Live Video Analytics at Scale with Approximation and Delay-Tolerance

Haoyu Zhang[⋆†], Ganesh Ananthanarayanan[⋆], Peter Bodik[⋆], Matthai Philipose[⋆]
Paramvir Bahl[⋆], Michael J. Freedman[†]

[⋆]*Microsoft* [†]*Princeton University*

Abstract

Video cameras are pervasively deployed for security and smart city scenarios, with millions of them in large cities worldwide. Achieving the potential of these cameras requires efficiently analyzing the *live* videos *in real-time*. We describe VideoStorm, a video analytics system that processes thousands of video analytics *queries* on live video streams over large clusters. Given the high costs of vision processing, resource management is crucial. We consider two key characteristics of video analytics: *resource-quality tradeoff with multi-dimensional configurations*, and *variety in quality and lag goals*. VideoStorm's offline profiler generates query resource-quality profile, while its online scheduler allocates resources to queries to *maximize performance* on quality and lag, in contrast to the commonly used fair sharing of resources in clusters. Deployment on an Azure cluster of 101 machines shows improvement by as much as 80% in quality of real-world queries and 7× better lag, processing video from operational traffic cameras.

1 Introduction

Video cameras are pervasive; major cities worldwide like New York City, London, and Beijing have millions of cameras deployed [8, 12]. Cameras are installed in buildings for surveillance and business intelligence, while those deployed on streets are for traffic control and crime prevention. Key to achieving the potential of these cameras is effectively analyzing the *live* video streams.

Organizations that deploy these cameras—cities or police departments—operate large clusters to analyze the video streams [5, 9]. Sufficient bandwidth is provisioned (fiber drops or cellular) between the cameras and the cluster to ingest video streams. Some analytics need to run for long periods (e.g., counting cars to control traffic light durations) while others for short bursts of time (e.g., reading the license plates for AMBER Alerts, which are raised in U.S. cities to identify child abductors [1]).

Video analytics can have *very high resource demands*. Tracking objects in video is a core primitive for many scenarios, but the best tracker [69] in the VOT Challenge 2015 [59] processes only 1 frame per second on an 8-core machine. Some of the most accurate Deep Neural Networks for object recognition, another core primitive, require 30GFlops to process a single frame [75]. Due to the high processing costs and high data-rates of video streams, resource management of *video analytics queries* is crucial. We highlight two properties of video analytics queries relevant to resource management.

Resource-quality trade-off with multi-dimensional configurations. Vision algorithms typically contain various parameters, or *knobs*. Examples of knobs are video resolution, frame rate, and internal algorithmic parameters, such as the size of the sliding window to search for objects in object detectors. A combination of the knob values is a query *configuration*. The configuration space grows exponentially with the number of knobs. *Resource demand* can be reduced by changing configurations (e.g., changing the resolution and sliding window size) but they typically also lower the output *quality*.

Variety in quality and lag goals. While many queries require producing results in real-time, others can tolerate *lag* of even many minutes. This allows for temporarily reallocating some resources from the lag-tolerant queries during interim shortage of resources. Such shortage happens due to a burst of new video queries or "spikes" in resource usage of existing queries (for example, due to an increase in number of cars to track on the road).

Indeed, video analytics queries have a wide variety of quality and lag goals. A query counting cars to control the traffic lights can work with moderate quality (approximate car counts) but will need them with low lag. License plate readers at toll routes [16, 17], on the other hand, require high quality (accuracy) but can tolerate lag of even many minutes because the billing can be delayed. However, license plate readers when used for AMBER Alerts require high quality results *without* lag.

Scheduling large number of streaming video queries with diverse quality and lag goals, each with many configurations, is computationally complex. Production systems for stream processing like Storm [4], StreamScope [62], Flink [2], Trill [36] and Spark Streaming [89] allocate resources among multiple queries only based on *resource fairness* [7, 10, 27, 43, 51] common to cluster managers like Yarn [3] and Mesos [49]. While simple, being agnostic to query quality and lag makes fair sharing far from ideal for video stream analytics.

We present VideoStorm, a video analytics system that scales to processing thousands of *live* video streams over large clusters. Users submit video analytics queries containing many *transforms* that perform vision signal processing on the frames of the incoming video. At its core, VideoStorm contains a scheduler that efficiently generates the query's *resource-quality profile* for its different knob configurations, and then *jointly maximizes the quality and minimizes the lag* of streaming video queries. In doing so, it uses the generated profiles, and lag and quality goals. It allocates resources to each query and picks its configuration (knob values) based on the allocation.

Challenges and Solution. The major technical challenges for designing VideoStorm can be summarized as follows: *(i)* There are no analytical models for resource demand and quality for a query configuration, and the large number of configurations makes it expensive to even estimate the resource-quality profile. *(ii)* Expressing quality and lag goals of *individual queries* and *across all queries* in a cluster is non-trivial. *(iii)* Deciding allocations and configurations is a computationally hard problem *exponential* in the number of queries and knobs.

To deal with the multitude of knobs in video queries, we split our solution into *offline* (or profiling) and *online* phases. In the offline phase, we use an efficient *profiler* to get the resource-quality profile of queries without exploring the entire combinatorial space of configurations. Using greedy search and domain-specific sampling, we identify a handful of knob configurations on the *Pareto boundary* of the profile. The scheduler in the online phase, thus, has to consider only these configurations.

We encode quality and lag goals of a query in a *utility function*. Utility is a weighted combination of the achieved quality and lag, with penalties for violating the goals. Penalties allow for expressing priorities between queries. Given utilities of multiple queries, we schedule for two natural objectives – maximize the *minimum* utility, or maximize the *total* utility. The former achieves fairness (max-min) while the latter targets performance.

Finally, in the online phase, we model the scheduling problem using the Model-Predictive Control [67] to predict the *future* query lag over a short time horizon, and use this predicted lag in the utility function. The scheduler considers the resource-quality profile of queries during allocation, and allows for lagging queries to "catch up." It also deals with inevitable inaccuracies in resource usages in the resource-quality profiles.

While we focus VideoStorm on video analytics using computer vision algorithms, approximation and lag are aspects that are fundamental to all machine learning algorithms. To that end, the techniques in our system are broadly applicable to all stream analytics systems that employ machine learning techniques.

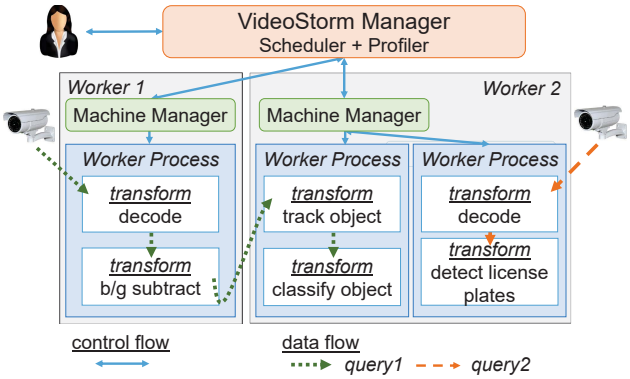


Figure 1: VideoStorm **System Architecture.**

Contributions. Our contributions are as follows:

1. We designed and built a system for large-scale analytics of live video that allows users to submit queries with arbitrary vision processors.
2. We efficiently identify the resource-quality profile of video queries without exhaustively exploring the combinatorial space of knob configurations.
3. We designed an efficient scheduler for video queries that considers their resource-quality profile and lag tolerance, and trades off between them.

We considered streaming databases with approximation [19,37,68] as a starting point for our solution. However, they only consider the sampling rate of data streams and used established analytical models [38] to calculate the quality and resource demand. In contrast, vision queries are more complex black-boxes with many more knobs, and do not have known analytical models. Moreover, they optimize only one query at a time, while our focus is on scheduling multiple concurrent queries.

Deployment on 101 machines in Azure show that VideoStorm's scheduler allocates resources in hundreds of milliseconds even with thousands of queries. We evaluated using real video analytics queries over video datasets from live traffic cameras from several large cities. Our offline profiling consumes 3.5× less CPU resources compared to a basic greedy search. The online VideoStorm scheduler outperforms fair scheduling of resources [3, 31, 49] by as much as 80% in quality of queries and 7× in terms of lag.

2 System Description

We describe the high-level architecture of VideoStorm and the specifications for video queries.

2.1 VideoStorm Architecture

The VideoStorm cluster consists of a centralized *manager* and a set of *worker machines* that execute *queries*,

```
"name": "LicensePlate",
"transforms": [
  {"id": "0",
   "class_name": "Decoder",
   "parameters": {
     "CameraIP": "134.53.8.8",
     "CameraPort": 8100,
     "@OutputResolution": "720P",
     "@SamplingRate": 0.75 }
  },
  {"id": "1",
   "input_transform_id": "0",
   "class_name": "OpenALPR",
   "parameters": {
     "@MinSize": 100,
     "@MaxSize": 1000,
     "@Step": 10 }
  } ]
```

Figure 2: VideoStorm **Query for license plate reader.**

see Figure 1. Every query is a DAG of *transforms* on *live video* that is continuously streamed to the cluster; each transform processes a time-ordered stream of messages (e.g., video frames) and passes its outputs downstream.

Figure 1 shows two example queries. One query runs across two machines; after decoding the video and subtracting the background, it sends the detected objects to another machine for tracking and classification. The other query for detecting license plates runs on a single machine. We assume there is sufficient bandwidth provisioned for cameras to stream their videos into the cluster.

Every worker machine runs a *machine manager* which start *worker processes* to host transforms. The machine manager periodically reports resource utilizations as well as status of the running transforms to the VideoStorm manager. The scheduler in the manager uses this information to allocate resources to queries. The VideoStorm manager and the machine managers are not on the query *data path*; videos are streamed directly to the decoding transforms and thereon between the transforms.

2.2 Video Queries Specification

Queries submitted to the VideoStorm manager are strung together as pipelines of *transforms*. Figure 2 shows a sample VideoStorm pipeline with two transforms. The first transform decodes the live video to produce frames that are pushed to the second transform to find license plate numbers using the OpenALPR library [13].

Each transform contains an id and class_name which is the class implementing the transform (§7). The input_transform_id field specifies the transform whose output feeds into this transform, thus allowing us to describe a pipeline. VideoStorm allows arbitrary DAGs including multiple inputs and outputs for a transform. *Source* transforms, such as the "Decoder", do not specify input transform, but instead directly connect to the camera source (specified using IP and port number).

Each transform contains optional knobs (parameters); e.g., the minimum and maximum window sizes (in pixels) of license plates to look for and the step increments to search between these sizes for the OpenALPR transform (more in §5). Knobs whose values can updated dynamically start with the '@' symbol. The VideoStorm manager updates them as part of its scheduling decisions.

C	D	Q
A1	1	0.6
A2	2	0.7
A3	3	0.8

(a) Query A

C	D	Q
B1	1	0.1
B2	2	0.3
B3	3	0.9

(b) Query B

		Query A					Query B				
Time	R	C	D	A	Q	L	C	D	A	Q	L
0	4	A2	2	2	0.7	-	B2	2	2	0.3	-
10	2	A1	1	1	0.6	-	B1	1	1	0.1	-
22	4	A2	2	2	0.7	-	B2	2	2	0.3	-

(c) Fair allocation

		Query A					Query B				
Time	R	C	D	A	Q	L	C	D	A	Q	L
0	4	A1	1	1	0.6	-	B3	3	3	0.9	-
10	2	A1	1	1	0.6	-	B3	3	1	0.9	-
22	4	A1	1	1	0.6	-	B2	2	3	0.3	8s
38	4	A1	1	1	0.6	-	B3	3	3	0.9	-

(d) Performance-based allocation

Table 1: Tables (a) and (b) show queries A and B with three configurations each, resource demand *D* and quality *Q*. Tables (c) and (d) show the time and capacity *R*, and for each query the chosen configuration *C*, demand *D*, allocation *A*, achieved quality *Q*, and lag *L* for the fair and performance-based schedulers. Notice in (d) that query B achieves higher quality between times 10 and 22 than with the fair scheduler in (c), and never lags beyond its permissible 8s.

3 Making the Case for Resource Allocation

We make the case for resource management in video analytics clusters using a simple example (§3.1) and real-world video queries (§3.2).

3.1 Motivating Example

Cluster managers such as Yarn [3], Apollo [31] and Mesos [49] commonly divide resources among multiple queries based on *resource fairness*. Being agnostic to query quality and lag preferences, fair allocation is the best they can do. Instead, *scheduling for performance*

leads to queries achieving better quality and lag.

The desirable properties of a scheduler for video analytics are: (1) allocate more resources to queries whose qualities will improve more, (2) allow queries with built-up lag in their processing to "catch up," and (3) adjust query configuration based on the resource allocated.

Tables 1a and 1b shows two example queries A and B with three knob configurations each (Ax and Bx, respectively). Query A's improvement in quality Q is less pronounced than B's for the same increase in resource demand D. Note that D is the resource to keep up with the incoming data rate. Query A cannot tolerate any lag, but B can tolerate up to 8 seconds of lag. Lag is defined as the difference between the time of the last-arrived frame and the time of the last-processed frame, i.e., how much time's worth of frames are queued-up unprocessed.

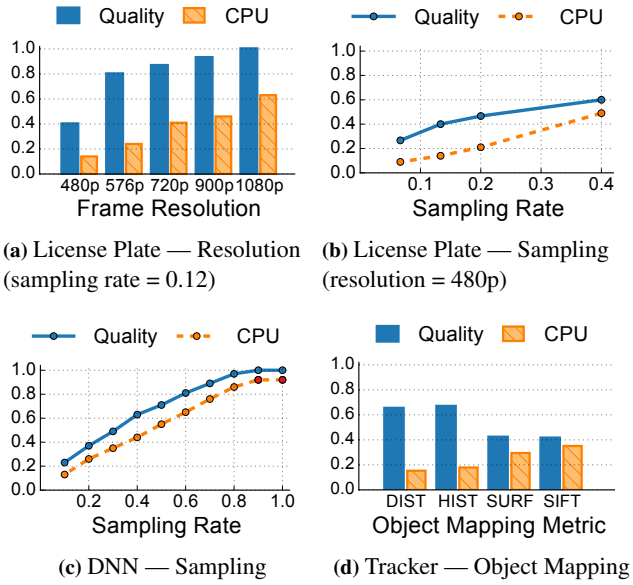
Let a single machine with resource capacity R of 4 run these two queries. Its capacity R drops to 2 after 10 seconds and then returns back to 4 after 12 more seconds (at 22 seconds). This drop could be caused by another high-priority job running on this machine.

Fair Scheduling. Table 1c shows the assigned configuration C, query demand D, resource allocation A, quality Q and lag L with a fair resource allocation. Each query selects the best configuration to keep up with the live stream (i.e., keeps its demand below its allocation). Using the fair scheduler, both queries get an allocation of 2 initially, picking configurations A2 and B2 respectively. Between times 10 to 22, when the capacity drops to 2, the queries get an allocation of 1 each, and pick configurations A1 and B1. At no point do they incur any lag.

Performance-based Scheduling. As Table 1d shows, a performance-based scheduler allocates resources of 1 and 3 to queries A and B at time 0; B can thus run at configuration B3, achieving higher quality compared to the fair allocation (while A's quality drops only by 0.1). This is because the scheduler realizes the value in providing more resources to B given its resource-quality profile.

At time 10 when capacity drops to 2, the scheduler allocates 1 unit of resource to each to the queries, but retains configuration B3 for B. Since resource demand of B3 is 3, but B has been allocated only 1, B starts to lag. Specifically, every second, the lag in processing will *increase* by $2/3$ of a second. However, query B will still produce results at quality 0.9, albeit delayed. At time 22, the capacity recovers and query B has built up a lag of 8 seconds. The scheduler allocates 3 resource units to B but switches it to configuration B2 (whose demand is only 2). This means that query B can now *catch up* – every second it can process 1.5 seconds of video. Finally, at time 38, all the lag has been eliminated and the scheduler switches B to configuration B3 (quality 0.9).

The performance-based scheduler exhibited the three properties listed above. It allocated resources to optimize for quality and allowed queries to catch up to built-up lag, while accordingly adjusting their configurations.



(a) License Plate — Resolution (sampling rate = 0.12)

(b) License Plate — Sampling (resolution = 480p)

(c) DNN — Sampling

(d) Tracker — Object Mapping

Figure 3: Resource-quality profiles for real-world video queries. For simplicity, we plot one knob at a time.

3.2 Real-world Video Queries

Video analytics queries have many *knob configurations* that affect output quality and resource demand. We highlight the resource-quality profiles of four real-world queries—*license plate reader, car counter, DNN classifier, object tracker*—of interest to the cities we are partnering with and obtained videos from their *operational* traffic cameras (§8.1). For clarity, we plot one knob at a time and keep other knobs fixed. Quality is defined as the F1 score $\in [0, 1]$ (the harmonic mean between precision and recall [83]) with reference to a *labeled ground truth*.

License Plate Reader. The OpenALPR [13] library scans the video frame to detect potential plates and then recognizes the text on plates using optical character recognition. In general, using higher video resolution and processing each frame will detect the most license plates accurately. Reducing the resolution and processing only a subset of frames (e.g., sampling rate of 0.25) dramatically reduces resource demand, but can also reduce the quality of the output (i.e., miss or incorrectly read plates). Figures 3a and 3b plots the impact of resolution and sampling rate on quality and CPU demand.[1]

Car Counter. Resolution and sampling rate are knobs that apply to almost all video queries. A car counter monitors an "area of interest" and counts cars passing the area. In general, its results are of good quality even with low resolution and sampling rates (plots omitted).

[1] Sampling rate of 0.75 drops every fourth frame from the video.

Deep Neural Network (DNN) Classifier. Vision processing is employing DNNs for key tasks including object detection and classification. Figure 3c profiles a Caffe DNN [54] model trained with the widely-used ImageNet dataset [41] to classify objects into 1,000 categories. We see a uniform increase in the quality of the classification as well as resource consumption with the sampling rate. As DNN models get *compressed* [45, 46], reducing their resource demand at the cost of quality, the compression factor presents another knob.

Object Tracker. Finally, we have also profiled an object tracker. This query continuously models the "background" in the video, identifies foreground objects by subtracting the background, and tracks objects *across frames* using a mapping metric. The mapping metric is a key knob (Figure 3d). Objects across frames can be mapped to each other using metrics such as distance moved (DIST), color histogram similarity (HIST), or matched over SIFT [14] and SURF [15] features.

Resource-quality profiles based on knob configurations is intrinsic to video analytics queries. These queries typically identify "events" (like license plates or car accidents), and using datasets where these events are labeled, we can empirically measure precision and recall in identifying the events for different query configurations.

In contrast to approximate SQL query processing, there are no analytical models to estimate the relationship between resource demand and quality of video queries and it *depends on the specific video feeds*. For example, reducing video resolution may not reduce OpenALPR quality if the camera is zoomed in enough. Hence queries need to be *profiled* using representative video samples.

3.3 Summary and Challenges

Designing a scheduler with the desirable properties in §3.1 for real-world video queries (§3.2) is challenging.

First, the configuration space of a query can be large and there are no analytical models to estimate the resource demand and result quality of each configuration.

Second, trading off between the lag and quality goals of queries is tricky, making it challenging to define scheduling objectives *across all queries in the cluster*.

Third, resource allocation across all queries in the cluster each with many configurations is computationally intractable, presenting scalability challenges.

4 Solution Overview

The VideoStorm scheduler is split into *offline profiling* and *online* phases (Figure 4). In the offline phase, for every query, we efficiently generate its *resource-quality profile* – a small number of configurations on the Pareto

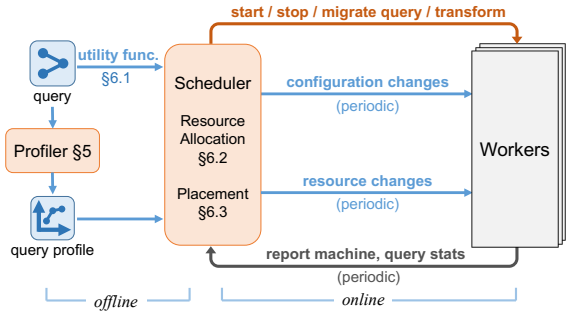


Figure 4: VideoStorm **Scheduler Components.**

curve of the profile, §5. This dramatically reduces the configurations to be considered by the scheduler.

In the online phase, the scheduler periodically (e.g., every second) considers all running queries and adjusts their resource allocation, machine placement, and configurations based on their profiles, changes in demand and/or capacity (see Figure 4). We encode the quality and lag requirements of *each individual query* into its utility function, §6.1. The performance goal across *all queries in a cluster* is specified either as maximizing the minimum utility or the sum of utilities, §6.2 and §6.3.

5 Resource-Quality Profile Estimation

When a user submits a new query, we start running it immediately with a default profile (say, from its previous runs on other cameras), while at the same time we run the query through the offline profiling phase. The query profiler has two goals. 1) Select a small subset of configurations (Pareto boundary) from the resource-quality space, and 2) Compute the query profile, $\mathcal{P}_k$, i.e., the resource demand and result quality of the selected configurations. The profile is computed either against a labeled dataset or using the initial parts of the video relative to a "golden" query configuration which might be expensive but is known to produce high-quality results.

5.1 Profile estimation is expensive

We revisit the license plate reader query from §3.2 in detail. As explained earlier, frame *resolution* and *sampling rate* are two important knobs. The query, built using the OpenALPR library [13], scans the image for license plates of size *MinSize*, then multiplicatively increases the size by *Step*, and keeps repeating this process until the size reaches *MaxSize*. The set of potential license plates is then sent to an optical character recognizer.

We estimate the quality of each knob configuration (i.e., combination of the five knobs above) on a labeled dataset using the F1 score [83], the harmonic mean between precision and recall, commonly used in machine

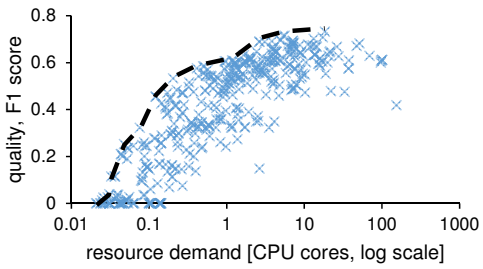


Figure 5: Resource-quality for license plate query on a 10 minute video (414 configurations); x-axis is resource demand to keep up with live video. Generating this took 20 CPU *days*. The black dashed line is the Pareto boundary.

Term	Description
$\mathcal{P}_k$	profile of query k
$c_k \in \mathcal{C}_k$	specific configuration of query k
$Q_k(c)$	quality under configuration c
$D_k(c)$	resource demand under configuration c
$L_{k,t}$	measured lag at time t
U_k	utility
Q_k^M	(min) quality goal
L_k^M	(max) lag goal
a_k	resources allocated

Table 2: Notations used, for query k.

learning; 0 and 1 represent the lowest and highest qualities. For example, increasing *MinSize* or decreasing *MaxSize* reduces the resources needed but can miss some plates and decrease quality.

Figure 5 shows a scatter plot of resource usage vs. quality of 414 configurations generated using the five knobs. There is four orders of magnitude of difference in resource usage; the most expensive configuration used all frames of a full HD resolution video and would take over 2.5 hours to analyze a 1 minute video on 1 core. Notice the vast spread in quality among configurations with similar resource usage as well as the spread in resource usage among configurations that achieve similar quality.

5.2 Greedy exploration of configurations

We implement a greedy local search to identify configuration with high quality (Q) and low demand (D); see Table 2. Our *baseline profiler* implements hill-climbing [74]; it selects a random configuration c, computes its quality $Q(c)$ and resource demand $D(c)$ by running the query with c on a *small subset* of the video dataset, and calculates $X(c) = Q(c) - \beta D(c)$ where β trades off between quality and demand. Next, we pick a neighbor configuration n (by changing the value of a *random* knob in c). If $X(n) > X(c)$, then n is better than c in quality or resource demand (or both); we set $c = n$ and repeat. When we cannot find a better neighbor (i.e., our exploration indicates that we are near a local optimum), we repeat by picking another random c.

Several enhancements significantly increase the efficiency of our search. To avoid starting with an expensive configuration and exploring its neighbors, (which are also likely to be expensive, thus wasting CPU), we pick k random configurations and start from the one with the highest $X(c)$. We found that using even $k = 3$ can successfully avoid starting in an expensive part of the search space. Second, we cache *intermediate results* in the query's DAG and reuse them in evaluating configurations with overlapping knob values.

While our simple profiler is sufficiently efficient for our purpose, sophisticated hyperparameter searches (e.g., [76]) can potentially further improve its efficiency.

Pareto boundary. We are only interested in a small subset of configurations that are on the *Pareto boundary* $\mathcal{P}$ of the resource-quality space. Let $Q(c)$ be the quality and $D(c)$ the resource demand under configuration c. If c_1 and c_2 are two configurations such that $Q(c_1) \geq Q(c_2)$ and $D(c_1) \leq D(c_2)$, then c_2 is not useful in practice; c_1 is better than c_2 in both quality and resource demand. The dashed line in Figure 5 shows the Pareto boundary of such configurations for the license plate query. We extract the Pareto boundary of the explored configurations and call it the resource-quality profile $\mathcal{P}$ of the query.

We can generate the same profile as the baseline profiler on the license plate query with 3.5× less CPU resources (i.e., 5.4 CPU hours instead of 19 CPU hours).

6 Resource Management

In the *online phase*, the VideoStorm cluster scheduler considers the utilities of individual queries and the cluster-wide performance objectives (defined in §6.1) and periodically performs two steps: resource allocation and query placement. In the resource *allocation step*, §6.2, the scheduler assumes the cluster is an aggregate bin of resources and uses an efficient heuristic to maximize the cluster-wide performance by adjusting query allocation and configuration. In the query *placement step*, §6.3, the scheduler places new queries to machines in the cluster and considers migrating existing queries.

6.1 Utility: Combining Quality and Lag

Each query has preferences on the desired quality and lag. What is the *minimum* quality goal (Q^M)? How much does the query benefit from higher quality than the goal? What is the *maximum* lag (L^M) it can tolerate and how sensitive are violations to this goal? (See Table 2 for notations.) We encode these preferences in

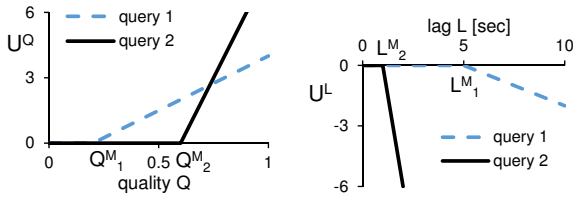


Figure 6: Examples for the second (U^Q) and third terms (U^L) in equation 1. (Left) Query 1's quality goal is relatively lenient, $Q_1^M = 0.2$, but its utility grows slowly with increase in quality beyond Q_1^M. Query 2 is more stringent, $Q_2^M = 0.6$, but its utility grows sharply thereon. (Right) Query 1 has lag target of $L_1^M = 5$ beyond which it incurs a penalty. Query 2 has a stricter lag goal of $L_2^M = 1$ and also its utility drops much faster with increased lag.

utility functions, an abstraction used extensively in economics [65, 73] and computer systems [22, 55].

Our utility function for a query has the following form, where $(x)_+$ is the positive part of x. We omit the query index k for clarity.

$$\begin{aligned} U(Q,L) &= U^B + U^Q(Q) + U^L(L) \\ &= U^B + \alpha^Q \cdot (Q - Q^M)_+ - \alpha^L \cdot (L - L^M)_+ \end{aligned} \quad (1)$$

U^B is the "baseline" utility for meeting the quality and lag goals (when $Q = Q^M$ and $L = L^M$). The second term U^Q describes how the utility responds to achieved quality Q above Q^M, the soft quality goal; the multiplier α^Q and Q^M are query-specific and set based on the application analyzing the video. Results with quality below Q^M are typically not useful to the users.

The third term, U^L, represents the penalty for results arriving later than the maximum lag goal of L^M. [2] Recall that lag is the difference between the current time and the arrival time of the last processed frame, e.g., if at time 10:30 we process a frame that arrived at 10:15, the lag is 15 minutes. Similar to latency SLOs in clusters, there is no bonus for lag being below L^M. See Figure 6 for examples of U^Q and U^L in queries.

Scheduling objectives. Given utilities of individual queries, how do we define utility or *performance* of the whole cluster? Previous work has typically aimed to maximize the minimum utility [61, 64] or sum of utilities [61, 63], which we adopt. When deployed as a "service" in the public cloud, utility will represent the revenue the cluster operator generates by executing the query; penalties and bonuses in utility translate to loss and increase in revenue. Therefore, *maximizing the sum of utilities* maximizes revenue. In a private cluster that is shared by many cooperating entities, achieving fairness is more desirable. Maximally improving the utility of the worst query provides *max-min fairness over utilities*.

[2] Multiplier α^L is in (1/second), making U^L dimensionless like U^Q.

To simplify the selection of utility functions in practical settings, we can provide only a few options to choose from. For example, the users could separately pick the minimum quality (40%, 60%, or 80%) and the maximum lag (1, 10, or 60 minutes) for a total of nine utility function *templates*. Users of cloud services already make similar decisions; for example, in Azure Storage [32], they separately select data redundancy (local, zone, or geo-distributed) and data access pattern (hot vs. cool).

6.2 Resource Allocation

Given a profile $\mathcal{P}_k$ and a utility function U_k for each query k, the scheduler allocates resources a_k to the queries and picks their query configuration ($c_k \in \mathcal{P}_k$). The scheduler runs periodically (e.g., every few seconds) and reacts to arrival of new queries, changes in query demand and lag, and changes in resource capacity (e.g., due to other high-priority non-VideoStorm jobs).

6.2.1 Scheduling Using Model-Predictive Control

The scheduler aims to maximize the minimum or sum of query utilities, which in turn depend on their quality and lag. A key point to understand is that while we can *near-instantaneously* control query quality by adjusting its configuration, query lag *accumulates* over time if we allocate less resources than query demand.

Because of this accumulation property, the scheduler cannot optimize the *current performance*, but only aims to improve performance in the *near future*. We formulate the scheduling problem using the Model-Predictive Control (MPC [67]) framework; where we model the cluster performance over a short time horizon T as a function of query configuration and allocation. In each step, we select the configuration and allocation to maximize performance over the near future (described in detail in §6.2.2).

To predict future performance, we need to predict query lag; we use the following formula:

$$L_{k,t+T}(a_k, c_k) = L_{k,t} + T - T\frac{a_k}{D_k(c_k)} \quad (2)$$

We plug in the predicted lag $L_{k,t+T}$ into the utility function (Equation 1) to obtain the predicted utility.

6.2.2 Scheduling Heuristics

We describe resource allocation assuming each query to contain only one transform, which we relax in §6.4.

Maximizing sum of utilities. The optimization problem for maximizing sum of utilities over time horizon T is as follows. Sum of allocated resources a_k cannot ex-

ceed cluster resource capacity R.

$$\max_{a_k, c_k \in \mathcal{P}_k} \quad \sum_k U_k(Q_k(c_k), L_{k,t+T}) \qquad (3)$$
$$s.t. \qquad \sum_k a_k \leq R$$

Maximizing the sum of utilities is a variant of the knapsack problem where we are trying to include the queries at different allocation and configuration to maximize the total utility. The maximization results in the best distribution of resources (as was illustrated in §3.1).

When including query k at allocation a_k and configuration c_k, we are *paying* cost of a_k and *receiving value* of $u_k = U_k(Q_k(c_k), L_{k,t+T})$. We employ a greedy approximation based on [40] where we prefer queries with highest value of u_k/a_k; i.e., we receive the largest increase in utility normalized by resource spent.

Our heuristic starts with $a_k = 0$ and in each step we consider increasing a_i (for all queries i) by a small Δ (say, 1% of a core) and consider all configurations of $c_i \in \mathcal{P}_i$. Among these options, we select query i (and corresponding c_i) with largest increase in utility.[3] We repeat this step until we run out of resources or we have selected the best configuration for each query. (Since we start with $a_k = 0$ and stop when we run out of resources, we will not end up with infeasible solutions.)

Maximizing minimum utility. Below is the optimization problem to maximize the minimum utility predicted over a short time horizon T. We require that all utilities be $\geq u$ and we maximize u.

$$\max_{a_k, c_k \in \mathcal{P}_k} \quad u \qquad (4)$$
$$s.t. \quad \forall k : U_k(Q_k(c_k), L_{k,t+T}) \geq u$$
$$\sum_k a_k \leq R$$

We can improve u only by improving the utility of the worst query. Our heuristic is thus as follows. We start with $a_k = 0$ for all queries. In each step, we select query $i = \arg\min_k U_k(Q_k(c_k), L_{k,t+T})$ with the lowest utility and increase its allocation by a small Δ, say 1% of a core. With this allocation, we compute its best configuration c_i as $\arg\max_{c \in \mathcal{P}_i} U_i(Q_i(c), L_{i,t+T})$. We repeat this process until we run out of resources or we have picked the best configuration for each query.

6.3 Query Placement

After determining resource allocation and configuration of each query, we next describe the placement of new queries and migration of existing queries. We quantify the suitability of placing a query q on machine m by computing a score for each of the following goals: high utilization, load balancing, and spreading low-lag queries.

(i) Utilization. High utilization in the cluster can be achieved by *packing* queries in to machines, thereby minimizing fragmentation and wastage of resources. Packing has several well-studied heuristics [44, 71]. We define *alignment* of a query relative to a machine using a weighted dot product, p, between the vector of machine's available resources and the query's demands; $p \in [0, 1]$.

(ii) Load Balancing. Spreading load across the cluster ensures that each machine has spare capacity to handle changes in demand. We therefore prefer to place q on a machine m with the smallest utilization. We capture this in score $b = 1 - \frac{M+D}{M_{max}} \in [0, 1]$, where M is the current utilization of machine m and D is demand of query q.

(iii) Lag Spreading. Not concentrating many low-lag queries on a machine provides *slack* to accumulate lag for some queries when resources are scarce, *without* having to resort to migration of queries or violation of their lag goal L^M. We achieve this by maintaining *high average* L^M on each machine. We thus compute score $l \in [0, 1]$ as the average L^M *after* placing q on m.

The final score $s_{q,m}$ is the average of the three scores. For each new query q, we place it on a machine with the largest $s_{q,m}$. For each existing query q, we migrate from machine m_0 to a new machine m_1 only if its score improves substantially; i.e., $s(q, m_1) - s(q, m_0) > \tau$.

6.4 Enhancements

Incorrect resource profile. The profiled resource demand of a query, $D_k(c_k)$, might not exactly correspond to the actual query demand, e.g., when demand depends on video content. Using incorrect demand can negatively impact scheduling; for example, if $D_k(c) = 10$, but actual usage is $R_k = 100$, the scheduler would estimate that allocating $a_k = 20$ would reduce query lag at the rate of $2\times$, while the lag would actually *grow* at a rate of $5\times$. To address this, we keep track of a running average of *mis-estimation* $\mu = R_k/D_k(c)$, which represents the multiplicative error between the predicted demand and actual usage. We then incorporate μ in the lag predictor from Equation 2, $L_{k,t+T}(a_k, c_k) = L_{k,t} + T - T\frac{a_k}{D_k(c_k)}(\frac{1}{\mu})$.

Machine-level scheduling. As most queries fit on a single machine, we can respond to changes in demand or lag at the machine-level, without waiting for the cluster-wide decisions. We therefore execute the allocation step from §6.2 on each machine, which makes the scheduling logic much more scalable. The cluster-wide scheduler still runs the allocation step, but only for the purposes of determining query placement and migration.

DAG of transforms. Queries consisting of a DAG of transforms could be placed across multiple machines.

[3] We use a *concave* version of the utility functions obtained using linear interpolation to ensure that each query has a positive increase in utility, even for small Δ.

We first distribute the query resource allocation, a_k, to *individual transforms* based on per-transform resource demands. We then place individual transforms to machines as described in §6.3 while accounting for the expected data flow across machines and network link capacities.

7 VideoStorm Implementation

We now discuss VideoStorm's key implementation details and the interfaces implemented by transforms.

7.1 Implementation Details

In contrast to widely-deployed cluster frameworks like Yarn [3], Mesos [49] and Cosmos [31], we highlight the differences in VideoStorm's design. First, VideoStorm takes the list of knobs, resource-quality profiles and lag goals as inputs to allocate resources. Second, machine-level managers in the cluster frameworks *pull* work, whereas the VideoStorm manager *pushes* new queries and configuration changes to the machine-managers. Finally, VideoStorm allows machine managers to autonomously handle short-term fluctuations (§6.4)

Flow control. We implemented flow control across transforms of a query to minimize the buffering inside the query pipeline, and instead push queuing of unprocessed video to the *front* of the query. This helps for two reasons. First, decoded frames can be as much as 300× larger than the encoded video (from our benchmarks on HD videos). Buffering these frames will significantly inflate memory usage while spilling them to disk affects overall performance. Second, buffering at the front of query enables the query to respond promptly to configuration changes. It prevents frames from being processed by transforms with old inconsistent knob values.

Migration. As described in §6.3, VideoStorm migrates queries depending on the load in the cluster. We implement a simple "start-and-stop" migration where we start a copy of a running query/transform on the target machine, duplicate its input stream to the copy, and stop the old query/transform after a short period. The whole process of migration is *data-lossless* and takes roughly a second (§8.3), so the overhead of duplicated processing during the migration is very small.

Resource Enforcement. VideoStorm uses Job Objects [18] for enforcing allocations. Similar to Linux Containers [11], Job Objects allow controlling *and resizing* the CPU/memory limits of running processes.

7.2 Interfaces for Query Transforms

Transforms implement simple interfaces to process data and exchange control information.

- *Processing*. Transforms implement byte[] Process(header, data) method. header contains metadata such as frame id and timestamp. data is the input byte array, such as decoded frame. The transform returns another byte array with its result, such as the detected license plate. Each transform maintains its own state, such as the background model.
- *Configuration*. Transforms can also implement Update(key, value) to set and update knob values to change query configuration at runtime.

8 Evaluation

We evaluate the VideoStorm prototype (§7) using a cluster of 101 machines on Microsoft Azure with real video queries and video datasets. Our highlights:

1. VideoStorm outperforms the fair scheduler by 80% in quality of outputs with 7× better lag. (§8.2)
2. VideoStorm is robust to errors in query profiles and allocates nearly the same as correct profiles. (§8.3)
3. VideoStorm scales to thousands of queries with little systemic execution overheads. (§8.4)

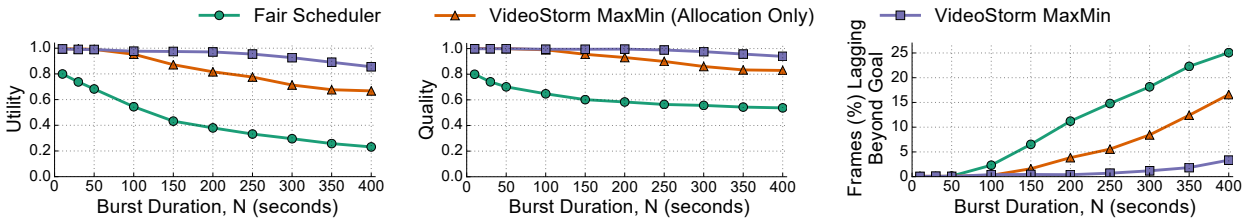
8.1 Setup

Video Analytics Queries. We evaluate VideoStorm using four types of queries described and profiled in §3.2 – license plate reader, car counter, DNN classifier, object tracker. These queries are of major interest to the cities we are partnering with in deploying our system.

Video Datasets. The above queries run on video datasets obtained from real and operational traffic cameras in Bellevue and Seattle cities for two months (Sept.–Oct., 2015). In our experiments, we stream the recorded videos at their original frame-rate (14 to 30 fps) and resolution (240P to 1080P) thereby mimicking live video streams. The videos span a variety of conditions (sunny/rainy, heavy/light traffic) that lead to variation in their processing workload. We present results on multiple different snippets from the videos.

Azure Deployment. We deploy VideoStorm on 101 D3 v2 instances on Azure's West-US cluster [6]. D3 v2 instances contain 4 cores of the 2.4GHz Intel Xeon processor and 14GB RAM. One machine ran the VideoStorm global manager on which no queries were scheduled.

Baseline. We use the work-conservative fair scheduler as our baseline. It's the widely-used scheduling policy for cluster computing frameworks like Mesos [49], Yarn [3] and Cosmos [31]. When a query, even at its best configuration, cannot use its fair share, it distributes the excess resources among the other queries. The fair scheduler places the same number of queries evenly on all available machines in a round-robin fashion.



(a) Utility **(b)** Quality **(c)** Lag

Figure 7: VideoStorm **outperforms the fair scheduler as the *duration* of burst of queries in the experiment is varied. Without its placement but only its allocation ("**VideoStorm **MaxMin (Allocation Only)"), its performance drops by a third.**

Metric. The three metrics of interest to us are quality, frames (%) exceeding the lag goal in processing, and utility (§6.1). We compare the improvement (%); if a metric (say, quality) with VideoStorm and the fair scheduler is X_V and X_f, we measure $\frac{X_V - X_f}{X_f} \times 100\%$.

8.2 Performance Improvements

Our workload consists of a mix of queries with lenient and stringent goals. We start with a set of 300 queries picked from the four types (§8.1) on 300 distinct video datasets at the beginning of the experiment. 60% of these queries have a lag goal L^M of 20s while the remaining are more lenient with a lag goal of 300s. All of them have a quality goal Q^M of 0.25. We set the lag multiplier $\alpha^L = 1$ for these long-lived video analyses.

Burst of N seconds: At a certain point, a burst of 200 license plate queries arrive and last for N seconds (which we will vary). These queries have a lag goal Q^L of 20s, a high quality goal (1.0), and higher $\alpha^L = 2$. They mimic short-term deployment of queries like AMBER Alerts with stringent accuracy and lag goals. We evaluate VideoStorm's reaction to the burst of queries up to several minutes; note that the improvements will carry over when tolerant delay and bursts are much longer.

8.2.1 Maximize the Minimum Utility (MaxMin)

We ran a series of experiments with burst duration N from 10 seconds to 400 seconds. Figure 7a plots the minimum query utility achieved in each of the experiments, when VideoStorm maximizes the minimum utility (§6.2.2). For each point in the figure, we obtain the minimum utility, quality and lag over an interval that includes a minute before and after the N second burst. VideoStorm's utility ("VideoStorm-MaxMin") drops only moderately with increasing burst duration. Its placement and resource allocations ensure it copes well with the onset of and during the burst. Contrast with the fair scheduler's sharp drop with N.

The improvement in utility comes due to smartly accounting for the resource-quality profile and lag goal of the queries; see Figures 7b and 7c. Quality (F1 score [83]; $\in [0, 1]$) with the fair scheduler is 0.2 lower than VideoStorm to begin with, but reduces significantly to nearly 0.5 for longer bursts (higher N), while quality with VideoStorm stays at 0.9, or nearly 80% better. The rest of VideoStorm's improvement comes by ensuring that despite the accumulation in lag, fewer than 5% of the frames exceed the query's lag goal whereas with the fair scheduler it grows to be 7× worse.

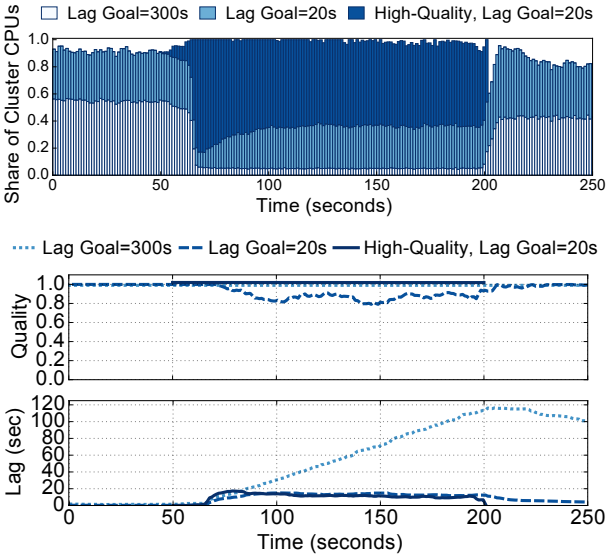


Figure 8: (Top) CPU Allocation for burst duration $N =$ 150s, and (bottom) quality and lag averaged across all queries in each of the three categories.

How valuable is VideoStorm**'s placement?** Figure 7 also shows the "VideoStorm MaxMin (Allocation Only)" graphs which lie in between the graphs for the fair scheduler and VideoStorm. As described in §6.3, VideoStorm first decides the resource allocation and then places them onto machines to achieve high utilization, load balancing and spreading of lag-sensitive and lag-tolerant queries. As the results show, not using VideoStorm's placement heuristic (instead using our baseline's round-robin placement) considerably lowers VideoStorm's gains.

Figure 8(top) explains VideoStorm's gains by plotting the allocation of CPU cores in the cluster over time, for burst duration $N = 150$s. We group the queries into

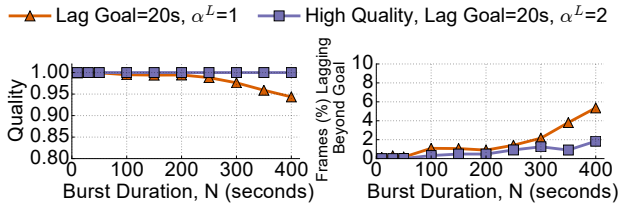
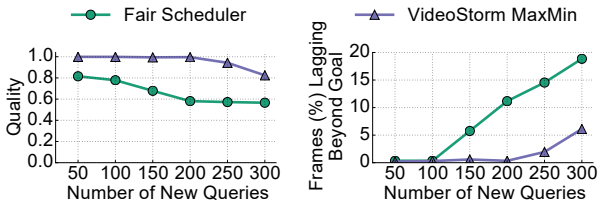


Figure 9: Impact of α^L. Queries with higher α^L have fewer frames lagging beyond their goal.



(a) Quality **(b)** Lag

Figure 10: VideoStorm **vs. fair scheduler as the number of queries in the burst during the experiment is varied.**

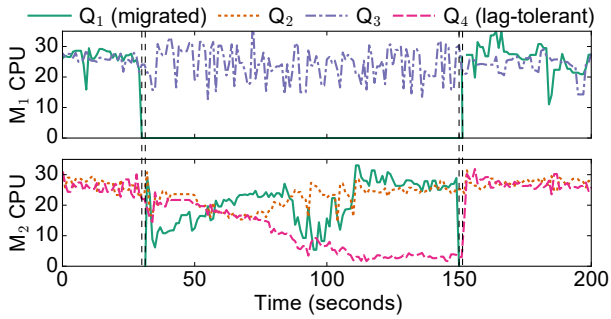


Figure 11: Q_1 migrated between M_1 and M_2. Resource for the only lag-tolerant query Q_4 (on M_2) is reduced for Q_1.

three categories — the burst of queries with 20s lag goal and quality goal of 1.0, those with 20s lag goal, and 300s lag goal (both with quality goal of 0.25). We see that VideoStorm adapts to the burst and allocates nearly 60% of the CPU cores in the cluster to the burst of license plate queries which have a high quality and tight lag goals. VideoStorm also delays processing of lag-tolerant queries (allocating less than 10% of CPUs). Figure 8(bottom) shows the resulting quality and lag, for queries in each category. We see that because the delay-tolerant queries have small allocation, their lag grows but stays below the goal. The queries with 20s lag goal reduce their quality to adapt to lower allocation and keep their lag (on average) within the bound.

Impact of α^L. Figure 9 plots the distinction in treatment of queries with the same lag goal (L^M) but different α^L and quality goals. While the figure on the left shows that VideoStorm does not drop the quality of the query with $Q^M = 1.0$, it also respects the difference in α^L; fewer frames of the query with $\alpha^L = 2$ lag beyond the goal of 20s (right). This is an example of how utility functions encode priorities.

8.2.2 Maximize the Total Utility (MaxSum)

Recall from §6.2.2 that VideoStorm can also maximize the *sum* of utilities. We measure the *average* utility, quality, and frames (%) exceeding the lag goal; maximizing for the total utility and average utility are equivalent. VideoStorm achieves 25% better quality and 5× better lag compared with the fair scheduler.

Per Query Performance. While MaxMin scheduling, as expected, results in all the queries achieving similar quality and lag, MaxSum priorities between queries as the burst duration increases. Our results show that the license plate query, whose utility over its resource demand is relatively lower, is de-prioritized with MaxSum (reduced quality as well as more frames lagging). With its high quality (1.0) and low lag (20s) goals, the scheduler has little leeway. The DNN classifier, despite having comparable resource demand does not suffer from a reduction in quality because of its tolerance to lag (300s).

8.2.3 Varying the Burst Size

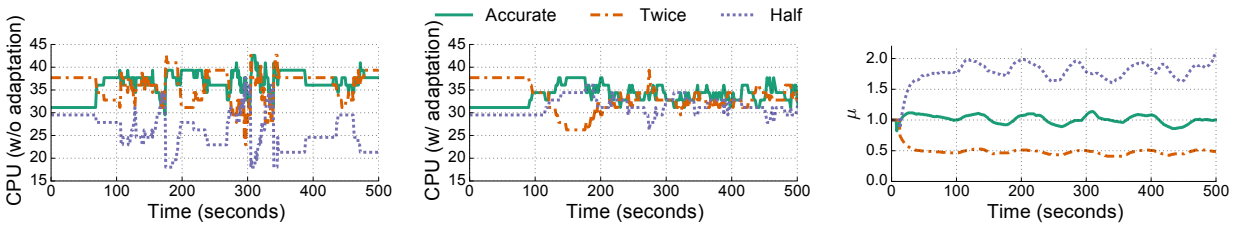
We next vary the *size* of the burst, i.e., number of queries that arrive in the burst. Note that the experiments above had varied the *duration* of the burst but with a fixed size of 200 queries. Varying the number of queries in the burst introduces different dynamics and reactions in VideoStorm's scheduler. We fix the burst duration to 200s. Figure 10 plots the results. The fair allocation causes much higher fraction of frames to exceed the lag goal when the burst size grows. VideoStorm better handles the burst and consistently performs better. Note that beyond a burst of 200 queries, resources are insufficient even to satisfy the lowest configuration (least resource demand), causing the degradation in Figure 10b.

8.3 VideoStorm's Key Features

We now highlight VideoStorm's migration of queries and accounting for errors in the resource demands.

8.3.1 Migration of Queries

Recall from §6.3 and §7 that VideoStorm migrates queries when necessary. We evaluate the value of migration by making the following addition to our experiment described at the beginning of §8.2. During the experiment, we allocate half the resources in 50% of our machines to other *non*-VideoStorm jobs. After a few minutes, the non-VideoStorm jobs complete and leave. Such jobs will be common when VideoStorm is co-situated



(a) Without Adaptation **(b)** With Adaptation **(c)** μ Over Time

Figure 12: We show three queries on a machine whose resource demands in their profiles are synthetically doubled, halved, and unchanged. By learning the proportionality factor μ (12c), our allocation adapts and converges to the right allocations (12b) as opposed to without adaptation (12a).

with other frameworks in clusters managed by Yarn [3] or Mesos [49]. We measure the migration time, and compare the performance with and without migration.

Figure 11 plots the timeline of two machines, M_1 and M_2; M_1 where a non-VideoStorm job was scheduled and M_2 being the machine *to* which a VideoStorm query Q_1, originally on M_1, was migrated. Q_1 shifts from running on M_1 to M_2 in only 1.3s. We migrate Q_1 back to M_1 when the non-VideoStorm job leaves at $\sim 150s$.

Shifting Q_1 to M_2 (and other queries whose machines were also allocated non-VideoStorm jobs, correspondingly) ensured that we did not have to degrade the quality or exceed the lag goals. Since our placement heuristic carefully spread out the queries with lenient and stringent lag goals (§6.3), we ensured that each of the machines had sufficient *slack*. As a result, when Q_1 was migrated to M_2 which already was running Q_2 and Q_4, we could delay the processing of the lag-tolerant Q_4 *without* violating any lag goals. The allocations of these delayed queries were ramped up for them to process their backlog as soon as the queries were migrated back.

As a consequence, the quality of queries with migration is 12% better than without migration. Crucially, 18× more frames (4.55% instead of 0.25%) would have exceeded the lag goal without migration.

8.3.2 Handling Errors in Query Profile

VideoStorm deals with difference between the resource demands in the resource-quality profile and the actual demand by continuously monitoring the resource consumption and adapting to errors in profiled demand (μ in §6.4). We now test the effectiveness of our correction.

We synthetically introduce *errors* in our profiles, as if they were profiles with errors, and use the erroneous profiles for our resource allocation. Consequently, the actual resource demands when the query executes do not match. In the workload above, we randomly make the profile to be half the actual resource demand for a third of the queries, twice the demand for another third, and unchanged (accurate) for the rest. VideoStorm's adaptive correction ensures that the quality and lag of queries with erroneous profiles are nearly 99.6% of results obtained *if* the profiles were perfectly accurate.

Action	Mean Duration (ms)	Standard Deviation (ms)
Start Transform	60.37	3.96
Stop Transform	3.08	0.47
Config. Change	15	2.0
Resource Change	5.7	1.5

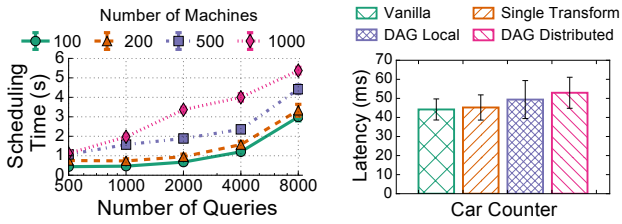
Table 3: Latency of VideoStorm's actions.

In Figure 12, we look at a single machine where VideoStorm placed three license plate queries, one each of the three different error categories. An ideal allocation (in the absence of errors) should be a third of the CPU to each of the queries. Figure 12a, however, shows how the allocation is far from converging towards the ideal *without* adaptation, because erroneous profiles undermine the precision of utility prediction. In contrast, with the adaptation, despite the errors, resource allocations converge to and stay at the ideal (Figure 12b). This is because the μ values for the queries with erroneous profiles are correctly learned as 2 and 0.5; the query without any error introduced its profile has its μ around 1 (Figure 12c).

8.4 Scalability and Efficiency

Latency of VideoStorm's actions. Table 3 shows the time taken for VideoStorm to start a new transform (shipping binaries, process startup), stop a transform, and change a 100-knob configuration and resource allocation of 10 running queries. We see that VideoStorm allows for near-instantaneous operations.

Scheduling Decisions. Figure 13a plots the time taken by VideoStorm's scheduler. Even with thousands of queries, VideoStorm make its decisions in just a few seconds. This is comparable to the scalability of schedulers in big data clusters, and video analytics clusters are unlikely to exceed them in the number of queries. Combined with the low latency of actions (Table 3), we believe VideoStorm is sufficiently scalable and agile.



(a) Scheduling Scalability **(b)** Overheads

Figure 13: Overheads in scheduling and running queries.

Transform Overheads. Finally, we measure the overhead of running a vision algorithm inside VideoStorm. We compare the latency in processing a frame while running as a vanilla process, inside a single transform, as a DAG of transforms on one machine, and as a DAG distributed across machines. Figure 13b shows that the overheads are limited. Running as a single transform, the overhead is $< 3\%$. When possible, VideoStorm places the transforms of a query DAG locally on one machine.

9 Related Work

Cluster schedulers. Cluster schedulers [3, 31, 39, 42, 44, 49, 86] do not cater to the performance objectives of streaming video analytics. They take resource demands from tasks (not the profiles), mostly allocate based on fairness/priorities, and do not resize running containers, key to dealing with resource churn in VideoStorm (§7).

Deadline-Based Scheduling. Many systems [22, 39, 42, 56, 85] adaptively allocate resources to meet deadlines of batch jobs or reduce lag of streaming queries. Scheduling in real-time systems [52,87] has also considered using utility functions to provide (soft) deadlines to running tasks. Crucially, these systems do not consider approximation together with resource allocation to meet deadlines and do not optimize across multiple queries and servers.

Streaming and Approximate Query Processing Systems. Load shedding has been a topic of interest in streaming systems [25, 68] to manage memory usage of SQL operators but they do not consider lag in processing. Aurora, Medusa, and Borealis [19, 33, 37] and follow-up works [78, 79, 81, 82, 88] use QoS graphs to capture lag and sampling rate but they consider them *separately* and do not trade-off between them, a key aspect in our solution. In contrast to JetStream [72], that degrades data quality based on WAN bandwidths, VideoStorm identifies the best knobs to use automatically and adjusts allocations *jointly* across queries. Stream processing systems used in production [2, 4, 62, 89] do not consider load-shedding, and resource-quality tradeoff and lag in their design; Google Cloud Dataflow [21] requires manual trade-off specifications. Approximation is also used by recent [20, 23, 84] and older [47, 53] batch querying systems using statistical models for SQL operators [38].

Relative to the above literature, our main contributions are three-fold: (i) considering quality and lag of video queries *together* for multiple queries using *predictive control*, (ii) dealing with multitude of knobs in vision algorithms, and (iii) profiling *black-box vision transforms* with arbitrary user code (not standard operators).

Utility functions. Utility functions are used extensively throughout economics [65, 73], compute science [48, 55, 57, 63], and other disciplines to map how users benefit from performance [50, 58, 80]. In stream processing systems, queries describe their requirements for throughput, latency, and fraction of dropped tuples [22,34,60,79]. With multiple entities, previous work has typically maximized the minimum utility [61, 64] or sum of utilities [61, 63], which is what we also use. Utility *elicitation* [28, 30, 35] helps obtain the exact shape of the utility function.

Autonomic Computing. Autonomic computing [24, 26, 29, 66, 70, 77] allocate resources to VMs and web applications to maximize their quality of service. While some of them used look-ahead controllers based on MPC [67], they mostly ignored our main issues on the large space of configurations and quality-lag trade-offs.

10 Conclusion

VideoStorm is a video analytics system that scales to processing thousands of video streams in large clusters. Video analytics queries can adapt the quality of their results based on the resources allocated. The core aspect of VideoStorm is its scheduler that considers the resource-quality profiles of queries, each with a variety of knobs, and tolerance to lag in processing. Our scheduler optimizes jointly for the quality and lag of queries in allocating resources. VideoStorm also efficiently estimates the resource-quality profiles of queries. Deployment on an Azure cluster of 101 machines show that VideoStorm can significantly outperform a fair scheduling of resources, the widely-used policy in current clusters.

Acknowledgments

We are grateful to Xiaozhou Li, Qifan Pu, Logan Stafman and Shivaram Venkataraman for reading early versions of the draft and providing feedback. We also thank our shepherd George Porter and the anonymous NSDI reviewers for their constructive feedback. This work was partially supported by NSF Awards CNS-0953197 and IIS-1250990.

References

[1] AMBER Alert, U.S. Department of Justice. http://www.amberalert.gov/faqs.htm.

[2] Apache Flink. https://flink.apache.org/.

[3] Apache Hadoop NextGen MapReduce (YARN). https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html.

[4] Apache Storm. https://storm.apache.org/.

[5] Avigilon. http://avigilon.com/products/.

[6] Azure Instances. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/.

[7] Capacity Scheduler. https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[8] China's 100 Million Surveillance Cameras. https://goo.gl/UK3Obl.

[9] Genetec. https://www.genetec.com/.

[10] Hadoop Fair Scheduler. https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[11] Linux Containers LXC Introduction. https://linuxcontainers.org/lxc/introduction/.

[12] One Surveillance Camera for Every 11 People in Britain, Says CCTV Survey. https://goo.gl/cHLqiK.

[13] Open ALPR. http://www.openalpr.com.

[14] OpenCV Documentation: Introduction to SIFT (Scale-Invariant Feature Transform). http://docs.opencv.org/3.1.0/da/df5/tutorial_py_sift_intro.html.

[15] OpenCV Documentation: Introduction to SURF (Speeded-Up Robust Features). http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html.

[16] SR 520 Bridge Tolling, WA. https://www.wsdot.wa.gov/Tolling/520/default.htm.

[17] Turnpike Enterprise Toll-by-Plate, FL. https://www.tollbyplate.com/index.

[18] Windows Job Objects. https://msdn.microsoft.com/en-us/library/windows/desktop/ms684161(v=vs.85).aspx.

[19] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, Jan. 2005.

[20] S. Agarwal, B. Mozafari, A. Panda, M. H., S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *ACM EuroSys*, Apr. 2013.

[21] T. Akidau et al. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-order Data Processing. *Proceedings of the VLDB Endowment*, Aug. 2015.

[22] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive Control of Extreme-Scale Stream Processing Systems. In *IEEE ICDCS*, July 2006.

[23] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *USENIX NSDI*, Apr. 2014.

[24] A. AuYoung, A. Vahdat, and A. C. Snoeren. Evaluating the Impact of Inaccurate Information in Utility-Based Scheduling. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov. 2009.

[25] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *IEEE ICDE*, Mar. 2004.

[26] A. Beloglazov and R. Buyya. Energy Efficient Resource Management in Virtualized Cloud Data Centers. In *IEEE CCGRID*, May 2010.

[27] A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical Scheduling for Diverse Datacenter Workloads. In *ACM SoCC*, Nov. 2014.

[28] J. Blythe. Visual Exploration and Incremental Utility Elicitation. In *AAAI*, July 2002.

[29] N. Bobroff, A. Kochut, and K. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *IFIP/IEEE International Symposium on Integrated Network Management*, 2007.

[30] C. Boutilier, R. Patrascu, P. Poupart, and D. Schuurmans. Regret-based Utility Elicitation in Constraint-based Decision Problems. In *IJCAI*, 2005.

[31] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *USENIX OSDI*, 2014.

[32] B. Calder et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *ACM SOSP*, 2011.

[33] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: a New Class of Data Management Applications. In *VLDB*, 2002.

[34] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB*, 2003.

[35] U. Chajewska, D. Koller, and R. Parr. Making Rational Decisions Using Adaptive Utility Elicitation. In *AAAI*, 2000.

[36] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. Wernsing, and D. Rob. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. In *USENIX NSDI*, 2014.

[37] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR*, Jan. 2003.

[38] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases*, Jan. 2012.

[39] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based Scheduling: If You're Late Don't Blame Us! Nov. 2014.

[40] G. B. Dantzig. Discrete-Variable Extremum Problems. *Operations Research 5 (2): 266288*, 1957.

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 2009.

[42] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *ACM EuroSys*, 2012.

[43] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*, 2011.

[44] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-Resource Packing for Cluster Schedulers. In *ACM SIGCOMM*, 2014.

[45] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *Computing Research Repository*, Nov. 2015.

[46] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *ACM MobiSys*, 2016.

[47] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *ACM SIGMOD*, 1997.

[48] A. Hernando, R. Sanz, and R. Calinescu. A Model-Based Approach to the Autonomic Management of Mobile Robot Resources. In *International Conference on Adaptive and Self-Adaptive Systems and Applications*, 2010.

[49] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *USENIX NSDI*, 2011.

[50] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing Risk and Reward in a Market-Based Task Service. In *IEEE International Symposium on High Performance Distributed Computing*, 2004.

[51] J. Jaffe. Bottleneck Flow Control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[52] E. D. Jensen, P. Li, and B. Ravindran. On Recent Advances in Time/Utility Function Real-Time Scheduling and Resource Management. *IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2005.

[53] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable Approximate Query Processing with the DBO Engine. *ACM Transactions on Database Systems*, 33(4):23, 2008.

[54] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *ACM International Conference on Multimedia*, 2014.

[55] R. Johari and J. N. Tsitsiklis. Efficiency Loss in a Network Resource Allocation Game. *Mathematics of Operations Research*, 29(3):407–435, 2004.

[56] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, Í. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: towards automated SLOs for enterprise clusters. In *USENIX OSDI*, 2016.

[57] F. P. Kelly, A. K. Maulloo, and D. K. Tan. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, 49(3):237–252, 1998.

[58] J. O. Kephart. Research Challenges of Autonomic Computing. In *ACM ICSE*, 2005.

[59] M. Kristan, J. Matas, A. Leonardis, M. Felsberg, L. Cehovin, G. Fernandez, T. Vojir, G. Hager, G. Nebehay, and R. Pflugfelder. The Visual Object Tracking (VOT) Challenge Results. In *IEEE ICCV Workshops*, Dec. 2015.

[60] V. Kumar, B. F. Cooper, and K. Schwan. Distributed Stream Management Using Utility-Driven Self-Adaptive Middleware. In *IEEE ICAC*, 2005.

[61] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster based web services. In *Integrated Network Management VIII*, pages 247–261. Springer, 2003.

[62] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams. In *USENIX NSDI*, Mar. 2016.

[63] S. H. Low and D. E. Lapsley. Optimization Flow Control-I: Basic Algorithm and Convergence. *IEEE/ACM Transactions on Networking*, 7(6):861–874, 1999.

[64] P. Marbach. Priority Service and Max-Min Fairness. In *IEEE INFOCOM*, 2002.

[65] R. C. Merton. *Continuous-Time Finance*. Blackwell, 1990.

[66] D. Minarolli and B. Freisleben. Utility-Based Resource Allocation for Virtual Machines in Cloud Computing. In *IEEE Symposium on Computers and Communications*, pages 410–417, 2011.

[67] M. Morari and J. H. Lee. Model Predictive Control: Past, Present and Future. *Computers & Chemical Engineering*, 23(4):667–682, 1999.

[68] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *CIDR*, 2003.

[69] H. Nam and B. Han. Learning Multi-Domain Convolutional Neural Networks for Visual Tracking. *Computing Research Repository*, abs/1510.07945, 2015.

[70] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 289–302, 2007.

[71] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder. Heuristics for Vector Bin Packing. In *Microsoft Research Technical Report*, Jan. 2011.

[72] A. Rabkin, M. Arye, S. Sen, V. Pai, and M. Freedman. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *USENIX NSDI*, 2014.

[73] B. T. Ratchford. Cost-Benefit Models for Explaining Consumer Choice and Information Seeking Behavior. *Management Science*, 28, 1982.

[74] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2nd edition, 2003.

[75] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *Computing Research Repository*, abs/1409.1556, 2014.

[76] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *NIPS*, Dec. 2012.

[77] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. Chess. Server Virtualization in Autonomic Management of Heterogeneous Workloads. In *IFIP/IEEE International Symposium on Integrated Network Management*, 2007.

[78] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying Fit: Efficient Load Shedding Techniques for Distributed Stream Processing. In *VLDB*, 2007.

[79] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB*, 2003.

[80] G. Tesauro, R. Das, W. E. Walsh, and J. O. Kephart. Utility-Function-Driven Resource Allocation in Autonomic Systems. In *ICAC*, 2005.

[81] Y.-C. Tu, M. Hefeeda, Y. Xia, S. Prabhakar, and S. Liu. Control-Based Quality Adaptation in Data Stream Management Systems. In *Database and Expert Systems Applications*, 2005.

[82] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load Shedding in Stream Databases: a Control-Based Approach. In *VLDB*, 2006.

[83] C. J. Van Rijsbergen. Information Retrieval. *Butterworth, 2nd edition*, 1979.

[84] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The Power of Choice in Data-aware Cluster Scheduling. In *USENIX OSDI*, 2014.

[85] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments. In *ICAC*, 2011.

[86] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *ACM EuroSys*, 2015.

[87] E. Wandeler and L. Thiele. Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 80–89. ACM, 2005.

[88] Y. Wei, V. Prasad, S. H. Son, and J. A. Stankovic. Prediction-Based QoS Management for Real-Time Data Streams. In *IEEE RTSS*, 2006.

[89] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *ACM SOSP*, 2013.