Autonomic Workload Change Classification and Prediction for Big Data Workloads

Mikhail Genkin* and Frank Dehne
School of Computer Science, Carleton University, Ottawa, Canada
* Contact Author. E-mail: michael.genkin@carleton.ca

***Abstract*—The big data software stack based on Apache Spark and Hadoop has become mission critical in many enterprises. Performance of Spark and Hadoop jobs depends on a large number of configuration settings. The manual tuning procedure is expensive and brittle. There have been efforts to develop on-line and off-line automatic tuning approaches to make the big data stack more autonomic, but many researchers noted that it is important to tune only when truly necessary because many parameter searches can reduce rather than enhance performance. Autonomic systems need to be able to accurately detect important changes in workload characteristics, predict future workload characteristics, and use this information to pro-actively optimise resource allocation and frequency of parameter searches. This paper presents the first study focusing on workload change detection, change classification and workload forecasting in big data workloads. We demonstrate 99% accuracy for workload change detection, 90% accuracy for workload and workload transition classification, and up to 96% accuracy for future workload type prediction on Spark and Hadoop job flows simulated using popular big data benchmarks. Our method does not rely on past workload history for workload type prediction.**

Keywords: Big data autonomic computing, on-line automatic tuning, YARN, Hadoop, Spark, workload change detection, workload forecasting, big data performance optimisation.

I. Introduction

A. Background

Big data analytics has emerged as one of the most important computing trends in high performance computing. Key big data technologies such as Hadoop map-reduce jobs, Spark applications, Hive, Hbase, and others run on large hardware clusters that are managed by open-source and commercial resource managers, such as YARN, Mesos, and Kubernetes.

Performance is a key challenge in this space. To be useful, analytic jobs need to run fast enough to produce valuable insights. This needs to be accomplished on very large and often loosely structured data sets distributed over a large number of compute nodes. Apache Hadoop and Spark job performance, however, is dependent on a large number of configuration settings, and has often been observed to be brittle and inconsistent. Resource managers also present many additional configuration parameters and scheduling policies that can significantly affect job performance under multi-user and multi-tenant conditions.

A poorly tuned configuration can result in order-of-magnitude slower performance than for an optimally-tuned one. Manual tuning involves experimenting with different combinations of tunable parameters. Considering that each experiment, at multi-Terabyte data scale, can take hours to days to run this can turn into a very long and expensive procedure. Also, parameters that are optimal for one job (input data set) may not be well suited to another, and then the experiments have to be repeated.

To address this issue there were a number of efforts to develop automatic tuning systems specifically for big data frameworks and resource managers [1], [2], [3]. While these studies were able to demonstrate improvements, it should be pointed out that in most cases relatively simple single-user workloads were used.

Real big data workloads tend to execute multiple jobs concurrently. The number and type of jobs will change over time. To be truly effective automatic tuning systems need to be able to detect significant workload changes and adapt to new workload characteristics. To accomplish this they generally need to be able to:

- Classify the currently executing workload.
- Find the ideal combination of tuning parameters for this workload.
- Detect workload changes.
- Adapt to workload changes by finding a new ideal combination of tuning parameters to suit new workload characteristics.

B. Problem

State of the art approaches from the 'small data' domain rely on past workload history to predict important changes in the workload characteristics. This may be good enough in those situations where the same, or similar, jobs are being executed day to day. In the big data space, however, analytic jobs tend to focus more on exploration and experimentation. This makes big data workloads less cyclical and repetitive than those being run on the more traditional 'small data' systems, making it difficult to leverage the past history to predict the workload type and performance.

Workload change detection and forecasting in big data systems present unique challenges. Analytic frameworks such as Hadoop MapReduce and Apache Spark tend to produce very abrupt changes in workload characteristics as they execute different data processing stages. For example, as a Hadoop MapReduce job transitions from the Map phase to the Reduce phase the nature of the workload changes from CPU-memory bound to network and disk I/O-bound. When multiple jobs are running concurrently these changes can mask other important workload changes, such as the start or completion of new types of analytic jobs.

Autonomic systems for big data need the ability to forecast at least the near term workload characteristics and job performance without relying on the past history. This information can be used to make more intelligent resource allocation decisions, and adjustments to the tuning parameters of the currently executing analytic jobs and those that are queued and about to start. Longer-term predictions are less useful.

C. Limitations Of Previous Approaches

To date there has been very little research focusing on the workload change detection and classification in big data workloads. Most research in this area comes from the more traditional Relational Database Management Systems (RDBMS) and Web services areas. These earlier studies rely on analysis of past database query and Web page request patterns to predict workload shifts and cycles, and detect changes by comparing workload characteristics at a given point in time with predicted characteristics. This approach has only limited applicability in the big data space.

In our earlier study [4] we demonstrated a simple technique for workload change detection in Spark and Hadoop workloads. To date this is was the only published study that demonstrated change detection in Spark and Hadoop workloads. However, in order to build intelligent autonomic applications it is not enough to simply detect the change in workload. It is also important for the automatic tuning system to be able to identify and classify the type of change that has occurred, and to predict performance metrics that can be expected in the subsequent observation windows.

D. Our Contribution

In this study we present a new autonomic architecture for big data that uses machine-learning algorithms to classify workload transitions as well as workloads, and applies this classification to accurately predict future workload type without relying on past workload history.

We demonstrate 99% change detection accuracy, 90% workload transition classification accuracy, and up to 96% workload type prediction accuracy.

II. Previous Work

To date there has been very little research focusing on detecting and predicting workload changes in big data applications. Almost all research studies specifically focusing on workload change detection and prediction come from researchers working on autonomic database management systems (DBMS) in the more established ”small data” domain. Most of these works focus on predicting shifts from On-Line Transaction Processing (OLTP) to Decision Support System (DSS) type of workload because these very different workloads require different tunings to achieve acceptable performance. Although big data frameworks have significant architectural differences from the more traditional ”small data” DBMS, techniques used for workload classification, change detection and prediction are relevant. Below we discuss the most relevant works in more detail.

Lin Ma et al. [5] describe QueryBot5000 (QB5000) - a system for query-based workload forecasting that can be used to implement autonomic qualities for RDBMS such as MySQL and PostgreSQL. The authors asserted that the best way to model the future workload was to build a model based on the past query type and arrival rate rather than resource utilization. The QB5000 architecture included 3 components: 1 - a Pre-processor that replaced SQL constants with symbols, and used heuristics to reduce millions of unique queries to a more manageable set comprising thousands of query templates; 2 - a Cluster that used an on-line version of DBSCAN non-supervised clustering algorithm to further reduce the total number of models that need to be constructed; 3 - a Forecaster component that used an ensemble comprising a Linear Regression (LR) algorithm and a Recurrent Neural Network (RNN) to predict query arrival rates for cyclical and evolving workloads, and Kernel Regression (KR) to predict query arrival rates for workloads with spikes.

QB5000 used a combination of off-line training and on-line prediction to provide input to the database optimizer. Ma et al. evaluate several prediction horizons. Their results showed improved accuracy when using LR+RNN ansemble, or KR for longer prediction horizons or 2 or more days. For short prediction horizon of 1 hr there does not seem to be much difference in accuracy among several techniques evaluated. Their approach relies on past workload history to predict the future query arrival rates, and sudden change in workload pattern would require re-training of the system.

Elnaffar and Martin [6] proposed a framework for predicting shifts in DBMS workload from predominantly OLTP-type to DSS-type. Their framework, called the Psychic-Skeptic Prediction (PSP) framework, included several components - the Workload Classifier, the Workload Predictor and the Skeptic.

The role of the Workload Classifier was to classify the type of workload at any given point in time based on a feature vector that includes features such as the number of pages read and the number of rows returned for a query. The Workload Classifier uses a technique called Decision Tree Induction to construct it’s classification model. This technique was chosen over other techniques, such as artificial neural networks (ANN), because it provides for high interoperability and produces a collection of rules that can be readily understood by a human. The Workload Classifier produced a metric called DSSness that expressed the OLTP vs. DSS nature of the workload in a quantitative way.

Elnaffar and Martin noted that running the Workload Classifier continuously resulted in a significant performance overhead [6]. Thus they introduced another component called the Workload Predictor, that would examine a time-series of DSSness, and predict a the timing of future workload characteristic shifts from OLTP to DSS. The Skeptic component of PSP would sample the workload characteristics at times close to the predicted shift in the workload characteristics to validate that the prediction was still accurate.

Although the PSP included a mechanism for updating the prediction model in those situations where differences between the predicted and the actual workload characteristics were observed, one main limitation of this approach was that it relied on the past performance to predict the future and thus required significant off-line re-training if a significant change in the workload pattern occurred.

Holze and Ritter [7] presented a continuous, light-weight solution for workload monitoring and workload shift detection using n-gram-models. The authors use the term *workload shift* to describe a number of situations: 1 - introduction of new applications; 2 - sun-setting of obsolete applications; 3 - modifications to existing applications as a result of new releases; 4 - application usage changes caused by, for example, increasing the number of users. Holtze and Ritter [7] contend that *workload shifts* can exhibit short-term and long-term patterns, and focus their study on the detection of long-term patterns.

Huang et al. [8] developed a deep recurrent model for server load and performance prediction in data center. These authors contend that since sequences of user requests are the root cause of server performance, they should be used as the basis for performance prediction. Their approach uses a type of RNN - the Long Short Term Memory (LSTM) neural network to analyse the server logs containing user requests and predict the future user requests. They use a second LSTM with a Multi-Layer Perceptron (MLP) output layer to predict the performance metrics, such as the throughput and the server CPU utilization based on the workload prediction.

Lei et al. [9] describe a method for detecting hot spots in a virtualized environment. This is an important issue in the cloud space because it affects workload balancing. Their architecture was based on two parts: 1 - an agent that collected system hardware utilization information from the hardware nodes; 2 - a Hadoop MapReduce-based algorithm that identified virtual machines (VM) with anomalous memory utilization. The algorithm scans the time-series of the hardware utilisation metrics collected during the monitoring interval and counts the number of times the average resource utilizations exceed a pre-defined threshold to identify the hot-spots. The importance of hot spot detection is relevant to big data workload analysis because big data analytic jobs run on scaled-out clusters, and the appearance of imbalance in the cluster utilization can denote an important workload change.

Cherkasova et al. [10] proposed a framework for automated anomaly detection and application change analysis. The main objective of their research was to provide a way to pro-actively identify poorly performing enterprise applications. These authors distinguish between the terms *performance anomaly* and *workload change*. They define the term *performance anomaly* to mean that the observed application performance (for example the CPU utilization) cannot be explained by the observed application workload. Their method uses an off-line statistical method - Non-linear Least Squares Regression (Non-negative LSQ) in conjunction with step-wise linear regression to identify significant transactions and model their CPU demand, and an on-line algorithm that computes the new application signature and compares it with the old application signature to identify *performance anomalies*.

Khanna et al. [11] describe a method for autonomic characterisation of workloads using workload fingerprinting. The authors focus on cloud computing and the need for the orchestration layer to forecast changing workload conditions in order to be able to meet Service Level Agreements (SLA). They describe the workload as consisting of a number of phases. Their approach to forecasting consists of two steps: 1 - workload detection and classification; 2 - identifying distinct workload phases. For step 1 they use the Adaboost [12] ensemble algorithm to synthesize the workload detection model. For step 2 they use the non-supervised K-means clustering algorithm to identify the distinct workload phases. The idea of describing workload as a series of phases or stages is interesting for big data applications because Spark and Hadoop analytic frameworks apply stages of distinct processing.

On reviewing the current state of the art it's possible to make the following generalized observations about the approaches used to identify the workload changes and predict performance:

- there is little standardization in the terminology used to describe workload changes
- researchers use a combination of off-line and on-line techniques
- off-line learning techniques are commonly variants of LR, RNN and LSTM, and Decision Trees and use historical data to construct prediction models
- unsupervised clustering algorithms, such as K-means and DBSCAN, are sometimes used as a pre-processing stage, to reduce the number of prediction models that need to be constructed
- on-line algorithms compare actual executing workload with predictions from historical data to detect workload changes, shifts and anomalies.

The main weakness of this approach is that it relies on the records of past performance to detect change in the currently executing workload and predict the future performance. If the current workload does not match the workloads executed in the past, this approach is not accurate.

In the big data space analytic workloads often focus on finding new insights and tend to be less repetitive and cyclical than the workloads associated with the OLTP Web applications and the more conventional 'small data' RDBMS. Therefore, below we discuss a different approach to workload change detection and performance prediction.

III. Workloads and Workload Changes In Big Data Applications

There are many definitions for the term 'workload' in the literature. There are also many different ways to describe workload changes. Due to this, before delving into the theoretic basis and the architecture of our approach, it is important to clarify the terminology and the definitions of these terms as treated in our study.

In this study, the term 'workload' is used to represent any continuous sequence of observation windows with feature vectors that do not show any statistically meaningful differences. Such a sequence represents a period of distinct steady-state processing. Using this definition, for example, a Map stage of a Hadoop MapReduce job would be a distinct workload. Similarly, the reduce-shuffle and reduce stages would be treated as distinct workloads as well. For Spark jobs, the job stages would be treated as distinct workloads from the shuffles.

Genkin et al. [4] define the term *workload transition* to indicate any significant change in workload characteristics. In this paper we expand on our previous work and provide a more rigorous description, definition, and classification of workloads and workload changes in big data applications.

The term 'workload change' is very broad and can be used to indicate a number of different conditions. We define the following terms to help discuss and analyse changes that can occur in big data systems:

1) *Workload Cycles*. This term refers to regular, repeating workload changes caused by shifts in the usage patterns. This includes, for example, the shift from DSS to OLTP processing, discussed in the section II. These changes can be gradual and symmetrical, or they can be abrupt, asymmetrical, or spiky.
2) *Workload Drift*. This term refers to long-term changes in workload performance characteristics caused by: 1 - changing volume of users; 2 - changing volume of data; 3 - systematic changes in the processing software stack; 4 - changes in the underlying hardware infrastructure.
3) *Workload Anomaly*. This term refers to a sudden, significant, non-repeating, not planned change in the workload characteristics. A sudden spike in the usage activity of a Web commerce site due to an on-line sale is one example.
4) *Workload Transition*. This term refers to a period of non-steady state processing that results when either the usage pattern changes during a *workload cycle*, *workload drift*, *workload anomaly* or as a result of the algorithmic cycles within the analytic framework. One example of this would be the transition from the map phase to the reduce phase during the execution of a map-reduce job.

A Hadoop MapReduce job can present the following sequence of workload transitions during it's processing:

1) The job start and the beginning of the map phase of processing. This workload transition is typically marked by a sharp spike in disk read activity, high level of container creation and completion, as well as high CPU and memory usage.
2) The completion of the map phase of processing and start of the reduce-shuffle phase of processing. Completion of the map phase is marked by sharp increase in container completion rate, a drop in the CPU utilisation, and an increase in the disk write rate produced by map tasks writing to intermediate files. This is followed by an increase in the container creation rate and a surge in the network read activity as the newly created reduce tasks fetch data from the other nodes.
3) The completion of the reduce-shuffle phase of processing and the start of the reduce phase of processing.This event is marked by a marked reduction in the network read rate, and a surge in the disk read rate as the reduce tasks load data fetched from other nodes.
4) The completion of the reduce phase of processing, and of the job itself. This event is marked by a surge in the container completion rate and a corresponding surge in the disk write rate caused by the reduce tasks writing out the final data.

A Hadoop MapReduce, single-user job flow can involve many jobs and many workload transitions described above. Workload transitions for a single-user job flow should show a regular and distinct temporal patterns. For example, the transition marking the start of a job will be typically followed by the transition from the map to the reduce phase.

For multi-user job flows temporal patterns of workload transitions may not be as clear-cut as for the single jobs and the single-user job flows. For example, the start-of-job workload transition may be followed by more start-of-job workload transitions. Nevertheless, for each start of job transition it is reasonable to expect that a map-to-reduce workload transition will follow.

Although Spark processes data differently than Hadoop MapReduce, similar reasoning can be applied to describe workload transitions that can be produced by that framework:

- The job start. Similar to Hadoop MapReduce job start, the start of a spark job is marked by a surge in container creation activity, and disk read activity.
- The job stage transition. Spark breaks the submitted job down into a number of stages for execution. Unlike Hadoop MapReduce, Spark requests containers at the beginning of the job, and then re-uses them until all of the required processing has been completed. Nevertheless, transitions from one stage of the job to the next can result in significant workload changes. Transition into the shuffle stage is marked by a surge in network read activity. End of the shuffle stage is marked by a surge in disk write, as intermediate results are saved.
- The job end. The result stage of the Spark job is marked by a surge in container completion activity coupled with a surge in disk write activity.

Table II summarizes the different types of workloads and workload transitions for Apache Hadoop and Spark.

IV. Architecture

Figure 1 shows how the different theoretical constructs in our analysis relate to one another. The automatic tuning engine aggregates real-time streaming data emitted by the resource manager and the cluster infrastructure into observation windows. Each observation window is described by it's start time, it's end time, and a feature vector. Observation window data are essentially multi-variate, real-time, time-series data. Each feature in the feature vector is a real number.

Transition Label	Description
	Workloads
HDMAP	Hadoop MapReduce Map phase processing.
HDSHUF	Hadoop MapReduce shuffle processing.
HDRED	Hadoop MapReduce Reduce phase processing.
SPSTAGE	Spark job stage processing.
SPSHUF	Spark shuffle processing.
	Workload Transitions
HDJSTART	Start of a Hadoop job. Could be under single or multi-user conditions.
HDMAPSHUFL	Map ends and reduce-shuffle starts.
HDSHUFRED	Reduce-shuffle ends.
HDJEND	Reduce ends, end of Hadoop MapReduce job. Could be under single or multi-user conditions.
SPJSTART	Spark job start.
SPSTSTART	Transition from shuffle to the start of a new job stage.
SPSTSTEND	Transition from a job stage to shuffle.
SPJEND	Spark job end.

TABLE I
WORKLOAD AND WORKLOAD TRANSITION CLASSES AND LABELS.

Feature	Type	Description
numContStarted	Integer	Number of containers created during this observation window.
meanStarted	Double	Average response-time of containers started in this observation window.
rsdStarted	Double	Relative standard deviation of containers started in this observation window.
numContainersFinished	Integer	Number of containers completed during this observation window.
meanFinished	Double	Average response-time of containers started in this observation window.
rsdFinished	Double	Relative standard deviation of containers finished in this observation window.
avgCPU	Double	Average actual CPU utilization of the cluster during this observation window.
averageActiveMem	Double	Average actual active memory utilization on the cluster during this observation window.
avgNetRead	Double	Average actual network read rate on the cluster during this observation window.
avgNetWrite	Double	Average actual network write rate on the cluster during this observation window.

TABLE II
OBSERVATION WINDOW FEATURES AND THEIR DESCRIPTIONS.

The real-time stream of observation window data can be treated as a sequence of steady state periods connected to each other by periods of non-steady-state processing, which we term workload transitions. During steady state processing differences in the feature vector data between adjacent windows do not show statistically meaningful differences. During workload transitions differences in the feature vector data between adjacent observation windows do show statistically meaningful differences. Observation windows that form workload transitions can be described by an additional feature vector that captures the rate of change that has occurred relative to previous observation windows. Each feature in this transition vector is also a real number.

Observation window data typically contain significant amounts of random statistical fluctuation. It is therefore necessary to smooth these fluctuations by applying a sliding analytical window. As shown in Figure 1, the sliding analytic window is defined as a multiple of an observation window. During our study we used sliding analytic window that aggregated data from 3 observation windows and would slide over 1 observation window at each step. Each analytic window is described by a feature vector containing the average and standard deviation values for each observation window feature.

Statistically meaningful changes in the workload characteristics can be detected by comparing feature vectors of the current analytic window with the feature vector of the previous analytic window. Feature vectors associated with the steady-state and the workload transition observation windows can be used as the basis for workload classification. This concept serves as the basis for our architecture.

Our high-level architecture is shown in Figure 2, and contains four main components: 1 - *ChangeDetector*; 2 - *WorkloadClassifier*; 3 - *TransitionClassfier*; 4 - *WorkloadPredictor*. We describe the design and the responsibilities of each component below.

The real-time stream of observation window data initially passes through the *ChangeDetector* component. The *ChangeDetector* is a binary classifier that classifies a given observation window as either steady-state or workload transition window. To accomplish this it implements a sliding analytic window that is n observation windows wide. As shown in Figure 1 the analytic window covers observation windows t-2 to t, where t is the current observation window. With the arrival of each new observation window the analytic window slides over one step.

The *ChangeDetector* is an ensemble of of statistical classifiers. Each statistical classifier performs the Welch's statistical significance test on a single feature from the analytic window's feature vector. It compares the mean and the standard deviation of the feature at observation window t with the mean and the

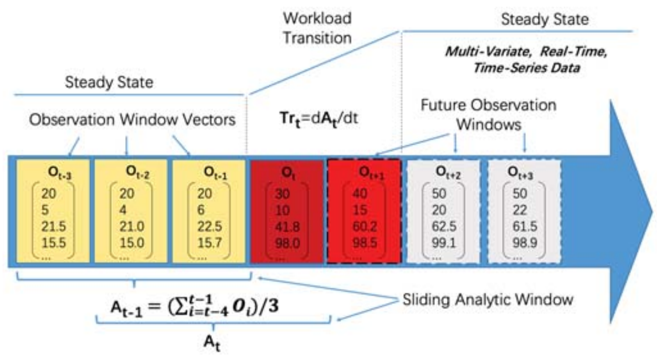

Fig. 1. Theoretical relationship between workloads, workload transitions, observation windows, and analytic windows.

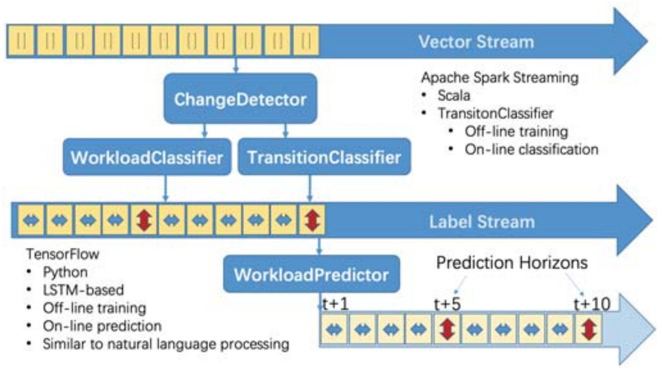

Fig. 2. High-level architecture for on-line automatic tuning system for big data.

standard deviation at observation window t-1. If the Welch's test indicates that there is a statistically meaningful difference, it registers one vote. Each feature classifier also has a weight assigned to it. This weight defaults to 1 and can be optionally adjusted by manual configuration. The *ChangeDetector* counts votes from all 10 feature classifiers and compares the sum of the votes against a threshold parameter. If the number of votes equals or exceeds the threshold, the *ChangeDetector* classifies the current analytic window and the observation window as workload transition windows. The *ChangeDetector* is an unsupervised classifier, and does not require off-line training to learn the weights and the threshold. These parameters need to be established experimentally during pre-production testing, and set manually.

If the *ChangeDetector* classifies the current observation window as steady-state, it will pass the observation window data to the *WorkloadClassifier* for further classification into the workload sub-types listed in Table II. If the *ChangeDetector* classifies the current observation window as a workload transition window, it will pass the corresponding analytic window feature matrix data to the *TransitionClassifier* for further classification into one of the workload transition sub-types listed also in Table II. The *TransitionClassifier* uses the analytic window data to calculate the transition feature vector, that is used for classification. This is further described in the section below that discusses the *TransitionClassifier*.

The *WorkloadClassifier* uses the random forest ensemble algorithm to classify the observation windows as workload sub-types. This classifier is trained off-line using a representative training set of the observation windows selected from the previously executed and recorded workloads. It classifies the observation windows in real-time, and outputs one of the workload labels from Table II.

The *TransitionClassifier*, like the *WorkloadClassifier*, also uses the random forest ensemble algorithm, and is also trained off-line. In fact these classifiers share a common abstract base class that wraps the random forest algorithm. Unlike the *WorkloadClassifier*, the *TransitionClassifier* operates not on the observation window, but on the associated analytic window. The sliding analytic window has the same time index as the current observation window, but references several (the number is configurable) previous observation windows in addition to the current observation window. This enables the *TransitionClassifier* to calculate the rate of change for the feature vector. The new derivative vector is used for the real-time classification. The *TransitionClassifier* outputs a transition label from Table II for each observation window.

The *ChangeDetector*, the *WorkloadClassifier*, and the *TransitionClassifier* are implemented in Scala using Apache Spark 2.4.3 Structured Streaming and MLib APIs.

After passing through the classification pipeline the real-time observation window stream containing feature vectors is transformed into a real-time stream of workload and transition labels. This is shown in Figure 2.

For example, a simple Hadoop MapReduce job that has all relevant stages could produce a label stream/sequence that reads something like this: "HDJSTART HDJSTART HDMAP HDMAP HDMAP HDMAP HDMAP HDMAP HDMAPSHUF HDMAPSHUF HDSHUF HDSHUF HDSHUF HDSHUFRED HDSHUFRED HDRED HDRED HDRED HDRED HDRED HDJEND". We now have a stream of data that very much resembles a textual representation of natural languages, and can be operated on using algorithmic techniques from that domain.

The *WorkloadPredictor* component uses a Long Short-Term Memory (LSTM) neural network to predict workload and workload transition types that are likely to occur in subsequent observation windows. LSTM neural networks have demonstrated considerable success predicting which words will come next in the real-time natural language processing applications, and the *WorkloadPredictor* uses the same approach. The LSTM is configured with a single layer, and is trained off-line using segments from the recorded workload time-series. During the processing of each current observation window, it predicts a sequence of workload and workload transition classes that can be expected in the next 10 (configurable) observation windows.

The *WorkloadPredictor* is implemented in Python using the popular TensorFlow machine-learning framework. It runs as a separate process and writes predictions into a file that is continuously read by the tuning engine, which uses the predictions to decide whether or not it makes sense to perform parameter searches.

V. Evaluation Methodology

Our evaluation methodology focused on simulating common Apache Hadoop and Spark workloads and workload transitions using well-understood big data benchmarks. The experiments were planned and performed in increasing order of complexity. Starting with simple single jobs, we then proceeded to the more complex single-user job flows, and then to multi-user job flows.

Before capturing performance statistics for each experiment, experimental runs were performed to establish the optimal observation window duration. The observation window duration was chosen so that the majority of windows had a statistically valid number of containers recorded. For example, if all container creation and completion events were recorded during a single very long window then this would not make for a compelling analysis.

All metrics were compiled using log analysis. Container performance metrics were compiled from KERMIT [2] logs. System performance metrics for the cluster were captured using the NMON utility that was executed during the run on all cluster nodes. A Spark application was developed that merged container performance metrics with system metrics, producing an integrated time-series of observation window data. Each observation window feature vector included all container and system performance metrics collected during that window.

A file containing the location of the known workloads and workload transitions was prepared for each time series. The exact time span for a particular workload, and the time at which workload transitions occurred, was established by a Spark application that performed log analysis on YARN logs, Spark event logs, and workload driver logs. The application looked for the key log entries indicating which type of workload was running, or that a workload transition had occurred. The time stamp was then converted to an observation window number, and an entry was added to the known transitions file to indicate the location and the type of workload or workload transition in the time-series. This file was later used to evaluate the accuracy of the classifications and the predictions.

During evaluation each time-series was replayed as a real-time stream of data. The KERMIT streaming engine was implemented as an Apache Spark structured streaming consumer configured to read a file source as a stream. The streaming engine would read the file contents into a streaming data frame. It would then apply SQL and custom stored procedure transformations to the data frame to execute the classification pipeline described in the section IV. The result would be a stream of classification labels listed in Table II, and it would be written into a separate text file. This file would be read by the *WorkloadPredictor* component, that would be running as a separate process.

This file, containing detected workloads and transitions was then compared with the known workloads and transitions file. The confusion matrix measures, such as accuracy, Positive Predictive Value (PPV) and error rate, were then calculated on the basis of this comparison. To evaluate the workload transition detection accuracy, the transition type was ignored, and detection was treated as a binary classification problem aiming to classify steady state vs. change from steady state. To evaluate workload and workload transition classification accuracy, one vs. the rest approach was used.

The time series were separated into a training and testing sub-sets using 70/30 ratio when evaluating the supervised learning algorithms. Five-fold cross-validation was used to evaluate variance in the classification results and to mitigate over-fitting.

Evaluation measures and approach varied depending on the component. Procedures specific to each component are further discussed below in the section VI.

A. Parameter Settings

Unless stated otherwise, the Hadoop MapReduce and Spark configurations used default values. For YARN, the yarn.nodemanager.resource.cpu-vcores parameter in the yarn-site.xml file was set to the total number of CPUs shown by the operating system on each of the cluster nodes. The yarn.nodemanager.resource.memory-mb and yarn.scheduler.maximum-allocation-mb parameters were set to the total amount of memory on each data node. In the mapred-site.xml file, the parameter mapreduce.job.reduces was set to 36. The parameters mapreduce.output.fileoutputformat.compress and mapreduce.map.output.compress were set to true. The parameters mapreduce.output.fileoutputformat.compress.codec and mapreduce.map.output.compress.code were set to org.apache.hadoop.io.compress.Default in order to avoid running out of space in the HDFS during longer runs. The parameter mapred.child.java.opts was modified to increase the maximum JVM heap size setting from the default to 850 MB. This was done to remove the possibility of a memory bottleneck impacting container performance. On the Spark side, the spark.executor.memory configuration parameter was set to 6G to ensure that most memory on our nodes was utilized.

B. Hardware And Software

All measurements were performed on an 8-node cluster comprising 1 management node and 7 compute/data nodes (all KVM virtual machines running on IBM S822L Power8 with Dual 10-core Power8 processors at 3.42GHz; one bare metal server was used for every two VMs). Each node was equipped with a 100 GB SSD drive for operating system and the Hadoop stack installation. All the nodes shared access to a 12TB network shared drive connected through a 10Gb fibre switch. Each virtual node was also configured with 48 GB RAM and 10 virtual cores. All nodes were running the Ubuntu 16.04 ppc64le operating system. The test cluster topology is shown in Figure 3. We used Hadoop 2.7.3 and Spark 2.4.3. In order to facilitate container metric collection, a jar file containing the

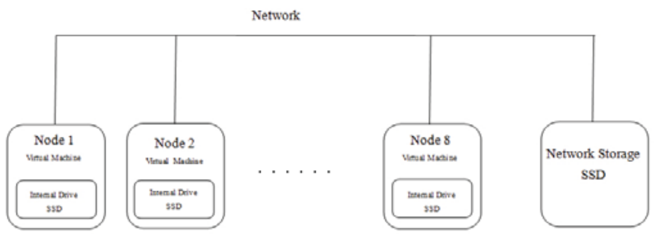

Fig. 3. Test-bed topology.

YARN resource manager and our KERMIT library [2] was built and deployed to replace the standard YARN jar.

VI. Results

We discuss our findings in the sections below. The sub-section VI-A discusses binary classification results observed when classifying observation windows as steady-state vs. non-steady state using the *ChangeDetector* component. The sub-section VI-B discusses the results achieved for the *WorkloadClassifier* component. The sub-section VI-C discusses the results achieved for the *TransitionClassifier* component. The sub-section VI-D discusses the results achieved for the *WorkloadPredictor* component.

A. Change Detection

Evaluation of the *ChangeDetector* component involved the following experiments:

- The experiments to determine the optimal threshold for the *ChangeDetector*, and the weights for feature classifiers.
- The experiments to determine the optimal width of the analytic window.

Figure 4 shows the Receiver Operating Characteristic (ROC) plot summarising the results of the experiments performed to establish the best threshold value and weights. The best results were achieved using a threshold value of 4 votes, and equal weights for the feature classifiers.

After ideal values for the threshold and the weights were established, additional experiments were performed to determine the ideal width of the analytic window. The best results were achieved using the smallest possible analytic window of 3 observation windows.

Figure 5 shows the summary of key confusion matrix measures for the *ChangeDetector*. A very high accuracy rate of 0.995 was achieved. It should be pointed out, however, that the observation window data displayed significant amount of class imbalance. There were roughly 100 times more steady-state windows than workload transition windows, and thus the accuracy measure was dominated by the negative case (no statistically significant difference). For this reason we also focused on the Positive Predictive Value (PPV) and the miss rate measures.

The PPV measure was considered to be a more interesting measure for the *ChangeDetector* than accuracy because it evaluates how well the classifier did vs. the positive case

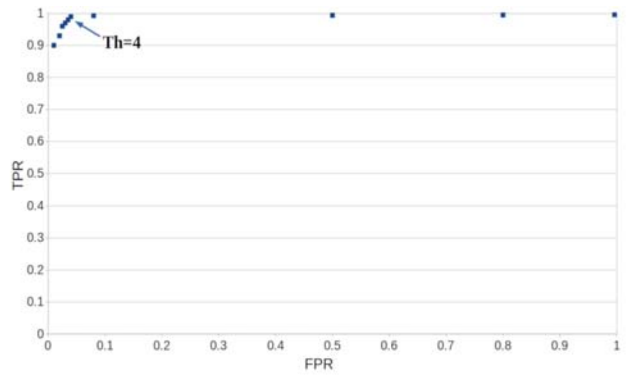

Fig. 4. ChangeDetector ROC for different thresholds, and equal weights.

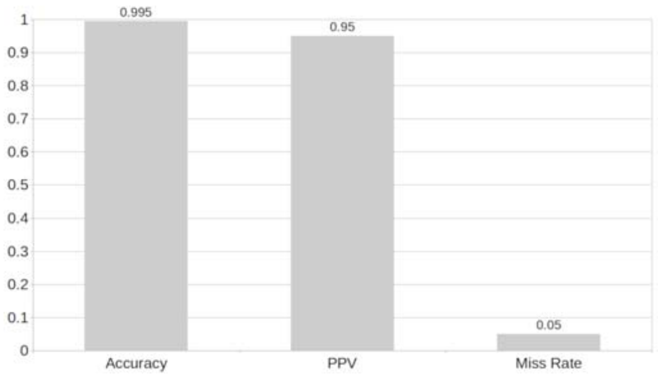

Fig. 5. ChangeDetector summary of key classifier performance metrics.

(workload transition). As shown in Figure 5 PPV was observed to be lower than accuracy. At 0.95 it was still observed to be very good. The miss rate, which evaluates the proportion of actual transitions missed by the classifier, was observed to be low, at 0.05.

B. Classifying Workloads

The training set for the *WorkloadClassifier* component was assembled manually. Time-series data were separated into training and testing time-series using a 70/30 split, as discussed above. The sample observation window data was extracted from the training time-series, and manually labelled using libsvm format. The observation window training set covered all of the workload classes listed in Table II. The classes in the training set were reasonably well balanced.

As with the *ChangeDetector*, multiple confusion matrix measures were used to evaluate the performance of the *WorkloadClassifier*. Accuracy, PPV and False Positive Rate (FPR) were decided to be the most important measures to focus on. One-vs-the-rest approach was used to calculate the measures for each class. From the class-specific data, average metric value and associated standard deviation were calculated, and are shown in Figure 6. Experiments were performed to determine the optimal number of trees and categories for the random forest ensemble algorithm used by the classifier. The best results were achieved using 100 trees and 10 categories.

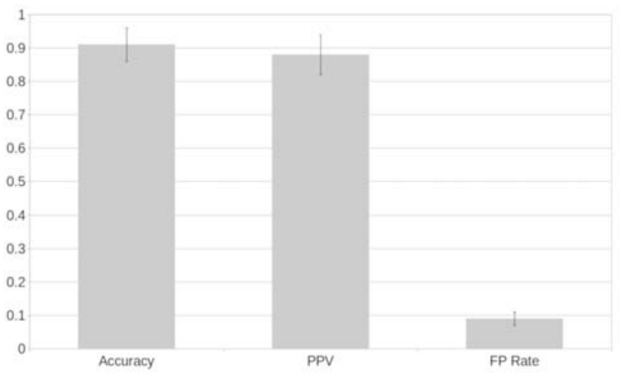

Fig. 6. Average Accuracy, PPV and FP Rate achieved for the WorkloadClassifier.

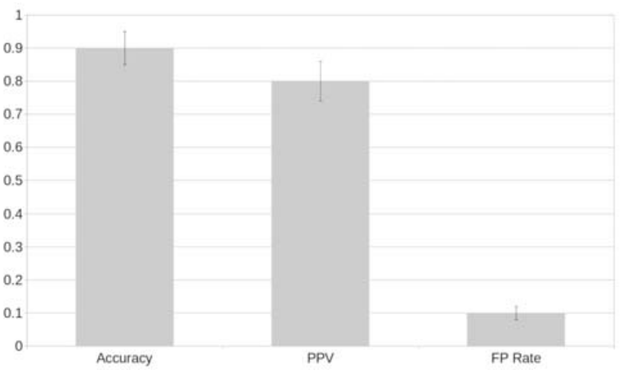

Fig. 7. Average Accuracy, PPV and FP Rate achieved for the TransitionClassifier.

The *WorkloadClassifier* achieved fairly high average accuracy (0.90) and PPV (0.88), and a fairly low average FPR (0.09). The standard deviation in these measures for all workload classes was observed to be reasonably small (0.02 to 0.06). This is shown in Figure 6.

C. Classifying Workload Transitions

The training procedure for the *TransitionClassifier* component was very similar to that used for the *WorkloadClassifier*. The only difference was due to the fact that the *TransitionClassifier* needs to be trained on the analytic window data, rather than the observation window data.

To accomplish this segments of observation window data containing known transitions were selected from the training time-series. A utility program was developed to generate and save corresponding analytic window data. The same confusion matrix measures and evaluation approach was used as for the *WorkloadClassifier*.

As with the *WorkloadClassifier*, experiments were performed to establish the ideal number of trees and the categories for the underlying random forest ensemble algorithm. In this case the best results were observed with 200 trees and 20 categories.

Figure 7 shows the summary of the key measures. Results were observed to be good, showing average accuracy of 0.91,

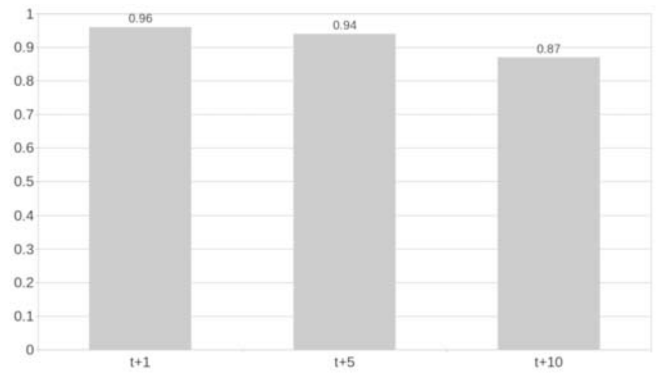

Fig. 8. WorkloadPredictor summary of accuracy by prediction horizon.

and quite similar to those achieved for the *WorkloadClassifier*. This is not very surprising considering that the classifiers use the same underlying algorithm.

D. Predicting Performance

After evaluation and optimization of the classification pipeline was complete, time-series with class labels given in Table II were generated for all time series, in both the training and evaluation set. The *WorkloadPredictor* LSTM was trained using complete time-series from the training set.

The testing data were then streamed through the *WorkloadPredictor*, and this component generated a sequence of 10 predicted class labels for each labelled observation window. This sequence was then compared to the actual sequence of labelled windows in the same time series, and accuracy at 3 prediction horizons was calculated. As with the classifiers, one-vs-the-rest approach was used to calculate the accuracy. In this case the workload and the workload transition classes were treated as one set.

The automatic tuning engines and the resource managers that comprise the orchestration layer care mostly about the near-term, and so the following prediction horizons were selected for analysis:

- t+1 observation window (i.e. the workload or the transition expected in the next observation window)
- t+5 (i.e. the workload or the transition expected in 5 observation windows from now)
- t+10 (i.e. the workload or the transition expected in 10 observation windows from now)

A summary of the accuracy results observed for the *WorkloadPredictor* are shown in Figure 8. As expected, the best accuracy was observed for the closest prediction horizon (0.96). For the longest prediction horizon of t+10 accuracy remained quite high at 0.87.

VII. Conclusion

In this paper we presented the first study focusing on workload change detection, classification and prediction for big data workloads. Unlike previous studies focusing on the more traditional 'small data' space, our approach does not rely on the longer-term past workload history to predict future workload transitions. Instead we treat big data workloads as sequences of more elemental steady state workload segments connected by non-steady state workload transitions.

We use a classification pipeline to segment multi-variate real-time workload data and convert it into a sequence of class labels, forming a 'workload language' that is similar in essence to natural languages. We were able to use an LSTM, a very popular natural language processing algorithm, to accurately predict the type of workload, or workload transition that can be expected to occur in the near to intermediate time horizon. The automatic tuning engines and the resource managers that comprise the future autonomic big data software infrastructure can leverage this information to pro-actively manage resource queues, container placement, and reduce search overhead.

Future research direction could focus on developing new on-line machine learning algorithms that could achieve classification and prediction performance equal to, or better than, the currently used supervised learning algorithms (random forest and LSTM) that require off-line training. Although training stages could be automated and could run in the background, the need for off-line training complicates overall system design, deployment and maintenance.

REFERENCES

[1] X. Ding, Y. Liu, and D. Qian, "Jellyfish: Online performance tuning with adaptive configuration and elastic container in hadoop yarn," in *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*. IEEE, 2015, pp. 831–836.

[2] M. Genkin, F. Dehne, M. Pospelova, Y. Chen, and P. Navarro, "Automatic, on-line tuning of yarn container memory and cpu parameters," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems*. IEEE, 2016, pp. 317–324.

[3] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics." in *CIDR*, vol. 11, 2011, pp. 261–272.

[4] M. Genkin, F. Dehne, P. Navarro, and S. Zhou, "Machine-learning based spark and hadoop workload classification using container performance patterns," in *2018 International Symposium on Benchmarking, Measuring and Optimizing (Bench' 18)*, vol. 1149, 2019, p. in print.

[5] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," in *In Proceedings of 2018 International Conference on Management of Data (SIGMOD'18)*, 2018.

[6] S. Elnaffar and P. Martin, "The psychic–skeptic prediction framework for effective monitoring of dbms workloads," *Data & Knowledge Engineering*, vol. 68, pp. 393–414, 2009.

[7] M. Holtze and N. Ritter, "Autonomic databases: Detection of workload shifts with n-gram-models," in *ADBIS 2008: Advances in Databases and Information Systems*, ser. Lecture Notes in Computer Science, vol. 5207, 2008, pp. 127 – 142.

[8] Z. Huang, J. Peng, H. Lian, J. Guo, and W. Qiu, "Deep recurrent model for server load and performance prediction in data center," *Hindawi Complexity*, vol. 2017, 2017.

[9] Z. Lei, B. Hu, J. Guo, L. Hu, W. Shen, and Y. Lei, "Scalable and efficient workload hotspot detection in virtualized environment," *Cluster Computing*, vol. 17, pp. 1253–1264, 2014.

[10] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2014, pp. 452 – 461.

[11] R. Khanna, M. Ganguli, A. Narayan, A. R., and P. Gupta, "Autonomic characterization of workloads using workload fingerprinting," in *In Proceedings of 2014 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2014.

[12] F. Yoav and S. R. E., "A decision-theoretic generalization of on-line learning and an application to boosting." *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119 – 139, 1997.