Autonomic Architecture for Big Data Performance Optimization

Mikhail Genkin[1], Frank Dehne[2], Anousheh Shahmirza[2], Pablo Navarro[2], and Siyu Zhou[2]

[1] Trent University, Peterborough ON, Canada,
mikhailgenkin@trentu.com
[2] Carleton University, Ottawa ON, Canada

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. Manual tuning is expensive and brittle. There have been prior efforts to develop on-line and off-line automatic tuning approaches to make the big data stack less dependent on manual tuning. These, however, demonstrated only modest performance improvements with very simple, single-user workloads on small data sets. This paper presents KERMIT - the autonomic architecture for big data capable of automatically tuning Apache Spark and Hadoop on-line, and achieving performance results 30% faster than rule-of-thumb tuning by a human administrator and up to 92% as fast as the fastest possible tuning established by performing an exhaustive search of the tuning parameter space. KERMIT can detect important workload changes with up to 99% accuracy, and predict future workload types with up to 96% accuracy. It is capable of identifying and classifying complex multi-user workloads without being explicitly trained on examples of these workloads. It does not rely on the past workload history to predict the future workload classes and their associated performance. KERMIT can identify and learn new workload classes, and adapt to workload drift, without human intervention.

Keywords: autonomic computing, big data, machine learning, high-performance computing

1 Introduction

1.1 Big Data Performance and Artificial Intelligence

Big data analytics have become mission critical in many enterprises. Big data is used extensively in baking and securities, communications, media and entertainment, health care, education, manufacturing, government, insurance and retail industries. The use of big data has become so pervasive that today many enterprises not only augment their more traditional Relational Database Management Systems (RDBMS) with big data technologies, but have completely re-based their analytic processing around the big data technologies.

Big data processing presents unique challenges due to the sheer volume of the data that needs to be analyzed and to the fact that these data are typically loosely structured. While the functional aspects of these important big data frameworks are very important, so is performance. In many cases analytic jobs must complete their processing within a specified time duration in order for the results to be useful. Some big data jobs must complete their processing in sub-seconds. Recommended systems are one example of systems that need to execute big data very quickly in order to maintain an acceptable end-user experience. Other jobs, especially those that aim to analyze large volumes of historical data to enable accurate projections of future events, may take hours, days, or even weeks to complete.

Today Apache Spark and Apache Hadoop form the foundation of the big data software stack. Other analytic technologies, such as Apache Hive, Apache Hbase, and Apache Tez, leverage these frameworks to implement their functionality. The performance of Spark and Hadoop jobs depends on a very large number of configuration settings. Manual tuning is expensive and brittle. Ideal combinations of tuning parameters have to be established experimentally. Each experiment often requires many hours to run due to the large volume of data involved. The introduction on new jobs, or changes in the nature or volume of the data, or the number of user executing jobs concurrently on the system often require the experiments to be repeated. A system capable of optimizing big data performance based on workload characteristics would be clearly beneficial, and would help greatly reduce operating costs of big data systems and improve the end user experience.

Recent advances in Artificial Intelligence (AI) and Machine Learning (ML) enabled new approaches to autonomic system design. Supervised ML algorithms can be used to classify objects and estimate their future quantifiable behavioral characteristics, such as speed and direction for example. Advanced supervised ML techniques, such as Zero-Shot Learning (ZSL), one-shot learning, and few-shot learning are now available, and can help reduce the effort associated with training data set construction. Unsupervised ML algorithms can be used effectively to discover patterns in data, thereby discovering previously unseen classes. Machine learning pipelines can be constructed to automate labeling and training of supervised ML algorithms. And so, the overarching research question behind this investigation is: "Can we combine ML techniques to construct a coherent architecture capable of optimizing big data performance without human intervention?"

In the sections below we begin by recapping the key research problem and our contribution to resolving it. We then discuss the most relevant previous works in this area. Subsequently, we delve into the details of our architecture. We begin with the definitions of key concepts and terms used in this paper. We then proceed to discuss the key sub-systems of our architecture, and how they operate.

2 Problem

To date there has been a significant amount of research focusing on autonomic computing. The majority of relevant research has focused on the more traditional, 'small data' RDBMS space [7] [9] [13]. Some research on this area also comes from network, cloud [10] [12], and web [6], but there has been no research to date focusing on big data specifically.

Supervised learning algorithms investigated in [4] and [2] require explicit labeling of each workload and workload transition type. The research investigation in [1] would significantly reduce the required labeling and training effort, but would not eliminate it completely.

We don't want to replace the manual tuning problem with a potentially equally expensive labeling and training problem. We need a solution that can automate labeling and training functions and minimize human intervention.

3 Limitations of Previous Approaches

This is the first study that focuses on autonomic performance optimization for big data workloads. Previous works focusing on cloud and traditional small data systems have the following limitations:

1. Coarse view of workload, such as DSS vs. OLTP, makes it impossible to optimize job by job. Big data jobs can have very different optimal configuration parameters.
2. Linear regression models typically used to predict workload characteristics perform poorly with abrupt workload transition common in the big data space.
3. Most methods depend on past workload history to predict future workload characteristics. Coupled with coarse view of workload, this makes them ineffective in situations where workload characteristics change frequently.
4. None of the previous works include the ability to anticipate new, previously unseen, workloads.

Most previous works focus on one of the aspects of autonomic computing but don't describe a complete architecture that implements an autonomic feedback loop.

4 Contribution

This work presents the first autonomic architecture specifically designed for autonomic optimization of big data workloads. This architecture implements the feedback loop based on pervasive implementation of machine learning algorithms. The Knowledge Extraction Resource Management Interface (KERMIT) architecture is able to interact with a wide variety of analytic frameworks and applications regardless of their internal architecture and resource usage pattern.

KERMIT can:

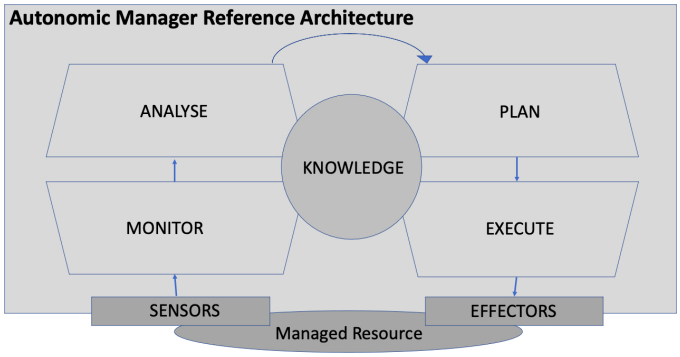

Fig. 1. The MAPE-K reference architecture for autonomic systems proposed by IBM. This figure was constructed based on the reference architecture description in [11].

1. Learn new workloads and their characteristics without human intervention.
2. Anticipate the appearance of new, unseen, workload classes.
3. Detect changes in workload characteristics and classify workloads and workload transitions in real-time.
4. Predict which workload types and transitions are likely to occur in the future, and when.
5. Minimize parameter search overhead.

Sections below will discuss previous work on autonomic architectures, and then go into a detailed description of the KERMIT architecture.

5 Previous Work

The field of autonomic computing was introduced by IBM in 2001 with the goal of creating computer systems capable of self-management. In 2005 IBM published it's reference architecture for autonomic systems [11]. This reference architecture, referred to as MAPE-K, is shown in Figure 1. The acronym MAPE-K stands for Monitor(M), Analyze(A), Plan(P), Execute(E), and Knowledge(K).

Since then, there has been considerable research focusing on developing autonomic systems, but this paper is the first study focusing specifically on autonomic workload optimization for big data. This section summarizes the most relevant studies from the other problem domains. Works focusing on the cloud space and networking are discussed first, followed by more relevant works focusing on small data autonomic databases.

Movahedi et al. [14] present a survey of autonomic network architectures (ANM). The authors classify ANM architectures into hierarchical and flat types. The authors noted that only one of the reviewed architectures - the Cognitive Network Architecture (CNA) - made use of learning techniques. The CNA architecture includes a cognitive plane responsible for data analysis and the decision-making process. They highlighted that the use of learning mechanisms could

significantly improve the performance of policy-based adaptation schemes towards finding the optimal solution [14].

Carrera et al. [5] present a study on autonomic placement of batch and transactional workloads. The authors present a technique that enables the existing middle ware to fairly manage mixed workloads comprised of both batch analytic jobs and transactional applications. The authors define a simple objective function to measure the difference between the actual response-time and the response-time goal for interactive applications. Their architecture defines the placement control loop and an application placement controller component that periodically inspects the system to determine if placement changes are needed in response to the changing workload. The period of the control loop is configurable. The placement algorithm uses a mathematical model to estimate application performance relative to a given CPU allocation. Their method extrapolates the applications' performance over the duration of the current control cycle and subsequent cycles. The authors claim that their technique improves mixed workload performance while providing service differentiation based on high-level performance goals.

This approach would have limited applicability in the big data space. This is because the mathematical model used to estimate future performance is linear and would not be able to predict the very abrupt workload transitions that big data jobs present, such as the map-to-reduce transition that results in a major change in workload characteristics. Their method does not have any provision for learning. The mathematical model needs to be executed every time, even if similar workload transitions recur.

Gergin et al. [8] describe a decentralized autonomic architecture for performance control in the cloud. Their architecture utilizes feedback loops. It uses a series of autonomic controllers to monitor virtual machine utilization under a Web OLTP-type workload and provisions new virtual machines as needed to achieve SLA objectives. Each controller independently regulates a tier of the application and implements the proportional, integrative, and derivative control laws. The mathematical model underlying each controller uses linear component to extrapolate near-term performance. This approach, as discussed above, would not work well for big data applications because they tend to produce very abrupt workload transitions. Their architecture does not include a learning mechanism.

In their recent paper Nouri et al. [15] focus on a cloud Infrastructure-as-a-Service (IaaS) use case. The authors describe a distributed architecture that aims to maximize performance of a large number of applications deployed on many servers. Each server is a virtual machine. Their view is that a centralized controller would become too complex because it would have to monitor a large number of applications on many server. It may not be able to respond in time when presented with a rapid change in load. A centralized controller would also become a single point of failure in the system.

Their architecture involves deploying an agent on each server. Each agent is responsible for monitoring the application performance on that server. The agents share a common knowledge base. Each agent has application and system

monitoring components which feed information to a learning core. The architecture monitors the application response-time statistics and system resource utilization values for the server.

The learning core is based on reinforcement learning. It maps a moving average of CPU utilization values to a set of states for the server. It then uses a utility function that converts that state into a scalar reward value that is used by reinforcement learning model to select from a set of actions for each state. The states actions and reward values are stored in the knowledge base on each server.

Nouri et al. [15] architecture is only of limited applicability to big data resource management because they consider a scenario whereby multiple applications running on a single server share a pool of resources. Thus exclusive resource allocation to the application is not possible. Containerized big data applications, on the other hand, rely on exclusive allocation of resources.

The architecture described in this work does not have any notion of searching the parameter space to optimize the application performance. Instead the controller can execute a limited set of actions to scale the number of application instances and/or servers up or down. Another aspect that limits this architectures applicability to KERMIT is the fact that this architecture uses linear regression to model system performance. Big data workloads present many abrupt and highly non-linear workload transitions. Nouri et al. architecture is reactive in nature. It does not predict future workload characteristics, and does not anticipate new workload classes.

In recent survey Raza et al. [16] works focusing on autonomic performance tuning in large scale data repositories. This survey explicitly excludes works for focusing on big data. It focuses on the more traditional RDBMS-based data warehouses and on DSS and OLTP workloads. The authors organize research into several categories, including workload classification, performance prediction, and self-adaptation. Most of the surveyed papers focus on one of these aspects and only a few combine them into an architecture that implements the full autonomic cycle. The most relevant ones are discussed in the paragraphs below.

Lin Ma et al. [13] 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. Although QB5000 was able to forecast future workload characteristics, the authors do not explicitly describe how an autonomic loop could be implemented.

Elnaffar and Martin [7] 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 a mechanism for updating the prediction model in those situations where differences between the predicted and the actual workload characteristics were observed - thus implementing the autonomic feedback loop. The main limitation of this approach is that the PSP relied heavily on past workload cycles to determine whether the model needs to be updated.

In summary, most previous works focus on one of the aspects of autonomic computing but don't describe a complete architecture that implements an autonomic feedback loop. Most of the described methods tend to be reactive in nature. They don't have the ability to classify workloads or anticipate workload changes. Most methods depend on past workload history to predict future workload characteristics. This makes them less effective in situations where workload characteristics change frequently, or do not follow a regular pattern. None of the previous works include the ability to anticipate new, previously unseen, workloads.

Furthermore, those works that do implement the ability to anticipate workload changes have a very coarse view of workload, such as DSS vs. OLTP, making it impossible to optimize job by job. Big data jobs can have very different optimal configuration parameters. Linear regression models typically used to predict workload characteristics perform poorly with abrupt workload transition common in the big data space. Our architecture aims to address these shortcomings.

6 Autonomic Architecture for Big Data Workload Optimization

Before we delve into the details of the KERMIT architecture it is important to establish a clear description of the key concepts and terms used throughout this work. Once this is done we describe the key autonomic architecture principles, and finally we discuss, in detail, the design techniques used to implement them in KERMIT.

6.1 Key Concepts and Terminology

The following key concept definitions underpin the KERMIT autonomic architecture:

1. *Workload.* Different researchers define the term *workload* differently. In this work we use the definition presented during our earlier investigation [2]. Consistent with our previous definition, this term refers to a multi-variate time-series of observation windows. Each observation window does not show statistically meaningful differences with neighboring observation windows. Thus, workloads are uniquely identifiable periods of steady-state processing. The concept of workload is represented with the symbol Ω.
2. *Workload transition.* A workload transition, like a workload, is a multivariate time series of observation windows. Unlike a workload, it represents a period of non-steady-state processing. Each observation window of the workload transition will show statistically meaningful changes relative to the neighboring windows.
3. *Workload drift.* Workload drift refers to changes, whether systematic or random, that can occur to a workload over time. This term is defined in [2]. Sections below will further elaborate on this important concept.

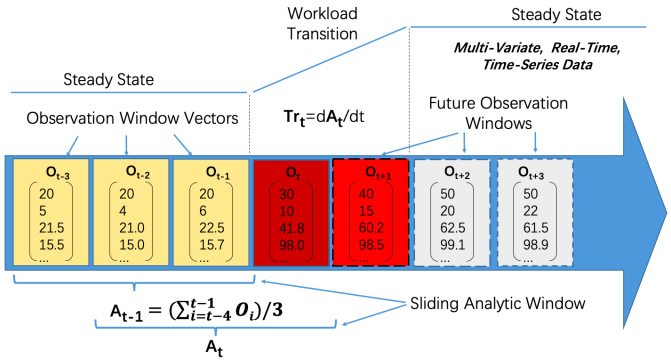

Fig. 2. Workloads are periods of steady state processing connected by workload transitions [2].

The broader set of processing performed by the system can be described as a sequence of workloads connected together by workload transitions. This is shown in Figure 2 [2].

This view of workload is more granular than that generally used by other researchers (for example in [9], [7]). Nevertheless, it allows for a more systematic and automated treatment of workload analysis and optimization.

6.2 Autonomic Architecture Principles

The MAPE-K architecture, shown in Figure 1 implements a feedback loop designed to achieve the key properties essential for an autonomic systems, as defined by IBM. These properties were defined as:

- *Self-configuration.* An autonomic system must be able to configure, or reconfigure, itself without requiring human intervention.
- *Self-healing.* In the event of a fault, an autonomic system must be able to find a way to mitigate the fault without human intervention.
- *Self-optimization.* An autonomic system must be able to optimize its performance within specified goals without human intervention.
- *Self-protection.* An autonomic system must be able to defend itself against misuse without human intervention.

Within the framework of the MAPE-K reference architecture (see Figure 1) the Autonomic Manager uses Sensors to Monitor a Managed Resource. Monitoring generates Knowledge that pertains to the Managed Resource. This knowledge is Analyzed and used to Plan the system's response to changes. During the Execute phase the Autonomic Manager uses the Effectors to affect the Managed Resource in accordance with the Plan.

The research presented in this paper focuses on achieving the self-optimization characteristic. By extension some aspects of the self-configuration, self-healing,

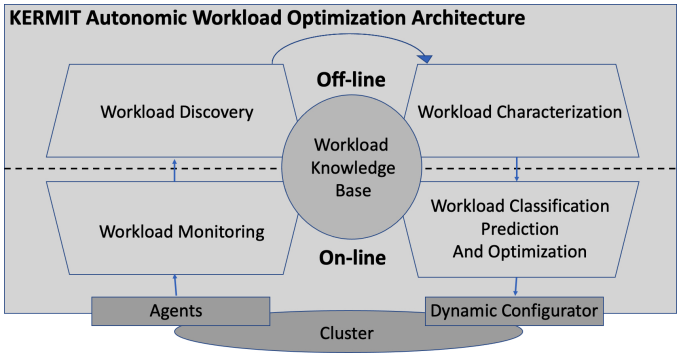

Fig. 3. Mapping the KERMIT sub-systems and components onto the MAPE-K reference architecture.

and self-protection characteristics are drawn in as well. This is due to the fact that in order to change the performance characteristics of the big data stack changes to the configuration are required.

The self-healing and self-protection properties were not directly the subject of this investigation. Nevertheless, it can be said that the KERMIT architecture partially addresses those properties as well. For example, failure of one or more nodes in the cluster can present itself as the appearance of new workload types because the loss of the CPU and memory resources will alter the observed feature vector of the observation windows. The KERMIT architecture, as described below, will be able to react to this and find a new optimum. This type of response can-be characterized as partial self-healing. The self-healing is partial in this case because KERMIT does not address bring replacement nodes on-line. Similarly, KERMIT's ability to respond to new workload types can be viewed as partial protection form a sophisticated type of DoS/DDos attack targeting back end systems.

Sections below describe how the KERMIT architecture layers map onto the MAPE-K reference architecture, and describe key components and algorithms in detail.

6.3 The KERMIT Architecture

Figure 3 shows the mapping of the key KERMIT sub-systems and components onto the MAPE-K reference architecture. The KERMIT architecture is broadly divided into two subsystems: 1 - On-line; 2 - Off-Line.

The On-line sub-system operates in real-time, and includes the Workload Monitoring sub-system, the Agents sub-system, the Dynamic Configurator subsystem, and the Workload Classification, Prediction and Optimization sub-system. The Off-line sub-system operates asynchronously in batch mode, and includes the Workload Discovery and the Workload Characterization sub-systems. All

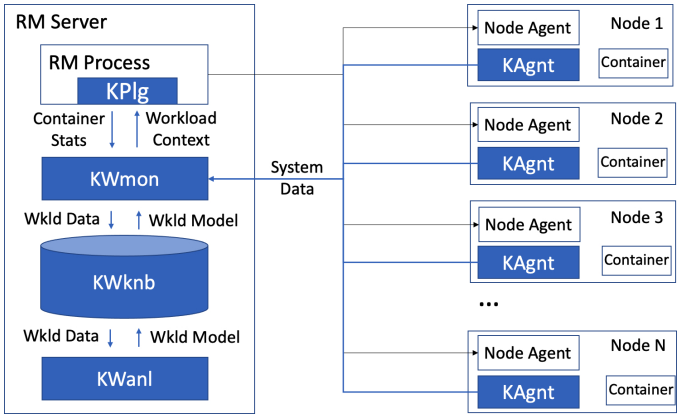

Fig. 4. The high-level KERMIT components shown in relation to the resource manager and the big data cluster.

KERMIT sub-systems both update, and read from the Workload Knowledge Base sub-system.

Figure 4 shows the high-level components of the KERMIT architecture, and how it relates to a typical big data cluster. The On-line sub-system interfaces with the Resource Manager process (RM Process in Figure 4) using a plug-in (KPlg). The core of the Workload Monitoring sub-system is the KERMIT Workload Monitor (KWmon) component. This component is a streaming engine that receives messages from the KPlg component and from the KERMIT system monitoring agents (KAgnt) deployed on cluster nodes.

The KERMIT Workload Analyzer (KWanl in Figure 4) is the main component in the Off-line sub-system. The KWmon and the KWanl components implement real-time and batch machine learning pipelines form the core of the KERMIT architecture. The details of each sub-system architecture are described in the sections below.

6.4 Workload Classification, Prediction, and Optimization

Figure 5 shows the logical architecture of the KERMIT workload knowledge base. The KERMIT workload knowledge base is implemented using a shared distributed file system, such as HDFS. The KERMIT workload knowledge base contains a Landing Zone (LZ in Figure 5), a Transformation Zone (TZ), and an Analytics Zone (AZ).

The raw time-stamped data generated by the KERMIT agents and the KERMIT plug-in components are stored in the LZ. These data are mostly loosely structured text files or log files. There is one file for each agent, and one for the KERMIT plug-in. The KERMIT workload monitor reads these data in real-time, treating each file as a streaming source, as new time-stamped data are appended. It transforms the time-stamped data into a structured format and

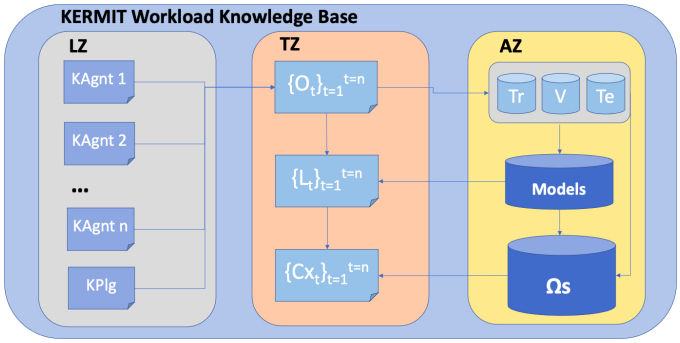

Fig. 5. Logical architecture of the KERMIT knowledge base.

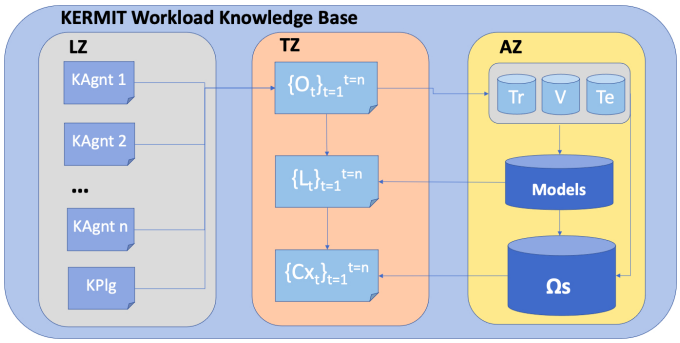

Fig. 6. The comparison of the classification accuracy for different machine learning algorithms [4].

aggregates them into observation windows O_t with the associated feature vector F_t.

The KERMIT workload monitor applies the workload classification pipeline described in [2] to transform the input stream of observation windows $\{O_t\}_{t=1}^{n}$ into a stream of labels $\{Y_t\}_{t=1}^{n}$, and writes out a sequence of workload context objects $\{\mathbb{C}_t\}_{t=1}^{n}$.

The workload context at observation window t, $\mathbb{C}_t$, contains the following information:

- The workload label for the current observation window t.
- The predicted workload label for time horizon t+1.
- The predicted workload label for time horizon t+5.
- The predicted workload label for time horizon t+10.

The KERMIT plug-in code is called whenever the resource manager responds to a resource request from an analytic framework. The integration between the KERMIT plug-in and the resource manager is described in [3]. The KERMIT code intercepts the resource managers' response to the analytic frameworks'

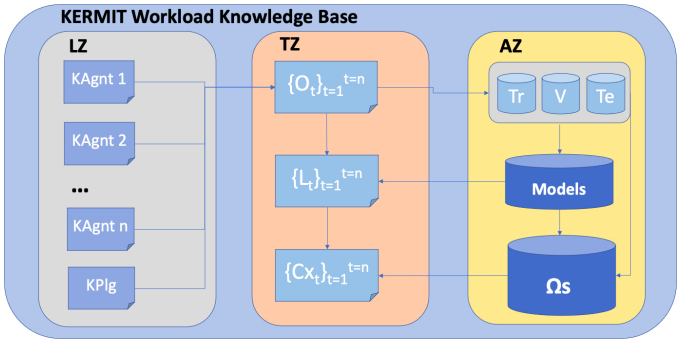

Fig. 7. TransitionClassfier algorithm performance [2].

resource request and engages the low-overhead, conceptually simple, Explorer search algorithm to find an optimal configuration [3].

The Explorer demonstrated that it was capable of achieving up to 30% better performance than rule-of-thumb tuning by a human practitioner, and up to 92.5% tuning efficiency relative to the best possible tuning by a human practitioner. However, many real-world workloads are repetitive in nature. For example, the job to tally up the daily financial results is run at the same time every day. Some jobs are executed many times daily. If we consider that our definition of the term workload is more granular than a job - it stands to reason that the same workload type may be encountered many times during the day, or even the hour.

It only makes sense to enhance the KERMIT plug-in architecture, described as the KERMIT Architecture in [3], with the ability to recognize workloads that have already been executed, and for which the optimal configuration has already been found. This allows KERMIT to avoid repeating the same parameter search multiple times, and achieve further performance gains under realistic workload conditions.

The KERMIT plug-in extends the KERMIT Analyzer component presented in [3] with the capability to read the workload context stream generated by the KERMIT workload monitor. When called by resource manager it first checks the workload monitor stream output and retrieves the workload context object. It then checks the label of the currently executing workload, and retrieves the workload descriptor object from KERMIT workload knowledge base.

The workload descriptor object contains the following items of information:

- Statistics for each feature in the workload feature vector (described further below).
- Centroid values for the workload (described further below).
- A true of false flag indicating whether or not an optimal configuration has been found.
- A a set of configuration values to be used.

The main high-level algorithm used by the KERMIT plug-in is described in Algorithm 1. When the resource manager calls the KERMIT plug-in code in response to a resource request from one of the analytic frameworks, the Analyzer component reads the workload context stream $\{\mathbb{C}_t\}_{t=1}^{n}$. It reads in the latest context $\mathbb{C}_t$, and compares the current time with the observation window associated with the context to make sure that the plug-in and the KERMIT workload monitor are in-sync. If they are not then an error is logged and a default configuration is used until the error is resolved.

Once the context has been read in, the plug-in checks the workload type label for the current observation window $\mathbb{C}_t.currentLabel$. When the KERMIT workload monitor first starts and the different workload types have not yet been determined, the type will be UNKNOWN. In this case the KERMIT plug-in will simply use the default configuration $\mathfrak{J}^D$ as the optimal configuration $\mathfrak{J}_i^o$ for this workload. The plug-in will wait until the off-line sub-system workload discovery catches up, and will continue to check the current workload type each time it is called by the resource manager.

Once the KERMIT plug-in encounters a current workload label that is known, it will check the KERMIT WorkloadDB to see if an optimal configuration for this label has already been established. If so, it will simply retrieve this configuration from the WorkloadDB.

If there is no optimal configuration associated with this workload, then the plug-in will check if workload drift has been detected. If so, then there will be a configuration in the WorkloadDB associated with this workload label, but this configuration will not be optimal. In this case the KERMIT plug-in will retrieve this configuration and pass it to the Explorer algorithm to initiate a local search described in [3]. Once the local search finds the optimal configuration, the plug-in will update the KERMIT WorkloadDB with this configuration, and set the optimal configuration field in the database to the value "True".

If workload drift has not been detected then the WorkloadDB will not have a sub-optimal configuration stored for the workload because the workload has just been detected by the off-line sub-system. In this case the KERMIT plug-in will start the Explorer algorithm's global search described in [3]. Once the global search finds the optimal configuration the KERMIT plug-in will update the WorkloadDB with this configuration, and set the optimal configuration field in the database to the value "True".

The algorithms used to identify different workload types and detect workload drift are discussed in the section below.

7 The KERMIT Off-Line Sub-System Architecture

Figure 8 shows the high-level processing pipeline flow in the off-line sub-system. The high-level component KERMIT workload analyzer (KWanl in Figure 4) implements the off-line analytic pipeline which performs the following stages of processing:

1. Workload discovery and labeling.

Algorithm 1 Main KERMIT plug-in algorithm.

Require: $\{\mathbb{C}_t\}_{t=1}^{n} \neq \{\}$ {Workload context stream must be started and available.}
Require: $O_t \neq \emptyset$ {Current observation window.}
Ensure: $\mathfrak{J}_i^o$ {The optimal configuration for this workload}.
  if $\mathbb{C}_t.currentLabel$ is UNKNOWN **then**
    $\mathfrak{J}_i^o \leftarrow \mathfrak{J}^D$ {Use the default configuration for unknown workloads.}
  else
    if $\mathbb{C}_t.currentLabel$ has optimal config in WorkloadDB **then**
      $\mathfrak{J}_i^o \leftarrow$ get optimal config from WorkloadDB
    else
      if $\mathbb{C}_t.currentLabel$ has workload drift **then**
        $\mathfrak{J}_i^o \leftarrow$ Explorer.localSearch($\mathfrak{J}_i$) {Do a local search starting with the last good configuration.}
      else
        $\mathfrak{J}_i^o \leftarrow$ Explorer.globalSearch()
      end if
      Update WorkloadDB with $\mathfrak{J}_i^o$
    end if
  end if

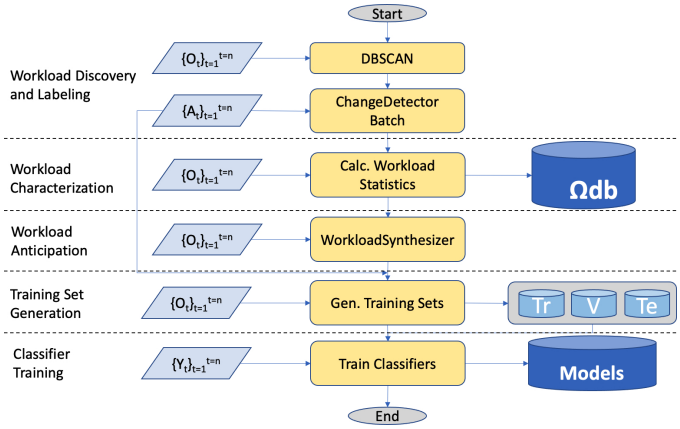

Fig. 8. The high-level steps in the off-line sub-system processing pipeline.

2. Workload characterization.
3. Workload anticipation.
4. Training set generation.
5. Classifier training.

Sections below begin by discussing processing pipeline and the algorithm used to detect and identify new workloads. The next section builds on this discussion to describe the algorithm for detecting workload drift.

7.1 Workload Discovery, Characterization, and Drift Detection

As discussed in the sections above, the KERMIT workload monitor stores the aggregated stream of workload windows $\{O_t\}_{t=1}^{n}$ in the transformation zone of the KERMIT knowledge base (see Figure 5). The high-level algorithm for workload discovery, characterization and drift detection is given in Algorithm 2.

Algorithm 2 The workload discovery and drift detection algorithm.

Require: $\{O_t\}_{t=1}^{n} \neq \emptyset$ {Landed observation window time-series.}
Ensure: $\{\mathfrak{Y}\}_{\mathfrak{j}}^{\mathfrak{k}}$ {Set of identified workload labels.}
  run ChangeDetector.batch() to identify transition windows
  extract transition windows from $\{O_t\}_{t=1}^{n}$
  run DBSCAN on $\{O_t\}_{t=1}^{n}$ to get a set of clusters
  for all clusters in the set **do**
    calculate workload characterization statistics
    if find match in WorkloadDB is True **then**
      Y_j gets matched label from WorkloadDB
      if L2 norm between the mean vectors of workload characterizations differ by more than ϵ **then**
        update isDrifting to True in WorkloadDB
        update workload characterization for matching label in WorkloadDB with new data
      end if
    else
      generate new label for the new workload
      insert new workload label and characterization data into WorkloadDB
    end if
  end for

The algorithm begins by using the ChangeDetector component in batch mode to scan the persisted time series of observation window data, and flag workload transition windows as described in [2]. The logic of the batch operation is exactly the same as in the real-time use case. The workload transition windows are then removed from the original set into a separate set, and clustering analysis is performed on the now filtered set of workload observation windows.

Figure 10 shows the key performance metrics for several different clustering algorithms. The effectiveness of each algorithm was evaluated using time-series workload data recorded during the execution of Apache Hadoop and Spark benchmarks. Clustering results were compared to ground truth interpretation made by a human specialist using Apache Hadoop and Spark logs.

The key metrics indicated in the figure are Awt and Purity. Purity indicates how many of the observation windows were classified correctly by the on-line sub-system. The Awt metric is also an accuracy-type metric. It measures how accurately the algorithm was able to identify different workload types. For example, if the benchmark executed 3 different workload types and the algorithm

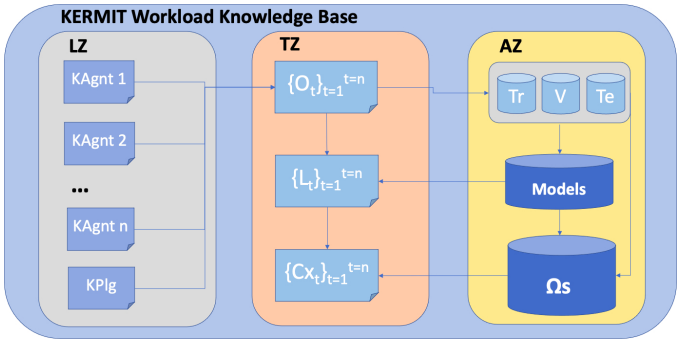

Fig. 9. ChangeDetector algorithm performance [2].

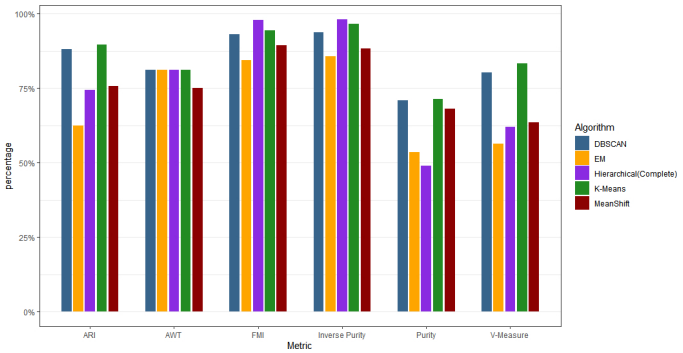

Fig. 10. Workload discovery performance for clustering algorithms.

detected 3 clusters whose centroids call within the observation window range of each workload type, then the Awt metric for this algorithm would be 100%.

Workload discovery in KERMIT is accomplished by running the DBSCAN algorithm on filtered observation window data. DBSCAN identifies clusters within the observation window data. Each cluster represents a distinct workload type.

The next step (see Algorithm 2) is to check if the newly identified workloads have been encountered before by calculating the workload characterization statistics for each cluster, and comparing with workload characterization statistics for workloads already identified and stored in WorkloadDB. This is accomplished by using the ChangeDetector off-line to compare statistical data.

Workload characterization involves calculating the relevant statistics for each subset of observation windows that were grouped by the DBSCAN algorithm into the corresponding cluster. A full set of statistics, including the mean, the standard deviation, the max, the min, the 90th percentile, and the 75th percentile are calculated. This set of statistics is the workload characterization.

If a matching workload characterization is found in WorkloadDB (as identified by the ChangeDetector), then this is an existing workload. All of the windows in the cluster get tagged with the matched workload label from WorkloadDB.

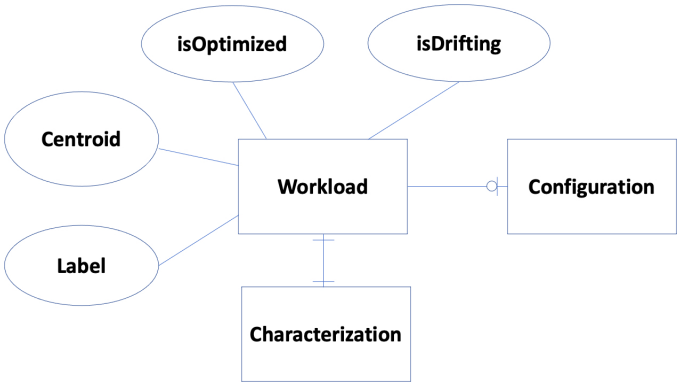

Fig. 11. Entity-relationship model of the WorkloadDB schema.

The next step is to check the workload for drift. This is accomplished by calculating the L2 norm of the distance between the mean vectors of the new cluster and the one stored in WorkloadDB. If the difference is larger than the configurable hyper-parameter ϵ, then drift has occurred, and the WorkloadDB is updated with the workload characterization for the new cluster.

If no matching workload is found in the WorkloadDB, then the KERMIT workload analyzer generates a unique integer label for the cluster. The generated workload labels do not need to be human-legible. They just need to be unique to each identified cluster of observation windows. Currently KERMIT implements a simple integer counter, because this facilitates the generation of libsvm files for model training. Then new label, along with the workload characterization, is inserted into WorkloadDB.

Figure 11 shows the WorkloadDB data model. Each workload is uniquely identified by its unique automatically generated label. Each workload contains the workload characterization statistics, a true/false field indicating whether the optimal configuration has been found, and a true/false field indicating whether workload drift has been detected. Each workload can have one configuration stored in the WorkloadDB. This configuration may or may not be the optimal configuration.

When a workload is initially identified, it will not have a configuration associated with it. This is because the configuration search is performed in real time by the on-line sub-system. Once the KERMIT plug-in performs the global search for the workload, it will update the WorkloadDB with the optimal configuration, and set the field indicating whether the optimal configuration has been found to True.

The next time clustering analysis is performed this occurs on a set interval, and on a new set of $\{O_t\}_{t=n+1}^{n+k}$ data collected during the last interval, where k is a constant hyper-parameter that controls the length or the batch used for clustering analysis. The entire process described in Algorithm 2 is repeated.

This approach allows KERMIT to continuously learn new workloads. New workloads are added to the WorkloadDB as they are detected during the off-line batch processing. Already known workloads are protected against workload drift because their characterizations are regularly updated. Workloads are never deleted from WorkloadDB. Thus KERMIT retains a long-term memory of workloads, and the ability of the KERMIT on-line sub-system to recognize workloads improves over time.

7.2 Automated Classifier Training

The KERMIT on-line analytic pipeline includes several classifiers described in [4], and [2]. Paragraphs below provide a quick overview of their function and purpose, discuss their training requirements and describe how the process is automated.

The KERMIT on-line classification pipeline uses the following classifiers:

- *ChangeDetector*. This statistical classifier is a binary classifier that simply uses the Welch's statistical test to distinguish steady state processing from workload transitions. This classifier does not require off-line training.
- *WorkloadClassifier*. This classifier is based on the random forest ensemble algorithm This is a supervised classifier that does require off-line training.
- *TransitionClassifier*. This classifier is also based on the random forest ensemble algorithm. It also is a supervised classifier that does require off-line training.
- *ZSL Workload Classifier*. This component, described in [1] re-uses the *WorkloadClassifier* class, and introduces the *WorkloadSynthesizer* component. This component needs to be trained off-line. It also generates synthetic class instances, which need to be merged into the training process for the *WorkloadClassifier*.
- *WorkloadPredictor*. This component is based on an LSTM neural network algorithm.

The training pipeline performs the following high-level steps (some of the steps performed as part of workload discovery are repeated for completeness):

1. Extract observation window range for each workload in WorkloadDB.
2. Use workload observation window id set to extract a set of analytic windows from the analytic window stream $\{A_t\}_{t=1}^{n}$ created by the Workload Monitor. For every observation window there is a matching analytic window. This becomes the training set for the WorkloadClassifier Dp_{Ω}^{Tr}.
3. Establish window ranges for workload transitions by scanning the sequence of analytic windows and marking ranges of windows that connect window sets that belong to each workload cluster, and correlating with workload transitions identified during the workload discovery phase.
4. Generate labels for each workload transition type. The same algorithm is used for label generation as that used for workloads. The labels don't need to be human-readable, just unique and consistent.

5. Transform the analytic window sequence $\{A_t\}_{t=1}^{n}$ to a rate of change sequence $\{A'_t\}_{t=1}^{n}$.
6. Extract transition windows from the transformed sequence - this forms the training set for the TransitionClassifier D_{Δ}^{Tr}.
7. Execute the WorkloadSyntesizer component on Dp_{Ω}^{Tr} to account for possible anticipated hybrid, multi-user workloads. This involves the following steps:
   (a) Generating the Class Descriptor file described in [1]. Each workload entry in the Workload DB is used as a pure class. Possible hybrid workloads are constructed by pairing the pure workloads.
   (b) Generate labels for the anticipated, hybrid workloads using the same algorithm as for pure workload classes.
   (c) Update the WorkloadDB with synthetic class prototypes - these contain the same information as the workload characterizations calculated for the seen classes.
   (d) Merge the synthetic workload instances with the observed workload instances to construct the final, merged, WorkloadClassifier training set D_{Ω}^{Tr}.
8. Generate the training set for the WorkloadPredictor component $D_{\mho}^{Tr}$ by extracting segments from the label sequence $\{Y_t\}_{t=n+1}^{n+k}$.
9. Train the classifiers.

Most of the steps described above, with the exception of steps needed to train the WorkloadClassifier, can be executed in parallel given sufficient compute resources.

8 Conclusion

This paper presents the first architecture intended for autonomic optimization of big data workloads. The KERMIT architecture implements an autonomic feedback look that includes on-line and off-line processing stages. It uses machine learning pervasively to analyze workload characteristics, identify new workload types, detect changes in real-time, classify workloads, and predict future workload types and characteristics. Change detection, workload classification, workload prediction, and parameter search are performed on-line, in real time. Classifier training is performed off-line as a batch machine learning pipe-line.

Experimental investigations focused on the critical proof points of the autonomic feedback loop demonstrate that the KERMIT architecture can:

- Real-time workload classification with up to 90% accuracy [2].
- Detect workload changes in real-time with up to 99% accuracy [2].
- Predict workload type with up to 96% accuracy [2].
- Anticipate new, unseen workload types, and classify them with up to 83% accuracy [1].

The KERMIT architecture, as discussed above, can anticipate the appearance of new unseen, multi-user, hybrid workloads that can present a mix of

characteristics observed with the currently identified workloads. This is a capability that has not been previously described in any of the earlier works focused on small data and cloud workload segments.

The ability to adjust to workload drift is another key element of the KERMIT architecture. The on-line sub-system will do its best to classify workloads in real-time. If a previously unseen and unanticipated workload is encountered, the KERMIT on-line sub-system will initially classify it as one of the known workload types, with the closest characteristics, and use the best available configuration for that workload. This is often better, in terms of reducing the tuning overhead, then immediately performing a global search for that workload.The new workload will be discovered by the off-line sub-system the next time it performs clustering analysis.

This architecture can operate with minimal configuration by a human administrator. Although there are still a number hyper-parameters that need to be set, these, unlike many of the Apache Hadoop and Spark configuration settings, do not require frequent tuning. For the most part these hyper-parameters can be left at their default settings. These default settings (for example the μ hyper-parameter for the DBSCAN algorithm) in many cases apply to a broad range of conditions and are well-documented in the scientific and technical literature.

Bibliography

[1] A1 (2020) Zero-shot machine learning technique for classification of multi-user big data workloads. In: 2020 IEEE International Conference on Big Data (Big Data), Atlanda, GA, USA., IEEE, p in print

[2] A1, A2 (2019) Autonomic workload change classification and prediction for big data workloads. In: 2019 IEEE International Conference on Big Data 2019 (Big Data), Los Angeles, CA, USA., IEEE, pp 2835 – 2844

[3] A1, A2, A3, A4, A5 (2016) 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, pp 317–324

[4] A1, A2, A3, A4 (2019) Machine-learning based spark and hadoop workload classification using container performance patterns. In: C. Zheng and J. Zhan (Eds) 2018 International Symposium on Benchmarking, Measuring and Optimizing (Bench' 18), Springer, vol 11459, pp 118 – 130

[5] Carrera D, Steinder M, Whalley I, Torres J, Ayguade E (2012) Autonomic placement of mixed batch and transactional workloads. IEEE Transactions on Parallel and Distributed Systems 23(2):219 – 231

[6] Cherkasova L, Ozonat K, Mi N, Symons J, Smirni E (2014) 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), IEEE, pp 452 – 461

[7] Elnaffar S, Martin P (2009) The psychic–skeptic prediction framework for effective monitoring of dbms workloads. Data & Knowledge Engineering 68:393–414

[8] Gergin I, Simmons B, Litoiu M (2014) A decentralized autonomic architecture for performance control in the cloud. In: 2014 IEEE International Conference on Cloud Engineering, Boston, MA, IEEE, pp 574 – 579

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

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

[11] IBM (2005) An architectural blueprint for autonomic computing. IBM 7:1 – 31

[12] Lei Z, Hu B, Guo J, Hu L, Shen W, Lei Y (2014) Scalable and efficient workload hotspot detection in virtualized environment. Cluster Computing 17:1253–1264

[13] Ma L, Aken DV, Hefny A, Mezerhane G, Pavlo A, Gordon GJ (2018) Query-based workload forecasting for self-driving database management systems. In: Proceedings of 2018 International Conference on Management of Data (SIGMOD'18), ACM

[14] Movahedi Z, Ayari M, Langar R, Pujolle G (2012) A survey of autonomic network architectures and evaluation criteria. IEEE Communications Surveys & Tutorials 14(2):464 – 490

[15] Nouri SMR, Li H, Venugopal S, Guo W, He M, Tian W (2019) Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications. Future Generation Computer Systems 94:765 – 780

[16] Raza B, Sher A, Afzal S, Malik AK, Anjum A, Kumar YJ, Faheem M (2019) Autonomic workload performance tuning in large-scale data repositories. Knowledge and Information Systems 61(2):27 – 63