Automated Performance Management for the Big Data Stack

Anastasios Arvanitis, Shivnath Babu, Eric Chu,
Adrian Popescu, Alkis Simitsis, Kevin Wilkinson
Unravel Data Systems
{tasos,shivnath,eric,adrian,alkis,kevinw}@unraveldata.com

ABSTRACT

More than 10,000 enterprises worldwide today use the *big data stack* that is composed of multiple distributed systems. At Unravel, we have worked with a representative sample of these enterprises that covers most industry verticals. This sample also covers the spectrum of choices for deploying the big data stack across on-premises datacenters, private cloud deployments, public cloud deployments, and hybrid combinations of these. In this paper, we aim to bring attention to the performance management requirements that arise in big data stacks. We provide an overview of the requirements both at the level of individual applications as well as holistic clusters and workloads. We present an architecture that can provide automated solutions for these requirements and then do a deep dive into a few of these solutions.

1. BIG DATA STACK

Many applications in fields like health care, genomics, financial services, self-driving technology, government, and media are being built on what is popularly known today as the *big data stack*. What is unique about the big data stack is that it is composed of multiple distributed systems. The typical evolution of the big data stack in an enterprise usually goes through the following stages (also illustrated in Figure 1).

Big Data Extract-Transform-Load (ETL): Storage systems like HDFS, S3, and Azure Blob Store (ABS) are used to store the large volumes of structured, semi-structured, and unstructured data in the enterprise. Distributed processing engines like MapReduce, Tez, and Pig/Hive (usually running on MapReduce or Tez) are used for data extraction, cleaning, and transformations of the data.

Big Data Business Intelligence (BI): MPP SQL systems like Impala, Presto, LLAP, Drill, BigQuery, RedShift, or Azure SQL DW are added to the stack; sometimes alongside incumbent MPP SQL systems like Teradata and Vertica. Compared to the traditional MPP systems, the newer ones have been built to deal with data stored in a different distributed storage system like HDFS, S3, or ABS. These systems power the interactive SQL queries that are common in business intelligence workloads.

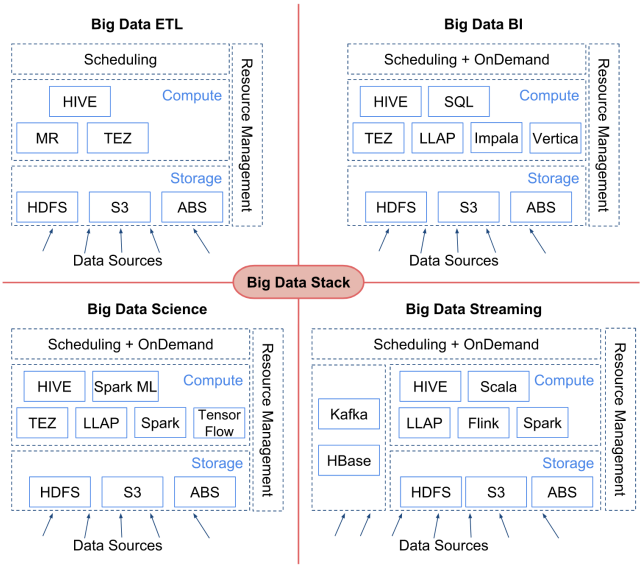

Figure 1: Evolution of the big data stack in an enterprise

Big Data Science: As enterprises mature in their use of the big data stack, they start bringing in more data-science workloads that leverage machine learning and AI. This stage is usually when the Spark distributed system starts to be used more and more.

Big Data Streaming: Over time, enterprises begin to understand the importance of making data-driven decisions in near real-time as well as how to overcome the challenges in implementing them. Usually at this point in the evolution, systems like Kafka, Cassandra, and HBase are added to the big data stack to support applications that ingest and process data in a continuous streaming fashion.

Industry analysts estimate that there are more than 10,000 enterprises worldwide that are running applications in production on a big data stack comprising three or more distributed systems [1]. At Unravel, we have worked closely with around 50 of these enterprises, and have had detailed conversations with around 350 more of these enterprises. These enterprises cover almost every industry vertical and run their stacks in on-premises datacenters, private cloud deployments, public cloud deployments, or in hybrid combinations (e.g., regularly-scheduled workloads like the Big Data

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2019.
9th Biennial Conference on Innovative Data Systems Research (CIDR '19) January 13-16, 2019 , Asilomar, California, USA.

ETL runs in on-premises datacenters while the non-sensitive data is replicated to one or more public clouds where ad-hoc workloads like Big Data BI run). Sizes of these clusters vary from few tens to few thousands of nodes. Furthermore, in some of the production deployments on the cloud, the size of an auto-scaling cluster can vary in size from 1 node to 1000 nodes in under 10 minutes.

The goal of this paper is to bring attention to the performance management requirements that arise in big data stacks. A number of efforts like *Polystore* [7], *HadoopDB* [3], and *hybrid flows* [18] have addressed challenges in stacks composed of multiple systems. However, their primary focus was not on the performance management requirements that we address. We split these requirements into two categories: application performance requirements and operational performance requirements. Next, we give an overview of these requirements.

1.1 Application Performance Requirements

The nature of distributed applications is that they interact with many different components that could be independent or interdependent. This nature is often referred to in popular literature as "having many moving parts." In such an environment, questions like the following can become nontrivial to answer:

- *Failure:* What caused this application to fail, and how can I fix it?
- *Stuck:* This application seems to have made little progress in the last hour. Where is it stuck?
- *Runaway:* Will this application ever finish, or will it finish in a reasonable time?
- *SLA:* Will this application meet its SLA?
- *Change:* Is the behavior (e.g., performance, resource usage) of this application very different from the past? If so, in what way and why?
- *Rogue/victim:* Is this application causing problems on my cluster; or vice versa, is the performance of this application being affected by one or more other applications?

It has to be borne in mind that almost every application in the big data stack interacts with multiple distributed systems. For example, a SQL query may interact with Spark for its computational aspects, with YARN for its resource allocation and scheduling aspects, and with HDFS or S3 for its data access and IO aspects. Or, a streaming application may interact with Kafka, Flink, and HBase (as illustrated in Figure 1).

1.2 Operational Performance Requirements

Many performance requirements also arise at the "macro" level compared to the level of individual applications. Examples of such requirements are:

- Configuring resource allocation policies in order to meet SLAs in multi-tenant clusters [9, 19].
- Detecting rogue applications that can affect the performance of SLA-bound applications through a variety of low-level resource interactions [10].
- Configuring the 100s of configuration settings that distributed systems are notoriously known for having in order to get the desired performance.
- Tuning data partitioning and storage layout.
- Optimizing dollar costs on the cloud. All types of resource usage on the cloud cost money. For example, picking the right node type for a cloud cluster can have a major impact on the overall cost of running a workload.
- Capacity planning using predictive analysis in order to account for workload growth proactively.
- Identify in an efficient way who (e.g., user, tenant, group) is running an application or a workload, who is causing performance problems, and so on. Such an accounting process is typically known as 'chargeback'.

2. ARCHITECTURE OF A PERFORMANCE MANAGEMENT SOLUTION

Addressing the challenges from Section 1 needs an architecture like the one shown in Figure 2. Next, we will discuss the main components of this architecture.

Full-Stack Data Collection: To answer questions like those raised in Sections 1.1 and 1.2, monitoring data is needed from every level of the big data stack. For example, (i) SQL queries, execution plans, data pipeline dependency graphs, and logs from the application level; (ii) resource allocation and wait-time metrics from the resource management and scheduling level; (iii) actual CPU, memory, and network usage metrics from the infrastructure level; (iv) data access and storage metrics from the file-system and storage level; and so on. Collecting such data in nonintrusive and low-overhead ways from production clusters remains a major technical challenge, but this problem has received attention in the database and systems community [8].

Event-driven Data Processing: Some of the clusters that we work with are more than 500 nodes in size and run multiple hundreds of thousands of applications every day across ETL, BI, data science, and streaming. These deployments generate tens of terabytes of logs and metrics every day. The *volume* and *velocity* challenges from this data are definitely nontrivial. However, the *variety* and *consistency* challenges here, to the best of our knowledge, have not been addressed by the database and systems community.

The Variety Challenge: The monitoring data collected from the big data stack covers the full sprectrum from unstructured logs to semistructured data pipeline dependency DAGs and to structured time-series metrics. Stitching this data together to create meaningful and useable representations of application performance is a nontrivial challenge.

The Consistency Challenge: Monitoring data has to be collected independently and in real-time from various moving parts of the multiple distributed systems that comprise the big data stack. Thus, no prior assumptions can be made about the timeliness or order in which the monitoring data arrives at the processing layer in Figure 2. For example, consider a Hive Query Q that runs two MapReduce jobs J_1 and J_2. Suppose, J_1 and J_2 in turn run 200 containers $C_1, \ldots, C_{200}$ and 800 containers $C_{201}, \ldots, C_{1000}$ respectively. One may expect that the monitoring data from these

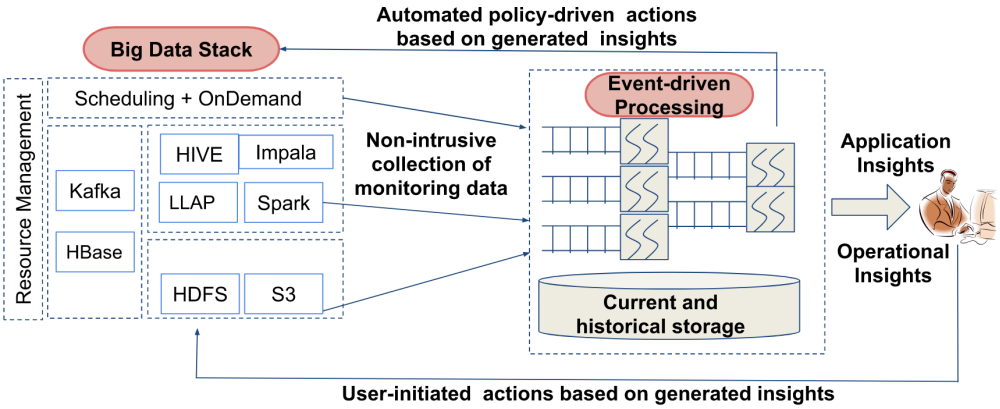

Figure 2: Architecture of a performance management platform for the big data stack

components arrives in the order Q, J_1, $C_1, \ldots, C_{200}$, J_2, $C_{201}, \ldots, C_{1000}$. However, the data can come in any order, e.g., J_1, $C_1, \ldots, C_{100}$, Q, J_2, $C_{101}, \ldots, C_{200}$, $C_{201}, \ldots, C_{1000}$.

As a result, the data processing layer in Figure 2 has to be based on event-driven processing algorithms whose outputs converge to same final state irrespective of the timeliness and order in which the monitoring data arrives. From the user's perspective, she should get the same insights irrespective of the timeliness and order in which the monitoring data arrives. An additional complexity that we do not have space to discuss further is the chance of some monitoring data getting lost in transit due to failure, network partitions, overload, etc. It is critical to account for this aspect in the overall architecture.

ML-driven Insights and Policy-driven Actions: Enabling all the monitoring data to be collected and stored in a single place opens up interesting opportunities to apply statistical analysis and learning algorithms to this data. These algorithms can generate insights that, in turn, can be applied manually by the user or automatically based on configured policies to address the performance requirements identified in Sections 1.1 and 1.2. Unlike the big data stack that we consider in this paper, efforts such as *self-driving databases* [2, 16] address similar problems for traditional database systems like MySQL, PostgreSQL, and Oracle, and in the cloud (e.g., [6, 12, 13, 14]). In the next section, we will use example problems to dive deeper into the solutions.

3. SOLUTIONS DEEP DIVE

3.1 Application Failure

In distributed systems, applications can fail due to many reasons. But when an application fails, users are required to fix the cause of the failure to get the application running successfully. Since applications in distributed systems interact with multiple components, a failed application throws up a large set of raw logs. These logs typically contain thousands of messages, including errors and stacktraces. Hunting for the root cause of an application failure from these messy, raw, and distributed logs is hard for experts, and a nightmare for the thousands of new users coming to the big data stack. The question we will explore in this section is how to automatically generate insights into a failed application in a multi-engine big data stack that will help the user get the application running successfully. We will use Spark as our example, but the concepts generalize to the big data stack.

Automatic Identification of the Root Cause of Application Failure: Spark platform providers like Amazon, Azure, Databricks, and Google Cloud as well as Application Performance Management (APM) solution providers like Unravel have access to a large and growing dataset of logs from millions of Spark application failures. This dataset is a gold mine for applying state-of-the-art artificial intelligence (AI) and machine learning (ML) techniques. Next, let us look at possible ways to automate the process of failure diagnosis by building predictive models that continuously learn from logs of past application failures for which the respective root causes have been identified. These models can then automatically predict the root cause when an application fails. Such actionable root-cause identification improves the productivity of Spark users significantly.

A distributed Spark application consists of a Driver container and one or more Executor containers. A number of logs are available every time a Spark application fails. However, the logs are extremely verbose and messy. They contain multiple types of messages, such as informational messages from every component of Spark, error messages in many different formats, stacktraces from code running on the Java Virtual Machine (JVM), and more. The complexity of Spark usage and internals make things worse. Types of failures and error messages differ across Spark SQL, Spark Streaming, iterative machine learning and graph applications, and interactive applications from Spark shell and notebooks (e.g., Jupyter, Zeppelin). Furthermore, failures in distributed systems routinely propagate from one component to another. Such propagation can cause a flood of error messages in the log and obscure the root cause.

Figure 3 shows our overall solution to deal with these problems and to automate root cause analysis (RCA) for Spark application failures. Overall, the solution consists of:

- Continuously collecting logs from a variety of Spark application failures
- Converting logs into feature vectors
- Learning a predictive model for RCA from these feature vectors

Data collection for training: As the saying goes: garbage in, garbage out. Thus, it is critical to train RCA models on representative input data. In addition to relying on logs

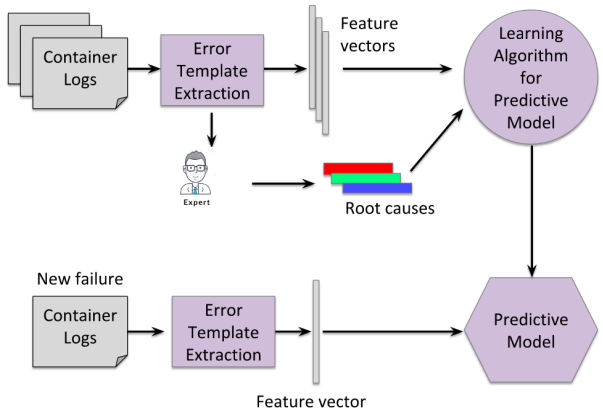

Figure 3: Approach for automatic root cause analysis (RCA)

from real-life Spark application failures observed on customer sites, we have also invested in a lab framework where root causes can be artificially injected to collect even larger and more diverse training data.

Structured versus unstructured data: Logs are mostly unstructured data. To keep the accuracy of model predictions to a high level in automated RCA, it is important to combine this unstructured data with some structured data. Thus, whenever we collect logs, we are careful to collect trustworthy structured data in the form of key-value pairs that we additionally use as input features in the predictive models. These include Spark platform information and environment details of Scala, Hadoop, OS, and so on.

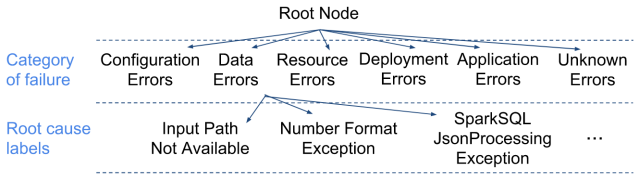

Figure 4: Taxonomy of failures

Labels: ML techniques for prediction fall into two broad categories: supervised learning and unsupervised learning. We use both techniques in our overall solution. For the supervised learning part, we attach root-cause labels with the logs collected from an application failure. This label comes from a taxonomy of root causes that we have created based on millions of Spark application failures seen in the field and in our lab. Broadly speaking, as shown in Figure 4, the taxonomy can be thought of as a tree data structure that categorizes the full space of root causes. For example, the first non-root level of this tree can be failures caused by: (i) Configuration errors, (ii) Deployment errors, (iii) Resource errors, (iv) Data errors, (v) Application errors, and (vi) Unknown factors.

The leaves of the taxonomy tree form the labels used in the supervised learning techniques. In addition to a text label representing the root cause, each leaf also stores additional information such as: (a) a description template to present the root cause to a Spark user in a way that she will easily understand, and (b) recommended fixes for this root cause. The labels are associated with the logs in one of two ways. First, the root cause is already known when the logs are generated, as a result of injecting a specific root cause we have designed to produce an application failure in our lab framework. The second way in which a label is given to the logs for an application failure is when a Spark domain expert manually diagnoses the root cause of the failure.

Input Features: Once the logs are available, there are various ways in which the feature vector can be extracted from these logs (recall the overall approach in Figure 3). One way is to transform the logs into a bit vector (e.g., 1001100001). Each bit in this vector represents whether a specific message template is present in the respective logs. A prerequisite to this approach is to extract all possible message templates from the logs. A more traditional approach for feature vectors from the domain of information retrieval is to represent the logs for a failure as a bag of words. This approach is mostly similar to the bit vector approach except for a couple of differences: (a) each bit in the vector now corresponds to a word instead of a message template, and (b) instead of 0s and 1s, it is more common to use numeric values generated using techniques like TF-IDF.

More recent advances in ML have popularized vector embeddings. In particular, we use the Doc2Vec technique [11]. At a high level, these vector embeddings map words (or paragraphs, or entire documents) to multidimensional vectors by evaluating the order and placement of words with respect to their neighboring words. Similar words map to nearby vectors in the feature vector space. The Doc2Vec technique uses a three-layer neural network to gauge the context of the document and relate similar content together.

Figure 5: Feature vector generation

Once the feature vectors are generated along with the label, a variety of supervised learning techniques can be applied for automatic RCA. We have evaluated both shallow as well as deep learning techniques, including random forests, support vector machines, Bayesian classifiers, and neural networks. The overall results produced by our solution are promising as shown in Figure 5. (Only one result is shown due to space constraints.) In this figure, 14 different types of root causes of failure are injected into runs of various Spark applications in order to collect a large set of logs. Figure 5 shows the accuracy of the approach in Figure 3 to predict the correct root cause based on a 75-25% split of training and test data. The accuracy of prediction is fairly high.

We are currently enhancing the solution in some key ways. One of these is to quantify the degree of confidence in the root cause predicted by the model in a way that users will easily understand. Another key enhancement is to speed up the ability to incorporate new types of application failures. The bottleneck currently is in generating labels. We are working on active learning techniques [4] that nicely pri-

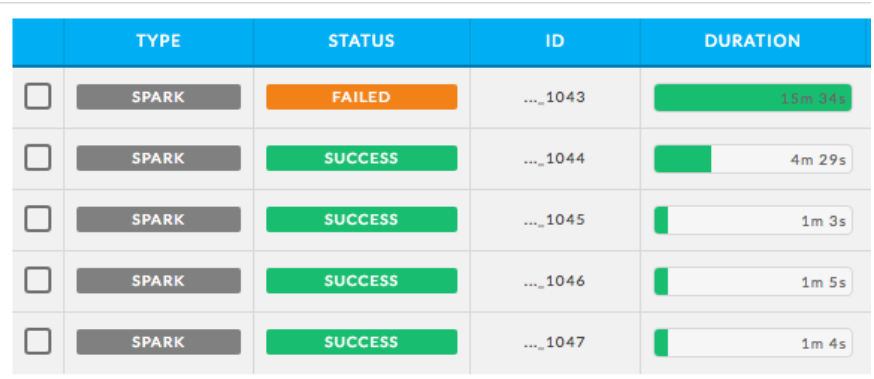

Figure 6: Automated tuning of a failed Spark application

oritize the human efforts required in generating labels. The intuition behind active learning is to pick the unlabeled failure instances that provide the most useful information to build an accurate model. The expert labels these instances and then the predictive model is rebuilt.

Automatic Fixes for Failed Applications: We did a deeper analysis of the Spark application failure logs available to us from more than 20 large-scale production clusters. The key findings from this analysis are:

- There is a "90-10" rule in the root cause of application failures. That is, in all the clusters, more than 90% of the failures were caused by less than 10 unique root causes.
- The two most common causes where: (i) the application fails due to out of memory (*OOM*) in some component; and (ii) the application fails due to timeout while waiting for some resource.

For application failures caused by OOM, we designed algorithms that, in addition to using examples of successful and failed runs of the application from history, can intelligently try out a limited number of memory configurations to get the application quickly to a running state; followed by getting the application to a resource-efficient running state.

As mentioned earlier, a Spark application runs one Driver container and one or more Executor containers. The application has multiple configuration parameters that control the allocation and usage of memory at the overall container level, and also at the level of the JVMs running within the container. If the overall usage at the container level exceeds the allocation, then the application will be killed by the resource management layer. If the overall usage at the Java heap level exceeds the allocation, then the application will be killed by the JVM.

The algorithm we developed to enable finding fixes automatically for OOM problems refines intervals based on successful and failed runs of the application. For illustration, let m represent the Executor container allocation. We define two variables, m_{lo} and m_{hi}, where m_{lo} is the maximum known setting of m that causes OOM; and m_{hi} is the minimum known setting of m that does not cause OOM. Given a run of the application that failed due to OOM while running with $m = m_{curr}$, we can update m_{lo} to:

$$m_{lo}^{new} = max(m_{lo}, m_{curr})$$

Given a run of the application that succeeded while running with $m = m_{curr}$, we can update m_{hi} to:

$$m_{hi}^{new} = min(m_{hi}, m_{obs})$$

Here, m_{obs} is the observed usage of m by the application in the successful run. At any point:

- A new run of the application can be done with m set to $\frac{m_{lo}+m_{hi}}{2}$
- m_{hi} is the most resource-efficient setting that is known to run the application successfully so far

The above approach is incomplete because the search space of configuration parameters to deal with OOM across the Driver, Executor, container, JVM, as well as a few other parameters that affect Spark memory usage is multi-dimensional. Space constraints prevent us from going into further details, but the algorithm from a related problem can be adapted to the OOM problem [5].

Figure 6 shows an example of how the algorithm works in practice. Note that the first run is a failure due to OOM. The second run, which was based on a configuration setting produced by the algorithm, managed to get the application running succcessfully. The third run—the next in sequence produced by the algorithm—was able to run the application successfully, while also running it faster than the second run. The third run was faster because it used a more memory-efficient configuration than the second run. Overallocation of memory can make an application slow because of the large wait to get that much allocated. Note how the algorithm is able to automatically find configurations that run the application successfully while being resource efficient. Thereby, we can remove the burden of manually troubleshooting failed applications from the hands of users, enabling them to focus entirely on solving business problems with the big data stack.

3.2 Cluster Optimization

Performing cluster level workload analysis and optimization on a multiplicity of distributed systems in big data stacks is not straightforward. Key objectives often met in practice include performance management, autoscaling, and cost optimization. Satisfying such objectives is imperative for both on-premises and cloud deployments and can serve different classes of users like Ops and Devs altogether.

Operational Insights: Toward this end, we analyze the metrics collected and provide a rich set of actionable insights, as for example:

- Insights into application performance issues; e.g., determine whether an application issue is due to code in-

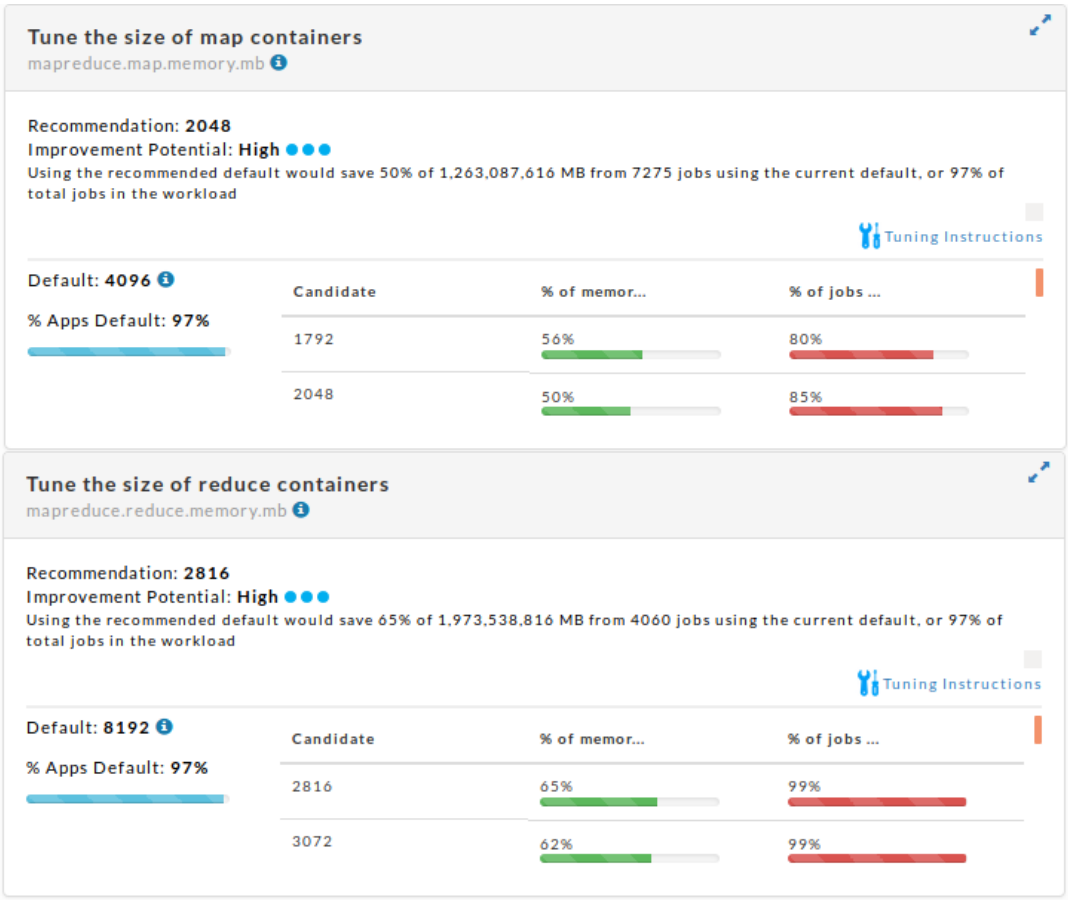

Figure 7: An example set of cluster wide recommendations

Before applying our cluster-level recommendations (8-day interval)

- Total 7275 jobs => 902 jobs/day
- Total vCore-Seconds = 18454324 (5126 vCore-Hours) => 641 vCore-Hours per day
- Total #containers = 395096 => 49387 containers/day
- Total #containers for MAP = 308371 (78%)
- Total #containers for REDUCE=86725(22%)

After applying our cluster-level recommendations (7-day interval)

- Total 13538 jobs => 1934 jobs/day
- Total vCore-Seconds = 8582425 (2384 vCore-Hours) => 341 vCore-Hours per day
- Total #containers = 292346 => 41764 containers/day
- Total #containers for MAP = 211375 (72%)
- Total #containers for REDUCE = 80971 (28%)

Figure 8: Example improvements of applying cluster level recommendations

efficiency, contention with cloud/cluster resources, or hardware failure or inefficiency (e.g., slow node)

- Insights on cluster tuning based on aggregation of application data; e.g., determine whether a compute cluster is properly tuned at both a cluster and application level

- Insights on cluster utilization, cloud usage, and autoscaling.

We also provide users with tools to help them understand how they are using their compute resources, as for example, compare cluster activity between two time periods, aggregated cluster workload, summary reports for cluster usage, chargeback reports, and so on. A distinctive difference from other monitoring tools (e.g., Ambari, Cloudera Manager, Vertica [17]) is that we offer a single pane of glass for supporting the entire big data stack, not just individual systems, and also employ advanced analytics techniques to unravel problems and inefficiencies, whilst we also recommend concrete solutions to such issues.

One of the most difficult challenges in managing multitenant Big Data stack clusters is understanding how resources are being used by the applications running in the clusters. We are providing a forensic view into each cluster's key performance indicators (KPIs) over time and how they relate to the applications running in the cluster. For example, we can pinpoint the applications causing a sudden spike in the total cpu (e.g., vcores) or memory usage. And then, we enable drill down into these applications to understand their behavior, and whenever possible, we also provide recommendations and insights to help improve how the applications run.

In addition to that, we also provide cluster level recom-

Figure 9: Example resource utilization for two queues

mendations to fine tune cluster wide parameters to maximize a cluster's efficiency based upon the cluster's typical workload. For doing so, we work as follows:

- Collect performance data of prior completed applications
- Analyze the applications w.r.t. the cluster's current configuration
- Generate recommended cluster parameter changes
- Predict and quantify the impact that these changes will have on applications that will execute in the future

Example recommendations involve parameters such as MapSplitSizeParams, HiveExecReducersBytesParam, HiveExecParallelParam, MapReduceSlowStartParam, MapReduceMemoryParams, etc. Figure 7 shows example recommendations for tuning the size of map containers (top) and reduce containers (bottom) on a production cluster, and in particular the allocated memory in MB. In this example, at the cluster level, the default value of the memory for map tasks was set to 4096MB. Our analysis of historical data has identified alternative memory sizes for map containers. The figure shows a distribution of applications over different memory sizes shown as a histogram, along with a reward (here, predicted number of memory savings shown as a green bar) and a calculated risk (here, percentage of jobs that are predicted to run if the candidate value is applied shown as a red bar). Based on these data, we make a recommendation to set the memory size to 2048MB and calculate the improvement potential: the recommended value could halve memory usage for 97% of the expected workload. Similar recommendation are made for the reduce containers shown at the bottom of Figure 7. Figure 8 shows example improvements of applying cluster level recommendations on a production cloud deployment of a financial institution: our cluster tuning enabled ∼200% more applications (i.e., from 902 to 1934 applications/day) to be run at ∼50% lower cost (i.e., from 641 to 341 vCore-Hours/day), increasing the organization's confidence in using the cloud.

Workload Analysis: Typically, how an application performs depends on what else is also running in the big data stack, altogether forming an application workload. A workload may contain heterogeneous applications in Hive, Spark SQL, Spark ML, etc. Understanding how these applications run and affect each other is critical.

We analyze queue[1] usage on a set of clusters and identify queue usage trends, suboptimal queue designs, workloads that run suboptimally on queues, convoys, 'problem'

[1]Various systems use different terms like 'queue' or 'pool' to characterize resource budget configurations.

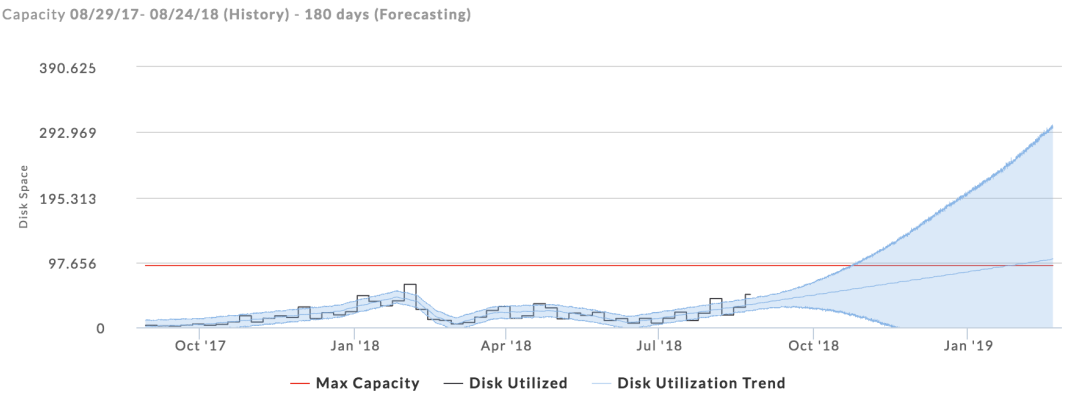

Figure 10: Disk capacity forecasting

applications (e.g., recognize and remediate excessive application wait times), 'problem' users (e.g., users who frequently run applications that reach max capacity for a long period), queue usage per application type or user or project, etc.

Figure 9 shows exemplar resource utilization charts for two queues over a time range. In this example, the workload running in the root.oper queue does not use all the resources allocated, here VCores and Memory, whilst the workload in the root.olap queue needs more resources; pending resources (in purple) go beyond the resources allocated (in black). Similar analysis can be done for other metrics like Disk, Scheduling, and so on.

Based on such findings, we generate queue level insights and recommendations including queue design/settings modifications (e.g., change resource budget for a queue or max/min limits), workload reassignment to different queues (e.g., move an application or a workload from one queue to another), queue usage forecasting, etc. Any of these recommendations could be applied to the situation shown in Figure 9. A typical big data deployment involves 100s of queues and such a task can be tedious.

We can enforce some of these recommendation using auto-actions, which enable complex actionable rules on a multiplicity of cluster metrics. Each rule consists of a logical expression and an action. A logical expression is used to aggregate cluster metrics and to evaluate the rule, and consists of two conditions:

- A prerequisite condition that causes a violation (e.g., number of applications running or memory used)
- A defining condition, who/what/when can cause a violation (e.g., user, application)

An action is a concrete, executable task such as kill an application, move an application to a different queue, send an HTTP post, notify a user, and so on.

Forecasting: Beside the current status of the big data stack systems, enterprises need to be able to provision for resources, usage, cost, job scheduling, and so on. One of the advantages of our architecture includes collecting a plethora of historical operational and application metrics. These can be used for capacity planning using predictive time-series models (e.g., [20]). Figure 10 shows an example disk capacity forecasting chart; the black line shows actual utilization and the light blue line shows a forecast within an error bound.

4. CONCLUSIONS

In this paper, we attempted to bring attention to the performance management requirements that arise in big data stacks. We provided an overview of the requirements both at the level of individual applications as well as holistic clusters and workloads. We also presented an architecture that can provide automated solutions for these requirements and discussed a few of the solutions.

The approach that we have presented here is complementary to a number of other research areas in the database and systems community such as *Polystore* [7], *HadoopDB* [3], and *hybrid flows* [18] (which have addressed challenges in stacks composed of multiple systems) as well as *self-driving databases* [2, 16] (which have addressed similar problems for traditional database systems like MySQL, PostgreSQL, and Oracle). Related complementary efforts also include the application of machine learning techniques to data management systems and cloud databases, such as (a) ML techniques for workload and resource management for cloud databases [13, 14], (b) a reinforcement learning algorithm for elastic resource management that employs adaptive state space partitionining [12], (c) a self-managing controller for tuning multi-tenant DBMS based on learning techniques for tenants' behavior, plans, and history [6], and (d) dynamic scaling algorithms for scaling clusters of virtual machines [15].

At Unravel, we are building the next generation performance management system by solving real-world challenges arising from the big data stack which is a gold mine of data for applied research, including AI and ML. We are working with many enterprises that have challenging problems; and by helping them understand and address these problems, we help them scale at the right cost.

5. REFERENCES

[1] Companies using the big data stack. https://idatalabs.com/tech/products/apache-hadoop[Online; accessed 24-August-2018].

[2] Oracle autonomous database cloud. https://www.oracle.com/database/autonomous-database.html[Online; accessed 24-August-2018].

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and DBMS

technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[4] S. Duan and S. Babu. Guided problem diagnosis through active learning. In *2008 International Conference on Autonomic Computing, ICAC 2008, June 2-6, 2008, Chicago, Illinois, USA*, pages 45–54, 2008.

[5] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *PVLDB*, 2(1):1246–1257, 2009.

[6] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan. Characterizing tenant behavior for placement and crisis mitigation in multitenant dbmss. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 517–528, 2013.

[7] V. Gadepally, P. Chen, J. Duggan, A. J. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker. The bigdawg polystore system and architecture. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, pages 1–6, 2016.

[8] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[9] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 117–134, 2016.

[10] P. Kalmegh, S. Babu, and S. Roy. Analyzing query performance and attributing blame for contentions in a cluster computing framework. *CoRR*, abs/1708.08435, 2017.

[11] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 1188–1196, 2014.

[12] K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris. Elastic management of cloud applications using adaptive reinforcement learning. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 203–212, 2017.

[13] R. Marcus and O. Papaemmanouil. Wisedb: A learning-based workload management advisor for cloud databases. *PVLDB*, 9(10):780–791, 2016.

[14] R. Marcus and O. Papaemmanouil. Releasing cloud databases for the chains of performance prediction models. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[15] J. Ortiz, B. Lee, and M. Balazinska. Perfenforce demonstration: Data analytics with performance guarantees. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2141–2144, 2016.

[16] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[17] A. Simitsis, K. Wilkinson, J. Blais, and J. Walsh. VQA: Vertica Query Analyzer. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 701–704, 2014.

[18] A. Simitsis, K. Wilkinson, U. Dayal, and M. Hsu. HFMS: managing the lifecycle and complexity of hybrid analytic data flows. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1174–1185, 2013.

[19] Z. Tan and S. Babu. Tempo: Robust and self-tuning resource management in multi-tenant parallel databases. *PVLDB*, 9(10):720–731, 2016.

[20] S. J. Taylor and B. Letham. Forecasting at Scale. https://peerj.com/preprints/3190.pdf[Online; accessed 24-August-2018].