Lube: Mitigating Bottlenecks in Wide Area Data Analytics

Hao Wang
University of Toronto

Baochun Li
University of Toronto

Abstract

Over the past decade, we have witnessed exponential growth in the density (petabyte-level) and breadth (across geo-distributed datacenters) of data distribution. It becomes increasingly challenging but imperative to minimize the response times of data analytic queries over multiple geo-distributed datacenters. However, existing scheduling-based solutions have largely been motivated by pre-established mantras (*e.g.*, bandwidth scarcity). Without data-driven insights into performance bottlenecks *at runtime*, schedulers might blindly assign tasks to workers that are suffering from unidentified bottlenecks.

In this paper, we present *Lube*, a system framework that minimizes query response times by detecting and mitigating bottlenecks at runtime. *Lube* monitors geo-distributed data analytic queries in real-time, detects potential bottlenecks, and mitigates them with a bottleneck-aware scheduling policy. Our preliminary experiments on a real-world prototype across Amazon EC2 regions have shown that *Lube* can detect bottlenecks with over 90% accuracy, and reduce the median query response time by up to 33% compared to Spark's built-in locality-based scheduler.

1 Introduction

With large volumes of data generated and stored at geographically distributed datacenters around the world, it has become increasingly common for large-scale data analytics frameworks, such as Apache Spark [30] and Hadoop [10] to span across multiple datacenters. Petabytes of data — including user activities, trending topics, service logs and performance traces — are produced on these geographically distributed datacenters every day, processed by tens of thousands data analytic queries.

Minimizing response times of geo-distributed data analytic queries is crucial, but far from trivial. Results of these analytics queries are typically used when making real-time decisions and online predictions, all of which depend upon the timeliness of data analytics. However, in contrast to data analytics in a single datacenter, the varying bandwidth on wide-area network (WAN) links and the heterogeneity of the runtime environment across geographically distributed datacenters impose new and unique challenges as query response times are minimized.

Known as *wide-area data analytics* in the literature, tasks (or data) are optimally placed across datacenters in order to improve data locality [15, 16, 20, 26, 27]. However, all previous works made the simplifying assumption that the runtime environment of wide-area data analytics is temporally stable, and that there are no runtime performance variations in these clusters. Naturally, this may not accurately reflect the reality. In addition, existing works have largely been motivated by a few widely accepted mantras, such as the scarcity of network bandwidth on access links from a datacenter to the Internet. With an extensive measurement study on analytic jobs, Ousterhout *et al.* [18] have convincingly pointed out that some of the widely held assumptions in the literature may not be valid in the context of a single cluster.

Delving into the fluctuating runtime environment of wide-area data analytics, this paper makes a strong case for analyzing and detecting performance bottlenecks in data analytics frameworks *at runtime*. Shifting gears from a single cluster to the context of wide-area data analytics, we believe that the conclusion from [18] still holds: it may not always be the same resource — such as bandwidth — that causes runtime performance bottlenecks in wide-area data analytic queries. To generalize a step further, the types of resource that cause performance bottlenecks may even vary over time at runtime, as analytic queries are executed across datacenters. It becomes intuitive that, if we wish to reduce the query response times in wide-area data analytics, these performance bottlenecks need to be detected *at runtime*, and

a new resource scheduling mechanism needs to be designed to mitigate them. Unfortunately, such a high-level intuition has not yet been well explored in the literature and remains a largely uncharted territory.

In this paper, we propose *Lube*, a new system that is designed to perform data-driven runtime performance analysis for minimizing query response times. *Lube* features a closed-loop design: the results of runtime monitoring are used for detecting bottlenecks, and these bottlenecks serve as input to the resource scheduling policy to mitigate them, again at runtime. Our original contributions in this paper are the following:

First, we propose effective and efficient techniques to detect resource bottlenecks at runtime. We investigate two bottleneck detection techniques, both driven by performance metrics collected in real-time. We start with a simple statistical technique, Autoregressive Integrated Moving Average (ARIMA) [6], and then propose machine learning techniques to further explore the implicit correlation between multiple performance metrics.[1] As one of the effective algorithms and a case study, we use the Sliding Hidden Markov Model (SlidHMM) [7], an unsupervised algorithm that takes time series as input and incrementally updates model parameters for detecting upcoming states.

Second, we propose a new scheduling policy that, when assigning tasks to worker nodes, mitigates bottlenecks by considering not only data locality (*e.g.*, [26]), but also the severity of bottlenecks. The upshot of our new scheduling policy is the use of a technique similar to late binding in Sparrow [19], that holds a task for a short while before binding it to a worker node. This is designed to avoid the negative implications of false positives when detecting bottlenecks.

We have implemented a prototype of *Lube* on a Spark SQL cluster over Amazon EC2 with 37 instances across nine regions. Our experiments of the Big Data Benchmark [23] with a 1.1 TB dataset show that *Lube* is able to detect bottlenecks with an accuracy over 90% and reduces the median query response time by as much as 33% (1.5× faster).

2 *Lube*: a Bird's-Eye View

Data analytics over geo-distributed datacenters may suffer from a highly volatile runtime environment, due to the lack of load distribution when using resources, or varying bandwidth availability over wide-area network links [1]. As a result, resource bottlenecks are more likely to occur at runtime, when data analytic queries are executed over the wide area.

As a motivating example of such runtime bottlenecks, Figure 1 presents a heat map of real-time memory utilization on the Java Virtual Machine (JVM) heap, captured on a 5-node Spark SQL [3] cluster running the Big Data Benchmark [23]. As we can observe, within a specific time window (marked by $t_{current}$), memory is heavily utilized on node_1, while other nodes are largely idle on their memory utilization. This implies that memory becomes a bottleneck on node_1, because the Spark SQL scheduler assigned more tasks to this node with no knowledge that its memory may be overloaded at runtime.

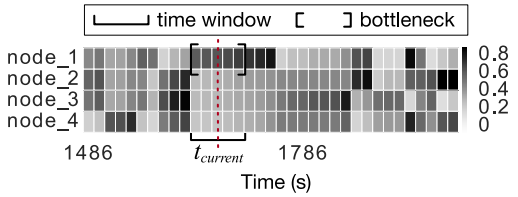


Figure 1: A potential bottleneck in memory.

Given the existence of resource bottlenecks, our ultimate objective is to reduce query response times by designing new task scheduling strategies that work around these bottlenecks. To achieve such an objective, we need to monitor performance attributes of data analytic queries at runtime and detect potential bottlenecks with very little overhead. To be more specific, we will need to design and implement the following components:

Lightweight performance monitors. A collection of performance monitors on each worker node is needed to capture process-level performance metrics in real-time. In *Lube*, rather than intrusively using code instrumentation, we choose to reuse existing lightweight system-level performance monitors on Linux (*e.g.*, `jvmtop`, `iotop`, `iperf` and `nethogs`).

Online bottleneck detection. With performance metrics collected in real-time, we will propose algorithms that analyze dependencies between performance metrics and detect potential bottlenecks at runtime.

Bottleneck-aware scheduling. To react to detected bottlenecks, a bottleneck-aware scheduler will make task assignment decisions by considering both bottleneck sever-

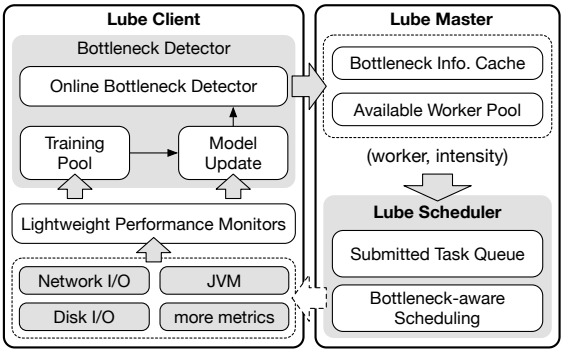


Figure 2: *Lube*: a closed-loop architecture involving performance monitors, bottleneck detection, and task scheduling.

ities and data locality. Besides, the scheduler should be able to tolerate inaccurate detections.

Figure 2 presents the closed-loop design architecture of *Lube*. On each worker node, a *Lube* client periodically collects runtime performance metrics, updates the machine learning model and reports detected bottlenecks to the *Lube* master; on the master node, the task scheduler makes task assignment decisions based on bottleneck intensities at the worker nodes, as well as data locality preferences of tasks. In return, the decisions made by the task scheduler will further influence the performance of data analytic queries at each worker node.

3 Detecting Bottlenecks

Performance bottlenecks may emerge anytime and anywhere in wide-area data analytics. To mitigate performance bottlenecks in time, we will first need to detect them correctly at runtime. *Lube* performs online bottleneck detection on performance metrics collected in real-time.

We investigate two techniques to detect bottlenecks from the time series of performance metrics. One is a simple statistical model — the Autoregressive Integrated Moving Average (ARIMA) algorithm that approximates the future value by a linear function of past values and past errors; the other is an unsupervised machine learning model: the Sliding Hidden Markov Model (SlidHMM) algorithm that can autonomously learn the implicit correlation between multiple performance metrics.

3.1 ARIMA

Introduced by Box and Jenkins [6], the ARIMA model has been widely applied in time series analysis. As a combination of autoregressive (AR) model and the moving average (MA) model, the ARIMA model is defined by the following equation:

$$y_t = \theta_0 + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \varepsilon_t - \theta_1 \varepsilon_{t-1} - \theta_2 \varepsilon_{t-2} - \cdots - \theta_q \varepsilon_{t-q}, \quad (1)$$

where y_t and ε_t denote the actual value and random error at time t respectively; ϕ_i $(i = 1,2,\ldots,p)$ and θ_j $(j = 1,2,\ldots,q)$ are the coefficients specified by the model. p and q are integers indicating the autoregressive (AR) and moving average (MA) polynomials respectively. A general ARIMA model is represented as ARIMA(p, d, q), in which d is the degree of difference transformation for data stationarity.

We build an univariate ARIMA model for each performance metric, rather than a vector ARIMA model for all metrics, as it usually becomes "overfitting" due to too many combinations of insignificant parameters [9]. To support online bottleneck detection, we periodically update the ARIMA model with continuously arriving performance metrics.

3.2 Sliding HMM

The Hidden Markov Model (HMM) [4] infers a sequence of hidden states that maps to the sequence of observation states. Through feeding a time series of observed performance metrics (O_1 to O_d) to HMM, we can infer the possible performance metrics O_k in the future (Figure 3). The HMM is usually defined as a three-tuple: (A, B, π) as the following notations (t is the time stamp):

$Q = \{q_1, q_2 \ldots, q_N\}$, hidden state sequence.

$O = \{O_1, O_2 \ldots, O_k\}$, observation state sequence.

$A = \{a_{ij}\}$, $a_{ij} = \Pr(q_j \text{ at } t+1 | q_i \text{ at } t)$, transition matrix.

$B = \{b_j(k)\}$, $b_j(k) = \Pr(O_k \text{ at } t | q_j \text{ at } t)$, emission matrix.

$\pi = \{\pi_i\}$, $\pi_i = \Pr(q_i \text{ at } t = 1)$, initial state distribution.

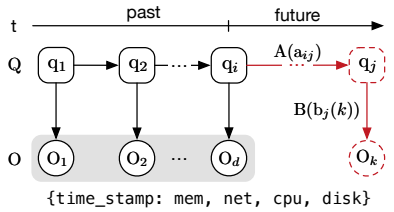


Figure 3: The Hidden Markov Model.

HMM learns the hidden states based on an expectation maximization algorithm, the Baum-Welch Algorithm [5]. This algorithm iteratively searches the model parameters (A, B, π) that maximizes the likelihood of $\Pr(O|\mu)$ — the best explanation of the observation sequence. Traces of JVM heap utilization in Figure 4 presents a clear periodical pattern. By learning the hidden states behind this pattern, the HMM infers the future performance metrics for bottleneck-aware scheduling.

To support bottleneck detection in runtime, HMM must be updated online. However, such online updates incur a heavy cost in both time and space, as the Baum-Welch algorithm needs to re-calculate both old and new time series input. Hence, we propose to use the Sliding Hidden Markov Model (SlidHMM) [7], which is a sliding version of the classic HMM, and is particularly designed for online characterization of high-density time

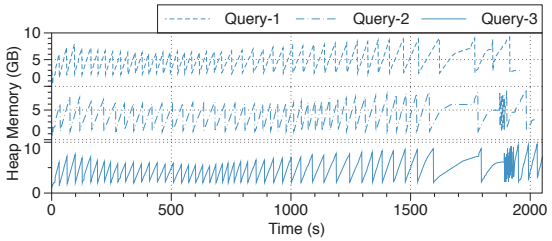


Figure 4: The JVM heap utilization traces of a Spark executor process.

series. The core of SlidHMM is a sliding window that accepts new observations and evicts outdated ones. A moving average approximation replaces the outdated observations during SlidHMM's training phase. Different from the traditional HMM, SlidHMM updates incrementally with the partial calculation on a fixed window of observations. Thus, it improves the efficiency of bottleneck detection in both time and space.

4 Bottleneck-Aware Scheduling

To justify the imperatives of bottleneck-aware scheduling, we visualized the performance metrics collected from a Spark SQL cluster running real-world workloads in a geographically distributed fashion on Amazon EC2. Figure 5 reveals the necessity and feasibility of performing bottleneck-aware scheduling in wide-area data analytics: **a single worker node is bottlenecked continuously while all nodes are rarely bottlenecked in chorus**. A bottlenecked node slows down the running tasks, and if we keep assigning tasks to the bottlenecked node, performance will be further degraded. Meanwhile, there usually exist available nodes to take over the tasks assigned to bottlenecked nodes.

Unfortunately, neither existing resource management platforms (*e.g.*, Mesos [13] and YARN [2]) nor scheduling solutions (*e.g.*, Iridium [20] and Sparrow [19]) support online detection of such performance bottlenecks. The built-in schedulers of Spark and Hadoop make decisions only based on data locality, with the objective of reducing network transmission times [29].

Bottlenecked nodes consume extra time to process tasks. To minimize the response times of data analytic queries, we propose a simple task scheduler to coordinate with our bottleneck detection algorithms and mitigate bottlenecks at runtime. A node will be marked as available if no upcoming bottlenecks have been detected; a task has several levels of locality preferences in descending order. When assigning a task, this scheduler jointly considers data locality and bottleneck severity. Essentially, it searches for an available node that satisfies the highest locality preference level of the task compared to all available nodes. Considering that bottlenecks may not be correctly detected, we introduce a *late-binding* algorithm to our bottleneck-aware scheduler. Sparrow [19] applies this algorithm to work around incorrect samplings. The intuition of *late-binding* is that the worker nodes first verify the correctness of bottleneck detection, and then launch the assigned tasks. If such verification fails, the task will be reassigned.

5 Implementation and Evaluation

We implement components of *Lube* in a decoupled fashion. The pluggable performance monitors, the standalone bottleneck detection module and the scheduler exchange messages via in-memory `redis` [21] servers. The messages are negligible small key-value pairs for network transmission. We modified the built-in scheduler of Spark to enable bottleneck-aware scheduling.

We conducted a preliminary evaluation of our design in realistic wide-area settings. The deployment includes 37 EC2 `m4.2xlarge` instances across 9 regions. Each instance has an 8-core CPU, 32 GB of memory, 1000 Mbps network[2] and a 100 GB SSD disk. All instances run on Ubuntu-14.04 installed with Oracle Java-1.8.0, Spark-1.6.1, HDFS-2.6.4 and Hive-1.2.1. The Big Data Benchmark [23] includes four queries: Query 1-3 each has three scale levels *a*, *b*, *c*, from small to large; Query 4 is a user-defined-function (UDF) running a python script to count URLs. We run this benchmark with a 1.1 TB dataset.

The results in Figure 6 show that with an accuracy of over 90% in bottleneck detection, *Lube* speeds up median query response times from 26.88% (1.37×) to 33.46% (1.5×). Figure 6(a) presents the accuracies of bottleneck detection under different settings. The *hit rate* is defined as the proportion of detected bottlenecks that are observed by monitors among all detected bottlenecks. We collect the time stamps and the bottleneck sequences during the running of the Big Data Benchmark for 15 times respectively. By comparing the time sequences of detected bottlenecks and observed bottlenecks, we calculate the *hit rate* offline. In our experiments, the average *hit rate* of the SlidHMM is 92.1%, while it's 83.57% for ARIMA. The *hit rate* of ARIMA tends to decrease with the increase of query scale. As a linear combination of autoregression and moving average, ARIMA ignores

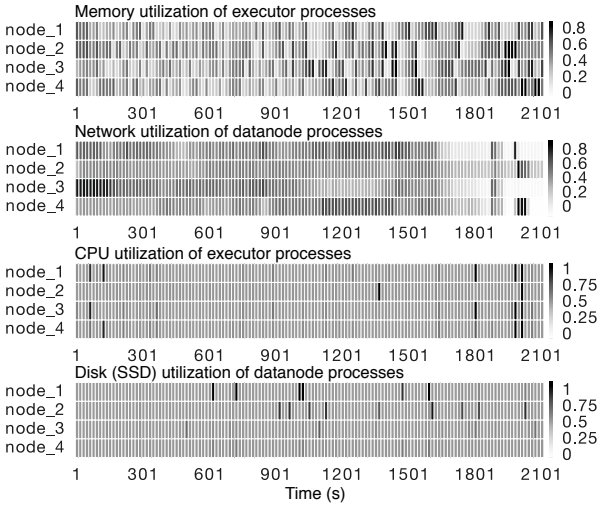
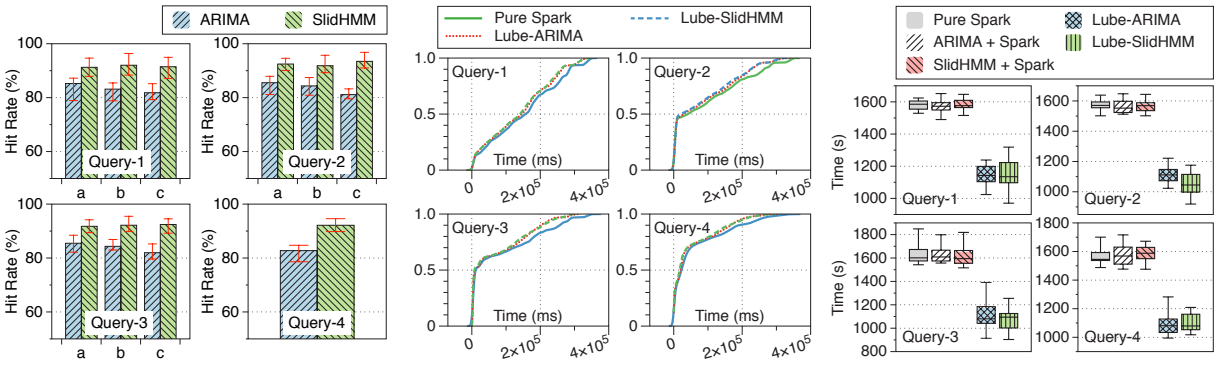


Figure 5: Heat maps of performance metrics.



(a) The accuracy of bottleneck detection. (b) CDF of task completion times. (c) Query response times.

Figure 6: Evaluation on a 37-node EC2 cluster running the Big Data Benchmark.

nonlinear patterns in the performance metric sequences, which may lower the accuracy of bottleneck detection.

We show that *Lube* achieves a faster query response and maintains a low overhead from the task level to query level. At the task level, Figure 6(b) plots the task completion times CDF of pure Spark, *Lube*-ARIMA and *Lube*-SlidHMM. For Query 1, the average (75th percentile) task completion time of pure Spark is 150.928 seconds (246.19 seconds). *Lube*-ARIMA saves 12.454 seconds (22.075 seconds) for average seconds (75th percentile) tasks compared to pure Spark, while *Lube*-SlidHMM saves 14.783 seconds (27.469 seconds) for average (75th percentile) tasks. Our bottleneck-aware scheduler brings a substantial improvement to the completion times of long tasks.

At the query level, we measure query response times under different control groups. Pure Spark is the baseline; *Lube*-ARIMA and *Lube*-SlidHMM show the reduction of query response times; and, the Spark default scheduler with *Lube*-ARIMA (ARIMA + Spark) and *Lube*-SlidHMM (SlidHMM + Spark) are the control group to evaluate *Lube*'s overhead. Figure 6(c) shows that running *Lube*-ARIMA or *Lube*-SlidHMM with the Spark default scheduler does not introduce much overhead since the query response times under these three settings are similar. In addition, for median query response times, *Lube* reduces 26.88% to 33.46% (1.37× to 1.5× faster) of time with the ARIMA algorithm, while reduces 28.41% to 33.18% (1.4× to 1.5× faster) of time with the SlidHMM algorithm. From these results, we can conclude that *Lube* reduces the overall query response times.

6 Related Work

There exists large volumes of existing research on optimizing the performance of wide-area data analytics. Clarinet [25] pushes wide-area network awareness to the query planner, and selects a query execution plan before the query begins. Graphene [12] presents a Directed Acyclic Graph (DAG) scheduler with awareness of DAG dependencies and task complexity. Iridium [20] optimizes data and task placement to reduce query response times and WAN usage. Geode [26] minimizes WAN usage via data placement and query plan selection. SWAG [14] adjusts the order of jobs across datacenters to reduce job completion times. These works develop their solutions based on a few widely-accepted mantras, which are shown to be skeptical in a systematic analysis on the performance of data analytics frameworks [18]. The *blocked time analysis* proposed in [18] calls for more attention to temporal performance variations.

However, there is still very little existing effort on optimizing the performance of data analytics with the awareness of variations in the runtime environment. Hadoop speculative task execution [28] duplicates tasks that are slow or failed, but not knowing the exact bottlenecks may lead to worse performance. As far as we know, *Lube* is the first work that leverages machine learning techniques to detect runtime bottlenecks and schedules tasks with awareness of performance bottlenecks.

Machine learning techniques have been actively applied to predict and classify data analytics workloads. NearestFit [8] establishes accurate progress predictions of MapReduce jobs by a combination of nearest neighbour regression and statistical curve fitting techniques. Ernest [24] applies a linear regression model to predict the performance of large-scale analytics.

7 Conclusion

In this paper, we have presented *Lube*, a closed-loop framework that mitigates bottlenecks at runtime to improve the performance of wide-area data analytics. *Lube* monitors runtime query performance, detects bottlenecks online and mitigates them with a bottleneck-aware scheduling policy. Experiments across nine EC2 regions show that *Lube* achieves over 90% bottleneck detection accuracy and, compared to the default Spark scheduler, reduces the median query response time by up to 33%.

8 Discussions and Future Work

Preliminary experiments highlight the performance of *Lube* in reducing query response times achieved through detecting bottlenecks, mitigating bottlenecks at runtime. While this motivates the research on performing data-driven runtime performance analysis to optimize data analytics frameworks, there are a few aspects to discuss.

Selection of runtime metrics. It is the selected runtime metrics that determine the efficacy of the runtime performance analysis. There are enormous runtime metrics from multiple hierarchies of wide-area data analytics frameworks. To efficiently detect and mitigate bottlenecks in low-level resources (*e.g.*, CPU, memory, disk I/O and network I/O *etc.*), we have studied several performance monitors and various combinations of performance metrics. However, the space of selecting appropriate metrics has still not been fully explored. We will put more efforts in the assessment of runtime metrics selection.

Bottleneck detection models. *Lube* achieves a substantial improvement by applying two simple models, ARIMA and SlidHMM. The emerging data-driven techniques broaden the horizon of data analytics optimization methodologies. We would like to further explore the latest data-driven techniques, such as Generative Adversary Network (GAN) [11] and Reinforcement Learning [22]. For example, DeepRM [17] builds a deep reinforcement learning model for strategies of cluster resource management. However, the surprising accuracy of machine learning models makes us wonder the practical boundary of their effectiveness, which is imperative for robust and reproducible solutions.

WAN conditions. Most recent work mainly considers the heterogeneity and the variance of wide-area network bandwidths [1, 12, 14, 20, 24, 26]. A few approaches have been applied to measure network conditions in these work. *Lube* captures the local network throughput by measuring network I/O on each node, which though only reveals a coarse-grained awareness of network; and, measures the pair-wise WAN bandwidths by a `cron` job running `iperf` on each node. We plan to exploit the capabilities of Software-Defined Network (SDN) to complement the global wide-area network conditions at runtime.

9 Acknowledgement

This work is supported by a research contract with Huawei Technologies Co. Ltd. and an NSERC Collaborative Research and Development (CRD) grant. We would like to thank the HotCloud anonymous reviewers for their valuable comments.

References

[1] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI* (2017).

[2] APACHE. Apache Hadoop Official Website. `http://hadoop.apache.org/`. [Online; accessed 1-May-2016].

[3] APACHE. Spark SQL. `https://spark.apache.org/sql/`. [Online; accessed 1-July-2016].

[4] BAUM, L. E., AND PETRIE, T. Statistical inference for probabilistic functions of finite state Markov chains. *The annals of mathematical statistics 37*, 6 (1966), 1554–1563.

[5] BAUM, L. E., PETRIE, T., SOULES, G., AND WEISS, N. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The Annals of Mathematical Statistics 41*, 1 (1970), 164–171.

[6] BOX, G. E., JENKINS, G. M., REINSEL, G. C., AND LJUNG, G. M. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[7] CHIS, T. Sliding Hidden Markov Model for Evaluating Discrete Data. In *10th Proceedings of Computer Performance Engineering European Workshop (EPEW)* (2013), vol. 8168.

[8] COPPA, E., AND FINOCCHI, I. On data skewness, stragglers, and MapReduce progress indicators. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), ACM.

[9] DE GOOIJER, J. G., AND HYNDMAN, R. J. 25 years of time series forecasting. *International Journal of Forecasting 22*, 3 (2006), 443–473.

[10] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. USENIX Symposium on Operating System Design and Implementation (OSDI)* (2004).

[11] GOODFELLOW, I., POUGET-ABADIE, J., MIRZA, M., XU, B., WARDE-FARLEY, D., OZAIR, S., COURVILLE, A., AND BENGIO, Y. Generative adversarial nets. In *Advances in neural information processing systems (NIPS)* (2014).

[12] GRANDL, R., KANDULA, S., RAO, S., AKELLA, A., AND KULKARNI, J. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).

[13] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R. H., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2011).

[14] HUNG, C., GOLUBCHIK, L., AND YU, M. Scheduling Jobs Across Geo-Distributed Datacenters. In *Proc. ACM Symposium on Cloud Computing (SoCC)* (2015).

[15] JALAPARTI, V., BODIK, P., MENACHE, I., RAO, S., MAKARYCHEV, K., AND CAESAR, M. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *ACM SIGCOMM Computer Communication Review* (2015), pp. 407–420.

[16] KLOUDAS, K., MAMEDE, M., PREGUIÇA, N., AND RODRIGUES, R. Pixida: Optimizing Data Parallel Jobs in Wide-Area Data Analytics. *VLDB Endowment 9*, 2 (2015), 72–83.

[17] MAO, H., ALIZADEH, M., MENACHE, I., AND KANDULA, S. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (2016), ACM.

[18] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making sense of performance in data analytics frameworks. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[19] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (2013).

[20] PU, Q., ANANTHANARAYANAN, G., BODIK, P., KANDULA, S., AKELLA, A., BAHL, P., AND STOICA, I. Low Latency Geo-Distributed Data Analytics. In *Proc. ACM SIGCOMM* (2015).

[21] REDIS. Redis Website. `http://redis.io/`. [Online; accessed 1-May-2016].

[22] SUTTON, R. S., AND BARTO, A. G. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[23] UC BERKELEY AMPLAB. The Big Data Benchmark. `https://amplab.cs.berkeley.edu/benchmark/`. [Online; accessed 1-July-2016].

[24] VENKATARAMAN, S., YANG, Z., FRANKLIN, M., RECHT, B., AND STOICA, I. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), USENIX Association.

[25] VISWANATHAN, R., ANANTHANARAYANAN, G., AND AKELLA, A. Clarinet: WAN-aAware optimization for analytics queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), USENIX Association.

[26] VULIMIRI, A., CURINO, C., GODFREY, P., JUNGBLUT, T., PADHYE, J., AND VARGHESE, G. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[27] VULIMIRI, A., CURINO, C., GODFREY, P., KARANASOS, K., AND VARGHESE, G. WANalytics: Analytics for A Geo-Distributed Data-Intensive World. In *Proc. Conference on Innovative Data Systems Research (CIDR)* (2015).

[28] WHITE, T. *Hadoop: The Definitive Guide*. " O'Reilly Media, Inc.", 2012.

[29] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proc. ACM European Conference on Computer Systems* (2010), pp. 265–278.

[30] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012).

Notes

[1]For example, a higher network I/O will lead to higher JVM heap swap frequencies, since network send/receive semantics will trigger memory load/dump operations.

[2]EC2 only guarantees intra-region bandwidth. The inter-region traffic runs on public links that are highly fluctuating and intensely competitive.