction of time-bounded and unbounded big data streaming applications, respectively. Our proposed methodology uses multi-dimensional HMM/HSMM at the lower-layer, which are well-trained and the states concatenated at the upper-layer for resource scaling. Experimental evaluation results show that LMD-HMM has a good potential for predicting the resource usage of time-bounded big data streaming applications. Thus, as long as the lower-layer produce consistent models during training and testing, the LMD-HMM has a better advantage than the single-layer HMM since it is able to perform well under varying streaming environments.

We also make use of the LMD-HSMM framework for predicting the resource usage of streaming applications by explicitly modeling the duration of states. A modified forward-backward algorithm is used to estimate the parameters of the MD-HSMM in which new variables are defined and the corresponding re-estimation formulae is based on the new variables derived. By incorporating explicit temporal structure into the framework, we are able to predict the resource usage of unbounded big data streaming applications. Evaluation of our proposed frameworks is carried out using developed big data streaming applications running within the Spark streaming environment.

Our experimental results show that the HSMM-based framework has a much better performance than the HMM-based model when applied to unbounded applications since it explicitly models the duration of states. In future, we will investigate applicable models for managing resources in a multi-tenant cloud environment.

References

[1] Ji, Changqing, Yu Li, Wenming Qiu, Uchechukwu Awada, and Keqiu Li. "Big data processing in cloud computing environments." 2012 12th International Symposium on Pervasive Systems, Algorithms and Networks, pp. 17-23. IEEE, 2012.

[2] Ranjan, Rajiv. "Streaming big data processing in datacenter clouds." IEEE Cloud Computing 1, no. 1 (2014): 78-83.

[3] Armbrust, Michael, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee et al. "A view of cloud computing." Communications of the ACM 53, no. 4 (2010): 50-58.

[4] Toshniwal, Ankit, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson et al. "Storm@ twitter." Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 147-156. ACM, 2014.

[5] Spark Streaming Programming Guide - Apache Spark. Available: https:llspark.apache.org/ accessed on August 2016.

[6] Xhafa, Fatos, Victor Naranjo, and Santi Caball. "Processing and analytics of big data streams with yahoo! s4." 2015 IEEE 29th International Conference on Advanced Information Networking and Applications, pp. 263-270. IEEE, 2015.

[7] Shariffdeen, R. S., D. T. S. P. Munasinghe, H. S. Bhathiya, U. K. J. U. Bandara, and HMN Dilum Bandara. "Adaptive workload prediction for proactive auto scaling in PaaS systems." In Cloud Computing Technologies and Applications (CloudTech), 2016 2nd International Conference on, pp. 22-29. IEEE, 2016.

[8] P.C. Brebner. "Is your cloud elastic enough?: performance modelling the elasticity of infrastructure as a service (IaaS) cloud applications." Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, ACM, Boston, MA, April 2012, pp. 263266.

[9] Nikravesh, Ali Yadavar, Samuel A. Ajila, and Chung-Horng Lung. "Towards an autonomic auto-scaling prediction system for cloud resource provisioning." In Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 35-45. IEEE Press, 2015.

[10] Biswas, Anshuman, Shikharesh Majumdar, Biswajit Nandy, and Ali El-Haraki. "A hybrid auto-scaling technique for clouds processing applications with service level agreements." Journal of Cloud Computing 6, no. 1 (2017): 29.

[11] Gong, Zhenhuan, Xiaohui Gu, and John Wilkes. "Press: Predictive elastic resource scaling for cloud systems." 2010 International Conference on Network and Service Management, pp. 9-16. IEEE, 2010.

[12] Roy, Nilabja, Abhishek Dubey, and Aniruddha Gokhale. "Efficient autoscaling in the cloud using predictive models for workload forecasting." 2011 IEEE International Conference on Cloud Computing (CLOUD), pp. 500-507. IEEE, 2011.

[13] Runsewe, Olubisi, and Nancy Samaan. "Cloud Resource Scaling for Big Data Streaming Applications Using A Layered Multi-dimensional Hidden Markov Model." In Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on, pp. 848-857. IEEE, 2017.

[14] Rabiner, Lawrence R. "A tutorial on hidden Markov models and selected applications in speech recognition." Proceedings of the IEEE 77, no. 2 (1989): 257-286.

[15] Gordon, Michael I., William Thies, and Saman Amarasinghe. "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs." ACM SIGOPS Operating Systems Review 40, no. 5 (2006): 151-162.

[16] Abadi, Daniel J., Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner et al. "The Design of the Borealis Stream Processing Engine." CIDR, vol. 5, pp. 277-289. 2005.

[17] STREAM Group. "STREAM: The Stanford Stream Data Manager." IEEE Data Engineering Bulletin, http://www-db. stanford. edu/stream 2, no. 003 (2003).

[18] Gedik, Bugra, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. "SPADE: the system s declarative stream processing engine." Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pp. 1123-1134. ACM, 2008.

[19] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51, no. 1 (2008): 107-113.

[20] Kreps, Jay, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing." Proceedings of the NetDB, pp. 1-7. 2011.

[21] Li, Jack, Calton Pu, Yuan Chen, Daniel Gmach, and Dejan Milojicic. "Enabling elastic stream processing in shared clusters." In Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on, pp. 108-115. IEEE, 2016.

[22] Amazon Web Services Autoscale, Available: https://aws.amazon.com/autoscaling/ August 2016.

[23] AzureWatch, Available: https://www.paraleap.com/ August 2016.

[24] Bi, Jing, Zhiliang Zhu, Ruixiong Tian, and Qingbo Wang. "Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center." 2010 IEEE 3rd International Conference on Cloud Computing, pp. 370-377. IEEE, 2010.

[25] Iqbal, Waheed, Matthew N. Dailey, David Carrera, and Paul Janecek. "Adaptive resource provisioning for read intensive multi-tier applications in the cloud." Future Generation Computer Systems 27, no. 6 (2011): 871-879.

[26] Shen, Zhiming, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. "Cloudscale: elastic resource scaling for multi-tenant cloud systems." Proceedings of the 2nd ACM Symposium on Cloud Computing, p. 5. ACM, 2011.

[27] Han, Rui, Li Guo, Moustafa M. Ghanem, and Yike Guo. "Lightweight resource scaling for cloud applications." 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 644-651. IEEE, 2012.

[28] Lorido-Botran, Tania, Jose Miguel-Alonso, and Jose A. Lozano. "A review of auto-scaling techniques for elastic applications in cloud environments." Journal of Grid Computing 12, no. 4 (2014): 559-592.

[29] Baldan, Francisco Javier, Sergio Ramirez-Gallego, Christoph Bergmeir, Jose M. Benitez-Sanchez, and Francisco Herrera. "A Forecasting Methodology for Workload Forecasting in Cloud Systems." IEEE Transactions on Cloud Computing (2016).

[30] Rao, Jia, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. "VCONF: a reinforcement learning approach to virtual machines auto-configuration." Proceedings of the 6th international conference on Autonomic computing, pp. 137-146. ACM, 2009.

[31] Vijayakumar, Smita, Qian Zhu, and Gagan Agrawal. "Dynamic resource provisioning for data streaming applications in a cloud environment." 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp. 441-448. IEEE, 2010.

[32] Nikravesh, Ali Yadavar, Samuel A. Ajila, and Chung-Horng Lung. "Cloud resource auto-scaling system based on hidden markov model (hmm)." 2014 IEEE International Conference on Semantic Computing (ICSC), pp. 124-127. IEEE, 2014.

[33] Kumbhare, Alok Gautam, Yogesh Simmhan, and Viktor K. Prasanna. "Plasticc: Predictive look-ahead scheduling for continuous dataflows on clouds." In 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 344-353. IEEE, 2014.

[34] Fu, Tom ZJ, Jianbing Ding, Richard TB Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. "DRS: dynamic resource scheduling for real-time analytics over fast streams." In Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on, pp. 411-420. IEEE, 2015.

[35] Lombardi, Federico, Leonardo Aniello, Silvia Bonomi, and Leonardo Querzoni. "Elastic symbiotic scaling of operators and resources in stream processing systems." IEEE Transactions on Parallel and Distributed Systems 29, no. 3 (2018): 572-585.

[36] Pahl, Claus, and Brian Lee. "Containers and clusters for edge cloud architectures–A technology review." In Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on, pp. 379-386. IEEE, 2015.

[37] Yu, Shun-Zheng. "Hidden semi-Markov models." Artificial intelligence 174.2 (2010): 215-243.

[38] Apache Flume, Available:https://flume.apache.org/ August 2016.

[39] Apache Hadoop, Available: http://hadoop.apache.org/ August 2016.

[40] Ganglia, Available: http://ganglia.info/. August 2016.

[41] Li, Xiaolin, Marc Parizeau, and Rjean Plamondon. "Training hidden markov models with multiple observations-a combinatorial method." IEEE Transactions on Pattern Analysis and Machine Intelligence 22, no. 4 (2000): 371-377.

Olubisi Runsewe received her MSc degree in Electronic Business Technologies from the University of Ottawa, Canada, in 2012. She is currently a PhD student and part-time professor at the University of Ottawa. Her current research interests include cloud computing, big data, machine learning, data mining and resource management. In 2015, she received the Ontario Government Scholarship (OGS) Award. She is a member of the IEEE.

Nancy Samaan received her PhD degree in computer science from the University of Ottawa, Canada, in 2007. She is currently an associate professor with the School of Electrical Engineering and Computer Science, University of Ottawa. Her current research interests include cloud computing, network virtualization network resource management and wireless communications. She is a recipient of the Natural Sciences and Engineering Research Council of Canada University Faculty Award. She is a member of the IEEE.