DRS: Auto-Scaling for Real-Time Stream Analytics

Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, *Senior Member, IEEE*,
Marianne Winslett, *Associate Member, IEEE, Fellow, ACM,* Yin Yang, and Zhenjie Zhang

***Abstract*— In a stream data analytics system, input data arrive continuously and trigger the processing and updating of analytics results. We focus on applications with real-time constraints, in which, any data unit must be completely processed within a given time duration. To handle fast data, it is common to place the stream data analytics system on top of a cloud infrastructure. Because stream properties, such as arrival rates can fluctuate unpredictably, cloud resources must be dynamically provisioned and scheduled accordingly to ensure real-time responses. It is essential, for existing systems or future developments, to possess the ability of scaling resources dynamically according to the instantaneous workload, in order to avoid wasting resources or failing in delivering the correct analytics results on time. Motivated by this, we propose DRS, a dynamic resource scaling framework for cloud-based stream data analytics systems. DRS overcomes three fundamental challenges: 1) how to model the relationship between the provisioned resources and the application performance, 2) where to best place resources, and 3) how to measure the system load with minimal overhead. In particular, DRS includes an accurate performance model based on the theory of *Jackson open queueing networks* and is capable of handling *arbitrary* operator topologies, possibly with loops, splits, and joins. Extensive experiments with real data show that DRS is capable of detecting sub-optimal resource allocation and making quick and effective resource adjustment.**

***Index Terms*— Cloud computing, queueing network model, resource auto-scaling, stream data analytics.**

I. Introduction

IN APPLICATIONS such as analytics over microblogs, video feeds or sensor readings, data records are not available beforehand, but gradually and continuously arrive in the form of streams. A stream data analytics system handles such streams, executes user-defined computation logics and produces analysis results and/or statistical updates. Often, users are interested in performing data analytics in *real time*, meaning that the results/updates must be produced within a given time period after any new input data arrives. For instance, for a stream data analytics system monitoring surveillance video streams in hospital wards, events such as a patient falling should be detected promptly to alarm doctors and nurses in time.

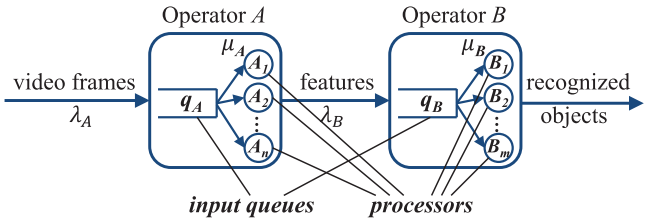


Fig. 1. An illustration of stream data analytics applications.

To handle fast, high-volume streams with stringent real-time constraints, it is increasingly common to place stream data analytics systems on top of a cloud infrastructure, which provides virtually unlimited computing resources on demand. Because key properties of a data stream such as its volume, arrival rates and value distribution, can fluctuate in an unpredictable manner, the cloud-based stream data analytics systems should dynamically provision cloud resources for each application so as to satisfy the real-time constraints with minimum resource consumption. Meanwhile, for each application, resources need to be carefully *scheduled* to different components to optimize performance. Misplacing resources may cause not only poor resource utilization, but system instability as well.

Figure 1 shows an example of video stream processing application with two operators A (which extracts features from input video frames) and B (which recognizes objects from the extracted features), with the output of A fed to B as input. The arrival rates of data records for A and B are denoted by λ_A and λ_B respectively, where λ_A depends on the input, e.g., 24 frames per second, while λ_B depends on the output rate of A, i.e., the number of features extracted in a unit time. Inside each operator, an input is first buffered at an *input queue*, i.e., q_A and q_B, before being processed by one of the parallel processors, i.e., $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$. Assuming the cloud provides identical processing units, each processor in A (respectively in B) can process μ_A (μ_B) data records in a unit of time. Clearly, an operator must have sufficient processors to keep up with its input rate; otherwise, inputs start to fill its input queue, resulting in increased latency

Manuscript received August 17, 2016; revised May 2, 2017; accepted July 27, 2017; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor L. Ying. Date of publication September 1, 2017; date of current version December 15, 2017. This work was supported by the research grant for the Human Centered Cyber-physical Systems Programme at the Advanced Digital Sciences Center from Singapore's Agency for Science, Technology and Research (A*STAR). The work of T. Z. J. Fu was supported in part by the Natural Science Foundation of China under Grant 61702113 and in part by the China Postdoctoral Science Foundation under Grant 2017M612613. The work of R. T. B. Ma was supported by the Ministry of Education of Singapore AcRF under Grant R-252-000-572-112. The work of Y. Yang was supported by the QNRF Project under Grant NPRP9-466-1-103. The work of Z. Zhang was supported by the Science and Technology Planning Project of Guangdong under Grant 2015B010131015. *(Corresponding author: Richard T. B. Ma.)*

T. Z. J. Fu is with the Advanced Digital Sciences Center, Illinois at Singapore Pte. Ltd., Singapore 138602, and also with the Guangdong University of Technology, Guangzhou, China (e-mail: fuzhengjia@gmail.com).

J. Ding is with the School of Information Science and Technology, Sun Yat-sen University, Guangzhou 510275, China (e-mail: dingsword@gmail.com).

R. T. B. Ma is with the School of Computing, National University of Singapore, Singapore 117418 (e-mail: tbma@comp.nus.edu.sg).

M. Winslett is with the Department of Computer Science, University of Illinois at Urbana–Champaign, Champaign, IL 61801 USA (e-mail: winslett@illinois.edu).

Y. Yang is with the College of Science and Engineering, Hamad Bin Khalifa University, Doha, Qatar (e-mail: yyang@qf.org.qa).

Z. Zhang is with the Advanced Digital Sciences Center, Illinois at Singapore Pte. Ltd., Singapore 138602 (e-mail: zhenjie@adsc.com.sg).

Digital Object Identifier 10.1109/TNET.2017.2741969

1063-6692 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

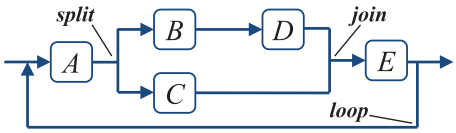


Fig. 2. Example complex operator topology.

due to queueing delay, and eventually, errors when the queue overflows. Since the arrival and processing rate for each processor are uncontrollable, the main resource scaling issue is to determine the number of processors for each operator, i.e., n and m in our example.

A simple approach to scaling resources is to monitor the workload in each operator and adjust the number of processors accordingly; however, this approach is insufficient for multi-operator applications. For instance, consider the case that at some point, many recognizable objects appear in the video stream. Although the number of frames per second in the input λ_A remains stable, each frame now contains more extractable features, requiring more work at operator A. Consequently, μ_A will decrease, which subsequently overloads operator A, causing inputs to wait longer in its queue q_A and slowing down output responses. If we naively add processors to A to flush q_A, operator A will suddenly produce a large amount of outputs, leading to a burst in the input rate λ_B and overload operator B. This problem is exacerbated when the application involves a complex processing topology of its operators. Figure 2 shows such an example with split (A to B, C), join (C, D to E) and feedback loop (E to A). Such topological features are key enablers for certain applications, e.g., feedback loop allows data reduction at the input based on the most updated results as we will show in an empirical example in Section V.

As we review in Section II, existing systems overlook the problem of dynamic resource scaling. Consequently, to meet the real-time constraint, they either require manual tuning at runtime (which is infeasible for dynamic streams), overprovisioning resources to operators (which wastes resources), or load shedding (which leads to incorrect results). Motivated by this, we design and implement DRS, a dynamic resource scaling framework that applies to general operator-based stream data analytics systems and allows operators to form an *arbitrary* topology, possibly with splits, joins and loops as shown in Figure 2. In particular, the support for loops is crucial for applications involving iterations. Meanwhile, from a semantics point of view, allowing arbitrary topologies is more general than the two-step MapUpdate in Muppet [1] and the DAG model in TimeStream [2]. Our main contributions include effective and efficient solutions to three fundamental problems in dynamic resource scaling: (i) how much resources are needed, (ii) where to best place the allocated resources to minimize average response time, and (iii) how to implement resource auto-scaling in a real system with minimal overhead. In particular, our solutions to the first two problems are based on the theory of extended Jackson networks, which provides an educated estimate of system performance.

The rest of the paper is organized as follows. Section II surveys related work. Section III presents our performance model and optimization algorithm. Section IV describes the system design and implementation of DRS. Section V contains an extensive set of experiments with real data. Section VI concludes with directions for future work.

II. Related Work

A. Resource Scaling in Cloud Systems

A cloud consists of a massive number of interconnected commodity servers. A key feature of the cloud is that its resources, such as CPU cores, memory, disk space and network bandwidth can be provisioned to applications on demand. In fact, most cloud infrastructure providers today offer pay-as-you-go options for resource usage. Hence, a fundamental requirement for a system to effectively use the cloud is *elasticity*, meaning that the system must be able to dynamically allocate and release cloud resources based on the current workload. Many traditional parallel and distributed systems, however, assume a fixed amount of resources available beforehand, rendering them unsuitable to be applied in a cloud platform. As a result, many novel elastic cloud-based paradigms and systems have emerged in the past decade.

The first wave of cloud-based systems were built for running a batch of (often slow) jobs offline. Notably, MapReduce [3] is a batch processing framework that hides the complexity of the cloud infrastructure, and exposes a simple programming interface to users consisting of two functions: *map* (e.g., for data filtering and transformation) and *reduce* (for aggregation and join). A plethora of MapReduce systems, improvements, techniques, and optimizations have been proposed in recent years, and we refer the reader to a comprehensive survey [4].

Resource scheduling has been a central problem in Map-Reduce like systems, and a plethora of schedulers have been developed and used in production, e.g., Fair Scheduler [5], Capacity Schedular [6]. Since tasks running on nodes without relevant data incur costly network transmissions, delay scheduling [7] reduces such non-local tasks by forcing nodes to wait until either a local task appears, or a specified period has passed. These scheduling strategies, however, do not apply to our problem, because they are designed for offline, batch processing of (semi-) static data, where the goal is to minimize *total* job completion time. Another direction investigates task/job scheduling in fork-join process network with task synchronization constraints [8], where queuing network models are applied for finding the optimal scheduling policy. Our work focuses on the optimal resource allocation of processors to individual operators where the service capacity of each processor is inseparable and pre-determined.

Recently, much attention has been shifted to real-time interactive systems for big data analytics, such as Dremel [9], Presto [10], OceanRT [11], [12], C-Cube [13], and newer versions of Hive [14]. Such systems deal with static rather than streaming data; meanwhile, the term "real-time" has a different meaning that each query is executed quickly enough so that the user can wait online for its results. Hence, resource scheduling in these systems resembles offline systems and their techniques do not apply to our problem for similar reasons.

Finally, there exist generic scheduling solutions for provisioning to multiple applications competing for cloud resources. Systems such as Mesos [15], YARN [16] are prominent

examples. Abacus [17] optimizes total utility by allocating resources via a truth-revealing auction. These methods generally assume that an application already knows the amount of resources it needs, and how to distribute these resources internally, which are the problems solved in this paper. Hence, they can be used in combination with the proposed solution.

B. Traditional Stream Data Processing Systems

Stream processing has been an important research topic in both academia and industry. Earlier work focuses on stream data analytics systems in a centralized setting, which resembles the traditional, centralized database management systems. For instance, STREAM [18] establishes formal semantics for queries over streams [19], and proposes efficient query processing algorithms, e.g., [20]. Similar systems include Aurora [21], Gigascope [22], TelegraphCQ [23], and System S [24]. Scheduling in such centralized systems means deciding the best order of operators to execute (by the central processor), e.g., in order to minimize memory consumption [25], [26]. Hence, scheduling strategies in these systems do not apply to our cloud-based setting, where operators are executed by multiple nodes in parallel, and computational resources are dynamically provisioned on demand. Stream systems built for traditional parallel settings, notably Borealis [27], also differ from cloud-based stream data analytics systems in that the former assume that a fixed amount of computational resources available beforehand, rather than dynamically allocated.

C. Cloud-Based Stream Processing

There are two general methodologies for processing streams in a cloud: using an operator-based stream data analytics systems, and discretizing stream inputs into mini-batches [28]. The former derives from traditional stream data processing, whereas the latter reduces stream processing to batch execution. In general, mini-batch systems are optimized for throughput, at the expense of increased query response time, since each input must wait until a full batch is formed. While it is possible to minimize this extra latency by having extremely small batches, doing so would lead to high overhead, defeating the purpose. We focus on operator-based stream data analytics systems since our target applications have real-time constraints, in which response time is key.

Popular open source operator-based stream data analytics systems include Storm [29], Heron [30] and S4 [31]. Their main difference is that Storm and Heron guarantee the correctness of results (through the Trident component), while S4 does not. All the systems rely on manual configurations for resource scheduling. Hence, to avoid slow responses due to operator overloading, the user has to either overprovision resources to every operator, which is wasteful, or continuously tuning the system, which is infeasible for dynamic streams.

Many research prototypes of operator-based stream data analytics systems are proposed, such as TimeStream [2], which features efficient fault recovery, and Samza [32]. None of these systems, however, addresses the resource scaling problem.

TABLE I
TABLE OF NOTATION

Symbol	Meaning
N	Total number of operators in an application
λ_0	Arrival rate of inputs to the application
λ_i	Arrival rate of inputs to operator i
μ_i	Processing rate of inputs to operator i
a_i	Squared coefficient of variation (SCV) of inter-arrival times of inputs at operator i
s_i	SCV of processing times of inputs at operator i
k_i	# of processors allocated to operator i
$\mathbf{k}$	A vector $(k_1, \ldots, k_N)$ containing all k_is
$T_{\max}$	Real-time constraint parameter: each input of the application is expected to be fully processed within $T_{\max}$.
$K_{\max}$	Resource constraint parameter: maximum number of available processors that can be allocated to operators
t	An input tuple to the streaming application
T	A random variable on the total sojourn time of a tuple t

In the following we present DRS, an effective resource scheduler for cloud-based stream data analytics systems.

III. Dynamic Resource Scaling

We focus on stream analytics applications, which are usually memory-based and computation intensive. For such applications, *processors* are the main type of resource, each of which contains a CPU (or one of its cores) and certain amount of RAM. Disk space is not critical as streaming inputs are computed on-the-fly. The goal of DRS is to fully process each input of the application in real time. Specifically, an input tuple to the application, e.g., a video frame in Figure 1, may lead to multiple intermediate results, e.g., features extracted by operator A and objects recognized by operator B. We say an input tuple t is fully processed, if and only if every intermediate result derived from t has been processed by its corresponding operator. We use the term *total sojourn time* to refer to the duration from t first arrives at the system till t is fully processed. Our goal is twofold: (i) to minimize the expected total sojourn time of each input t, when the total number of available processors, denoted by $K_{\max}$, are specified by users; or (ii) to minimize the total number of processors while ensuring that the expected total sojourn time is no more than a user-specified duration, denoted by $T_{\max}$. Table I summarizes the frequently used notation in the paper.

A. Performance Model

Given an application's operator topology, e.g., the one in Figure 2, the existing resource allocation and the characteristics of streaming data, the DRS performance model estimates the average total sojourn time of an input of the application. For the ease of presentation, we assume that all processors in the cloud have *identical computational power*. Nevertheless, the proposed models and algorithms can also support settings with heterogeneous processors, and we will explain how this can be done whenever necessary. Meanwhile, we assume that *load balancing* is achieved in every operator, i.e., each processor inside the same operator performs roughly equal amount of work. How to achieve load balancing is an orthogonal topic under active research [33]–[35]. Under these assumptions, the processing speed of an operator depends

mainly on the number of processors, and therefore, the existing resource allocation in the system is the number of processors assigned to each operator. Formally, we use N to denote the number of operators of an application and a resource allocation solution is represented by a vector $\mathbf{k} = (k_1, k_2, \ldots, k_N)$, where $k_i (1 \leq i \leq N)$ corresponds to the number of processors allocated to operator i.

Regarding the characteristics of data, the important variables include the rate at which tuples arrive at each operator, and how fast the tuples can be processed by one processor. Notice that the inter-arrival and processing (or service) times of the data tuples are random and our model can handle fluctuations in arrival rates and processing rates; however, we do assume that the system remains in a relatively steady state during the short time-span that DRS performs resource scaling. This means that the instantaneous tuple arrival rate and processing time at each operator remains stable, and we obtain these quantities through the measurement module of the system, described in Section IV. Specifically, for operator i, we use λ_i to denote the arrival rate of its inputs, and μ_i to denote the processing rate of each of its processors. For instance, if $k_i = 3$, $\lambda_i = 10$ and $\mu_i = 4$, on average 10 tuples arrive at operator i per unit time, and each of its 3 processors processes 4 tuples per unit time. For operators with multiple input streams, e.g., join operators, λ_i is the aggregate arrival rate of all its input streams, and μ_i is the average processing rate of the operator, regardless of where the tuple comes from. In addition, we denote the external arrival rate of inputs that flow into the application's operator topology by λ_0. When there are clear "source" operators in the topology, λ_0 is simply the total arrival rates of these sources. In general, however, there may not be a simple relationship between λ_0 and λ_is. For the example in Figure 2, λ_0 is the arrival rate of tuples that arrive (from outside the system) at operator A; the input arrival rate λ_A for A on the other hand is the sum of λ_0 and the arrival rate of A's other input stream, produced by operator E.

We use random variable T to denote the total sojourn time of an input tuple to the application. Our goal is to estimate $\mathbb{E}[T]$, i.e., the expected value of T. The basic idea is to apply the theory and established results of open queueing networks (OQNs). In an OQN, the total sojourn time of an input tuple t is computed by summing up its service times, i.e., total time spent on processing t and intermediate results derived from t, and queueing delays, i.e., total time that t and its derived tuples wait in the operators' queues. The challenge, however, is that there are numerous OQN models in the queuing literature, and selecting an appropriate one is non-trivial. On the one hand, complex queueing network models generally do not have known solutions, and even among the ones that do, most have only numerical solutions rather than closed-form analytical ones, which makes effective optimization hard. On the other hand, an overly simplified model may rely on strong assumptions that do not hold in our setting.

After comparing various options and testing them through experiments, we chose to build our model based on the *Jackson networks* ($M/M/k$ OQNs) [36], [37], which not only enable effective analysis of each individual operator, but also help aggregate these analyses to estimate $\mathbb{E}[T]$. In addition, our model provides an analytical solution and efficient optimizations, and has only mild limitations, which will be discussed shortly. We use random variable T_i to denote the sojourn time of an input at operator i, representing the time between its arrival at this operator till it is fully processed. In a steady state, the average sojourn time $\mathbb{E}[T_i]$ is a function of the number of allocated processes k_i and consists of two parts: (i) the expected processing time which is equal to $1/\mu_i$ and (ii) the expected queuing delay, denoted by $\mathbb{E}[Q_i](M/M/k_i)$, i.e.,

$$\mathbb{E}[T_i](k_i) = \mathbb{E}[Q_i](M/M/k_i) + \frac{1}{\mu_i}. \tag{1}$$

By applying the Erlang's delay formula [38], the expected queuing delay of a $M/M/k$ service node can be calculated by

$$\mathbb{E}[Q_i](M/M/k_i) = \begin{cases} \dfrac{\pi_0 (k_i \rho_i)^{k_i}}{k_i!(1-\rho_i)^2 \mu_i k_i} & \text{for} \rho_i < 1; \\ +\infty & \text{for} \rho_i \geq 1, \end{cases} \tag{2}$$

where $\rho_i = \frac{\lambda_i}{k_i \mu_i}$ defines the utilization of operator i and π_0 is a normalization term, given by:

$$\pi_0 = \left[\sum_{l=0}^{k_i - 1} \frac{(k_i \rho_i)^l}{l!} + \frac{(k_i \rho_i)^{k_i}}{k_i!(1-\rho_i)} \right]^{-1}. \tag{3}$$

Equation (2) intuitively shows that because tuples arrive at a rate λ_i and each processor processes them at a rate μ_i, when $\rho_i \geq 1$, the processors cannot keep up with incoming tuples and the number of tuples in the operator's queues increases with time, leading to infinite queuing delays. Conversely, when $\rho_i < 1$, tuples are expected to be handled faster than they arrive. However, due to the randomness in the inter-arrival and processing times, queues may still be built up temporarily.

Based on the theory of OQNs [36], [39], $\mathbb{E}[T]$ of the entire topology can be computed as a weighted sum of the $\mathbb{E}[T_i]$s as

$$\mathbb{E}[T](\mathbf{k}) = \mathbb{E}[T](k_1, k_2, \ldots, k_N) = \frac{1}{\lambda_0} \sum_{i=1}^{N} \lambda_i \mathbb{E}[T_i](k_i). \tag{4}$$

Discussion: Since the above described DRS performance model relies on *Jackson networks* with $M/M/k$ service nodes, it inherits two limitations. First, the model implicitly assumes that both the inter-arrival times of external tuples (that come from outside the system) and the service times of the operators are independently and identically distributed (iid) exponential random variables. Second, Jackson network does not explicitly model the pipelining between different operators, nor the split of jobs [40]. Hence, our model may give an inaccurate estimate of $\mathbb{E}[T]$, when the inter-arrival or service time deviates significantly from the expected exponential distribution, or when pipelining or job splitting affects total processing time considerably.

To alleviate the effects caused by the first limitation, we can simply extend our performance model to the *generalized Jackson networks* ($GI/G/k$ OQNs) [36], where the tuple arrival and processing are approximated by renewal processes and each operator is modeled as a stochastically independent

$GI/G/k$ service node. By applying the parametric decomposition approach developed by Whitt [41], for any operator i, the expected sojourn time $\mathbb{E}[T_i]$ can be approximated as

$$\mathbb{E}[T_i](k_i) \approx \left(\frac{a_i + s_i}{2}\right) \mathbb{E}[Q_i](M/M/k_i) + \frac{1}{\mu_i}, \qquad (5)$$

where $E[Q_i](M/M/k_i)$ is defined in Equation (2) and a_i (resp. s_i) is the *squared coefficient of variation* or the variability of the tuples' inter-arrival times (resp. processing times), and is defined as the variance divided by the squared mean of the inter-arrival times (resp. processing times) at operator i [42].

According to Equation (5), the expected sojourn time of a $GI/G/k$ service node, of which both tuple inter-arrival times and processing times are non-Markovian, is a straightforward extension of that of an $M/M/k$ service node in Equation (1), with the second moment information, i.e., the squared coefficient of variation of the inter-arrival times and processing times, taken into consideration. It can be easily verified that $M/M/k$ is a special case of $GI/G/k$, where $a_i = s_i = 1$ for operator i. In the meantime, Equation (4) is general enough that it can be applied directly for aggregating all the $\mathbb{E}[T_i]$s of $GI/G/k$ service nodes to calculate $\mathbb{E}[T]$.

Nevertheless, as will be shown in our experimental evaluations, (i) the value of $\mathbb{E}[T]$ predicted by the $M/M/k$ OQNs model is sufficiently accurate, when the underlying application is computation intensive; (ii) even if the estimation deviates from the measurement, it is still strongly correlated with the exact value of $\mathbb{E}[T]$, implying that DRS remains capable of identifying the best resource allocation with the predicted value; and (iii) for most cases, the extended $GI/G/k$ OQNs model shows very similar capability to the $M/M/k$ OQNs model in estimating $\mathbb{E}[T]$ as well as identifying the optimal resource allocation.

Our model does not explicitly consider networking costs due to the fact that (i) recent research [43] on performance analysis on data analytics frameworks has revealed that network is often not the bottleneck, but CPU is; (ii) data centers today are increasingly equipped next-generation networking hardware that provide significantly higher bandwidth and lower latency, such as 10G Ethernet [44] and InfiniBand [45], whose prices have been dropping rapidly. In contrast, processor speed in terms of CPU clock rate and RAM latency has stagnated in the past few years. Hence, we consider CPU to be the bottleneck of the system, not the network bandwidth; (iii) as the network bandwidth increases rapidly, it brings more challenges and higher costs in measuring the network delay, e.g., clock synchronization among physical nodes can be expensive in a real-time application, but very limited benefits according to (i) and (ii). As will be shown in our experimental evaluations, even when network costs lead to deviations in the estimation of $\mathbb{E}[T]$ made by our model, they do not affect the capability of DRS to identifying the optimal resource allocation.

In the rest of the section, we first show how DRS adjusts resources based on the $M/M/k$ OQNs model, i.e., Equations (1)-(4), and then how the scaling algorithm can easily adapt to the extended model based on $GI/G/k$ OQNs.

B. Scaling Algorithm

In a nutshell, DRS (i) monitors the current performance of the system (more details in Section IV), (ii) checks whether the performance degrades (or is about to degrade) under real-time constraints, or when the system can fulfill the constraints with less resource, and (iii) reschedules resources when the conditions of (ii) are triggered. The main challenge lies in (ii), which needs to answer two questions: 1) *how many* processors are needed to satisfy the real-time constraints, and 2) *where* to place them in the operator topology. We first focus on the latter question. Specifically, given a number (say, $K_{\max}$) of processors, we are to find an optimal assignment of these processors to the operators of an application that obtains the minimum expected total sojourn time $\mathbb{E}[T]$. The optimization problem can be mathematically formalized as follows:

$$\begin{aligned} \min_{\mathbf{k}} \quad & \mathbb{E}[T](\mathbf{k}) \\ \text{s.t.} \quad & \sum_{i=1}^{N} k_i \leq K_{\max},\ k_i \text{ is interger}, i = 1, 2, \ldots, N. \end{aligned} \qquad (6)$$

A naive approach to solve the above problem is to view it as an integer program and apply standard solvers. However, existing integer programming solvers are prohibitively slow, especially considering that DRS itself has to run in real time. In the following we describe a greedy algorithm (presented as Algorithm 1) to solve (6) based on the $GI/G/k$ OQNs model.

The idea of the proposed algorithm is to start from the smallest possible value of each k_i (lines 1-3) and iteratively add one processor to the operator that leads to the largest decrease in $\mathbb{E}[T]$ (lines 7-14). According to Equation (2), each k_i must be greater than λ_i/μ_i; otherwise, $\mathbb{E}[T_i](k_i)$ becomes infinitely large, leading to infinity on $\mathbb{E}[T]$ as well.

Algorithm 1 $AssignProcessors$

Input: $K_{\max}$, λ_0, $\{\lambda_i, i = 1, \ldots, N\}$, $\{\mu_i, i = 1, \ldots, N\}$, $\{a_i, i = 1, \ldots, N\}$, $\{s_i, i = 1, \ldots, N\}$.
Output: $\mathbf{k} = (k_1, k_2, \ldots, k_N)$
1: **for all** $i \leftarrow 1, \ldots, N$ **do**
2: $k_i \leftarrow \left\lfloor \frac{\lambda_i}{\mu_i} \right\rfloor + 1$ /* Initialize each k_i */
3: **end for**
4: **if** $\sum_{i=1}^{N} k_i > K_{\max}$ **then**
5: throw an exception that the number of processors are not sufficient for the application.
6: **end if**
7: **while** $\sum_{i=1}^{N} k_i \leq K_{\max}$ **do**
8: **for all** $i \leftarrow 1, \ldots, N$ **do**
9: $\delta_i \leftarrow \lambda_i \cdot \left[\mathbb{E}[T_i](k_i) - \mathbb{E}[T_i](k_i + 1)\right]$
10: **end for**
11: /* find the operator with the largest marginal benefit. */
12: $j \leftarrow \arg\max_i \delta_i$
13: $k_j \leftarrow k_j + 1$
14: **end while**
15: **return** $\mathbf{k} = (k_1, k_2, \ldots, k_N)$

Before we come to the theorems stating the effectiveness of our proposed algorithms to solving the optimization problems,

we state the following assumption widely adopted by previous works [36], [42].

Assumption 1: The squared coefficient of variations (scvs) a_i and s_i for all $i = 1, \ldots, N$ are independent of the number of processors k_i assigned to each operator i.

Theorem 1: Under Assumption 1, Algorithm 1 always returns one of the optimal solutions of (6).

Proof: First of all, by applying the marginal analysis technique, $\mathbb{E}[Q_i](M/M/k_i)$ is proved to be a decreasing and convex function in k_i, i.e., the number of processors assigned to operator i [46].

Secondly, based on Assumption 1, we have $\mathbb{E}[T_i](k_i)$ of Equation (5) being a decreasing and convex function in k_i, too. This implies that the marginal benefit for increasing k_i drops monotonously as k_i becomes larger. Formally, for all $k_i' > k_i$, we have

$$\mathbb{E}[T_i](k_i) - \mathbb{E}[T_i](k_i + 1) > \mathbb{E}[T_i](k_i') - \mathbb{E}[T_i](k_i' + 1). \quad (7)$$

Finally, let $\mathbf{k}$ be the output of $AssignProcessors$ shown in Algorithm 1, and $\mathbf{k}^*$ be an optimal assignment that minimizes $\mathbb{E}[T]$. Suppose $\mathbf{k} \neq \mathbf{k}^*$, there must exist two operators x and y satisfying that $k_x^* > k_x$ and $k_y^* < k_y$. According to the facts that (i) $AssignProcessors$ always increments the number of processors for the operators with the highest marginal benefit (lines 8-13); and (ii) the diminishing marginal benefit property in Inequality (7), we derive the following inequality:

$$\lambda_y \Big[\mathbb{E}[T_y](k_y^*) - \mathbb{E}[T_y](k_y^* + 1)\Big] \geq \lambda_x \Big[\mathbb{E}[T_x](k_x^* - 1) - \mathbb{E}[T_x](k_x^*)\Big]$$

In other words, in $\mathbf{k}^*$, taking one processor away from operator x and assigning it to operator y leads to a value of $\mathbb{E}[T]$ that is no worse than before. This can be done repeatedly to gradually change $\mathbf{k}^*$ to $\mathbf{k}$, without increasing $\mathbb{E}[T]$. Hence, $\mathbb{E}[T](\mathbf{k}) \leq \mathbb{E}[T](\mathbf{k}^*)$. Since $\mathbf{k}^*$ is optimal, $\mathbf{k}$ must be optimal as well. □

As a corollary, by setting $a_i = s_i = 1, i = 1, \ldots, N$, Algorithm 1 and Theorem 1 can be directly applied to the basic $M/M/k$ OQNs model [37], i.e., $\mathbb{E}[T_i]$ in Equation (1).

Next, we focus on the question about how to determine the minimum number of processors that are expected to achieve real-time processing, i.e., the expected total sojourn time $\mathbb{E}[T]$ is no larger than a user-defined threshold $T_{\max}$. This can be modeled by the following optimization problem.

$$\min_{\mathbf{k}} \quad \sum_{i=1}^{N} k_i,$$
$$\text{s.t.} \quad \mathbb{E}[T](\mathbf{k}) \leq T_{\max},\ k_i \text{ is interger}, i = 1, 2, \ldots, N. \quad (8)$$

Similar to (6), we can solve (8) using a greedy strategy as shown in the following Algorithm 2. Specifically, we start by initializing each k_i with the minimal resource requirement (lines 2-6). It is necessary to check the feasibility of user-configured $T_{\max}$ by comparing it with the lower bound (lines 7-9). Subsequently, we repeatedly add one processor and obtain the optimal processor assignments by calling Algorithm 1, until $\mathbb{E}[T]$ is no larger than $T_{\max}$ (lines 10-13).

Theorem 2: Under Assumption 1, Algorithm 2 always returns the optimal solution of (8).

Algorithm 2 $MinProcessors$

Input: $T_{\max}$, λ_0, $\{\lambda_i, i = 1, \ldots, N\}$, $\{\mu_i, i = 1, \ldots, N\}$, $\{a_i, i = 1, \ldots, N\}$, $\{s_i, i = 1, \ldots, N\}$.
Output: $\mathbf{k} = (k_1, k_2, \ldots, k_N)$
1: $K \leftarrow 0$, $T_L \leftarrow 0$
2: **for all** $i \leftarrow 1, \ldots, N$ **do**
3: $k_i \leftarrow \left\lfloor \frac{\lambda_i}{\mu_i} \right\rfloor + 1$ /* Initialize each k_i */
4: $K \leftarrow K + k_i$
5: $T_L \leftarrow T_L + \frac{1}{\mu_i}$
6: **end for**
7: **if** $T_L > T_{\max}$ **then**
8: throw an exception that $T_{\max}$ is not achievable.
9: **end if**
10: **while** $\mathbb{E}[T](\mathbf{k}) > T_{\max}$ **do**
11: $K \leftarrow K + 1$
12: $\mathbf{k} \leftarrow AssignProcessors(K, \lambda_0, \{\lambda_i\}, \{\mu_i\}, \{a_i\}, \{s_i\})$
13: **end while**
14: **return** $\mathbf{k} = (k_1, k_2, \ldots, k_N)$

Proof: Let $\mathbf{k} = (k_1, k_2, \ldots, k_N)$ be the output of $MinProcessors$ and $K = \sum_{i=1}^{N} k_i$ be the total number of processors of the assignment. According to Algorithm 2, line 12, we use $\mathbf{k}' = (k_1', k_2', \ldots, k_N')$ to denote the output of execution on $AssignProcessor(K - 1, \lambda_0, \{\lambda_i\}, \{\mu_i\}, \{a_i\}, \{s_i\})$, resulting in (i) $\sum_{i=1}^{N} k_i' = K - 1$ and (ii) $\mathbb{E}[T](\mathbf{k}) \leq T_{\max} < \mathbb{E}[T](\mathbf{k}')$.

We assume the optimal $K^* < K$. Without loss of generality, we consider $K^* = K - 1$ and its corresponding assignment $\mathbf{k}^* = (k_1^*, k_2^*, \ldots, k_N^*)$ satisfying that (i) $\sum_{i=1}^{N} k_i^* = K^* = K - 1$ and (ii) $\mathbb{E}[T](\mathbf{k}^*) \leq T_{\max}$. However, this contradicts with Theorem 1 that $\mathbf{k}' = (k_1', k_2', \ldots, k_N')$ is the optimal solution to Program 6 with inputs $K_{\max} = K - 1$, $\lambda_0, \{\lambda_i\}, \{\mu_i\}, \{a_i\}, \{s_i\}$. □

In practice, the solution of (8) may not provide us with the precise amount of resources needed for meeting the real-time constraints all the time for two reasons. First, the total sojourn time can be different for every input, and $\mathbb{E}[T]$ is merely a measure of its average value. Second, the performance model described in Section III-A outputs only an estimate of $\mathbb{E}[T]$, rather than its exact value. To address this problem, DRS starts with the number of processors suggested by the solution of (8), monitors the actual mean sojourn time $\mathbb{E}[\hat{T}]$, and continuously fine-tune the number of processors based on the measured value of $\mathbb{E}[\hat{T}]$. In next section, we discuss the system design and implementation issues with DRS.

IV. System Design

An overview of the system architecture is presented in Figure 3, which generally consists of two layers, the DRS layer and the CSP (cloud-based streaming processing) layer. Specifically, DRS layer is responsible for performance measurement, resource scaling and resource allocation control, while the CSP layer consists of the primitive streaming processing logic, e.g. running instances of *Storm* [29], *Heron* [30] and *S4* [31], and the cloud-based resource pool service, e.g. Mesos [15], Hadoop YARN [16] and Amazon EC2.

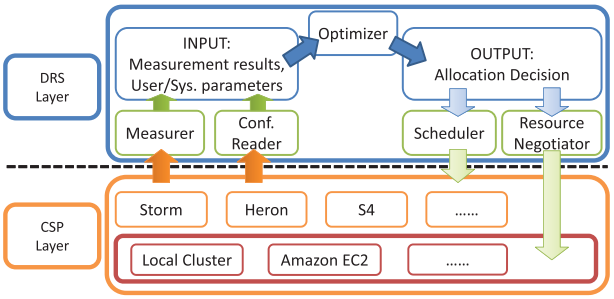


Fig. 3. The architecture overview.

We will focus on the major functionalities of the DRS layer and its connection to the CSP layer, as implementation of the CSP layer mainly reuses existing open source softwares or external commercial services. While the core of the DRS layer is responsible for optimizing the resource scaling based on the performance model developed in the previous section, its implementation is not a straightforward task. Given the heterogeneous underlying infrastructure and the complicated streaming processing applications running on the CSP layer, it is crucial to collect the accurate metrics from the infrastructure, aggregate the statistics, make online decisions and control the resource allocation in an efficient manner.

To seamlessly combine the optimization model and the concrete streaming processing system, we build a number of independent functional modules, which bridge the gap between the physical infrastructure and abstract performance model. As shown in Figure 3, on the input side of the optimizer component, we have *measurer* module and *configuration reader* module, which generate the statistics needed by the optimizer based on the data/control flow from CSP layer. On the output end of the workflow diagram, the *scheduler* module and *resource negotiator* module transform the decisions of the optimizer into executable commands for different streaming processing platforms and resource pools. In the following, we provide the technical details and key features of these modules.

A. Measurer and Configuration Reader Modules

The measurer module is mainly responsible for the measurement on the CSP layer and the pre-processing of the metrics before sending them to the optimizer component. Recall from Algorithms 1 and 2 that for each operator i of a running application, it is essential to collect two local metrics of the operator: the arrival rate, denoted by $\hat{\lambda}_i$, and the service rate, denoted by $\hat{\mu}_i$. In addition, the optimizer component also needs certain global metrics for its optimization algorithm, i.e., metrics related to individual tuples and multiple operators such as the external arrival rate of tuples, denoted by $\hat{\lambda}_0$, and the average total sojourn time, denoted by $\mathbb{E}[\hat{T}]$ of the tuples. Notice that the performance model derived in Section III-B allows the system to estimate $\mathbb{E}[T]$. We, however, believe that direct measurements from the infrastructure help reduce errors between the estimated and actual values.

There exist two major technical challenges to implement the measurer module in the DRS layer. First, the operators and the parallel instances within the operators might be running on different physical machines during online stream processing. Therefore, the measurement must be conducted collaboratively in a distributed environment. Second, it is also important for the measurer module to minimize the overhead and maintain the high availability of the streaming processing service itself.

To tackle the challenges listed above, the measurer module in our system is designed as an independent system operator, mostly invisible to the system user and programmer. To collect the local metrics, a group of optional measurement logics are injected into the executables on each instance of the operators, such that specified local metrics are forcefully collected and kept in the memory of the distributed nodes. A pull-based mechanism is employed to control the data flow from the operators of the topology to the measurement operator. To limit the overhead of distributed metric collection, a bi-layer sampling strategy is applied to the system. Specifically, each instance of the operators records the metric of a tuple every N_m local input tuples, while the centralized measurement operator pulls updates from the other operators every T_m seconds.

To collect the global metric of the external arrive rate of tuples coming into the system, the measurement operator tracks the processing tree of the tuple, using existing techniques, e.g. acknowledgment mechanism. Therefore, the measurement operator receives notifications from the underlying infrastructure about the completion of the processing trees of the tuples, and calculates the global metric based on the notification times.

After collecting the raw measurements, the system still needs to apply pre-processing operations to eliminate the effects of noises, message loss and outliers. These operations include result aggregation at the operator level and results smoothing. Result aggregation is crucial because the metrics defined in our performance model are at the operator level rather than the instance level, which may only represent some proportion and underestimate the operator level metrics. Results smoothing helps reduce the effects of noise and improve stability of the system. We implemented two options of smoothness operations. We denote $d(n)$ as the measurement results of the nth interval collected and aggregated by the controller, and denote $D(n)$ as the smoothed results after the nth interval. The first smoothness option is *α-weighed averaging*, in which we update $D(n) = \alpha D(n-1) + (1-\alpha)d(n)$, with $\alpha \in [0, 1)$ as a tunable parameter controlling the fading rate of the outdated metrics. The second smoothness option is *window-based averaging*, in which we update $D(n) = \frac{1}{w}\sum_{j=n-w+1}^{n} d(j)$, with w as the windows size parameter. We will discuss experimental results based on the window-based option in the next section.

The configuration reader module is designed to be a general interface for managing a data structure containing the configuration parameters provided by either the users or the CSP layer. We list part of the parameters: (i) the type of the optimization problem, i.e., (6) or (8); (ii) the corresponding parameter $K_{\max}$, for Algorithm 1; $T_{\min}$ and $T_{\max}$ for Algorithm 2; and T_{ra} in the optimizer; (iii) for the measurer, e.g., sampling rate N_m, trigger interval T_m and α or w for the smooth processing; (iv) for the scheduler, e.g., the current running allocation and the re-allocation cost.

B. Scheduler and Negotiator Modules

Based on user requirements, the optimizer returns two types of optimization results, which minimize the latency given fixed amount of available resource and minimize the amount of computation resource given the maximal allowable latency. Since these results only indicate the amount of resources assigned to the particular operators, to execute the results, the system still needs to determine a concrete mapping between the available resources and the operators, which is handled by the scheduler module and resource negotiator module.

In the implementation of DRS, we deployed two schedulers corresponding to the two different optimization problems. For optimization (6), the scheduler runs Algorithm 1 on a regular basis, e.g., once per minute. In particular, assuming a user configures the K_{max}, if the number of processors assigned to each operator of the application is different to the optimal solution, the scheduler invokes the re-allocation procedure if the time elapsed since the previous re-allocation event was invoked exceeds a pre-defined threshold, denoted by T_{ra}.

For optimization (8), the scheduler runs Algorithm 2 on a regular basis, i.e., once per minute: (i) when it detects resource shortage, i.e., the measured $\mathbb{E}[\hat{T}]$ is larger than the user configured $T_{\max}$, it triggers *MinProcessors* in Algorithm 2 to adding extra processors; or (ii) when it detects resource overprovisioning, i.e., $\mathbb{E}[\hat{T}]$ is below the user configured $T_{\min}$, which is actually a critical point where users start to shift their attention from tuple complete latency to the total resource consumption, it triggers a similar procedure as the *MinProcessors* to removing existing processors; (iii) when the total amount of resources in use is appropriate, but the assignment of processors to each operator is different to the optimal, it triggers the re-allocation procedure by applying the optimal solution returned by *AssignProcessors* in Algorithm 1. Note that the time between any two consecutive triggered events must be greater than the user defined threshold T_{ra}.

The rationale of controlling the frequency of triggering re-allocation procedures through T_{ra} is because the re-allocation procedure inevitably incurs noticeable costs in terms of total sojourn time, e.g., state migration. How to optimize the re-allocation procedure so as to minimize these costs [47] and when to trigger the re-allocation procedure for balancing between the re-allocation costs and the performance gain are interesting future extensions to the design of the scheduler. Furthermore, in the long run the optimal solutions of (6) and (8) may oscillate among a small set of configurations, due to the periodic behaviors of stream analytics applications, e.g., data arrival rate and characteristics periodically changing with the time of each day. Thus, it is possible to use machine learning based techniques to predict the changing trends, which could further help the scheduler module to achieve better performance. However, these future directions of research are out of the scope of this paper.

The resource negotiator module works at a lower layer than the scheduler module. When users submit new configurations on $K_{\max}$ or $T_{\max}$ either for saving budget or power consumption, or raising the QoS service level, the currently available physical processors, i.e., threads/CPU cores, might be either insufficient or over-provisioned. At these moments, the resource negotiator module will take the responsibility of interacting with the lower level resource managers of cloud platforms, e.g., Mesos [15] and Hadoop Yarn [16]. It is usually required to implement a series of dedicated APIs defined by the deployed managers. For example, one of the most common APIs must be launching (resp. stopping) new (resp. existing) virtual/physical computing nodes. It is sometimes non-trivial to design and implement the resource negotiator module because the interactions with resource mangers such as Mesos [15] may involve a complicated *bargaining* process, e.g., the final agreement on resource provision accepted by both the requester and the provider might be worked out after many rounds of negotiations. In consequence, how to design and implement a smart resource negotiator becomes another very interesting and important research problem which we consider as future works. To simplify the implementation of resource negotiator for DRS, we directly applied an existing open-source resource negotiator, called Storm-on-Yarn (https://github.com/yahoo/storm-yarn/).

V. Empirical Studies

To test and demonstrate the effectiveness of the DRS framework, we have implemented a prototype and integrated it with *Apache Storm* [29]. Source codes and complied executable JAR files are released in GitHub repository at *https://github.com/ADSC-Cloud/resa-drs/releases/*.

As no available API was provided by Apache Storm to measure the queue related metrics such as the arrival rate to each operator, the implementation and integration of DRS with Storm were quite complicated, because we had to modify the source codes of the Storm core to add such measurement logics. To further make our released DRS an independent and pluggable tool, we proposed modifications of the Storm core to the Apache Storm community (available at *https://github.com/apache/storm/pull/716/*), which has been accepted and merged into the master develop stream of the Storm codebase, and is effective since Storm version 1.0.0. The overview of the important concepts and architectural aspects of Storm, and more detailed descriptions of how we implemented the measurer, scheduler and resource negotiator modules of DRS in Storm are provided in Appendix VI.

A. Test Applications

We implemented two real-time stream analytics applications on top of Storm[1]: video logo detection (VLD) and frequent pattern detection (FPD) from different application domains. An application running on Storm is defined by a *topology*, with vertices as user-defined operators containing computation logics and edges as data flows between the operators. There are two types of operators in Storm, *spouts* and *bolts*. Spouts act as data sources, which are connected to external streaming sources; bolts represent all other non-source operators. Each operator contains one or more processors, called *executors*, running on different computing nodes.

1) Logo Detection From a Video Stream: Given a set of query logo images, the logo detection application identifies these images from an input video stream. Although much work

[1]We have developed a couple of real-time stream analytics applications based on Storm and published video demos at: http://www.resa-project.com/.

Fig. 4. The topology of real-time video logo detection application.

Fig. 5. The topology of the stream frequent pattern detection application.

has been done to improve the accuracy and efficiency of VLD, performing it in real time remains a major challenge, due to its high computational complexity.

Figure 4 illustrates the topology of the real-time VLD application, which is a chain of operators containing a spout, a feature extractor, a feature matcher, and an aggregator. The spout extracts frames from the raw video stream. The output rate of frames may vary from time to time according to the generation algorithm and the original video contents. We employ scale-invariant feature transform (SIFT) [48] algorithm to extract features from each frame, which is time-consuming as it involves convolution operators over the 2-dimensional image space. Moreover, the number of resulting SIFT features may vary dramatically on different frames, causing significant variance on the computation overhead over time. The feature matcher measures L_2 distance between its input SIFT features to those pre-generated logo features, and outputs matching pairs with distance lower than a predefined threshold. Finally, the aggregator judges whether a logo appears in a video frame by aggregating all input matching feature pairs, i.e., if the number of matched features in a video frame exceeds a threshold, the logo is considered to appear in the frame.

2) Frequent Pattern Detection over a Microblog Stream: This application maintains the frequent patterns [49] over a sliding window over a microblog stream from Twitter. For each input sentence, we append an additional label "+/-", indicating it is entering/leaving the sliding window. Given a set of input item groups in the sliding window and a threshold, we define a maximal frequent pattern (MFP) to be the itemset satisfying: (i) the number of item groups containing this itemset, called its occurrence count, is above the threshold; and (ii) the occurrence count of any of its superset is below the threshold.

Figure 5 illustrates the corresponding operator topology with two spouts, which generate tuples as itemsets enter/leave the current processing window, respectively. The pattern generator generates candidate patterns, i.e., itemsets. These candidates include an exponential number of possible non-empty combinations of items. Hence, its computation varies, according to the number of items in recent transactions. The detector maintains the state records containing (i) the occurrence counts and (ii) MFP indicator, of all the candidate itemsets. When the MFP indicator of an itemset changes from *False* to *True*, the detector outputs a notification to the reporter, and also to itself through the loop back link. Since (i) each processor in the detector maintains only a portion of the state records; and (ii) a state change can affect the states of other itemsets stored at a different processor, the loop ensures that the state change notifications be sent to all the instances. Finally, the reporter writes the updates of the detection results to an HDFS file.

B. Experiment Setup

The experiments were run on a cluster of 6 Ubuntu Linux machines interconnected by a LAN switch. Each machine is equipped with an Intel quad-core CPU 3.4GHz and 8GB of RAM. Following common configurations of Storm, we allocated one machine to host the Nimbus and Zookeeper Server; the remaining 5 machines host executors for the experimental applications. We also configured each of these 5 machines so that one machine can host at most 5 executors. The main purpose of such configuration is to mitigate the interference caused by other executors running on the same machine, and the resource contention due to the over-allocation of executors on a single machine. As a result, there are 25 executors in total.

For both applications, namely video logo detection (VLD) and frequent pattern detection (FPD), we allocated two executors as spouts, and one executor for DRS. The remaining $25-3=22$ executors are used as bolts, i.e., $K_{\max}=22$. For VLD, the input data are a series of videos clips of the soccer games, and we selected 16 logos as the detection targets. The frame rate simulates a typical Internet video experience, which is uniformly distributed in the interval $[1,25]$ with a mean of 13 frames/second. For FPD, we use a real dataset containing 28,688,584 tweets from 2,168,939 users collected from Oct. 2006 to Nov. 2009. We set the sliding window to 50,000 tweets, and simulated the arrival of tweets to the topology following the Poisson process with an average arrival rate of 320 tweets per second.

C. Experimental Results

We have conducted four groups of experiments aiming to (i) test the quality of the performance model; (ii) evaluate the effectiveness of the resource scaling algorithms 1 and 2; and (iii) investigate the running overhead of DRS. The experimental results show that DRS is capable of detecting resource shortage, overprovisioning, or sub-optimal allocation and making quick and effective re-allocation with neglectable running overheads.

1) Evaluations on the Quality of the Performance Model: In this set of experiments, each of which lasts for 10 minutes, we let $T_{ra}=+\infty$, i.e., the DRS runs passively by monitoring the system performance and recommending new (if better) resource allocations, but its scheduler module never triggers any re-allocation procedure. Figure 6 shows the mean and standard deviation of the total sojourn times under 6 different allocations for each application. The x-axis (x_1:x_2:x_3) denotes an resource configuration (in a partial order of x_1, x_2, x_3), where x_1, x_2, x_3 are the number of executors allocated to the operators *SIFT Feature Extractor*, *Feature matcher*, and *Matching aggregator* in Figure 4, or the *Pattern generator*, *detector*, and *reporter* in Figure 5. The two configurations with "$*$", i.e., (10:11:1) for VLD and (6:13:3) for FPD, are the recommended allocations provided by the DRS.

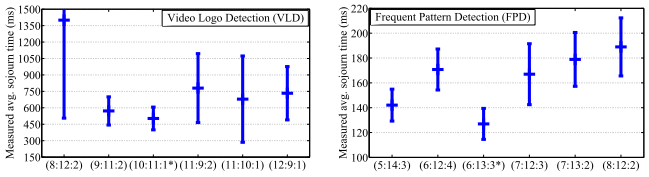


Fig. 6. The mean and standard deviation of the total sojourn time under different resource configurations without re-allocation, where the configurations with "$*$" are the recommended allocations by the passively running DRS.

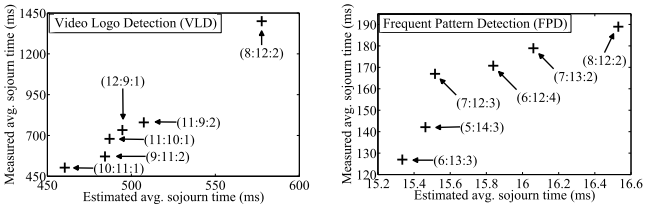


Fig. 7. Comparing average sojourn time estimated by the model and measured in the experiment.

From Figure 6, we make the following observations. The resource configurations (10:11:1) for VLD and (6:13:3) for FPD achieve the best performance according to the measured average sojourn time. This turns out to be consistent with the recommendations provided by DRS, which validates the accuracy and effectiveness of our performance model and resource scaling algorithms. In particular, these two configurations not only obtain the smallest average sojourn times, but also the minimum standard deviation, leading to the smallest performance oscillations. Different configurations, including the five closest ones in terms of the L_1 distance shown in the figures, exhibit considerably worse performance. These results demonstrate that it is not trivial to find the optimal resource allocation especially when the application topology becomes more complicated, e.g., having more than three bolt operators, and hence reveal the importance and usefulness of DRS.

To take a closer look at how DRS provides resource configuration recommendations correctly, we show the relationship between the measured average sojourn times and the estimated average sojourn time derived by our performance model for the six resource allocation configurations for both VLD and FPD in Figure 7.

In Figure 7, the x-axis represents the estimated average sojourn time, the y-axis represents the measured average sojourn time and each point represents one experiment initialized by a particular allocation configuration. As shown in Figure 7, the points of both VLD and FPD applications are positively correlated, which evidences that the performance model is capable of suggesting the best resource allocation configuration. Moreover, the performance model outputs accurate estimates for VLD; though, with some slight underestimation comparing to the measured values. It is worth noting that the estimates are quite accurate even though the underlying conditions for the Jackson network theory are not satisfied, i.e., the frame rate is uniformly distributed rather than exponentially distributed as required by the Jackson OQNs. Furthermore, the operator input queues do not follow strict FIFO rule; instead, tuples are hashed to processors and different operators are also run in parallel, leading to

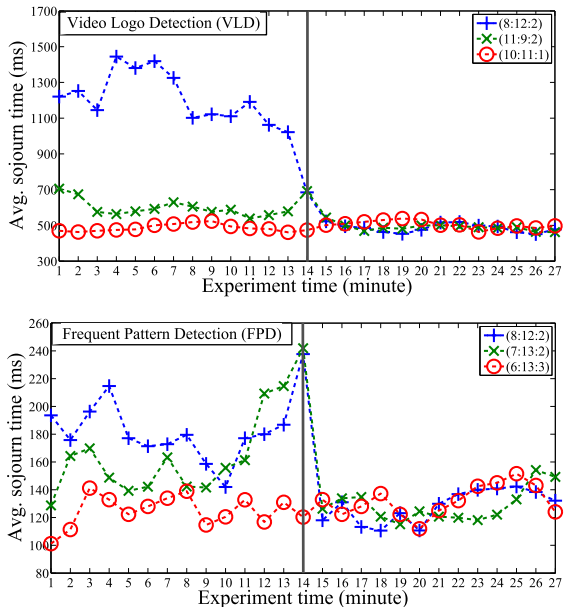


Fig. 8. The average sojourn times of three initial allocations for both VLD and FPD applications, where $K_{\max} = 22$ and $T_{ra} = 14$ minutes.

pipelining. For FPD, the estimated sojourn times show larger deviations than that of the measured ones. This is mainly because the model does not consider network transmission cost, which takes a dominant portion of the total query latency in this particular application. In other words, the FPD is de facto the type of data intensive rather than the computation intensive application that we focus on. Nevertheless, our model still correctly indicates the relative performance order of the resource allocation configurations. Since the estimates are strongly correlated with the true values, polynomial regressions can be used to make accurate predictions on the true latency values given the estimated ones.

2) Evaluations on the Optimization (6) and Algorithm 1: In this set of experiments, each of which lasts for 27 minutes, we let $T_{ra} = 14$ minutes, i.e., throughout the whole experiment, DRS invokes the re-allocation procedure at most once when it detects sub-optimal resource allocations given the maximal number of available executors $K_{\max} = 22$. In this way, we are able to have a clear view of the performance in terms of the average sojourn time across the re-allocation events.

As shown in Figure 8, x-axis represents the experiment time and y-axis represents the measured sojourn time averaged over each minute. For both VLD and FPD applications, three experiments with different initial allocation configurations, each of which is represented by a curve, were conducted. In particular, the curve marked by "o" represents the experiment initialized with the optimal allocation while the remaining two curves, marked by "+" and "x" represent the experiment initialized with sub-optimal allocations. We observe from Figure 8 that for both applications, the re-allocation procedure was triggered at the 14th minute, i.e., the earliest possible time, for the two experiments started with sub-optimal allocations, quickly responding to the less promising resource scheduling plan. After the re-allocation, all three experiments were scheduled with the unique optimal solution. This observation is supported

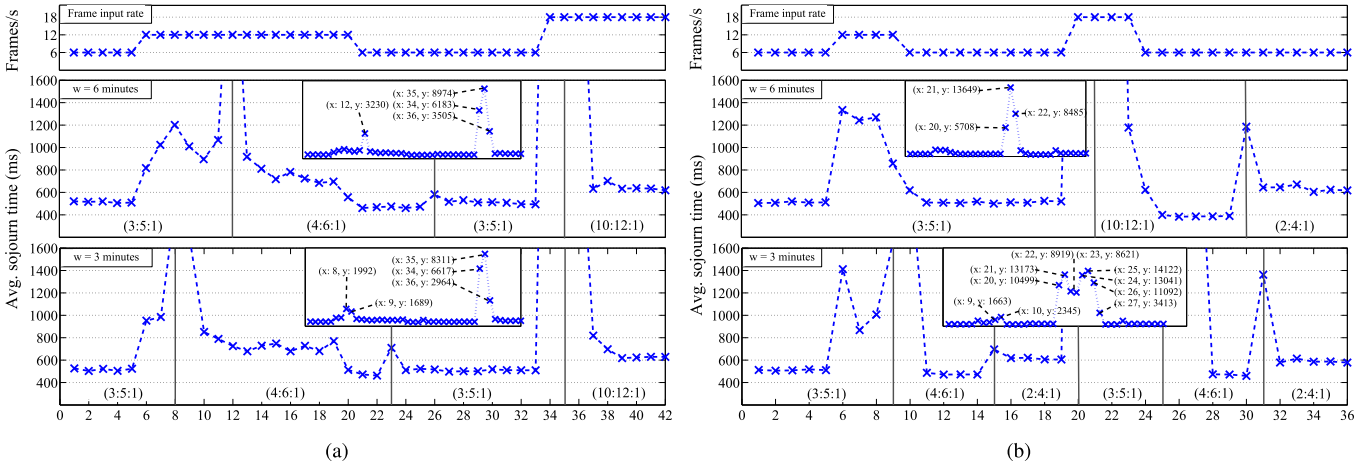


Fig. 9. Average sojourn time and resource re-allocation events of the VLD application under long-term (left) and short-term (right) changes in video input rate, where $T_{ra} = w$ minutes, $T_{\max} = 1000$ ms and $T_{\min} = 450$ ms. (a) Long-term change in video input rate. (b) Short-term change in video input rate.

by two facts. First, after the 14th minute, all the three curves have similar average sojourn times and performance trends. Especially for the two curves with re-allocation triggered, they show a clear decrease in the average sojourn time. Second, the plans kept in the log files further verify this observation.

It is worth noting that the built-in re-allocation procedure provided by Storm leads to serious performance degradation, i.e, the average sojourn time increases dramatically and lasts for 1-2 minutes. This is not affordable for real-time applications; therefore, we have developed our improved version [50] of re-allocation procedure for these experiments. As shown in the configurations (8:12:2) and (11:9:2) of VLD and (8:12:2) and (7:13:2) of FPD in Figure 8, our improved version of re-allocation procedure led to remarkably low cost, i.e., a neglectable increment in the average sojourn time within the 14th minute only. Besides, it only takes a few seconds, comparing to the 1-2 minutes taken by Storm's default version.

3) Evaluations on the Optimization (8) and Algorithm 2: We investigate how DRS adjusts resources when it detects resource shortage/overprovisioning. Four experiments on the VLD application are conducted under dynamic video arrival rates, i.e., two for long-term and two for short-term changes, and the experimental results are presented in Figure 9.

The top sub-figures of Figure 9 illustrate two changing patterns of external video input rates. Three discrete values, i.e., 6, 12 and 18 fps are used for simulating the 200% and 300% increment/decrement in the processing workloads. We set the threshold $T_{ra} = w$, where the middle and bottom sub-figures plot the instantaneous tuple sojourn times averaged in each minute for $w = 6$ and $w = 3$, respectively. In addition, each solid vertical line represents a DRS triggered event of resource re-allocation. For example, the left most vertical line in the middle subplot of Figure 9(a) shows that DRS has triggered the re-allocation procedure at the 12th minute. Consequently, the number of executors assigned to the *SIFT Feature Extractor*, *Feature matcher* and *Matching aggregator* of the VLD application, denoted by the triple (x_1:x_2:x_3), changes from (3:5:1) to (4:6:1), indicating that a resource shortage situation is detected by DRS and two extra executors are added during the re-allocation procedure. Based on the results shown in Figure 9, we make the following observations:

(i) DRS is capable of detecting resource shortage or over-provisioning and making effective resource adjustment. For example, the three re-allocation events (vertical lines) in the middle subplot in Figure 9(a) exactly correspond to the three changes in the video input rate at the 6th, 21st and 34th minute, respectively. In particular, DRS detects resource shortage for the 1st (a 200% increment) and 3rd (a 300% increment) changes of input rate, and triggers re-allocation for adding extra executors, while for the 2nd (a 200% decrement) change in the input rate, resource overprovisioning is detected and existing executors are, in consequence, removed by DRS.

(ii) By comparing the two curves in the middle ($w = 6$) and bottom ($w = 3$) subplots in Figure 9(a), we observe that a small value in w, i.e., the window size of smoothing measurement results, helps DRS to take quicker responses to the dynamic changes in the video input rate. For example, the time between the change in the input rate and the time when DRS triggers re-allocation equals 3 minutes for both the 1st and 2nd events under $w = 3$, whereas the corresponding times are 7 and 6 minutes, respectively, under $w = 6$. However, a large value in w, nonetheless, reduces the probability that DRS triggers an unnecessary re-allocation, and therefore, reduces re-allocation costs. An example can be found in Figure 9(b), where from the 6th to 9th minute, the video frames arrive at 12 fps (200% of the base rate). In the middle subplot, due to the large w, the measured sojourn time, after smoothed by a 6-minute window, stays constantly below the threshold $T_{\max}$; and therefore, no re-allocation is triggered. The bottom subplot of Figure 9(b), on the contrary, shows that under $w = 3$, two re-allocation events at the 9th and 15th minute are triggered. In summary, the window-size w provides a tradeoff between quick response and system stability. In real implementation, a sufficiently large w is often needed to ensure the system stability and save costs by lowering the probability of triggering unnecessary re-allocations.

(iii) Similar to the observations made in Figure 8, the costs incurred by our re-allocation procedure in all the experiments shown in Figure 9 are much lower than that of Storm's default.

TABLE II
COMPUTATION OVERHEADS IN MILLISECONDS UNDER DIFFERENT $K_{\max}$

$K_{\max}$	12	24	48	96	192
Scaling	0.083	0.158	0.323	0.665	1.250
Measurement	0.100	0.100	0.100	0.100	0.100

However, the overhead of adding executors is considerable higher than that of removing existing executors. For example, the middle subplot of Figure 9(a) shows overheads of more than 3000 (ms) increase in the average sojourn time in both the 12th and 35th minute, triggered by DRS under resource shortage; whereas, an increase of about 600 (ms) incurred during the 2nd re-allocation event at the 26th minute under overprovisioned resources. This is mainly due to the different actions taken during the re-allocation procedure. In the resource shortage case, new machines are initialized and added to the running topology, and during this period the whole running application has to be suspended; whereas in the resource overprovision case, the termination and removal of the working machines can be carried out simultaneously without affecting the running application, hence no substantial impacts on the average sojourn time.

4) The Running Overhead of the DRS: To evaluate the computation overhead of the overall DRS layer, we report the CPU time spent by the whole DRS module, including the processing on measurement results and calculating the optimal allocation. In this experiment, we only test on the VLD topology composed of three bolt operators with fixed parameters λ_0, λ_i and $\mu_i, i = 1, 2, 3$. We try different $K_{\max}$, i.e., total number of executors for all operators. For each value of $K_{\max}$, we run the procedure 100,000 times and report the average running time of the whole DRS layer. The results are listed in Table II, with *Scaling* as the allocation computation and *Measurement* as the metric processing computation.

Generally speaking, the computation done by DRS is almost neglectable, with overhead less than milliseconds in most of the cases. Moreover, the results are consistent with our intuition that the computation consumption is linear in $K_{\max}$, as analyzed in Algorithm 1. The time consumed on processing the measurement results is irrelevant to $K_{\max}$. In fact, it is affected by the total number of tasks of the topology, as we will discuss in Appendix VI that this number is kept immutable when the topology is running.

D. DRS With the Extended Model Based on $GI/G/k$ OQNs

According to our log files, in most cases, the values of $\mathbb{E}[T_i]$s and $\mathbb{E}[T]$ predicted by the extended model based on $GI/G/k$ OQNs are close to those predicted by the basic model based on $M/M/k$ OQNs. In consequence, the suggested optimal allocations by both models are exactly the same too. In terms of the accuracy of estimation on the values of $\mathbb{E}[T_i]$s and $\mathbb{E}[T]$, by definition, the extended model (based on $GI/G/k$) shall outperform the basic model (based on $M/M/k$) when the standard deviations of inter-arrival times and tuple processing times deviate from the corresponding means noticeably, since when they are equal or close, we have $a_i = s_i = 1$ and the extended model falls back into the basic one.

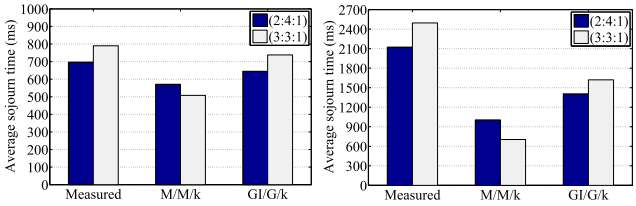


Fig. 10. Illustration of special cases where the extended model based on $GI/G/k$ OQNs outperforms the basic model based on $M/M/k$ OQNs. There are two additional experiments on VLD application with $T_{ra} = +\infty$. (a) $K_{\max} = 7$; $\lambda_0 = 5$. (b) $K_{\max} = 7$; $\lambda_0 = 6$

Plenty of prior studies on optimizing manufacturing operations showed that in most realistic cases the standard deviations of inter-arrival times and job process times are much smaller than the corresponding means [36], [42], [51]. This is also true for many computation intensive stream data analytics applications. According to Equation (5), when $a_i < 1$ and/or $s_i < 1$, the values of $\mathbb{E}[T_i]$s and $\mathbb{E}[T]$ calculated by the extended model are smaller than by the basic model, because "less" stochastic arrival and service processes, i.e., smaller a_i and s_i, should result in smaller sojourn times. As we mentioned in Section III-A and evidenced by the experimental results in Figure 7, the basic model tends to underestimate the true $\mathbb{E}[T]$ because the network overhead is not included. Consequently, the extended model produces even poorer (smaller) estimates on $\mathbb{E}[T]$ for these cases as expected.

Nevertheless, there exist certain cases where the extended model based on $GI/G/k$ OQNs outperforms the basic model. We take the VLD application illustrated in Figure 4 as an example. The tuple arrival and service processes of its two computation intensive operators, i.e., *SIFT Feature Extractor* and *Feature Matcher*, are more "stochastic" than exponential distribution. Because there are bulk arrivals of features output by *SIFT Feature Extractor* and fed into *Feature Matcher*, and the processing times for both feature extraction and feature matching largely depend on the number of features contained in each video frame, or equivalently the video contents, which may change rapidly in contexts such as a soccer game.

In the following experiments, we fix the maximum number of available processes $K_{\max} = 7$ and vary the workload in terms of the input frame rate λ_0. In particular, the inter-arrival times of the frames in seconds are uniformly distributed in [1/8, 1/2] and [1/9, 1/3] so that λ_0 equals 5 and 6 frames per second for Figure 10(a) and Figure 10(b), respectively.

As shown in Figure 10, the bars labeled "Measured" represent the real tuple average sojourn times measured during the experiment; the bars labeled "$M/M/k$" represent $\mathbb{E}[T]$ calculated by the basic $M/M/k$ OQNs model in Equation (1); and the bars labeled "$GI/G/k$" represent $\mathbb{E}[T]$ calculated by the extended $GI/G/k$ OQNs model in Equation (5). In each subplot, we show the best two configurations in terms of the average sojourn times,[2] i.e., (2:4:1) vs. (3:3:1), where (x_1:x_2:x_3) denotes the number of executors assigned to the *SIFT Feature Extractor*, *Feature matcher* and

[2]To simplify the presentation, we only show the results of the best and second-best configurations with the results of other configurations omitted.

Matching aggregator, respectively. We make the following two observations:

(i) The $M/M/k$ model produces a higher degree of underestimation on $\mathbb{E}[T]$ with respect to the measured average sojourn time when the input rate λ_0 increases from Figure 10(a) to Figure 10(b). This is because heavier workload leads to higher proportion of queuing time in the total sojourn time. The extended $GI/G/k$ model provides more accurate estimations due to the compensation term $(a_i+s_i)/2$ on the queuing delay in Equation (5). This observation is also consistent with the conclusions made in [42].

(ii) As shown in both sub-figures, the $M/M/k$ model would prefer the configuration (3:3:1), because it results in lower $\mathbb{E}[T]$ than that of (2:4:1); in contrast, the extended $GI/G/k$ model suggests (2:4:1), which is the de facto optimal configuration according to the measured average sojourn time.

Based on the above observations, we summarize that DRS equipped with the $M/M/k$ basic model is applicable of inferring optimal configurations for most stream data analytics applications. Only when the tuple inter-arrival and processing times have larger variances than those of the exponential distributions, and the operators have heavy workloads, i.e., λ_0 is not small, do we consider the extended $GI/G/k$ model.

VI. Conclusion

This paper proposes DRS, a dynamic resource scaling framework for real-time cloud-based stream data analytics systems. DRS overcomes several fundamental challenges, including the estimation of the required resources necessary for satisfying real-time requirements, effective and efficient resource provisioning and scheduling, and efficient implementation of such a scheduler in a cloud-based stream data analytics system. The performance model of DRS is based on rigorous queuing theory, and it demonstrates robust performance even when the underlying conditions of the theory are not fully satisfied. In addition, we have integrated DRS into a popular system Apache Storm, and evaluated it by conducting extensive experiments based on real applications and datasets.

Regarding future work, we plan to investigate efficient strategies for migrating the system from the current resource configuration to the new one recommended by DRS. This step should minimize additional overhead and result latency during migration, as well as the migration duration, (e.g., [47]).

Appendix
Overview of Storm and DRS Implementations

An application running on Storm is defined by a *topology*, with vertices as user-defined operators (containing computation logics) and edges as indicators of data flows between operators. There are two types of operators in Storm, *spouts* and *bolts*. A spout acts as a data source, which connects to external streaming sources. Bolts include all other (i.e., non-source) operators. Each operator contains one or more processors, called *executors*, running on different servers.

Storm supports dynamically "re-scaling" an operator (spout or a bolt), which changes its number of executors. This is implemented by decoupling the routing logics from the computation logics. The routing logics remain the same even when new executors are added. Storm's implementation is based on a partitioning scheme on each operator (spout or bolt), in which each partition is called a *task*. When an operator scales out (respectively in), the number of executors of the operators increases (decreases), with the tasks reassigned to the executors. In particular, there are different partitioning rules supported by Storm, e.g., shuffle, field and direct grouping. We refer the reader to [29] for details of the partitioning rules.

Given the architecture of Storm system, resource allocation/re-allocation can be controlled by assigning different numbers of executors to operators. Storm also provides an internal mechanism for migrating to a new resource configuration, called *re-balancing*. Simply put, the re-balancing mechanism suspends the entire system, e.g., by shutting down all the Java Virtual Machines (JVMs), modifies the executor to operator mappings and routings, and finally resumes the system. Hence, the response time becomes very high during re-balancing. Therefore, in the real implementation of DRS and the experiment, we developed and used our own version, which involves coding at the Storm core layer in Clojure, of the re-balancing mechanism, with significant improvements over Storm's default version. Discussions on how to migrate to a new resource configuration without such costly system-wide suspensions is out of the scope of this paper. The most essential improvement we have made is to re-use the JVMs [50]. Finally, Storm provides a scheduler interface that enables customized executor assignment strategy, and allows users to specify the operation frequency of the scheduler.

A. Measurer

We implemented two new system operators (not visible to users) into the Storm system, called *MeasurableSpout* and *MeasurableBolt*. They wrap a normal bolt/spout, and add measurement logics. The measurement for bolts mainly records the elapsed time volumes the execution function spends on each of the incoming tuples. These measured results are collected periodically by the "DRSMetricCollecor" module, which is implemented using the Measurement APIs provided by Storm. To measure the queue related metrics, e.g., the average tuple arrival rate to each operator i, is more complicated, because there is no available API we can make use of. Therefore, we had to modify the source code of the Storm core to add the measurement logics. Note we have made a pull request (available at *https://github.com/apache/storm/pull/716/*) for the modifications. It has been accepted and merged into the master develop stream of the Storm codebase, and is effective since Storm version 1.0.0.

B. Configuration Reader

Similarly, the configuration reader reuses the APIs of the Storm system, which shares the configuration in Zookeeper.

C. Scheduler

Since Storm provides the scheduling APIs, we overrode the default scheduler implementation by ours which calls our version of the re-balancing function only when the two

conditions are satisfied (i) the current allocation is sub-optimal; (ii) the elapsed time since the last call on re-allocation exceeds a pre-defined threshold.

D. Negotiator

The negotiator is at a lower level than the resource manager of the Storm. It is in charge of starting/shutting down extra/existing physical resources (e.g., physical machines or virtual machines). Our negotiator module is based on the APIs of YARN, on top of a Hadoop cluster.

REFERENCES

[1] W. Lam *et al.*, "Muppet: MapReduce-style processing of fast data," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1814–1825, Aug. 2012.

[2] Z. Qian *et al.*, "TimeStream: Reliable stream computation in the cloud," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, Apr. 2013, pp. 1–14.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *ACM Commun.*, vol. 51, no. 1, pp. 107–113, 2008.

[4] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu, "Distributed data management using MapReduce," *ACM Comput. Surv.*, vol. 46, no. 3, Jan. 2014, Art. no. 31.

[5] *Hadoop: Fair Scheduler*. Accessed on Apr. 8, 2013. [Online]. Available: http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html

[6] *Hadoop: Capacity Scheduler*. Accessed on Apr. 8, 2013. [Online]. Available: http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html

[7] M. Zaharia *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th Eur. Conf. Comput. Syst.*, Apr. 2010, pp. 265–278.

[8] R. Pedarsani, J. Walrand, and Y. Zhong, "Scheduling tasks with precedence constraints on multiple servers," in *Proc. 52nd Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Sep. 2014, pp. 1196–1203.

[9] S. Melnik *et al.*, "Dremel: Interactive analysis of Web-scale datasets," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 330–339, Sep. 2010.

[10] M. Traverso. *Presto: Interacting With Petabytes of Data at Facebook*. 2013. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920/

[11] S. Zhang, Y. Yang, W. Fan, L. Lan, and M. Yuan, "OceanRT: Real-time analytics over large temporal data," in *Proc. ACM SIGMOD*, Jun. 2014, pp. 1099–1102.

[12] S. Zhang, Y. Yang, W. Fan, and M. Winslett, "Design and implementation of a real-time interactive analytics system for large spatio-temporal data," *Proc. VLDB Endowment*, vol. 7, no. 13, pp. 1754–1759, Aug. 2014.

[13] Z. Zhang, H. Shu, Z. Chong, H. Lu, and Y. Yang, "C-Cube: Elastic continuous clustering in the cloud," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 577–588.

[14] A. Thusoo *et al.*, "Hive—A petabyte scale data warehouse using Hadoop," in *Proc. IEEE 26th Int. Conf. Data Eng. (ICDE)*, Mar. 2010, pp. 996–1005.

[15] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX NSDI*, 2011, pp. 1–14.

[16] V. K. Vavilapalli *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, Art. no. 5.

[17] Z. Zhang, R. T. B. Ma, J. Ding, and Y. Yang, "ABACUS: An auction-based approach to cloud service differentiation," in *Proc. IEEE Int. Conf. Cloud Eng.*, Mar. 2013, pp. 292–301.

[18] A. Arasu *et al.*, "STREAM: The stanford stream data manager (demonstration description)," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2003, p. 665.

[19] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *Int. J. Very Large Data Bases*, vol. 15, no. 2, pp. 121–142, Jun. 2006.

[20] S. Babu, K. Munagalat, J. Widom, and R. Motwani, "Adaptive caching for continuous queries," in *Proc. 21st Int. Conf. Data Eng. (ICDE)*, Apr. 2005, pp. 118–129.

[21] D. J. Abadi *et al.*, "Aurora: A new model and architecture for data stream management," *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.

[22] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A stream database for network applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2003, pp. 647–651.

[23] S. Chandrasekaran *et al.*, "TelegraphCQ: Continuous dataflow processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2003, p. 668.

[24] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu, "Processing high data rate streams in system S," *J. Parallel Distrib. Comput.*, vol. 71, no. 2, pp. 145–156, Feb. 2011.

[25] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas, "Operator scheduling in data stream systems," *Int. J. Very Large Data Bases*, vol. 13, no. 4, pp. 333–353, Dec. 2004.

[26] L. Wang *et al.*, "Elastic pipelining in an in-memory database cluster," in *Proc. ACM SIGMOD*, Jun./Jul. 2016, pp. 1279–1294.

[27] D. J. Abadi *et al.*, "The design of the borealis stream processing engine," in *Proc. Conf. Innov. Data Syst. Res.*, vol. 5. 2005, pp. 277–289.

[28] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *Proc. USENIX Conf. Hot Topics Cloud Comput.*, 2012, p. 10.

[29] A. Toshniwal *et al.*, "Storm@twitter," in *Proc. ACM SIGMOD*, Jun. 2014, pp. 147–156.

[30] S. Kulkarni *et al.*, "Twitter heron: Stream processing at scale," in *Proc. ACM SIGMOD*, Jun. 2015, pp. 239–250.

[31] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proc. IEEE Int. Conf. Data Mining Workshops (ICDMW)*, Dec. 2010, pp. 170–177.

[32] S. A. Noghabi *et al.*, "Samza: Stateful scalable stream processing at LinkedIn," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[33] R. L. Collins and L. P. Carloni, "Flexible filters: Load balancing through backpressure for stream programs," in *Proc. 7th ACM Int. Conf. Embedded Softw.*, Oct. 2009, pp. 205–214.

[34] Y. Xing, S. Zdonik, and J.-H. Hwang, "Dynamic load distribution in the borealis stream processor," in *Proc. 21st Int. Conf. Data Eng. (ICDE)*, Apr. 2005, pp. 791–802.

[35] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *VLDB J.*, vol. 23, no. 4, pp. 517–539, Aug. 2014.

[36] G. R. Bitran and R. Morabito, "State-of-the-art survey: Open queueing networks: Optimization and performance evaluation models for discrete manufacturing systems," *Prod. Oper. Manage.*, vol. 5, no. 2, pp. 163–193, Jun. 1996.

[37] T. Z. J. Fu *et al.*, "DRS: Dynamic resource scheduling for real-time analytics over fast streams," in *Proc. IEEE 35th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2015, pp. 411–420.

[38] H. C. Tijms, *Stochastic Modelling and Analysis: A Computational Approach*. Hoboken, NJ, USA: Wiley, 1986.

[39] J. R. Jackson, "Jobshop-like queueing systems," *Manage. Sci.*, vol. 10, no. 1, pp. 131–142, Oct. 1963.

[40] R. Nelson, D. Towsley, and A. N. Tantawi, "Performance analysis of parallel processing systems," *IEEE Trans. Softw. Eng.*, vol. 14, no. 4, pp. 532–540, Apr. 1988.

[41] W. Whitt, "Approximations for the GI/G/m queue," *Prod. Oper. Manage.*, vol. 2, no. 2, pp. 114–161, Jun. 1993.

[42] M. van Vliet and A. H. G. R. Kan, "Machine allocation algorithms for job shop manufacturing," *J. Intell. Manuf.*, vol. 2, no. 2, pp. 83–94, Apr. 1991.

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

[44] M. A. Soliman *et al.*, "Orca: A modular query optimizer architecture for big data," in *Proc. ACM SIGMOD*, Jun. 2014, pp. 337–348.

[45] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, CPU-efficient key-value store," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 103–114.

[46] M. E. Dyer and L. G. Proll, "On the validity of marginal analysis for allocating servers in M/M/c queues," *Manage. Sci.*, vol. 23, no. 9, pp. 1019–1022, May 1977.

[47] J. Ding *et al.*, "Optimal operator state migration for elastic data stream processing," *CoRR*, vol. abs/1501.03619, 2015. [Online]. Available: http://arxiv.org/abs/1501.03619

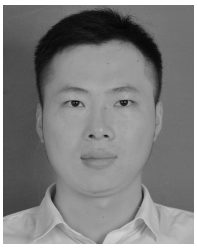
[48] T. Lindeberg, "Scale invariant feature transform," *Scholarpedia*, vol. 7, no. 5, p. 10491, 2012.

[49] D. Burdick, M. Calimlim, and J. Gehrke, "MAFIA: A maximal frequent itemset algorithm for transactional databases," in *Proc. 17th Int. Conf. Data Eng. (ICDE)*, Apr. 2001, pp. 443–452.

[50] M. Yang and R. T. B. Ma, "Smooth task migration in apache storm," in *Proc. ACM SIGMOD*, Jun. 2015, pp. 2067–2068.

[51] O. Boxma, A. H. G. R. Kan, and M. Van Vliet, "Machine allocation problems in manufacturing networks," *Eur. J. Oper. Res.*, vol. 45, no. 1, pp. 47–54, Mar. 1990.

Tom Z. J. Fu received the B.Eng. degree from Shanghai Jiao Tong University in 2006, and the M.Phil. and Ph.D. degrees from The Chinese University of Hong Kong in 2008 and 2013, respectively, all in information engineering. He is currently a Research Scientist with the Advanced Digital Sciences Center, Illinois at Singapore Pte Ltd. His research interests include cloud computing and real-time stream analytics, software defined network (SDN), P2P streaming systems, Internet measurement, and academic social network.



Jianbing Ding received the B.E. degree from the Nanjing University of Post and Telecommunications in 2008, the master's degree from the University of Science and Technology of China in 2011, and the Ph.D. degree from the School of Data and Computer Science, Sun Yat-sen University, in 2016. His research interests are mainly in cloud computing and large-scale information retrieval systems.

Richard T. B. Ma (SM'16) received the B.Sc. degree (Hons.) in computer science and the M.Phil. degree in computer science and engineering from The Chinese University of Hong Kong, in 2002 and 2004, respectively, and the Ph.D. degree in electrical engineering from Columbia University in 2010. During his Ph.D. study, he was a Research Intern with the IBM T. J. Watson Research Center, NY, USA, and Telefonica Research, Barcelona, Spain. He is currently an Assistant Professor with the School of Computing, National University of Singapore, and a Research Scientist with the Advanced Digital Science Center, University of Illinois at Urbana–Champaign. His current research interests include economics and evolution of the Internet, performance evaluation, big data analytics, and cloud computing. He was a co-recipient of the Best Paper Award in the IEEE IC2E 2013, the IEEE ICNP 2014, and the IEEE Workshop on Smart Data Pricing (SDP) 2015.

Marianne Winslett (A'05) received the Ph.D. degree in computer science from Stanford University in 1987. She was with Bell Labs. She has been a Professor with the Department of Computer Science, University of Illinois at Urbana-Champaign, since 1987, as an Assistant, Associate, Full, Adjunct, and currently a Research Professor. Her research interests lie in information security and in the management of scientific data. She is a fellow of the ACM. She received a Presidential Young Investigator Award from the U.S. National Science Foundation in 1989. She is currently on the editorial boards of the *ACM Transactions on the Web* and the *ACM Transactions on Information and System Security*. She served as the SIGMOD Vice-Chair from 2001 to 2005. She was on the editorial boards of the *ACM Transactions on Database Systems* from 1994 to 2004 and the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING from 1994 to 1998.

Yin (David) Yang is currently an Assistant Professor with the College of Science and Engineering, Hamad Bin Khalifa University. He has authored extensively in top venues on differentially private data publication and analysis, and on query authentication in outsourced databases. His main research interests include cloud computing, database security and privacy, and query optimization. He is actively involved in cloud-based big data analytics, with a focus on fast streaming data.

Zhenjie Zhang received the B.S. degree from the Department of Computer Science and Engineering, Fudan University, in 2004, and the Ph.D. degree in computer science from the School of Computing, National University of Singapore, in 2010. He is currently a Senior Research Scientist with Advanced Digital Sciences Center. He has authored over 20 research papers in database and data mining venues, including SIGMOD, VLDB, and ICML. His research interests cover a variety of different topics, including clustering analysis, nonmetric indexing, game theory, and data privacy. He has served as a Program Committee Member of WWW 2010, VLDB 2010, KDD 2010, and APWeb 2011. He received the Presidents Graduate Fellowship of Singapore in 2007.