Mitigating Bottlenecks in Wide Area Data Analytics via Machine Learning

Hao Wang and Baochun Li, *Fellow, IEEE*

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 percent accuracy, and reduce the median query response time by up to 33 percent compared to Spark's built-in locality-based scheduler.

Index Terms—Wide area, data analytics, performance prediction, machine learning, bottleneck detection, task scheduling

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 [47] and Hadoop [15] 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 [26], [27], [36], [42], [43]. 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. [33] 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 [33] 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. Lubefeatures 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:

• *The authors are with the Department of Electrical and Computer Engineering, University of Toronto ON M5S, Canada. E-mail: {haowang, bli}@ece.utoronto.edu.*

Manuscript received 15 Oct. 2017; revised 25 Feb. 2018; accepted 14 Mar. 2018. Date of publication 19 Mar. 2018; date of current version 5 Mar. 2020. (Corresponding author: Hao Wang.)
Recommended for acceptance by W. Zhang.
Digital Object Identifier no. 10.1109/TNSE.2018.2816951

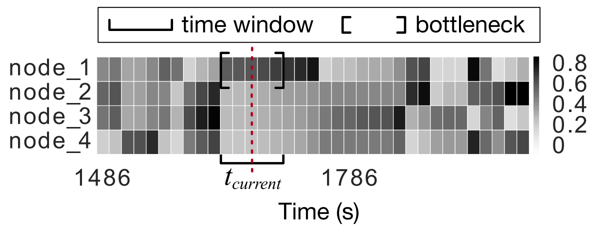


Fig. 1. A potential bottleneck in memory.

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) [10], 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) [12], 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., [42]), 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 [34], 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 Lubeon a Spark SQL cluster over Amazon EC2 with 37 instances across nine regions. Our experiments of the Big Data Benchmark [39] with a 1.1 TB dataset show that Lubeis able to detect bottlenecks with an accuracy over 90 percent and reduces the median query response time by as much as 33 percent ($1.5\times$ 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 [4]. 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, Fig. 1 presents a heat map of real-time memory utilization on the Java Virtual Machine (JVM) heap, captured on a 5-node Spark SQL [7] cluster running the Big Data Benchmark [39]. 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.

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.

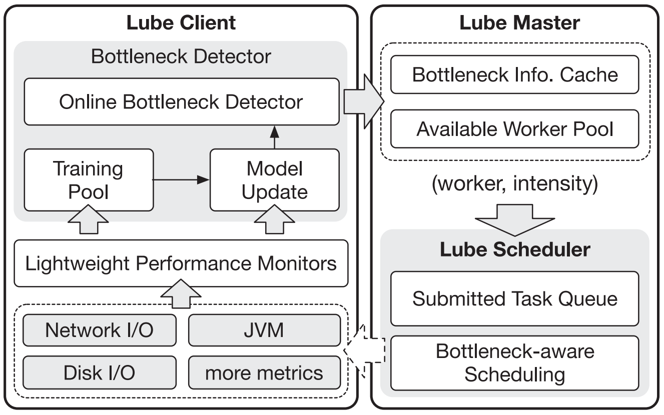


Fig. 2. *Lube*: A closed-loop architecture involving performance monitors, bottleneck detection, and task scheduling.

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

Fig. 2 presents the closed-loop design architecture of Lube. On each worker node, a Lubeclient periodically collects runtime performance metrics, updates the machine learning model and reports detected bottlenecks to the Lubemaster; 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. Lubeperforms 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

TABLE 1
The Metrics Used to Detect the Potential Performance Bottlenecks, Which Are Collected by Performance Monitors of Linux and Data Analytic Frameworks

Metrics	Description
`disk_io`	Disk read/write throughput
`disk_iops`	Disk I/O operations per second
`net_io`	Network throughput
`cpu_util`	CPU utilization per core
`mem_util`	Memory utilization
`jvm_gc_util`	JVM garbage collection utilization
`jvm_heap_util`	JVM heap utilization
`pending_task_num`	The number of waiting tasks
`completed_task_num`	The number of completed tasks
`failed_task_num`	The number of failed tasks
`task_retry_num`	The retry times of failed tasks
`pending_stage_num`	The number of waiting stages
`completed_stage_num`	The number of completed stages

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 Dataset Collection

We collect runtime performance metrics from both system-level monitors and monitoring interfaces of data analytic frameworks. Table 1 summarizes all features we have considered in the bottleneck detection. It should be noted that all features are obtained in a non-intrusive way by reusing exiting toolkits and interfaces. Besides, it is easy to incrementally expand the training dataset by including performance metrics collected during the execution of queries in reality. The bottleneck detection models can be improved with the updated dataset. All metrics are recorded as a 2-tuple including the timestamp and the value. A master node aggregates metrics from each worker node and updates the dataset on each worker node periodically.

In addition to the performance metrics, we also include the progress metrics collected from the data analytic frameworks. The progress metrics indicates the runtime execution status of workers. For example, a high `task_retry_num` on a worker implies that the worker is in severe bottleneck. Zero `pending_task_num` implies the worker is available or becoming available.

We perform data preprocessing on the raw data of performance metrics, because the range of values of features varies widely. For performance metrics such as `disk_io` and `net_io`, they are normalized by dividing the theoretical maximum performances defined by the hardwares. For utilization metrics such as `cpu_util` and `mem_util`, they are inherently normalized values between 0 and 1. Fig. 3 shows heap maps of the normalized metrics.

3.2 ARIMA

Introduced by Box and Jenkins [10], the ARIMA model has been widely applied in time series analysis. As a combination of autoregressive (AR) model and the moving average

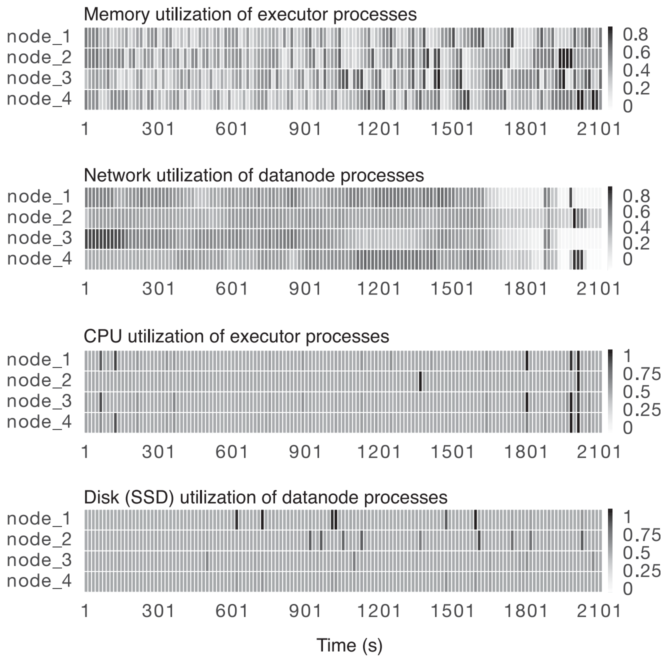


Fig. 3. Heat maps of performance metrics.

(MA) model, the ARIMA model is defined by the following equation:

$$\begin{aligned} y_t = \theta_0 + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} \\ + \epsilon_t - \theta_1 \epsilon_{t-1} - \theta_2 \epsilon_{t-2} - \cdots - \theta_q \epsilon_{t-q}, \end{aligned} \tag{1}$$

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. For example, if $q = 0$, then Eq. (1) reduces to an AR model of order p; If $p = 0$, then Eq. (1) reduces to a MA model of order q. A general ARIMA model is represented as ARIMA(p, d, q), in which d is the degree of difference transformation for data stationarity.

The ARIMA methodology basically has three iterative steps. First, we test the stationarity of the input time series by the Dickey-Fuller Test [17]. If the time series is not stationary, we transform it into a stationary time series by applying a suitable degree (defined by d) of differencing. Second, we examine the autocorrelation function (ACF) and the partial autocorrelation function (PACF) of the input time series to estimate appropriate p and q [10]. Third, we check the residuals (actual values minus fitted values) to diagnose whether the model is adequate. If the model is not adequate, then we repeat the previous steps.

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 [14]. To support online bottleneck detection, we periodically update the ARIMA model with continuously arriving performance metrics.

3.3 Sliding HMM

The Hidden Markov Model (HMM) [8] infers a sequence of hidden states that maps to the sequence of observation

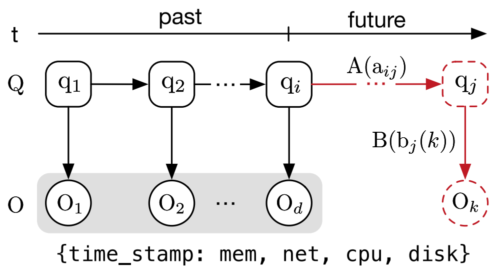


Fig. 4. The Hidden Markov Model.

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 (Fig. 4). 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.

HMM learns the hidden states based on an expectation maximization algorithm, the Baum-Welch Algorithm [9]. 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 Fig. 5 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) [12], which is a sliding version of the classic HMM, and is particularly designed for online characterization of high-density time 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.

3.4 Accuracy-Overhead Trade-Off

A natural conflict lies between the accuracy of bottleneck detection and the overheads of algorithms. More training iterations lead to more accurate detection, but introduces more overheads and latency. Designed to coexist with costly data analytics engines, bottleneck detection should work under the radar.

Considering the ultimate goal of bottleneck detection is to enable bottleneck-aware scheduling, it is fairly unnecessary to enforce an accuracy of one decimal place. A number of bottleneck severity levels are enough for bottleneck-aware scheduling. To this end, we shape the input

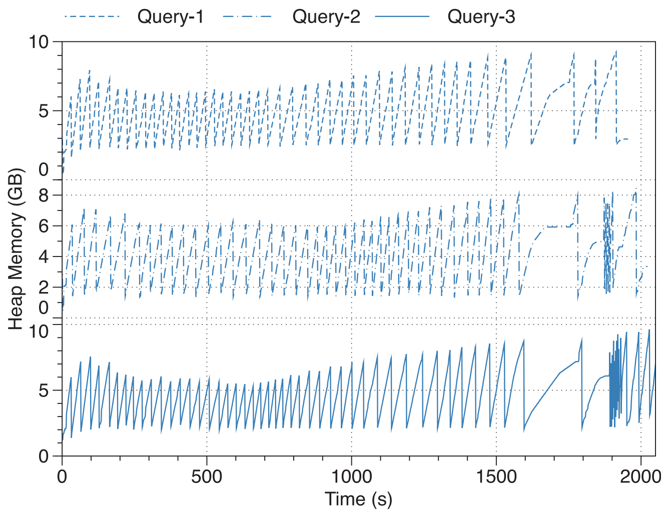


Fig. 5. The JVM heap utilization traces of a Spark executor process.

performance metrics from continuous values to discrete ones (e.g., integers from 1 to 100) and classify the output into several bottleneck severity levels. It results in a coarse inference of bottleneck severity, though, which is adequate for our error-tolerant scheduling strategy.

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. Fig. 3 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 [23] and YARN [6]) nor scheduling solutions (e.g., Iridium [36] and Sparrow [34]) 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 [46].

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 [34] 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.

Our bottleneck-aware scheduling algorithm has a prospective bird-view over bottleneck severities of all nodes, thanks to the online bottleneck detection. Basically, the bottleneck-aware scheduling algorithm is seeking a worker that not only best matches the locality preferred by the task, but also keeps bottleneck-free in future period of time. Pseudocode for this algorithm is shown in Algorithm 1. Note that a task usually has a list of preferred localities sorted by preference levels in descending order. However, if no bottleneck-free worker meets the task's locality preference, the task will be assigned to a random picked worker that is bottleneck-free. In case of potential false positives in bottleneck detection, our scheduler will call an error-tolerance algorithm first before the task is sent to the matched worker.

Algorithm 1. Bottleneck-Aware Scheduling Algorithm

```
    Input: taskList,                              ▷ pending task list
           workerList,                            ▷ running worker list
           btlOfWorker,                   ▷ bottleneck severity hashmap
    Output: taskToWorker                       ▷ assignment decisions
 1: for task in taskList do
 2:   for locality in task.localityList do
 3:                          ▷ search a worker that matches the locality
 4:     worker = search(workerList, locality);
 5:     btlLevel = btlOfWorker[worker];
 6:     if btlLevel is low then                        ▷ bottleneck-free
 7:                           ▷ for error-tolerance, call Algorithm 2
 8:       binding = lateBind(task, worker);
 9:       add binding to taskToWorker;
10:       break;                                   ▷ go to the next task
11:     else                                            ▷ in bottleneck
12:       continue;                                 ▷ try next locality
13:   if no matched worker then
14:                                       ▷ randomly pick a free worker
15:     randFreeWorker = randFree(btlOfWorker);
16:                           ▷ for error-tolerance, call Algorithm 2
17:     binding = lateBind(task, randFreeWorker);
18:     add binding to taskToWorker;
```

4.1 Error Tolerance

Our bottleneck detection algorithms are not always accurate. It is impossible to have one hundred percent accuracy. A false positive detection further worsens the bottleneck severity, since a node detected as bottleneck-free will be assigned with more tasks. Similar to Sparrow [34], we introduce a *late-binding* algorithm to overcome detection inaccuracies in bottleneck-aware scheduling.

The intuition of *late-binding* is that the workers do not immediately launch the assigned tasks and instead place reservations for the requested resources (i.e., CPU cores and memory). Meanwhile, the scheduler probes performance metric observations and justifies whether the detected bottlenecks are false positive. The scheduler will adopt a back-off strategy when false positives happen. For example, if the observed performance metrics increase rapidly, the detected bottleneck-free will be identified as false positive. The scheduler then releases the resource reservation on the nodes and puts back the assigned tasks to the list of pending tasks. Pseudocode for this algorithm is shown in Algorithm 2. For false negative cases that mistakenly mark a worker node as non-available, the worker will still receive tasks if there are no bottleneck-free worker nodes because the Lube scheduler is work-conserving. In other words, such false negative worker nodes will not be under-utilized, as if there were pending tasks in the queue.

Algorithm 2. Late-Binding Algorithm

```
    Input: task,                                 ▷ a task to be binded
           taskList,                             ▷ pending task list
           worker,                       ▷ detected as bottleneck-free
           obsOfWorker           ▷ observed performance hashmap
    Output: binding                                   ▷ task-to-worker
 1: reserve worker.cpu and worker.mem;
 2: while tick in timePeriod do            ▷ probe for a time period
 3:   obsVal = obsOfWorker[worker];                       ▷ observation
 4:   add obsVal to trend;
 5: if trend is ascending then                         ▷ false positive
 6:   add task to taskList;                      ▷ back to pending list
 7:   release worker.cpu and worker.mem;
 8:   return empty binding;
 9: else
10:   return(task, worker);                            ▷ late-binding
```

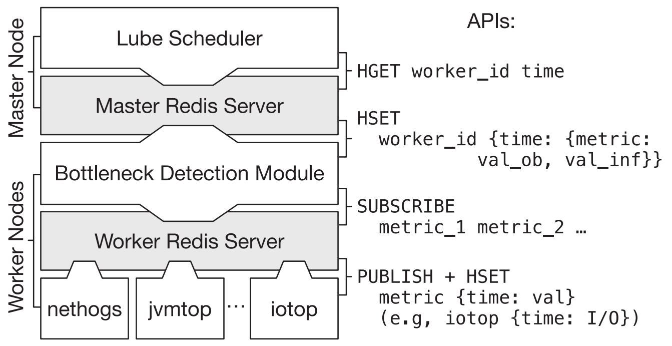
5 Implementation

In order to implement a practical closed-loop framework that enables online bottleneck-aware scheduling, we specifically set our implementation goals as follows:

- *Low-latency:* To detect bottlenecks accurately and make scheduling decisions effectively, the closed-loop of Lubemust operate at fine time granularity. Therefore, modules and pipelines integrated in this loop must act in low latency.
- *Extensible:* Data analytics tasks consume various resources. The framework should provide simple interfaces to plug in customized performance monitors, in case of missing potential bottlenecks in different resources.
- *Compatible:* The framework should be able to open the bottleneck detection service to prevailing data analytics engines. It must provide compatible loose-coupled APIs.
- *Low-overhead:* All components in Lubeframework must perform under the radar. They must be lightweight enough, introducing negligible impact on running analytics tasks.

To achieve these goals, we implement the software stack and open APIs of Lubeas Fig. 6. In essence, components are Python lightweight daemon processes that interact with each other asynchronously. `Redis` [37], an in-memory cache, works as the underlying low-latency and reliable messaging channels. Messages exchanged between different components are tagged with an epoch timestamp. To guarantee consistent timing on the whole cluster, a NTP [3] clock synchronization service is deployed on each node. Before cached in `redis`, messages are all serialized in a lightweight data-interchange format, `JSON` [2].

On each worker node, performance monitors and the bottleneck detection module communicate in a publish/



```
HGET worker_id time
HSET
  worker_id {time: {metric:
              val_ob, val_inf}}
SUBSCRIBE
  metric_1 metric_2 …
PUBLISH + HSET
  metric {time: val}
  (e.g, iotop {time: I/O})
```

Fig. 6. The software stack and APIs of Lube.

subscribe messaging paradigm. The monitors are continuously publishing performance metric streams to local redis server. As a subscriber, the local bottleneck detection module receives runtime performance metrics and detects potential upcoming bottlenecks. Subsequently, the detected bottleneck severities will be published to a remote redis server on master node. For error-tolerant (Algorithm 2), both observed performance metrics (i.e., val_ob) and detected bottlenecks (i.e., val_inf) will be directed to the redis server on master node.

On the master node, a redis server is deployed to cache detected bottleneck severities and observed performance metrics from each worker node. When a task is ready to run, the LubeScheduler will make task assignment decisions based on runtime bottleneck severities cached on the redis server.

Our implementation reuses the redis APIs as our interfaces, because the redis APIs have simple and clear definitions as well as a broad support for various programming languages. Inherently, Lubeis implemented in a well-decoupled structure with high flexibility and extensibility for further development.

5.1 Pluggable Performance Monitors

A performance monitor is implemented as a daemon process that monitors specific metrics in the background. As an instance of Python multiprocessing.Process, a performance monitor process is optimized for multiple CPU cores and non-blocking. Though implementation based on Python threading consumes less memory, it suffers high context-switching latency introduced by the Global Interpreter Lock (GIL) of Python.

Plugging in another customized performance monitor is easy. We have provided a base class PerfMonitor, with which performance monitor developers only need to: 1) create a child class; 2) pass in a monitor command (e.g., iotop) and a filter function (for filtering out redundant output). By tens of lines of code, one can plug a new performance monitor into the Lubeframework.

In our prototype of Lube, we have developed performance monitors based on nethogs, jvmtop and iotop, which monitor network I/O, JVM performance and disk I/O at process-level respectively. The output of performance monitors is organized as a HashMap following the API style of redis:

$$\{'\mathtt{monitor}_{\mathtt{n}}\mathtt{ame}' : \mathtt{timestamp}, \mathtt{metric\_1}, \mathtt{metric\_2}, \ldots\}.$$

The overhead of performance monitors is negligible. Monitoring toolkits like nethogs, jvmtop and iotop are well-tested and widely-deployed on Linux. Our experiments in Section 6 also show the overhead introduced by monitoring per second can be ignored.

5.2 Bottleneck Detector

The Bottleneck Detector is also a daemon process, running locally on each worker. As a subscriber of the local redis server, the Bottleneck Detector periodically updates the machine learning model based on new monitored performance metrics, and asynchronously pushes predicted bottleneck severities to the remote redis server on master node.

The core wrapped by the Bottleneck Detector process is bottleneck detection algorithms. We build statistics model ARIMA from Python statsmodels.tsa library and the machine learning model SlidHMM from Python scikit-learn library. The models are updated online and serialized as a file saved persistently by Python pickle. The serialization is necessary for potential failures on worker nodes, and avoid training a model from scratch everytime.

The bottleneck information is recorded in a UNIX-style number, each digit indicating the severity level of a specific performance metric. This digit-based structure is easy to extend, efficient to transfer and safe to cache. In our prototype, we define a 4-digit number as follows:

```
                  n   m d  c
                  |   | |  |
network : 0 - 4 < - +   | |  + - > cpu  : 0 - 4
memory  : 0 - 4 < - - -  + + - - - > disk : 0 - 4
```

As an instance, 4201 indicates the intensity levels of the bottleneck in network, memory, disk and CPU are 4, 2, 0 and 1 respectively (the highest level is 4).

5.3 LubeScheduler

We have to integrate the LubeScheduler natively into the codebases of data analytics engines. As one of the core components in data analytics engines, the schedulers have few open APIs exposed to the third-party schedulers. Our prototype scheduler is implemented for Spark-1.6.1, one of the most prevailing data analytics engines. Note that our APIs of bottleneck detection are aligned with standard redis APIs, implementing bottleneck-aware schedulers on other data analytics engines like Hadoop [15] and Cosmos [11] will not take much effort.

To implement an efficient and compatible LubeScheduler, we fully reuse the original codebase of Spark-1.6.1. We implement the LubeScheduler as a subclass of the Spark default task scheduler, inheriting most functions and data structures we do not need to refactor for the bottleneck-aware scheduling. The default task scheduler maintains two queues to respectively track available worker nodes and pending tasks. Each available worker is represented as a slot of CPU cores and memory capacities. A worker node enters the queue when it has free CPU cores and memory capacities. A task and a worker node will both be dequeued after a task assignment decision is made by the scheduler. The default scheduler applies a delay scheduling strategy [46] for data locality. To make a task assignment decision, the scheduler first dequeues a task from the queue of pending tasks, and iteratively looks for a worker that matches the task's locality preference. Based on this intuition, to enable bottleneck-aware scheduling, we extend the scheduler a little bit—to further check the bottleneck severities after the node and execute late-binding as Algorithm 2.

6 Evaluation

In this section, we present evaluation results on Lube. We demonstrate that Lube(i) provides accurate bottleneck detection for Spark jobs, (ii) significantly reduces completion times of queries, and (iii) is robust and scalable to large-scale clusters and workloads. The highlights of our evaluation are as follows.

- *Bottleneck detection accuracy:* Experiments on running different workloads show that Lubewith SlidHMM achieves over 90 percent bottleneck detection accuracy.
- *Bottleneck detection robustness:* We show that SlidHMM can hold a bottleneck detection accuracy around 90 percent when the scales of the cluster and the query increase.
- *Performance improvement:* Lubespeeds up the median query response times from 26.88 percent ($1.4\times$) to 33.46 percent ($1.5\times$) at negligible overheads.
- *Error-tolerance effectiveness:* LubeScheduler achieves approximated speedup rate with different bottleneck detection accuracies by ARIMA and SlidHMM.
- *Scalability:* We evaluate Lubewith a variety of queries on clusters in different scales. The results show Lubescales well and maintains stable performance benefits upon the increasing scales of clusters and workloads.

6.1 Methodology

EC2 Deployment. We deploy Lubeacross 9 EC2 regions in N. Virginia, N. California, Oregon, Ireland, Frankfurt, Tokyo, Seoul, Singapore and Sydney [5]. We launch 37 `m4.2xlarge` instances in total, with 4 workers in each region and 1 master in one region (N. Virginia). Each instance has a 8 cores CPU, 32 GB memory, 1000 Mbps network[2] and a 100 GB SSD disk.

Software Environment: All instances run on Ubuntu-14.04 with Oracle Java-1.8.0. The whole data analytics platform is build with Spark-1.6.1, HDFS-2.6.4 and Hive-1.2.1. For Spark, we configure each worker with 6 CPU cores and 24 GB memory; For HDFS, we configure each data chunk with 3 replicas, the same as the default.

Workloads. We use the Big Data Benchmark [39] as our workload, with datasets from Intel HiBench [24] and Common Crawl [1] document corpus. The dataset is around 1.1 TB (36 datanodes, $30s$ GB per each). Derived from Pavlo et al. [35], Big Data Benchmark contains a mix of Hive and Spark SQL queries for evaluating performance of large-scale parallel queries. There are four major workloads: Query 1 and Query 2 are both exploratory SQL queries; Query 3 is a join query; Query 4 is a user-defined-function

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

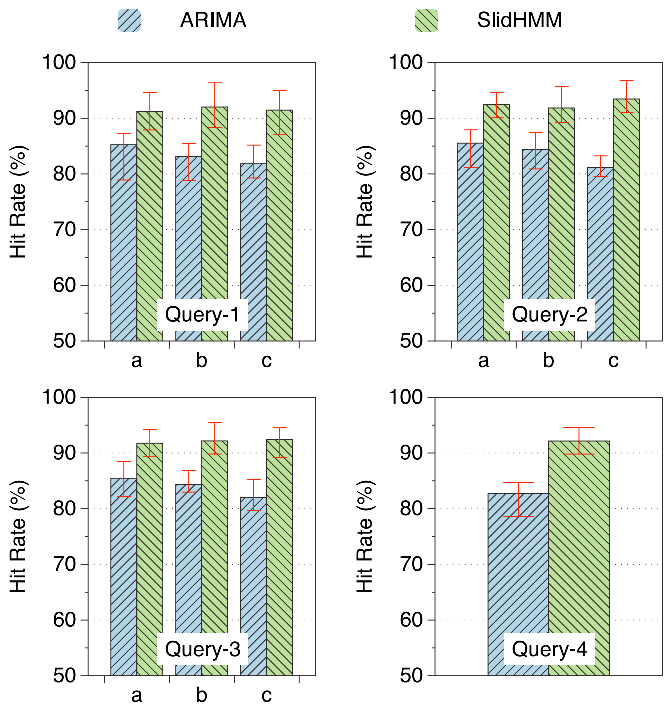


Fig. 7. The accuracy of bottleneck detection.

(UDF) query executing an external python script to count URLs. Query 1-3 each has three scale levels a, b, c, from small to large.

Evaluation Metrics. As for bottleneck detection accuracy, we use *hit rate* to represent the accuracy of our bottleneck detection algorithms. A *hit rate* is defined as the proportion of detected bottlenecks that are observed by monitors among all detected bottlenecks. As for benefits of bottleneck-aware scheduling, we compare the query response times and task completion times on pure Spark and Spark with Lubeenabled. In addition, we measure the overhead by running Lubewith default Spark task scheduler instead of the LubeScheduler. For effectiveness of Lube's error-tolerance, we compare query response times and task completion times under different detection accuracies.

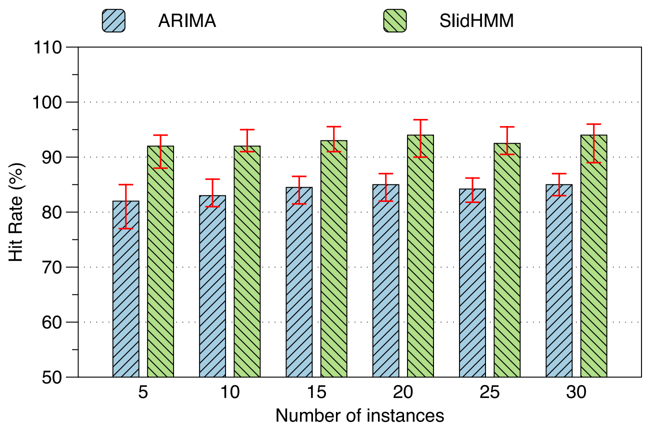
6.2 Bottleneck Detection Accuracy

We use *hit rate* to represent the accuracy of bottleneck detection accuracy. The *hit rate* is formulated as:

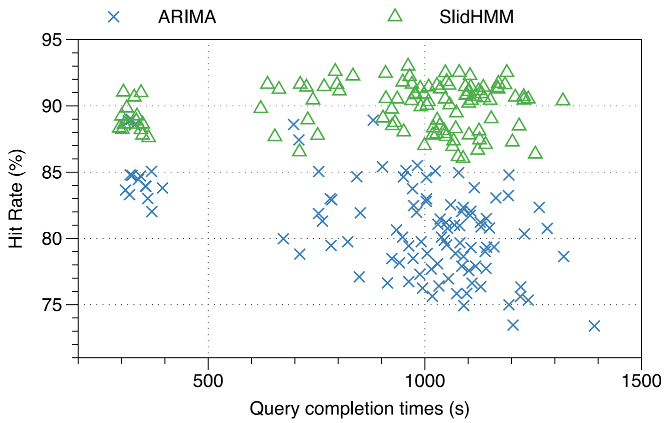
$$hit\ rate = \frac{\#((t, \text{detection}) \cap (t, \text{observation}))}{\#(t, \text{detection})} \times 100\%,$$

which calculates the proportion of verified bottleneck detections. We run each workload of Big Data Benchmark with the ARIMA algorithm and the SlidHMM algorithm for 15 times respectively, tracing both detected bottleneck severity sequences and observation sequences. By calculating *hit rate* offline, we plot the bottleneck detection accuracies under different queries in Fig. 7.

Fig. 7 presents the accuracies of bottleneck detection under different settings. We collect the time stamps and the bottleneck sequences during the running of the Big Data Benchmark for 15 times respectively. During the collection, it should be noted that the Lube scheduler is disabled to order to observe the bottlenecks. By comparing the time



(a) The average accuracy *vs* the scale of clusters.



(b) The accuracies *vs* the scale of queries.

Fig. 8. Analysis of Lube's bottleneck detection accuracy.

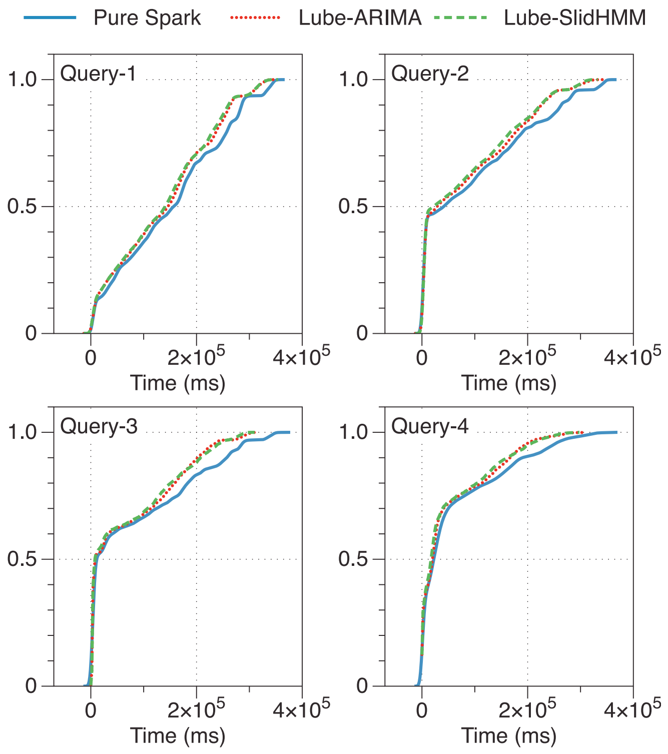


Fig. 9. CDF of task completion times.

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 percent, while it's 83.57 percent 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 nonlinear patterns in the performance metric sequences, which may lower the accuracy of bottleneck detection.

6.3 Bottleneck Detection Robustness

We evaluate accuracies of both ARIMA and SlidHMM on clusters consisting of different numbers of instances, when running each query of the Big Data Benchmark. We plot the average accuracies achieved by ARIMA and SlidHMM under different scales of clusters as Fig. 8a. The accuracies of both ARIMA and SlidHMM are stable in terms of an increasing cluster scale, credited to the distributed design of Lube's bottleneck detection module running locally on each worker node.

Fig. 8b shows that SlidHMM achieves accuracy around 90 percent for long-running queries, which last more than 1000 seconds. However, the accuracy of ARIMA decreases with the increase of query completion times.

6.4 Scheduling Effectiveness and Overheads

The results show that with an accuracy of over 90 percent in bottleneck detection, Lubespeeds up median query response times from 26.88 percent ($1.37\times$) to 33.46 percent ($1.5\times$). Lube achieves a faster query response and maintains a low overhead from the task level to query level.

At the task level, Fig. 9 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. Fig. 10 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, Lubereduces 26.88 to 33.46 percent ($1.37\times$ to $1.5\times$ faster) of time with the ARIMA algorithm, while reduces 28.41 to 33.18 percent ($1.4\times$ to $1.5\times$ faster) of time with the SlidHMM algorithm. From these results, we can conclude that Lubereduces the overall query response times.

In addition, though there is a gap of nearly 10 percent in accuracies of ARIMA algorithm and SlidHMM algorithm, Lubeachieves close performance in reducing task completion times and query response times with these two algorithms. Since the LubeScheduler applies a late-binding

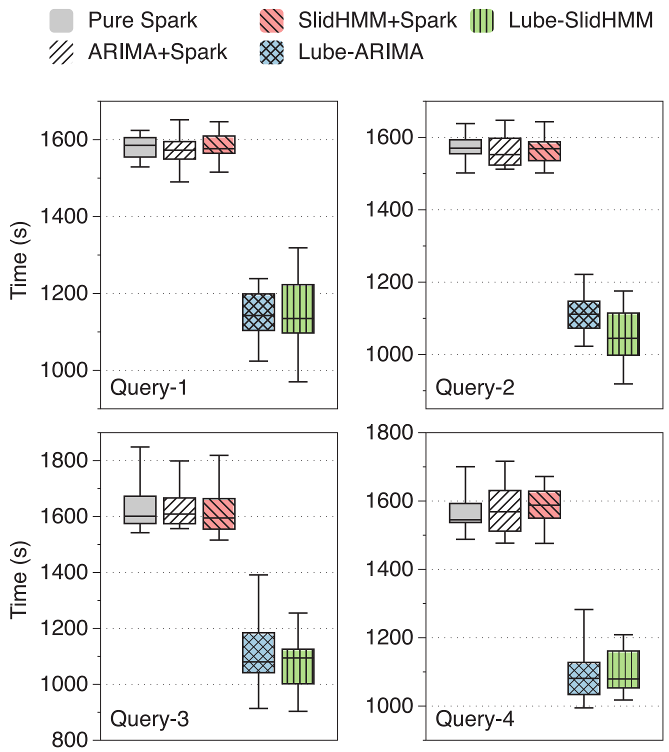


Fig. 10. Query response times.

algorithm for error-tolerance, 10 percent accuracy loss has little impact on query performance.

6.5 Scalability

We evaluate the scalability of Lubeby analyzing the reduction of query completion times when scale up the input data sizes and cluster instance numbers. Fig. 11 presents the scalability of Lubein three dimensions: (i) the cluster size, (ii) the query input data size, and (iii) the query complexity.

The reduction rate of query completion times is calculated in this way:

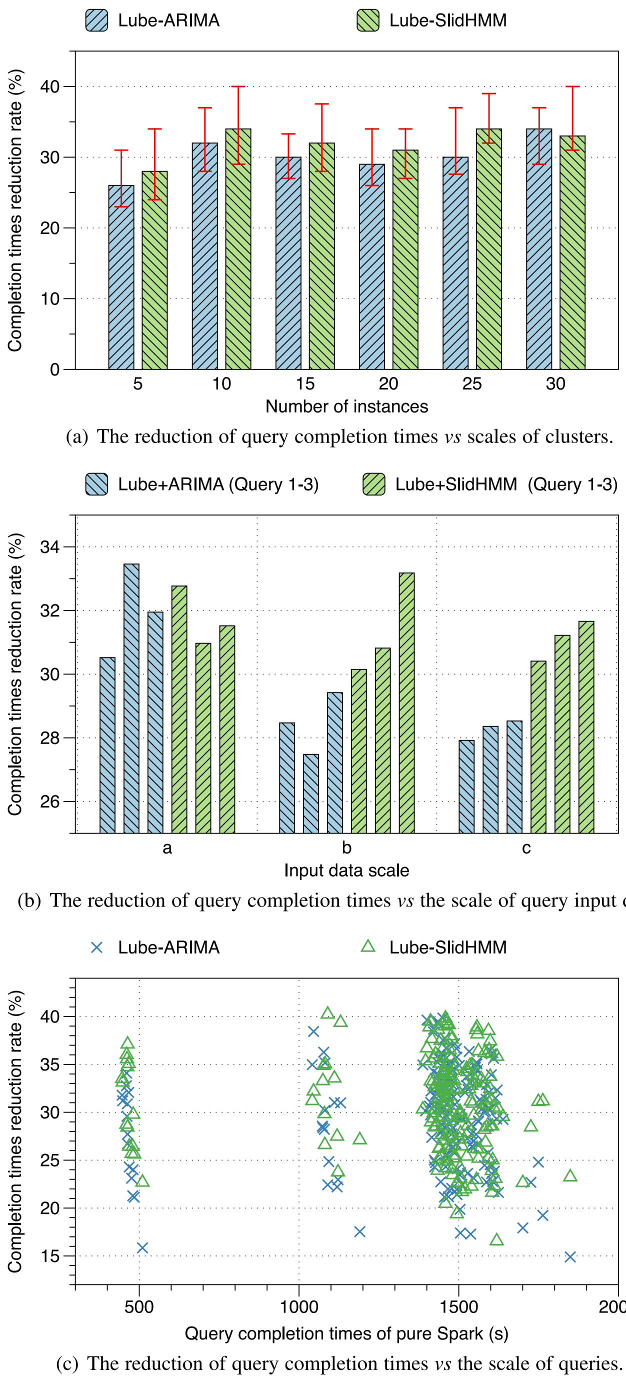
$$reduction\ rate = \frac{\text{average}(t_{pure}) - t_{lube}}{\text{average}(t_{pure})} \times 100\%,$$

in which the $\text{average}(t_{pure})$ is the average completion times of queries running on pure Spark, the t_{lube} is the completion times of queries running on Lube-ARIMA or Lube-SlidHMM.

Fig. 11a plots the reduction rates of query completion times when queries are executed on different sizes of clusters. The completion time reduction rates achieved by both Lube-ARIMA and Lube-SlidHMM float within a steady range of 26 to 34 percent. The scale-up of the cluster has negligible effects on the reduction rates.

Fig. 11b plots the reduction rates of completion times when queries take different scales of input data. Lube-SlidHMM is more robust to the increase of input data sizes, compared to Lube-ARIMA, which tends to have less performance improvement (i.e., query completion time reductions).

Fig. 11c plots the relationships between the reduction rates and the $\text{average}(t_{pure})$, which implies the inherent complexity of queries. There's no obvious decrease in reduction rates for both Lube-ARIMA and Lube-SlidHMM when the $\text{average}(t_{pure})$ increases.



(a) The reduction of query completion times *vs* scales of clusters.

(b) The reduction of query completion times *vs* the scale of query input data.

(c) The reduction of query completion times *vs* the scale of queries.

Fig. 11. Analysis of query performance improvements of Lube.

7 Related Work

There exists large volumes of existing research on optimizing the performance of wide-area data analytics. Clarinet [41] pushes wide-area network awareness to the query planner, and selects a query execution plan before the query begins. Graphene [22] presents a Directed Acyclic Graph (DAG) scheduler with awareness of DAG dependencies and task complexity. Iridium [36] optimizes data and task placement to reduce query response times and WAN usage. Geode [42] minimizes WAN usage via data placement and query plan selection. SWAG [25] 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 [33]. The *blocked time analysis* proposed in [33] 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 [45] duplicates tasks that are slow or failed, but not knowing the exact bottlenecks may lead to worse performance. As far as we know, Lubeis the first work that leverages machine learning techniques to detect runtime bottlenecks and schedules tasks with awareness of performance bottlenecks. Lube is orthogonal to schedulers of the existing wide-area data analytic frameworks. These schedulers can enable bottleneck awareness at runtime by integrating with Lube.

Machine learning techniques have been actively applied to predict and classify data analytics workloads. Nearest-Fit [13] establishes accurate progress predictions of MapReduce jobs by a combination of nearest neighbor regression and statistical curve fitting techniques. Ernest [40] applies a linear regression model to predict the performance of large-scale analytics. CherryPick [4] uses Bayesian Optimization to build cloud performance models for specific applications and cloud configurations.

A Kernel Canonical Correlation Analysis (KCCA) model is applied to predict query properties including completion time and resource demands [19]. ASpR [29] applied Artificial Neural Networks (ANNs) to forecast overhead of function calls and loops in script languages.

8 Discussions

Our preliminary experiments have highlighted the performance of Lubein reducing query response times achieved through detecting and 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 that need additional discussions.

Selection of Runtime Metrics. It is the selection of runtime metrics that determine the efficacy of the runtime performance analysis. There exist an enormous volume of runtime metrics from multiple hierarchies within 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 exploration of runtime metrics and the practice of feature selection techniques such as LASSO Path [18] and principal component analysis (PCA) [28].

Bottleneck Detection Models. Lubeachieves a substantial improvement by applying two simple models, ARIMA and SlidHMM. The emerging data-driven techniques have broadened the horizon of data analytics optimization methodologies. We would like to further explore the latest data-driven techniques, such as Generative Adversary Network (GAN) [20] and Reinforcement Learning [38]. For example, DeepRM [31] 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.

Fine-Grained Scheduling. The scheduling policy applied by Lubeis bottleneck-aware but coarse-grained due to the absence of task resource demand. Recent work on profiling tasks of data analytic applications characterized resource utilization of tasks by an analytical or statistical model [16], [21], [30], [32]. Besides, users can also provide the application resource requirements [23]. By jointly considering the task resource demand and the runtime bottleneck severity, Lubecan enforce a fine-grained task scheduling policy that mitigates the bottleneck of a type of resource required by particular tasks. This fine-grained scheduling policy is expected to achieve higher resource utilization and to further reduce query completion times.

WAN Conditions. Most recent work considered the heterogeneity and the variance of wide-area network bandwidths [4], [22], [25], [36], [40], [42]. A few approaches have been applied to measure network conditions in these works. Lubecaptures the local network throughput by measuring network I/O on each node, which only revealed a coarse-grained awareness of network. It also attempts to measure the pair-wise WAN bandwidth by a `cron` job running `iperf` on each node. In our future work, we plan to exploit the capabilities of Software-Defined Networking (SDN) to complement the global wide-area network conditions at runtime.

9 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. Lubemonitors runtime query performance, detects bottlenecks online and mitigates them with a bottleneck-aware scheduling policy. Experiments across nine EC2 regions show that Lubeachieves over 90 percent bottleneck detection accuracy and, compared to the default Spark scheduler, reduces the median query response time by up to 33 percent.

As a highlight, this work delves into runtime performance of wide area data analytics and delivers a responsive strategy as a workaround. It introduces an orthogonal solution to today's scheduling strategies in the wide area—detecting and mitigating bottlenecks online. Lubeis instrumental to improving performance profiling techniques, machine learning on time series analysis, and optimized resource scheduling strategies. For those who work on scheduling strategies in distributed data analytics engines, our pluggable performance sensor module will provide valuable insights on the real-time performance metrics they care about.

Acknowledgments

This work is supported by a research contract with Huawei Technologies Co. Ltd. and an NSERC Collaborative Research and Development (CRD) grant. Preliminary results have been presented in USENIX HotCloud'17 [44]

References

[1] Common Crawl Web Crawl Data. [Online]. Available: http://commoncrawl.org, Accessed on: Jul. 1, 2016.
[2] JSON Website. [Online]. Available: http://json.org, Accessed on: May 1, 2016.

[3] NTP: The Network Time Protocol. [Online]. Available: http://www.ntp.org, Accessed on: Jul. 1, 2016.

[4] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "CherryPick: Adaptively unearthing the best cloud configurations for big data analytics," in *Proc. USENIX Symp. Networked Syst. Des. Implementation*, 2017, pp. 2–4

[5] Amazon, Amazon Web Services. [Online]. Available: https://aws.amazon.com/about-aws/global-infrastructure/, Accessed on: Jul. 1, 2016.

[6] Apache. Apache Hadoop Official Website. [Online]. Available: http://hadoop.apache.org/, Accessed on: May 1, 2016.

[7] Apache. Spark SQL. [Online]. Available: https://spark.apache.org/sql/, Accessed on: Jul. 1, 2016.

[8] L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state Markov chains," *Ann. Math. Statist.*, vol. 37. no. 6, pp. 1554–1563, 1966.

[9] L. E. Baum, T. Petrie, G. Soules, and N. Weiss, "A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains," *Ann. Math. Statist.*, vol. 41, no. 1, pp. 164–171, 1970.

[10] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*. Hoboken, NJ, USA: Wiley, 2015.

[11] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: Easy and efficient parallel processing of massive data sets," *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1265–1276, 2008.

[12] T. Chis, "Sliding hidden markov model for evaluating discrete data," in *Proc. 10th Comput. Perform. Eng. Eur. Workshop*, 2013, vol. 8168, pp 251–262.

[13] E. Coppa and I. Finocchi, "On data skewness, stragglers, and MapReduce progress indicators," in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, pp. 139–152.

[14] J. G. De Gooijer and R. J. Hyndman, "25 years of time series forecasting," *Int. J. Forecasting*, vol. 22, no. 3, pp. 443–473, 2006.

[15] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2004, pp. 10–10.

[16] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware Cluster Management," in *Proc. 19th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2014, pp. 127–144.

[17] D. A. Dickey and W. A. Fuller, "Distribution of the estimators for autoregressive time series with a unit root," *J. Amer. Statistical Assoc.*, vol. 74, no. 366a, pp. 427–431, 1979.

[18] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning*. New York, NY, USA: Springer, 2001.

[19] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *Proc. IEEE 25th Int. Conf. Data Eng.*, 2009, pp. 592–603.

[20] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Proc. Advances Neural Inf. Process. Syst.*, 2014, pp. 2672–2680.

[21] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 455–466, 2015.

[22] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "Graphene: Packing and dependency-aware scheduling for data-parallel clusters," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, p. 81.

[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2011, pp. 295–308.

[24] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. Int. Conf. Data Eng. Workshops*, 2010, pp. 41–51.

[25] C. Hung, L. Golubchik, and M. Yu, "Scheduling jobs across geo-distributed datacenters," in *Proc. ACM Symp. Cloud Comput.*, 2015, pp. 111–124.

[26] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, pp. 407–420, 2015.

[27] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: Optimizing data parallel jobs in wide-area data analytics," *Proc. VLDB Endowment*, vol. 9, no. 2, pp. 72–83, 2015.

[28] K. Pearson, "On lines and planes of closest fit to systems of points in space," *London, Edinburgh, Dublin Philosophical Magazine J. Sci.*, vol. 2, no. 11, pp. 559–572, 1901.

[29] J. Li, X. Ma, K. Singh, M. Schulz, B. R. de Supinski, and S. A. McKee, "Machine learning based online performance prediction for runtime parallelization and task scheduling," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2009, pp. 89–100.

[30] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2009, pp. 45–55.

[31] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Topics Netw.*, 2016, pp. 50–56.

[32] V. S. Marco, B. Taylor, B. Porter, and Z. Wang, "Improving spark application throughput via memory aware task co-location: A mixture of experts approach," in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.*, 2017, pp. 95–108.

[33] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *Proc. USENIX Symp. Netwo. Syst. Des. Implementation*, 2015, pp. 293–307.

[34] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 69–84.

[35] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proc. ACM Int. Conf. Manag. Data*, 2009, pp. 165–178.

[36] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency Geo-Distributed data analytics," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 421–434.

[37] Redis, Redis, [Online]. Available: http://redis.io/, Accessed on: May 1, 2016.

[38] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press Cambridge, 1998.

[39] UC Berkeley AMPLab, The Big Data Benchmark. [Online]. Available: https://amplab.cs.berkeley.edu/benchmark/, Accessed on: Jul. 1, 2016.

[40] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. 13th USENIX Symp. Networked Syst. Des. Implementation*, 2016, pp. 363–378.

[41] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "Clarinet: WAN-aAware optimization for analytics queries," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 363–378.

[42] A. Vulimiri, C. Curino, P. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global analytics in the face of bandwidth and regulatory constraints," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2015, pp. 323–336.

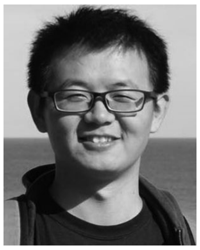
[43] A. Vulimiri, C. Curino, P. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for a geo-distributed data-intensive world," in *Proc. Conf. Innovative Data Syst. Res.*, 2015, pp. 1087–1092

[44] H. Wang and B. Li, "Lube: mitigating bottlenecks in wide area data analytics," in *Proc. 9th USENIX Conf. Hot Topics Cloud Comput.*, 2017, p. 1.

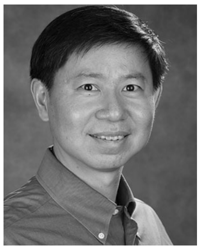
[45] T. White, *Hadoop: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2012.

[46] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.

[47] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2012, pp. 2–2.



Hao Wang received the BE degree in information security and the ME degree in software engineering from Shanghai Jiao Tong University, Shanghai, China, in 2012 and 2015 respectively. His research interests include large-scale data analytics, distributed computing, machine learning, and datacenter networking.



Baochun Li (F'15) received the BE degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995 and the MS and PhD degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 1997 and 2000. Since 2000, he has been with the Department of Electrical and Computer Engineering, University of Toronto, where he is currently a professor. He held the Nortel Networks junior chair in Network Architecture and Services from October 2003 to June 2005, and the Bell Canada Endowed chair in Computer Engineering since August 2005. His research interests include cloud computing, large-scale data processing, computer networking, and distributed systems. In 2000, He was the recipient of the IEEE Communications Society Leonard G. Abraham Award in the Field of Communications Systems. In 2009, he was a recipient of the Multimedia Communications Best Paper Award from the IEEE Communications Society, and a recipient of the University of Toronto McLean Award. He is a fellow of the IEEE and a member of ACM.