Reducto: On-Camera Filtering for Resource-Efficient Real-Time Video Analytics

Yuanqi Li*, Arthi Padmanabhan*, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, Ravi Netravali
UCLA

ABSTRACT

To cope with the high resource (network and compute) demands of real-time video analytics pipelines, recent systems have relied on frame filtering. However, filtering has typically been done with neural networks running on edge/backend servers that are expensive to operate. This paper investigates *on-camera filtering*, which moves filtering to the beginning of the pipeline. Unfortunately, we find that commodity cameras have limited compute resources that only permit filtering via frame differencing based on low-level video features. Used incorrectly, such techniques can lead to unacceptable drops in query accuracy. To overcome this, we built Reducto, a system that dynamically adapts filtering decisions according to the time-varying correlation between feature type, filtering threshold, query accuracy, and video content. Experiments with a variety of videos and queries show that Reducto achieves significant (51–97% of frames) filtering benefits, while consistently meeting the desired accuracy.

CCS CONCEPTS

• **Information systems** → *Data analytics*; • **Computing methodologies** → **Object detection**;

KEYWORDS

video analytics, deep neural networks, object detection

ACM Reference Format:
Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, Ravi Netravali. 2020. Reducto: On-Camera Filtering for Resource-Efficient Real-Time Video Analytics. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3387514.3405874

1 INTRODUCTION

Video cameras are pervasive in today's society, with cities and organizations steadily increasing the size and reach of their deployments. For example, cities now deploy tens of thousands of cameras, each continually collecting and streaming rich video data [2, 15, 17, 39]. As camera deployments expand, organizations increasingly rely on analyzing *live* video feeds to guide long-running tasks such as traffic monitoring, customer tracking, and surveillance. Key to the success of such applications has been recent advances in computer vision, particularly neural network (NN)-based techniques for highly accurate object detection and recognition [22, 45, 47, 49, 65].

In a typical real-time video analytics pipeline [23, 41, 71, 79], a camera streams live video to cloud servers, which immediately run object detection models (*e.g.*, YOLO [62]) to answer user queries about that video. Such pipelines aim to deliver query results with *high accuracy* and *low latency*, but doing so is challenging due to the high compute and network resource demands of video streaming and NN-based analysis [23, 41, 79]. To make matters worse, organizations commonly operate and analyze video from many cameras [40], further amplifying computation and network overheads.

Significant work has been expended to improve the efficiency of video analytics pipelines [23, 24, 36, 41, 53, 79]. Across these systems, a prevailing (and natural) strategy is to improve efficiency by *filtering out* frames that do not contain relevant information for the query at hand [23, 24, 36, 43]. Conceptually, filtering out a frame requires understanding how that frame would affect a query result. To make such decisions without needing the actual query results (which would negate filtering benefits), existing systems employ various levels of *approximations* based on either (1) *compressed* object detection models (e.g., Tiny YOLO [62]) that compute lower-confidence results [36], (2) specialized *binary classification* models that eliminate frames that do not contain an object of interest [23, 43], or (3) simple *frame differencing* to eliminate frames whose low-level features (*e.g.*, pixel values) have not changed substantially (based on a static threshold) and are expected to produce the same results [24].

On-camera filtering. In this paper, unlike prior filtering approaches that typically run on edge [54] or backend servers, we seek to filter frames at the beginning of the analytics pipeline – *directly on cameras*. Like edge server approaches, on-camera filtering has the potential to alleviate not only backend *computation overheads* (by reducing the number of frames that must be processed by the backend object detector), but also end-to-end *network bottlenecks* between cameras and backend servers, particularly for wireless cameras [23, 33, 81]. Furthermore, an on-camera approach can also sidestep the management and cost overheads of operating edge servers [52, 63]. We note that in targeting on-camera filtering, our aim is to eliminate the reliance on edge servers for filtering by making use of currently unused resources. Our on-camera filtering techniques could also run on edge servers (if present), outperforming existing strategies while consuming fewer resources (§5).

Despite the potential benefits, our study of commodity cameras and surveillance deployments paints a bleak resource picture (§2.1). In contrast to edge servers, smartphones, or recent smart cameras that possess GPUs and AI hardware accelerators, deployed cameras often have low-speed CPUs (1 GHz) and modest amounts of RAM (256 MB). These resources preclude even compressed NNs for filtering (*e.g.*, Tiny YOLO runs at $<$ 1 fps), and instead can only tolerate specialized binary classification NNs or frame differencing strategies. Unfortunately, we find that these approaches are far more limited for filtering (§2.2). Binary classification strategies forego between 17-74% of potential frame filtering opportunities by filtering based on

* These authors contributed equally to this work.

SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-7955-7/20/08…$15.00
https://doi.org/10.1145/3387514.3405874

object presence rather than changes in query result, *e.g.*, a parking lot can contain parked cars, but the overall count or locations of the cars may not change. In contrast, existing frame differencing strategies consistently violate query accuracy requirements by filtering out necessary frames (reasons described below).

Goal and insight. This paper asks: can we integrate on-camera filtering into video analytics pipelines in a way that achieves most of the potential filtering benefits without violating accuracy goals? Due to the aforementioned filtering limitations inherent to binary classification, we turn to *frame differencing with low-level features*.

Our key insight is that the lack of accuracy preservation with existing frame differencing strategies is *not* a problem inherent to low-level features, but rather a problem of these features not being used appropriately. For example, Glimpse [24] filters by comparing pixel-level frame differences against a static threshold, and is unable to adapt to the heterogeneous queries (*e.g.*, detection, counting) and dynamic video content that analytics pipelines are faced with [41, 50]. This is because the same difference values may carry different meanings (in terms of changes in query results) for different video content and query types, *e.g.*, a traffic light may warrant a lower threshold than a busy highway. We assert that if we can (1) establish a correlation between feature types, their filtering thresholds, and query accuracy, and (2) dynamically adjust this correlation in response to changes in queries and video content, these cheap features can be surprisingly effective (more than NN-based techniques!) in indicating if filtering a frame will cause accuracy violations.

Reducto. Based on this insight, we developed Reducto, a simple and yet inexpensive solution to the real-time video analytics efficiency problem, that tackles three main challenges.

(C1) What low-level video features to use? The computer vision (CV) community [20, 24, 44, 46, 59, 60, 66, 68, 82, 83] has discovered a slew of low-level video features that extract frame differences [21], such as `Edge` and `Pixel`. To find the most appropriate features for on-camera filtering, we carefully studied a representative set of them (§3). An important observation we make is that the "best" feature (i.e., the one that most closely tracks changes in query results) to use varies across query classes more so than across different videos (see §4.2). This is because each feature uniquely captures a certain low-level video property; different query classes are interested in different video properties, and hence fit the best with different features. Based on this observation, the Reducto server performs offline profiling of historical video data to determine the best feature *for each query class*. The server notifies the camera of the feature it should use for each new query. Note that this is in contrast to existing strategies that always use the `Pixel` feature [24].

(C2) How to select filtering thresholds? Filtering frames using a differencing feature inherently requires cameras to select a parameter (i.e., a differencing threshold). Selecting the appropriate threshold is paramount as this value directly impacts the accuracy and filtering benefits of Reducto: too low of a threshold will sacrifice filtering benefits, while too high of a threshold may sacrifice accuracy. However, selecting this threshold value is difficult as the optimal threshold varies rapidly, on the order of seconds, due to the inherent dynamism in video content (§4.3). This rapid variance precludes the static thresholds used by prior systems [24], and also prohibits servers from making threshold decisions. Instead, threshold selection must be *adaptive* and be done by *cameras, online*.

To overcome these challenges, we use lightweight machine learning techniques to predict, at a fine granularity on the camera (*e.g.*, every few frames), which threshold to use for the selected feature. To do this, we train a cluster-based model *for each query and server-specified feature*, based on the observation that there is a strong correlation between the thresholds of the feature and the query accuracy (see §4.3). Clustering is done over all pairs of observed difference values (in the training set) and their highest feature values that hit the accuracy target. For each observed difference value, the camera selects the cluster in which the value falls and performs filtering using the average filtering threshold of that cluster. Note that such models are cheap regression models that can run in real time even under the camera's tight resource constraints.

(C3) What if the model is incomplete? The model used to predict thresholds for the selected feature may *lack sufficient coverage*, particularly when video characteristics drastically change (*e.g.*, rush hour starts). Unfortunately, how and where to detect such scenarios is challenging because detection relies on analyzing the accuracy of recent frames; for example, a change may have occurred if we see a significant accuracy drop for recent frames. However, the question is how to see the accuracy drop – the camera is unaware of the true accuracy as it does *not* run DNN object detectors, while the server only receives a subset of frames that the camera deems as relevant.

To address this issue, the Reducto camera constantly checks if the feature value for the current frame falls into an existing cluster in the model. If not, this indicates a potentially significant (and previously unseen) change in video characteristics, so the camera halts filtering and notifies the server to retrain the model. Note that our linear model is not only efficient to run but also efficient to *train*, enabling the server to train a new model *online* upon a request from the camera. Once trained, the new model is streamed back to the camera, which uses it until a subsequent update is required.

Result summary. We evaluated Reducto using multiple datasets of live video feeds covering 24 hours from 7 live traffic and surveillance cameras. We consider three classes of queries that track people and cars: *tagging*, *object counting*, and *bounding box detection*. Running on both Raspberry Pi and VM environments similar to commodity camera settings, we find that Reducto is able to filter out 51–97% of frames compared to traditional pipelines, resulting in bandwidth savings of 21–86%, 50-96% reductions in backend computation overheads, and 66–73% lower query response times. Importantly, in our experiments, Reducto achieves such filtering benefits while *always* meeting the specified query accuracy targets. Reducto also outperforms two recent video analytics systems: Reducto filters out 93% more frames than the FilterForward [23] edge filtering system, and achieves 37% more backend processing improvements than Chameleon [41]. Source code and experimental data for Reducto are available at https://github.com/reducto-sigcomm-2020/reducto.

2 MOTIVATION

This section explores two questions: (1) what compute/memory resources do commodity and state-of-the-art smart cameras possess (§2.1)?, and (2) how well do existing filtering techniques perform in such settings (§2.2)?

2.1 Smart Camera Resource Overview

To better understand the available resources for filtering on cameras, we analyzed publicly available information about multiple surveillance deployments, and conducted a small-scale study of local city and campus-wide camera installations. We found that there is a large resource divide between state-of-the-art cameras and commodity cameras which are widely deployed. Given that large-scale camera deployments are financially expensive to install and maintain, we do

RAM	Tiny YOLO: Object Detection			NoScope: Binary Classification		
	0.5 GHz	1.0 GHz	1.5 GHz	0.5 GHz	1.0 GHz	1.5 GHz
128 MB	*NA*	*NA*	*NA*	*NA*	*NA*	*NA*
256 MB	*NA*	*NA*	*NA*	*NA*	*NA*	*NA*
512 MB	0.19	0.39	0.64	28.39	56.25	85.9
1024 MB	0.20	0.42	0.66	26.9	58.36	84.1

Table 1: Inference speed (in fps) of compressed object detection and binary classification models in resource-constrained (camera-like) environments. *NA* means the model lacked sufficient resources to run. Pixel-based frame differencing (omitted for space) always ran at over 300 fps.

not anticipate an immediate overhaul that replaces commodity cameras with state-of-the-art ones. Instead, we expect a more gradual shift, and thus believe that camera-based filtering must consider the resource availability on both classes of devices. Note that we only focused on smart cameras, or those with some non-zero amount of general purpose compute resources; cameras without such resources are unable to handle any on-device filtering.

State-of-the-art smart cameras. Recent smart cameras commonly include AI hardware accelerators built into their processors, which speed up tasks such as DNN execution and video encoding [6, 14, 51, 67]. For example, Ambarella [14]'s CV22 System on Chip includes a quad-core processor (1 GHz) along with the CVflow vector processor designed explicitly for vision-based CNN/DNN tasks (*e.g.*, object tracking on 4k videos at 60 fps). Some cameras also ship with small on-board GPUs as an alternate way to accelerate similar workloads [11, 13]. For instance, DNNCam [11] ships with an NVIDIA TX2 GPU and 32 GB of flash storage and has a unit price of $2,418. These resources support real-time object recognition (*i.e.*, 30 fps or higher) and thus can be used to run NN-based filtering techniques directly on cameras.

Commodity and deployed cameras. In contrast to the promising filtering resources on state-of-the-art cameras, deployed surveillance cameras paint a much bleaker resource picture [16, 72, 75]. These cameras are considerably cheaper (generally $20–100), and ship with far more modest compute resources typically involving a single CPU core, CPU speeds of 1-1.4 GHz, and 64-256 MB of RAM. We verified the widespread deployment of such low-resource cameras by speaking with security teams for UCLA and Los Angeles—none of their deployed cameras included AI hardware accelerators, GPUs, or colocated edge servers, but they all possessed cheap CPU resources.

2.2 Limitations of Existing Filtering Techniques

We now explore how existing filtering techniques would fare on deployed smart cameras in terms of speed and filtering benefits. From §1, there are three main classes of existing filtering techniques:

- The first approach runs a *compressed object detection* model (*e.g.*, Focus [36]) to obtain approximate query results. This approach determines whether or not to send each frame for full model execution (rather than just sending the computed result) based on the confidence in the result that the compressed model produces.
- The second approach runs a cheaper and less general (*e.g.*, trained for a specific query and video content) *binary classification model*, which detects whether an object of interest (for the current query) is present in a frame or not. Only frames with the object of interest are sent to the server for processing (*e.g.*, FilterForward [23], NoScope [43]).
- The third approach is to compute pixel-level *frame differences* and filter out frames which, according to a static/pre-defined differencing threshold, are largely unchanged from their predecessor and expected to yield the same query result (*e.g.*, Glimpse [24]).

Speed. We started by evaluating the feasibility of running these three techniques *on cameras* for real-time filtering. We considered the canonical query of counting the number of cars in each frame. To evaluate frame differencing, we directly ran Glimpse's trigger frame selection algorithm using an arbitrary static threshold (more on this below) [24]. For compressed object detection, we used Darknet [61] to train a Tiny YOLO model [62] (8 convolutional layers) that only recognizes cars based on data labeled with YOLOv3. For binary classification, we trained a model that mimics the lightest classification model developed in NoScope [43]; this model has 2 convolutional layers (32 filters each) and a softmax hidden layer.[1] In both cases, training was done for each camera in our video dataset (§5.1) using 9 10-minute video clips from that camera.

We ran each technique on a new 10 minute clip from each camera under a sweep of resource configurations: a single core, 0.5-1.5 GHz CPU speed, and 128-1024 MB of RAM. Experiments were performed on a Macbook Pro laptop with a virtual machine that restricted resources to the specified parameters. Table 1 lists the filtering speeds in each setting. As shown, both NN models require at least 512 MB of RAM to operate, which precludes them from being used on many deployed cameras. Tiny YOLO is unable to achieve even 1 fps in any setting; note that even with the 11× speedup reported when also using background subtraction [36], Tiny YOLO is still far below real-time speeds. In contrast, when it has sufficient memory to run, the binary classification model consistently achieves real-time speeds, *e.g.*, 28 fps and 86 fps with 0.5 GHz and 1.5 GHz processors, respectively. Further, pixel-based frame differencing is able to hit real-time speeds across all camera settings. Thus, the rest of the section focuses on frame differencing and binary classification (which is at least tenable in some camera settings).

Filtering efficacy. Now that we have identified potential filtering candidates for our resource-constrained environment, we ask, how effective are they at filtering out frames? We discuss the two candidates, binary classification and frame differencing, in turn.

To evaluate the *potential* filtering benefits with binary classification, we analyzed object detection results (captured by YOLO [62]) for all videos in our dataset and computed the fraction of frames that do not contain any object of interest. Note that this represents an *upper bound* on the benefits that systems such as NoScope [43] and FilterForward [23] can achieve. As a reference, we also considered an *offline optimal* strategy, where each frame is filtered if its query result is identical to that of its predecessor. As shown in Figure 1, binary classification is very limited in its filtering abilities: compared to the offline optimal, binary classification filters out **73.5%** and **16.7%** fewer frames for the car and person queries, respectively. The reason is that there exist many scenarios where query results remain unchanged across consecutive frames but have non-zero objects of interest. For example, a car count will be consistently greater than 1 if the camera is facing parked cars — although one frame would be sufficient to accurately count the number of cars, binary classifiers would send all such frames since they all contain objects of interest. Figures 1(b) and 1(c) illustrate this property for several representative video clips.

Quantifying the potential filtering benefits with frame differencing techniques is challenging as they vary based on the tunable filtering threshold. Instead, the key limitation with respect to filtering efficacy is that existing frame differencing systems employ static and

[1] We consider NoScope rather than FilterForward here because FilterForward's reliance on a DNN for feature extraction precludes its use on a camera; we empirically compare Reducto's filtering with that of FilterForward in §5.

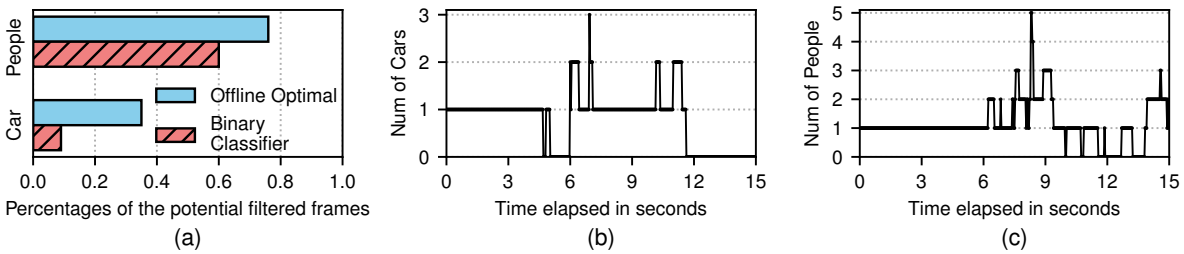


Figure 1: Binary classification yields limited filtering benefits: (a) the potential fraction of filtered frames, for standard people and car counting queries, as compared to an offline optimal (which filters based on query results), (b/c) representative video clips highlighting missed filtering opportunities with binary classification (*i.e.*non-zero but stable object counts).

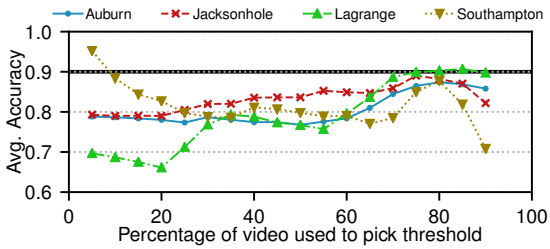


Figure 2: Glimpse [24] is unable to meet query accuracy requirements due to its use of a static threshold. The x-axis lists the fraction of each video used to select the best static threshold (i.e., max filtering while meeting the accuracy goal of 90%); the remainder of each video is used for evaluating the threshold.

pre-defined filtering thresholds, which complicate accuracy preservation. To illustrate the limitations of static thresholds, we evaluated Glimpse [24] on 4 random videos in our dataset. For each video, to pick the static threshold to use, we varied the amount of video (from the start) to use for selecting the *best possible static threshold*, i.e., the threshold that filtered the most frames while achieving the target accuracy. We then evaluated the query accuracy on the rest of the video that was not used for threshold selection. As shown in Figure 2, even with the best possible static threshold, Glimpse is almost never able to meet the target accuracy. Note that this is true even when we used 90% of each video for threshold selection, and despite the fact that this evaluation was done on adjacent video from the same camera. The reason, which we will elaborate on in §4.2, is that the best filtering threshold depends heavily on video content, which can be highly dynamic.

Key takeaway. These results collectively paint a challenging picture for on-camera filtering. Due to resource restrictions, to use existing techniques in real time, cameras must resort to either binary classification models or frame differencing. However, binary classification is largely suboptimal as it hides many filtering opportunities (*i.e.*, where objects are present but query results do not change across frames). Existing frame differencing strategies, on the other hand, use static thresholds and are unable to reliably meet accuracy targets.

To make effective use of frame differencing, the key question is whether it is possible to correlate frame differences with *pipeline accuracy* so that we can make a more informed decision as to whether a frame can be filtered out. We answer this question affirmatively in the next sections, where we describe how *lightweight differencing features* across video frames can serve as cheap monitoring signals that are highly correlated with changes in query results. If applied judiciously (and dynamically), these strong correlations enable large filtering benefits that are even comparable to those with the ideal baseline described earlier in this section.

Feature	On-camera tracking speed	Server tracking speed
SURF	1.27	26.55
SIFT	1.83	10.71
HOG	2.86	5.90
Corner	27.93	144.86
Edge	65.72	799.14
Area	71.80	1105.11
Pixel	308.60	2714.26

Table 2: Tracking speed (fps) for our candidate raw video features for frame differencing. High- and low-level features are shown on the top and bottom, respectively. Camera resources were 1 core, 1.0 GHz, and 512 MB RAM, while servers had 4 cores, 4 GHz, and 32 GB of RAM.

3 Filtering using Cheap Vision Features

Given the limitations of existing filtering strategies for on-camera filtering (§2.2), we seek a clean-slate approach to filtering based on frame differencing. In this section, we focus on identifying candidate features, and in §4, we present Reducto, which determines when and how to use those features for effective on-camera filtering.

Our goal is to identify *a set of raw video features* (1) that are cheap enough to be tracked on cameras in real-time, and (2) whose values are highly correlated with changes in query results for broad ranges of queries and videos (unlike prior systems that purely focus on detection [24]). We began with a representative list of differencing features used by the CV community [21], and grouped them in terms of the amount of computation required for extraction. *Low-level features* such as pixel or edge differences can be observed directly from raw images, but contain moderate amounts of noise. The main concern of using these features is whether or not this noise outweighs the true differencing values in certain cases. In contrast, *high-level features*, such as *scale-invariant feature transform* (SIFT) and *speeded up robust features* (SURF), aim to extract highly distinctive qualities of an image that are invariant to light, pose, etc., by analyzing properties such as local pixel intensities and shapes; these features have more semantic information, and many applications use such information to relate specific contents across frames. These features require multiple steps of computation on raw video values for extraction, but contain less noise as the noise is often smoothed out by the computation. The main concern of using high-level features is, clearly, their high extraction overheads.

Table 2 shows a representative list [37] of (high- and low-level) features we considered, and summarizes their computation overheads (in terms of fps) in both on-camera and server settings. Due to space constraints, we elide their detailed description and refer the interested reader to Table 8 (§A) and these survey papers [19, 37, 84] for details. Further, we note that other features may meet the aforementioned goals and can be easily plugged into Reducto.

Tracking speed. As shown in Table 2, tracking high-level features is far too slow to operate in real-time on cameras; many of these

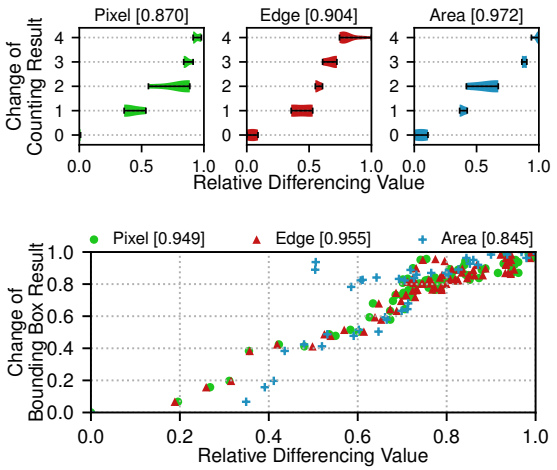


Figure 3: Correlations between differencing values and changes in query results for a 10 seconds clip in Auburn [3]. *Top* shows a car counting query where each line includes tick marks for min, max, and average feature value, with ribbons summarizing the distribution; *bottom* shows a car bounding box detection query. Results are for a random video. The legend lists the Pearson correlation coefficient per feature.

features cannot be extracted fast enough even on servers! For instance, SURF and SIFT are restricted to frame rates under 2 fps. In contrast, low-level features can be extracted on cameras at 28-309 fps. Overall, these results eliminate high-level features from consideration for on-camera filtering and direct us to focus on identifying the appropriate low-level features that can satisfy our correlation requirements. We also exclude the low-level `Corner` feature that falls just short of our real-time (30 fps) tracking goal.

Correlation with changes in query results. Recall that our goal is to use differencing features to "predict" whether a change in query results may occur. Thus, the features we use need not capture the precise *change in magnitude* between query results for two frames, but instead must have strong correlation with *whether* a change occurs. Figure 3 summarizes the correlation between the values for each feature that can operate in real-time and changes in query results. The two figures highlight the fact that the three low-level features `Pixel` (*i.e.*, directly compares pixels), `Edge` (*i.e.*, captures differences in contours of objects), and `Area` (*i.e.*, captures differences in areas) are indeed highly correlated (to varying degree–see §4) with changes in query results despite being potentially noisy on short time intervals. For example, on counting queries, a change of just 1 in object count leads to average changes of 0.42, 0.38, and 0.44 for the differencing values *w.r.t.* the `Pixel`, `Area`, and `Edge` features, respectively. As a reference, changes in these feature values are only 0.01, 0.11, and 0.09 when the count results are unchanged. For the bounding box query, even though the precise bounding box coordinates for an object change progressively across frames, the correlation remains strong, with easily visible differences in feature values for even minor adjustments in bounding box coordinates. Note that these trends hold for the other videos in our dataset as well (Figures 14-19 in §A).

4 Reducto Design and Implementation

4.1 Overview

Figure 4 depicts the high-level query execution workflow with Reducto. Currently, Reducto supports the three primary classes of

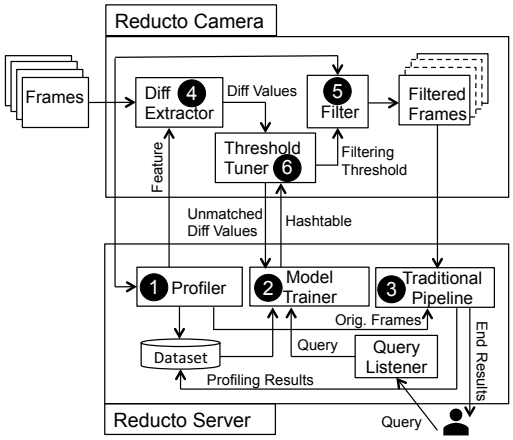


Figure 4: Overview of Reducto.

queries used in prior video analytics systems [42, 50]: tagging, counting, and bounding box detection. Descriptions of these queries are presented in §5.1.

***Offline* server profiling ❶ (§4.2).** The Reducto server first runs an *offline profiler* ❶ over several minutes of video that characterize the typical scenes for that camera. The profiler ❶ then runs traditional pipelines ❸ on that video and stores the results for subsequent feature selection. As this characterization data is collected, the profiler processes each frame in the video to extract the three low-level differencing features presented in §3. Our observation (Figure 5) is that there often exists a single feature that works the best for a query class across different videos, cameras, and accuracy targets. Hence, during profiling, the server finds the best feature for each query class that it wishes to support. At the end of this phase, the best feature for each query class is identified and stored at the server.[2]

***Per-frame* diff extraction ❹ (§4.4).** The camera does not stream any frames until it receives a query. Upon the arrival of a user-specified query and target accuracy, the server informs the camera of the best feature for that query. To filter, the diff extractor ❹ continuously tracks the differences in the specified feature between consecutive frames. The key question at this point is how to know, for each pair of consecutive frames, if the difference between them is *sufficiently insignificant* so that if the camera sends only the first one to the server (which reuses its query result for the second), the accuracy would not drop below the target. In other words, what is the right filtering threshold to use?

***Per-query* model training ❷ (§4.3).** To answer this question, the server uses a model trainer ❷ that quickly trains, for each query, a simple (regression) model characterizing the relationships between differencing values, filtering thresholds, and query result accuracy. The model is trained by performing K-means-based clustering over the original frames sent by the camera during a short period after the query arrives. Training typically takes several seconds to finish due to the simple models used. The generated model is encoded as a hash table, where each entry represents a cluster of differencing values whose corresponding thresholds are within the same neighborhood — each key is the average differencing value and each value is the threshold for that cluster which delivers the required accuracy. Together with the selected feature, this hash table is also sent to the camera for each query.

[2]Best features for common query types (*e.g.*, detection) can be pre-programmed or shared across servers, thereby avoiding profiling.

***Per-frame* threshold tuning ❻ and filtering ❺ (§4.4).** When the camera receives the feature and the hash table for the query, it starts filtering frames. To do so, the filter ❺ queries the threshold tuner ❻ for the threshold to use. The tuner looks up the hash table using the differencing value produced by the diff extractor ❹, finds the matching key-value entry, and applies the listed threshold (*i.e.*, the value of the entry).

***Occasional* model retraining ❷ (§4.5).** In some cases, the differencing value may not map to any table entry (*e.g.*, the distances between the value and the existing keys are too large). This indicates a potential change in video dynamics and implies that the new scene cannot be effectively captured by the existing clusters. As an example, the burst of cars at the start of rush hour can lead to a differencing value significantly different from those seen during training. In these cases, the threshold tuner ❻ sends these unmatched values (together with their original frames) to the model trainer ❷, which adds these new data points into its dataset (along with the generated query results), re-trains the model, and sends the tuner ❻ an updated hash table to ensure that the model stays applicable despite changes in the video.

The user can decide whether the camera deletes the frames that are not sent to the server. If the user wishes to save the frames for later retrieval or retrospective queries [36, 76], the camera archives all frames onto cheap local storage.

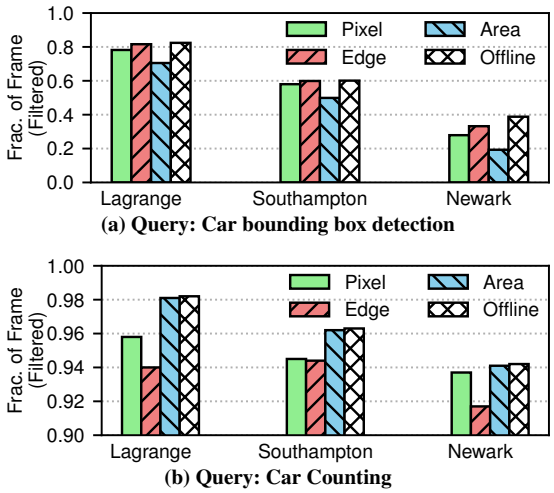
Tracking granularity. Since Reducto's goal is to ensure that the specified accuracy is continuously met, Reducto analyzes differencing features at the granularity of *video segments* rather than individual frames. Video segments represent small windows (*e.g.*, N seconds) of consecutive frames. Analyzing features over segments enables Reducto to smooth out intermittent noise in feature values (§3). Thus, the Reducto camera buffers frames for each segment and selects the filtering threshold for the feature (using the hash table) when all frames of the segment arrive. The camera then applies the filter with the selected threshold to each buffered frame to decide whether it needs to be sent.

Selecting the right segment size is important: a small segment size is susceptible to inaccuracy due to noisy feature values, while a large segment size better handles noise but requires more frames to be buffered prior to making filtering decisions (delaying query results). We empirically observe that $N = 1$ *second* sufficiently balances these properties, and we present results analyzing how sensitive Reducto's results are to segment size in §5.

Discussion. We note that the presented design for Reducto (and our current implementation) focuses on single queries for a given camera's video feed. However, the described filtering approach can be extended to handling multiple queries in a straightforward manner: filtering decisions can initially be made independently per query (as described), and then aggregated by taking the union of frames deemed important for *any* query. Additionally, we note that Reducto currently targets detection-based queries that do not carry over information across frames. For instance, in its current form, Reducto does not support activity detection queries. We leave support for these more complex queries to future work.

4.2 Feature Selection via Server-side (Offline) Profiling

During the offline profiling phase, the profiler ❶ uses several minutes of representative video frames to compute, for each frame, (1) the object detection results using traditional pipelines ❸, and (2) the low-level differencing values for each candidate feature. Using these results, the server determines which feature the camera should use.



(a) Query: Car bounding box detection

(b) Query: Car Counting

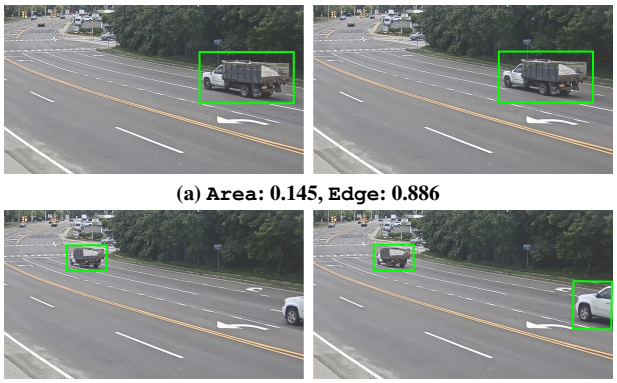
Figure 5: Filtering efficacy of the 3 low-level features across 3 videos and 2 queries. Y-axis reports the percentages of frames filtered (the higher the better). Across these videos, `Area` is best for counting, but `Edge` is best for bounding box detection. Results used YOLO and a target accuracy of 90%.

The best feature to use is the one that maximizes the filtering benefits (*i.e.*, filters out the most frames) while meeting the accuracy requirement specified by the user. In order to identify the best feature, the server analyzes the profiling results on a per-segment basis. For each segment and for each feature, the server considers a large range of possible thresholds for the feature. For each candidate feature, the server then aggregates the largest filtering benefits (obtained from using the best performed threshold on each segment) across all segments. These aggregated benefits are used to pick the best feature for each query class supported by Reducto.

Observation 1: Interestingly, we observe that the best feature tends to vary across query classes, but remains stable across cameras, videos, and target accuracies for each class. For example, consider Figure 5, which shows that the `Area` feature provides the largest filtering benefits for counting queries across 3 representative videos. In contrast, the `Edge` feature provides the most filtering benefits for bounding box queries. For reference, `Area` outperforms `Pixel` and `Edge` on counting queries by **11–70%**; and `Area` trails the two other features by **5–41%** on bounding box queries.

The reason is that different features and queries operate at different granularities, and their values change at varied levels with respect to changes across frames. In other words, minor frame differences may affect certain queries and feature values more than others. For example, consider the definitions of the `Area` and `Edge` features (Table 8 in §A). `Area` compares the size of the areas of motion across frames, but does not consider the distance that those areas move. In contrast, `Edge` is finer-grained and observes changes in the locations of the edges of objects.

Figure 6 illustrates how these divergences affect the suitability of each feature with respect to filtering for two query types: bounding box detection and counting queries. As shown in Figure 6(a), *any* motion for an object of interest can alter the corresponding bounding box coordinates. Whereas the `Area` feature is largely insensitive to such minor changes (making it ill-suited for filtering, as it suggests that the query result should not change), the `Edge` feature will detect (even minor) movements to the object's edges and yield a high differencing value. In contrast, Figure 6(b) shows that the coarse-grained nature of the `Area` feature is well-suited for counting



(a) **Area: 0.145, Edge: 0.886**

(b) **Area: 0.830, Edge: 0.908**

Figure 6: Car detection results for two sets of adjacent frames from the Southampton video; subcaptions list the corresponding differencing feature values. For bounding box detection queries, slight variations can change the query result; Edge picks up on these subtle changes (top) but Area does not. In contrast, counting queries are better served by Area, which reports significant differences when counts change (bottom), but not when counts stay fixed (top).

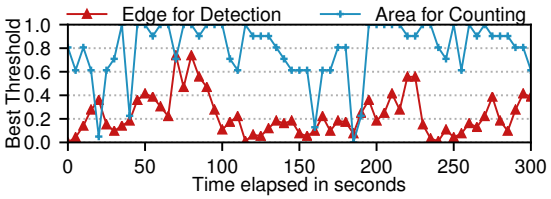


Figure 7: Best filtering thresholds vary across (even adjacent) video contents. This experiment used the Southampton video, and two features over two queries (Area over counting and Edge over bounding box detection, both for cars); the target accuracy is 90%. Trends hold for other queries, videos, and accuracy targets (§A).

queries: when a new object enters a scene, it represents a new area of motion and results in a high differencing value. Thereafter, until the object count changes, the `Area` value remains low. The `Edge` feature, on the other hand, reports significant frame differences even when the overall object count stays unchanged (*e.g.*, Figure 6(a)), making it too conservative for filtering for counting queries.

We verified that this stability in best feature holds across other query classes (*e.g.*, tagging), objects of interest (*e.g.*, people), target accuracies (*e.g.*, 80%), and detection models (*e.g.*, Faster R-CNN) as well; results are shown in §A (Figures 21- 22) due to space restrictions. This observation implies that the server need not select features dynamically, and instead can make one-time feature decisions for all the query classes it wishes to support.

4.3 Model Training for Threshold Tuning

Knowing which feature to use is not enough; the camera also needs to know how to tune the filtering threshold for the feature so that filtering does *not* create unacceptable degradation in query accuracy.

Observation 2: While for each query class the best feature remains stable over time, the best threshold (*i.e.*, highest one which meets the accuracy target) to use for a given feature does not. Figure 7 illustrates this point for two different query classes and their corresponding best features. As shown, the best threshold for each feature varies rapidly, on the order of segments. Thus, the camera needs a way to dynamically tune the threshold of the feature to prevent any unacceptable accuracy drops. However, making this decision requires understanding how different thresholds relate to the accuracy of query results. If the server can establish a mapping between

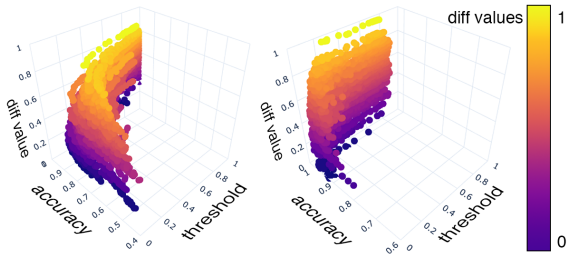


Figure 8: Simplified clustering results for two car queries: detection (left) and counting (right) over the Jackson Hole video.

differencing values, thresholds, and result accuracy, the camera can use such information to quickly find the best thresholds to use.

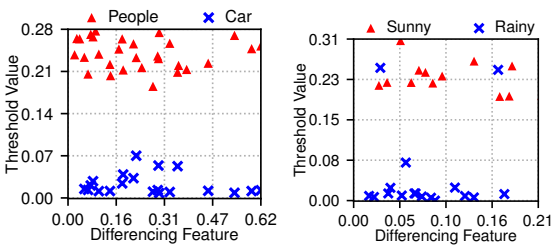
To generate this mapping, the server requires the camera to send unfiltered frames over a short window right after the query is registered. These frames are used as an initial training set — the server runs the full pipeline on them, producing *complete results* about each segment of frames — including query accuracy, fraction of frames filtered, and extracted feature values — for a broad range of candidate thresholds. For each segment, we compose a 29-dimension vector for the segment. This vector contains the average differencing feature value across the pairs of adjacent frames in the segment, *i.e.*, a 1-second segment contains 30 frames (30 fps), resulting in 29 differencing values. We then add a data point to our training set for each candidate threshold; each data point is keyed at the corresponding 29-dimension differencing vector, and labeled with the tested threshold and the resulting query accuracy. Lastly, we remove any data points whose accuracy falls below the target accuracy for the query. The server then clusters these data points using the standard K-means algorithm based on their differencing vectors. Selecting the number of clusters entails balancing the overhead of the clustering algorithm and robustness of the resulting clusters to noisy inputs; we empirically observe that setting a target of 5 clusters strikes the best balance between these factors, and we leave an exploration of more adaptive tuning strategies to future work [31, 32].

Figures 8 illustrates the clustering results for a random 10-minute clip. As shown, the data is highly amenable to such clustering, and the results follow a fairly intuitive pattern: to meet a given accuracy target, the filtering threshold *decreases* as the differencing feature value *increases*. This is because high feature values imply that frames are changing significantly (*e.g.*, due to motion) — these changes give Reducto a reason to believe that the query result *may* change and thus the camera needs to send more frames.

Once clustering is done, the results are encoded into a hash table where each entry encodes information about a cluster — keys represent aggregated differencing values in the cluster augmented with the size measurement of the cluster (discussed shortly), while values represent the aggregated labels (*i.e.*, thresholds). In particular, each key is of the form $\langle center, variance \rangle$, where *center* is a 29-dimension vector computed by performing element-wise averaging across the vectors in the cluster and *variance* is another 29-dimension vector where the i^{th} element represents the *longest distance* between the i^{th} elements in any possible pairs of data points in the cluster. In other words, *center* encodes the *central point* of the cluster while *variance* measures the *size* of the cluster (*i.e.*, how far apart data points can be). Each value is the averaged filtering threshold of all data points in the corresponding cluster.

4.4 On-Camera Filtering

To filter out frames in real time, the camera continuously tracks differencing values for the selected feature. At the end of each



(a) Different objects **(b) Different video contents**

Figure 9: Offline training would be limited: comparisons of hash table entries (*i.e.*, clusters) between (a) detection of different objects (*i.e.*, people and car) and (b) different video contents (*i.e.*, sunny and rainy) show that the clusters differ significantly under these circumstances; results were obtained from analyzing the entire Auburn video.

segment, the camera decides which frames in the segment should be sent to the server. To do this, the camera simply looks up the hash table provided by the server. Specifically, the camera composes a similar 29-dimension vector a for the segment (by averaging the vectors for the constituent frames) and queries the hash table. The lookup algorithm finds the key-value pair $\langle\langle c, v\rangle, l\rangle$ such that (1) the euclidean distance between a and c is $\leq$ to that between a and any other key in the hash table, and (2) the distance between the i^{th} elements in a and c is $\leq$ the i^{th} element in v, which represents the longest distance for the i^{th} dimension in the cluster. This indicates that the new data point falls well into the cluster (*i.e.*, video contents changed in a similar way as in the past). Once such a table entry is found, the camera uses the threshold (*i.e.*, the entry's value) to filter out frames in the segment. The remaining frames are compressed using H.264 at the original video's bitrate, and sent to the server.

4.5 Online Model Retraining

In scenarios where no matching key-value pair can be found (*i.e.*, a does not belong to any cluster listed in the hash table), Reducto speculates that the current video properties are different from those used by the server to compute the table. In order to prevent degradations to below the accuracy target, the camera halts filtering and sends all frames in the segment to the server. The server computes query results over these original frames so no accuracy loss can occur. The server also adds these unfiltered frames to its dataset and re-clusters. The updated hash table is streamed back to the camera once it is computed, and upon reception, the camera resumes filtering.

Online vs. offline training. In our implementation, model training (*i.e.*, hash table generation) and retraining (*i.e.*, hash table updates) are handled in the same way. Upon receiving a query, the server sends the selected feature and an *empty hash table* to the camera. The threshold tuner ❻ would not find any matching entry in the table and thus would have to stop filtering and send all frames for model training. Similarly, retraining is also triggered by misses in table lookups. A question the careful reader may ask is: is it necessary to perform model training/retraining online? In other words, does an *offline-learned* linear model suffice? To answer this question, we compared the hash table entries (*i.e.*, clusters) generated under different queries and video contents. The results are illustrated in Figure 9. As shown, the clusters (and threshold values) differ significantly under these different circumstances, indicating that an offline training approach would be limited for unseen queries and video contents. Thus, even though Reducto's initial hash table can benefit from historical video data, in order to cope with the fact that it is impractical to foresee all possible queries and video properties, Reducto also supports online training/retraining.

Camera location	FPS	Resolution
Jackson Hole, WY [5]	15	1920×1080
Auburn, AL [3]	15	1920×1080
Banff, Canada [1]	15	1280×720
Southampton, NY [10]	30	1920×1080
Lagrange, KY [7]	30	1920×1080
Casa Grande, AZ [4]	30	640×360
Newark, NJ [8]	10	640×360

Table 3: Summary of our video dataset.

5 Evaluation

5.1 Methodology

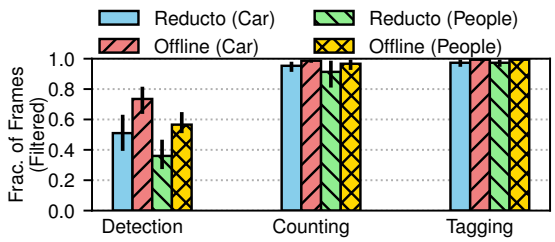
Table 3 summarizes the video dataset on which we evaluated Reducto. Our dataset comprises public video streams from 7 live surveillance video cameras deployed around North America. From each data source, we collected 25 10-minute video clips that cover a 24-hour period. As a result, video content for a given camera varied over time with respect to illumination, weather characteristics, and density of people and cars. Video content also varied across cameras *w.r.t.* quality, orientation (*e.g.*, certain cameras were mounted on traffic lights, while others were recording streets from a side angle), and speed/density of objects (*e.g.*, rural vs. metropolitan). Figure 20 (§A) provides some example screenshots.

In our evaluation, we considered three main classes of queries, each with a unique definition of accuracy:

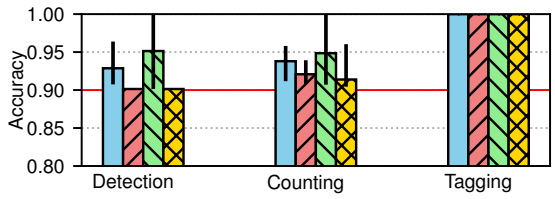
- **Tagging queries** return a binary decision regarding whether or not an object of a given type appears in a frame. Accuracy is defined as the percentage of frames which are tagged with the correct binary value.
- **Counting queries** return a count for the number of objects of a given type that appear in a frame. Accuracy for a frame is defined as the absolute value of the percent difference between the correct and returned values.
- **Bounding box detection queries** return bounding box coordinates around each instance of a given object that is detected in a frame. Accuracy is measured using the standard mAP metric [28] that evaluates, for each returned bounding box, whether the enclosed object is of the correct type and whether the bounding box has sufficiently large overlap (intersection over union) with the correct box.

We ran each query class across our entire video dataset for two types of objects: people and cars. Unless otherwise noted, ground truth for all video frames and queries was computed using YOLO [62]. Reported accuracy numbers for each of Reducto's segments were computed by averaging the accuracy values for each of the segments' constituent frames; Reducto used segments of 1 second unless further specified.

Server components ran on an Ubuntu Amazon EC2 p3.2xlarge instance with 8 CPU cores and 1 NVIDIA Tesla V100 GPU. The camera was either a Raspberry Pi or a VM whose resources were provisioned based on the RAM and CPU speeds observed in our study of deployed cameras (§2); the recorded video was fed into the camera sequentially and in real time. For brevity, in the VM scenario, we present results for the resource configuration of 256 MB of RAM and a 1 GHz CPU (single core). However, we note that the reported trends persist in the other settings in Table 1. The camera and server were connected over a variety of live (LTE and WiFi) and emulated networks via Mahimahi [55].

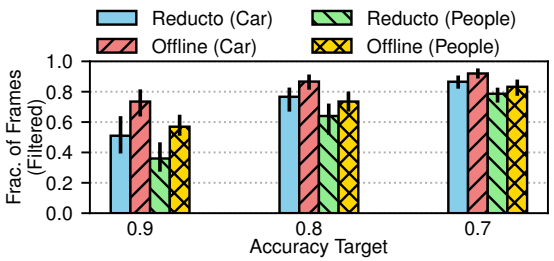


(a) Fraction of filtered frames

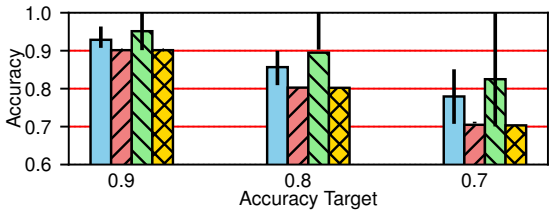


(b) Achieved accuracy

Figure 10: Comparing Reducto and the offline optimal filtering strategy for three query types and two objects of interest across our entire dataset. Results are for the distribution across all Reducto segments. Each bar reports the median with the error bar showing the 25th and 75th percentiles. The target query accuracy is 90%.



(a) Fraction of filtered frames



(b) Achieved accuracy

Figure 11: Analyzing Reducto's results for different accuracy targets. Results are for bounding box detection queries of cars and people across our entire video dataset.

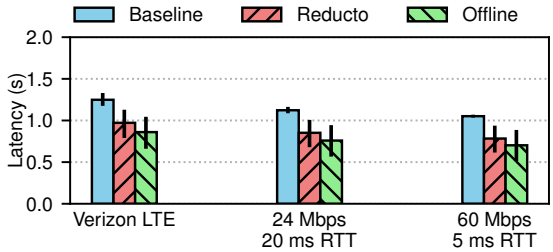


Figure 12: Distribution of per-frame query response times on different camera-server networks. Each bar reports the median, with error bars showing 25th and 75th percentiles. Results are for detecting cars on our entire video dataset, and the target accuracy is 90%.

5.2 Overall Performance

To understand Reducto's filtering efficacy, we first compared it to a *baseline* video analytics pipeline in which cameras do not perform any filtering, and servers compute query results for all frames. To contextualize our results, we compared both systems with the *offline optimal* (§2) that uses actual query results to perfectly filter out each frame whose result sufficiently matches that of its predecessor. Note that the offline optimal represents an *upper bound* of what Reducto can hope to achieve without direct knowledge of query results.

Figure 10a shows that, across our entire video dataset, a target accuracy of 90%, and a variety of query types, Reducto is able to filter out a median of 51-97% of frames, which is within 2.8-36.7% of the offline optimal. As expected, filtering benefits vary based on query type, object of interest, and video. For instance, across the dataset, Reducto filtered out a median of 97% and 51% for tagging and detecting cars, respectively. This follows from the fact that the bounding box position for a moving car changes very *quickly* (*e.g.*, across consecutive frames), while the presence of any car (*i.e.*, what a tagging query searches for) often remains stable for long durations. Indeed, almost all frames could be filtered for tagging queries because query results changed very *infrequently*.

Despite this aggressive filtering, Figure 10b illustrates that Reducto is able to *always* deliver per-segment accuracy values above the target (90%). Reducto consistently delivers higher accuracy than the offline optimal, which nearly perfectly matches the target (due to knowing the ground truth). This is a result of Reducto's cautious selection of filtering thresholds (§4.4). In other words, whereas Reducto conservatively selects the filtering threshold to overshoot the accuracy target (filtering out fewer frames than possible), the offline optimal perfectly hovers over the target, thereby optimizing the fraction of frames that can be filtered within the accuracy constraint.

Varying accuracy targets. We also evaluated how Reducto's filtering benefits vary with different accuracy targets. In this experiment, we primarily focused on bounding box detection queries which show the largest variation across accuracy targets due to their fine-grained nature. As expected, Reducto's filtering benefits increase as the accuracy target decreases (Figure 11). For instance, when the object of interest is people, filtering benefits rise from 36% to 79% as accuracy drops from 90% to 70%. The reason is that Reducto can be more aggressive with filtering and tolerate more substantial inter-frame differences in feature values, without violating a lower accuracy target. Importantly, Reducto always met the specified accuracy target.

System	Accuracy (%)	Fraction Filtered (%)	Bandwidth Saving (%)	Backend Processing (fps)
Baseline	100.00	0.00	0.00	41.13
Reducto	90.49	53.42	22.30	86.21
Optimal	90.16	72.80	39.33	140.04

Table 4: Breaking down the impact of Reducto's filtering on network and backend computation overheads. Results are for detecting cars and are averaged across our entire dataset. The target accuracy is 90%.

Query response times. The promise of frame filtering is ultimately to reduce resource overheads and deliver (highly accurate) query results with low latency. Figure 12 illustrates that, across several network conditions, Reducto is able to reduce median per-frame response times by 22-26% (0.26-0.28s) compared to the baseline pipeline; Reducto's response times are within 12–13% the offline optimal. Table 4 further breaks down these query response time speedups into network and backend improvements. As shown, Reducto's filtering results in an average bandwidth saving of 22% compared to the baseline pipeline; backend processing speeds, on the other hand, more than doubled due to the decrease in frames to be processed.

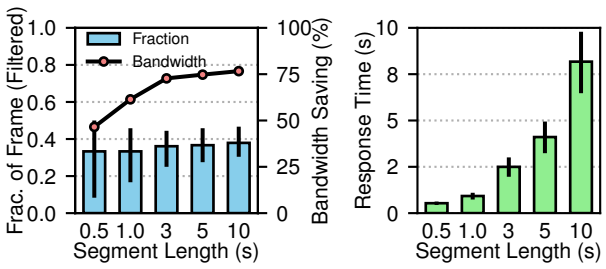


Figure 13: Impact of Reducto's on-camera segment length on filtering benefits for object detection of cars on two randomly selected videos in our dataset; the target accuracy was 90%. Results are distributions across segments, with bars representing medians and error bars spanning 25th to 75th percentile.

On-camera evaluation. Our experiments thus far have considered a resource-constrained VM as the camera component of the video analytics pipeline. In order to evaluate the feasibility of running Reducto directly on a camera, we replaced the aforementioned VM in our pipeline with a Raspberry Pi Zero [9] that embeds a 1.0 GHz single-core CPU and 512 MB of RAM; this resource profile falls into the range of on-camera resources that we observed in our study of commodity cameras and surveillance deployments (Table 1 and §2.1). We note that Raspberry Pi computing boards are intended to run alongside sensor devices (*e.g.*, cameras) to provide minimal and affordable computation resources. We implemented Reducto on the Raspberry Pi using OpenCV [12] for feature extraction and frame differencing calculations, and a hash table lookup to make threshold selections and filtering decisions. Unfiltered frames were encoded using Raspberry Pi's hardware-accelerated video encoder for the H.264 standard. As we did with the VM, we fed in each recorded video in Table 3 sequentially and in real-time to the Raspberry Pi.

Overall, we observed that Reducto's filtering results for each video *identically* matched those from our VM-based implementation (*i.e.*, results in Figure 11). More importantly, Reducto was able to operate at 47.8 fps on the Raspberry Pi, highlighting the ability to perform real-time filtering. Digging deeper, we found that the bulk of the processing overheads were due to per-frame feature extraction with OpenCV; this task could operate at 99.7 fps, as compared to frame differencing calculations and hash table lookups that ran at 129.5 and 318.6 fps, respectively.

Sensitivity to segment size. We varied the segment size that Reducto used for on-camera filtering between 0.5-10 seconds. Figure 13 illustrates three trends. First, as segment size decreases, Reducto's median filtering benefits are largely unchanged. We note that the distribution of filtering benefits widens largely because there are fewer opportunities to experience different video conditions within a small segment. For instance, a segment of 4 frames may be mostly unchanged and require only 1 frame to be sent; such a filtering fraction is less likely as segment sizes grow. Second, as segment size increases, bandwidth savings increase. This is because larger segments enable more aggressive bandwidth savings from standard video encodings: more frames can avoid redundant transmission due to fewer key frames. Third, per-frame query response times grow as segment sizes increase. Recall that Reducto cameras only filter out and ship frames to servers *after* a segment is captured. Thus, frames that are early in a given segment must experience query response times that are at least as long as the segment size.

Sensitivity to different object detection models: We verified (Figure 23 in §A) that Reducto's overall filtering benefits and accuracy preservation persist across other models, *i.e.*, SSD ResNet, Faster R-CNN with Inception ResNet.

System	Accuracy (%)	Fraction Filtered (%)	Bandwidth Saving (%)	Backend Processing (fps)
Reducto	90.49	53.42	22.30	86.21
Tiny YOLO	90.22	24.46	13.68	53.66
FilterForward	90.10	27.70	14.49	56.32

Table 5: Comparing Reducto with existing real-time filtering systems. Results are for detecting cars in our entire dataset, and the target accuracy is 90%.

5.3 Comparison with Other Filtering Strategies

We also compared Reducto with two existing filtering approaches that are both able to consistently meet a desired accuracy target; recall from §2.2 that Glimpse [24] was unable to do so due to its static threshold approach.

Tiny YOLO. We considered a filtering system that computes approximate query results using a compressed detection model (Tiny YOLO). Frames whose result confidence is sufficiently high (80% in this experiment; tuned to the target accuracy) can benefit from (1) filtering, if the frame does not contain an object of interest, or (2) result reuse which avoids running the backend detector. This approach is *loosely* inspired by Focus's ingest-time processing [36] which targets retrospective queries; we omit Focus' clustering strategy, which is primarily useful for the tagging queries that Focus targets. We trained a Tiny YOLO model on 90 minutes of video from each feed in our dataset to detect cars; we then tested on separate 30-minute clips from the same feed.

FilterForward. We also ran FilterForward [23], a binary classification-based filtering system designed for edge servers. With FilterForward, micro-classifers ingest feature maps computed by different layers of a full-fledged object detector, and determine whether an object of interest is present or not in each frame; if not, the frame is filtered at the edge server. FilterForward reports comparable performance to NoScope [43], which is intended for retrospective queries. In our experiments, we directly ran FilterForward's open-source code and trained a micro-classifer in the same way as Tiny YOLO above.

Results. Table 5 shows that Reducto achieves significantly larger filtering benefits compared to both systems. Average frame savings with Reducto are 53.42%, while Tiny YOLO and FilterForward filter only 24.46% and 27.7%, respectively. This translates to improvements of 54-63% and 53-61% in network bandwidth expenditure and backend processing costs, respectively. Key to this performance discrepancy is the limitation in binary classification-based filtering (§2.2). We note that, unlike Reducto, neither FilterForward nor Tiny YOLO can run in real time on a camera; the filtering benefits described here, however, are unaffected by resource constraints.

5.4 Comparison with Complementary Video Analytics Systems

We also compared Reducto with two systems that improve the efficiency of real-time video analytics pipelines, CloudSeg [57] and Chameleon [41]. Each system aims to improve a different aspect of the analytics pipeline, and both approaches are conceptually complementary to Reducto.

CloudSeg. CloudSeg uses super resolution techniques to significantly compress live video prior to shipping it to servers for analytics tasks; super resolution models at the server are used to (mostly) recover the original high resolution image, which is then fed into the analytics pipeline. To implement CloudSeg, we used bilinear interpolation in OpenCV [12] to compress all frames by 2-4× on

System	Accuracy (%)	Fraction Filtered (%)	Bandwidth Saving (%)	Backend Processing (fps)
Reducto	90.49	53.42	22.30	86.21
Cloudseg 2x	85.78	0.00	56.82	32.33
Cloudseg 4x	60.86	0.00	82.46	31.13
Reducto	99.10	97.11	80.23	1360.71
Cloudseg 2x	99.67	0.00	56.82	32.19
Cloudseg 4x	99.55	0.00	82.46	31.57

Table 6: Comparing Reducto with CloudSeg [57]. Results are for detecting cars (top) and tagging cars (bottom), both with an accuracy target of 90%.

System	Accuracy (%)	Bandwidth Saving (%)	Backend Processing (fps)
Baseline	100.00	0.00	13.04
Reducto	90.08	32.16	103.40
Chameleon	92.00	0.00	93.75

Table 7: Comparing Reducto with Chameleon [41] on a car counting query. The target accuracy was 90%.

our camera VM. We then used the same super resolution model as CloudSeg, CARN [48], to recover the original video on the server.

As shown in Table 6, we initially tried a compression factor of 4× for CloudSeg. Despite heavy tuning, we were unable to hit our accuracy target for detection. Thus, we focused our discussion on the 2× compression which narrowly misses the 90% accuracy goal. As expected, for detection, CloudSeg achieves superior bandwidth savings compared to Reducto (57% compared to 22%). However, CloudSeg does not filter out frames, and instead opts purely for compression, *i.e.*, all frames must go through costly backend processing. As a result, Reducto's filtering results in 2.7× improvements in backend processing overheads. Results for tagging follow a similar pattern, but we note that Reducto achieves superior bandwidth savings because most frames can be filtered out; for the same reason, the discrepancy in backend processing overheads is more pronounced. These approaches are complementary in that Reducto can also apply super resolution encoding on cameras (in real time) after filtering.

Chameleon. Systems such as Chameleon [41] and VideoStorm [79] reduce backend computation costs by profiling different configurations of pipeline knobs (*e.g.*, video resolution, frame sampling rate, etc.) and selecting those that are predicted to minimize resource utilization while meeting the user-specified accuracy requirement. Chameleon improves upon VideoStorm in that it profiles periodically rather than once, upfront. To implement Chameleon, we considered configurations based on the following knobs: 5 levels of image resolution (1080p, 960p, 720p, 600p, 480p), 2 pre-trained object detection models (Faster R-CNN and YOLOv3), and 5 levels of frame rate (30fps, 10fps, 5fps, 2fps, 1fps). For each video in our dataset, we selected the best configuration for each 4-second segment (which is Chameleon's profiling rate); we used the same segment size for Reducto. For ease of implementation, profiling for each segment was done offline. For fair comparison, Reducto used the more expensive Faster R-CNN model, which Chameleon treats as ground truth.

As shown in Table 7, both systems significantly outperform the baseline pipeline, but Reducto achieves 37% better backend processing speeds. Further, by filtering directly at the video source, Reducto is also able to achieve network bandwidth improvements that Chameleon cannot. While both systems reap filtering benefits (*e.g.*, decreased sample rates with Chameleon), they are largely complementary in that Chameleon considers knobs which Reducto does not, *i.e.*, detection model, image resolution.

6 Related Work

Edge-cloud split. One class of edge-based approaches, exemplified by FilterForward [23], sends frames to the server based on the objects present, approximated by a light-weight neural network running at the edge. Wang et. al. [69] use MobileNet [35] on drones. Similarly, Vigil [80] uses an edge node that can run object detection and sends frames with a higher object count. Gammeter et al. [30] send a frame only when object tracking on the mobile device has consistently low confidence. Alternatively, a server could receive partial information from the edge and decide whether it needs more based on inference results [57]. While this model can save significant bandwidth, round trips between server and edge impedes the system's ability to respond to queries in real time. Chinchali et al. [25] also use a server-driven approach, but the edge device can adapt (for DNN input) both the information it sends and the encoding method based on feedback from the server. Finally, Emmons et al. [27] propose a DNN split inference, where the edge runs as many layers as possible before sending the intermediate values to the cloud. In contrast to all of these solutions, Reducto, is aimed at cameras with resources that do not even support small NNs.

Resource scheduling. VideoStorm [79] and Chameleon [41] profile pipeline knobs to identify cheap and accuracy-preserving configurations (§5), while VideoEdge [38] also considers placement plans over a hierarchy of clusters. DeepDecision [58] and MCDNN [34] treat resource scheduling as an optimization problem and maximize key metrics such as accuracy or latency, while LAVEA [77] allocates computation among multiple edge nodes, optimizing for latency. These systems are largely complementary to Reducto, as the resource-accuracy tradeoff could be further tuned on the set of Reducto-chosen frames. Another complementary class of systems focuses on efficient GPU task scheduling [64].

Querying video. NoScope [43], BlazeIt [42], and Focus [36] lower resource consumption for efficient retrospective video querying. In contrast, Reducto uses the relationship between video features and query result, rather than presence of objects, for an early determination of relevant frames.

Computer vision. The idea of filtering frames based on their features is widely seen in the CV community [18, 29, 44, 60, 70, 73, 74]. Many of these methods are used for the task of retrospectively classifying or recognizing events in videos [26, 56, 74]. AdaFrame, for example, trains a Long Short-Term Memory network to adaptively select frames with important information. Others are used for key frame extraction [60, 78]. The are two main barriers to directly using these methods to filter frames in a setting like Reducto. One is that the task of choosing which frames a DNN should process is different from choosing frames for video classification or key frame extraction, because the importance of a frame in Reducto is determined solely by whether the DNN output changes. Second, these methods rely on neural networks that are too expensive to run on a camera.

7 Conclusion

This paper presents Reducto, a video analytics system that supports efficient real-time querying by leveraging previously unused resources to perform on-camera frame filtering. This work does not raise any ethical issues.

Acknowledgements. We thank Frank Cangialosi, Amy Ousterhout, Anirudh Sivaraman, and Srinivas Narayana for their valuable feedback on earlier drafts of this paper. We also thank our shepherd, Ganesh Ananthanarayanan, and the anonymous reviewers for their constructive comments. This work is supported in part by NSF grants CNS-1613023, CNS-1703598, CNS-1943621, and CNS-1763172, and ONR grants N00014-16-1-2913 and N00014-18-1-2037.

REFERENCES

[1] Banff Live Cam, Alberta, Canada. https://www.youtube.com/watch?v=9HwSNgcdQ7k.

[2] Can 30,000 Cameras Help Solve Chicago's Crime Problem? https://www.nytimes.com/2018/05/26/us/chicago-police-surveillance.html.

[3] City of Auburn Toomer's Corner Webcam. https://www.youtube.com/watch?v=hMYIc5ZPJL4.

[4] Gebhardt Insurance Traffic Cam Round Trip Bike Shop. https://www.youtube.com/watch?v=RNi4CKgZVMY.

[5] Jackson Hole Wyoming USA Town Square Live Cam. https://www.youtube.com/watch?v=1EiC9bvVGnk.

[6] JeVois Smart Machine Vision Camera. http://jevois.org.

[7] La Grange, Kentucky USA - Virtual Railfan LIVE. https://www.youtube.com/watch?v=pJ5cg83D5AE.

[8] Newark Police Citizen Virtual Patrol. https://cvp.newarkpublicsafety.org.

[9] Raspberry Pi Zero. https://www.raspberrypi.org/products/raspberry-pi-zero.

[10] TwinForksPestControl.com SOUTHAMPTON TRAFFIC CAM. https://www.youtube.com/watch?v=y3NOhpkoR-w.

[11] DNNCam^{TM} AI camera. https://groupgets.com/campaigns/429-dnncam-ai-camera.

[12] Open Source Computer Vision Library. https://https://opencv.org.

[13] Amazon. AWS DeepLens. https://aws.amazon.com/deeplens/.

[14] Ambarella. CV22 - Computer Vision SoC for Consumer Cameras. https://www.ambarella.com/wp-content/uploads/CV22-product-brief-consumer.pdf.

[15] James Areddy. One Legacy of Tiananmen: China's 100 Million Surveillance Cameras. https://blogs.wsj.com/chinarealtime/2014/06/05/\one-legacy-of-tiananmen-chinas-100-million-surveillance\-cameras/.

[16] AXIS. Axis for a safety touch at the Grey Cup Festival. https://www.axis.com/files/success_stories/ss_stad_greycup_festival_58769_en_1407_lo.pdf.

[17] David Barrett. One surveillance camera for every 11 people in Britain, says CCTV survey. https://www.telegraph.co.uk/technology/10172298/\One-surveillance-camera-for-every-11-people-in-Britain\-says-CCTV-survey.html.

[18] Shweta Bhardwaj, Mukundhan Srinivasan, and Mitesh M. Khapra. 2019. Efficient Video Classification Using Fewer Frames. *CoRR* abs/1902.10640 (2019). arXiv:1902.10640 http://arxiv.org/abs/1902.10640

[19] D. Brezeale and D. J. Cook. 2008. Automatic Video Classification: A Survey of the Literature. *Trans. Sys. Man Cyber Part C* 38, 3 (May 2008), 416–430. https://doi.org/10.1109/TSMCC.2008.919173

[20] S. Brutzer, B. Hoferlin, and G. Heidemann. 2011. Evaluation of Background Subtraction Techniques for Video Surveillance. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '11)*. IEEE Computer Society, Washington, DC, USA, 1937–1944. https://doi.org/10.1109/CVPR.2011.5995508

[21] N. Buch, S. A. Velastin, and J. Orwell. 2011. A Review of Computer Vision Techniques for the Analysis of Urban Traffic. *Trans. Intell. Transport. Sys.* 12, 3 (Sept. 2011), 920–939.

[22] Zhaowei Cai, Mohammad Saberian, and Nuno Vasconcelos. 2015. Learning Complexity-Aware Cascades for Deep Pedestrian Detection. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV) (ICCV '15)*. IEEE Computer Society, Washington, DC, USA, 3361–3369. https://doi.org/10.1109/ICCV.2015.384

[23] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G. Andersen, Michael Kaminsky, and Subramanya R. Dulloor. 2019. Scaling Video Analytics on Constrained Edge Nodes. In *2nd SysML Conference*.

[24] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. 155–168.

[25] Sandeep P. Chinchali, Eyal Cidon, Evgenya Pergament, Tianshu Chu, and Sachin Katti. 2018. Neural Networks Meet Physical Networks: Distributed Inference Between Edge Devices and the Cloud. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets '18)*. Association for Computing Machinery, New York, NY, USA, 50–56. https://doi.org/10.1145/3286062.3286070

[26] Chong-Wah Ngo, Yu-Fei Ma, and Hong-Jiang Zhang. 2005. Video summarization and scene detection by graph modeling. *IEEE Transactions on Circuits and Systems for Video Technology* 15, 2 (Feb 2005), 296–305. https://doi.org/10.1109/TCSVT.2004.841694

[27] John Emmons, Sadjad Fouladi, Ganesh Ananthanarayanan, Shivaram Venkataraman, Silvio Savarese, and Keith Winstein. 2019. Cracking Open the DNN Black-Box: Video Analytics with DNNs across the Camera-Cloud Boundary. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges (HotEdgeVideo'19)*. Association for Computing Machinery, New York, NY, USA, 27–32. https://doi.org/10.1145/3349614.3356023

[28] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. 2010. The Pascal Visual Object Classes (VOC) Challenge. *Int. J. Comput. Vision* 88, 2 (June 2010), 303–338. https://doi.org/10.1007/s11263-009-0275-4

[29] Hehe Fan, Zhongwen Xu, Linchao Zhu, Chenggang Yan, Jianjun Ge, and Yi Yang. 2018. Watching a Small Portion could be as Good as Watching All: Towards Efficient Video Classification. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 705–711. https://doi.org/10.24963/ijcai.2018/98

[30] S. Gammeter, A. Gassmann, L. Bossard, T. Quack, and L. Van Gool. 2010. Server-side object recognition and client-side object tracking for mobile augmented reality. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*. 1–8. https://doi.org/10.1109/CVPRW.2010.5543248

[31] Anil K Ghosh. 2006. On optimum choice of k in nearest neighbor classification. *Computational Statistics & Data Analysis* 50, 11 (2006), 3113–3123.

[32] Peter Hall, Byeong U Park, Richard J Samworth, et al. 2008. Choice of neighbor order in nearest-neighbor classification. *The Annals of Statistics* 36, 5 (2008), 2135–2152.

[33] Bo Han, Feng Qian, Lusheng Ji, and Vijay Gopalakrishnan. 2016. MP-DASH: Adaptive Video Streaming Over Preference-Aware Multipath. In *Proceedings of the 12th International on*

Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16). ACM, New York, NY, USA, 129–143. https://doi.org/10.1145/2999572.2999606

[34] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. ACM, New York, NY, USA, 123–136. https://doi.org/10.1145/2906388.2906396

[35] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *ArXiv* abs/1704.04861 (2017).

[36] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 269–286. https://www.usenix.org/conference/osdi18/presentation/hsieh

[37] Weiming Hu, Nianhua Xie, Li, Xianglin Zeng, and Stephen Maybank. 2011. A Survey on Visual Content-Based Video Indexing and Retrieval. *Trans. Sys. Man Cyber Part C* 41, 6 (Nov. 2011), 797–819. https://doi.org/10.1109/TSMCC.2011.2109710

[38] C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose. 2018. VideoEdge: Processing Camera Streams using Hierarchical Clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 115–131. https://doi.org/10.1109/SEC.2018.00016

[39] LDV Capital Insights. 45 Billion Cameras by 2022 Fuel Business Opportunities. https://www.ldv.co/insights/2017.

[40] Samvit Jain, Junchen Jiang, Yuanchao Shu, Ganesh Ananthanarayanan, and Joseph Gonzalez. 2018. ReXCam: Resource-Efficient, Cross-Camera Video Analytics at Enterprise Scale. *CoRR* abs/1811.01268 (2018). arXiv:1811.01268 http://arxiv.org/abs/1811.01268

[41] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: Scalable Adaptation of Video Analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 253–266. https://doi.org/10.1145/3230543.3230574

[42] Daniel Kang, Peter Bailis, and Matei Zaharia. 2018. BlazeIt: Fast Exploratory Video Queries using Neural Networks. *CoRR* abs/1805.01046 (2018). arXiv:1805.01046 http://arxiv.org/abs/1805.01046

[43] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1586–1597. https://doi.org/10.14778/3137628.3137664

[44] Hanme Kim, Stefan Leutenegger, and Andrew J. Davison. 2016. Real-Time 3D Reconstruction and 6-DoF Tracking with an Event Camera. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VI*. 349–364. https://doi.org/10.1007/978-3-319-46466-4_21

[45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386

[46] B. Kueng, E. Mueggler, G. Gallego, and D. Scaramuzza. 2016. Low-latency visual odometry using event-based feature tracks. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 16–23. https://doi.org/10.1109/IROS.2016.7758089

[47] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua. 2015. A convolutional neural network cascade for face detection. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 5325–5334.

[48] Yawei Li, Eirikur Agustsson, Shuhang Gu, Radu Timofte, and Luc Van Gool. 2018. CARN: Convolutional Anchored Regression Network for Fast and Accurate Single Image Super-Resolution. In *The European Conference on Computer Vision (ECCV) Workshops*.

[49] T. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. 2017. Feature Pyramid Networks for Object Detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 936–944. https://doi.org/10.1109/CVPR.2017.106

[50] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. 2016. Optasia: A Relational Platform for Efficient Large-Scale Video Analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY, USA, 57–70. https://doi.org/10.1145/2987550.2987564

[51] M5STACK. K210 RISC-V 64 AI Camera. https://m5stack.com/blogs/news/introducing-the-k210-risc-v-ai-camera-m5stickv.

[52] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief. 2017. A Survey on Mobile Edge Computing: The Communication Perspective. *IEEE Communications Surveys Tutorials* 19, 4 (Fourthquarter 2017), 2322–2358. https://doi.org/10.1109/COMST.2017.2745201

[53] IHS Markit. IHS Markit's Top Video Surveillance Trends for 2018. https://cdn.ihs.com/www/pdf/Top-Video-Surveillance-Trends-2018.pdf.

[54] Microsoft. Microsoft Azure Data Box. https://azure.microsoft.com/en-us/services/databox/.

[55] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP *(Proceedings of ATC '15)*. USENIX.

[56] Mayu Otani, Yuta Nakashima, Esa Rahtu, and Janne Heikkilä. 2019. Rethinking the Evaluation of Video Summaries. *CoRR* abs/1903.11328 (2019). arXiv:1903.11328 http://arxiv.org/abs/1903.11328

[57] Chrisma Pakha, Aakanksha Chowdhery, and Junchen Jiang. 2018. Reinventing Video Streaming for Distributed Vision Analytics. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotcloud18/presentation/pakha

[58] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen. 2018. DeepDecision: A Mobile Deep Learning Framework for Edge Video Analytics. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 1421–1429. https://doi.org/10.1109/INFOCOM.2018.8485905

[59] Henri Rebecq, Timo Horstschaefer, Guillermo Gallego, and Davide Scaramuzza. 2017. EVO: A Geometric Approach to Event-Based 6-DOF Parallel Tracking and Mapping in Real Time. *IEEE Robotics and Automation Letters* 2, 2 (2017), 593–600. https://doi.org/10.1109/LRA.2016.2645143

[60] Henri Rebecq, Timo Horstschaefer, and Davide Scaramuzza. 2017. Real-time Visual-Inertial Odometry for Event Cameras using Keyframe-based Nonlinear Optimization. In *British Machine Vision Conference 2017, BMVC 2017, London, UK, September 4-7, 2017*. https://www.dropbox.com/s/ijvhc2hsdh85kcb/0534.pdf?dl=1

[61] Joseph Redmon. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[62] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, Faster, Stronger. *CoRR* abs/1612.08242 (2016).

[63] Y. Ren, F. Zeng, W. Li, and L. Meng. 2018. A Low-Cost Edge Server Placement Strategy in Wireless Metropolitan Area Networks. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. 1–6. https://doi.org/10.1109/ICCCN.2018.8487438

[64] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 322–337. https://doi.org/10.1145/3341301.3359658

[65] Yi Sun, Xiaogang Wang, and Xiaoou Tang. 2013. Deep Convolutional Network Cascade for Facial Point Detection. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '13)*. IEEE Computer Society, Washington, DC, USA, 3476–3483. https://doi.org/10.1109/CVPR.2013.446

[66] Z. Tang, G. Wang, H. Xiao, A. Zheng, and J. Hwang. 2018. Single-Camera and Inter-Camera Vehicle Tracking and 3D Speed Estimation Based on Fusion of Visual and Semantic Features. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 108–115.

[67] Tencent. DeepGaze AI Camera. https://open.youtu.qq.com/#/open/solution/hardware-ai.

[68] Antoni Rosinol Vidal, Henri Rebecq, Timo Horstschaefer, and Davide Scaramuzza. 2018. Ultimate SLAM? Combining Events, Images, and IMU for Robust Visual SLAM in HDR and High-Speed Scenarios. *IEEE Robotics and Automation Letters* 3, 2 (2018), 994–1001. https://doi.org/10.1109/LRA.2018.2793357

[69] Junjue Wang, Ziqiang Feng, Zhuo Chen, Shilpa George, Mihir Bala, Padmanabhan Pillai, Shao-Wen Yang, and Mahadev Satyanarayanan. 2018. Bandwidth-Efficient Live Video Analytics for Drones Via Edge Computing. 159–173. https://doi.org/10.1109/SEC.2018.00019

[70] Shiyao Wang, Hongchao Lu, Pavel Dmitriev, and Zhidong Deng. 2018. Fast Object Detection in Compressed Video. *CoRR* abs/1811.11057 (2018). arXiv:1811.11057 http://arxiv.org/abs/1811.11057

[71] Paul N. Whatmough, Chuteng Zhou, Patrick Hansen, Shreyas K. Venkataramanaiah, Jae-sun Seo, and Matthew Mattina. 2019. FixyNN: Efficient Hardware for Mobile Computer Vision via Transfer Learning. *CoRR* abs/1902.11128 (2019). arXiv:1902.11128 http://arxiv.org/abs/1902.11128

[72] Wi4Net. Axis is on the case in downtown Huntington Beach. http://www.wi4net.com/Resources/Pdfs/huntington%20beach%20case_study%5BUS%5Dprint.pdf.

[73] Chao-Yuan Wu, Christoph Feichtenhofer, Haoqi Fan, Kaiming He, Philipp Krähenbühl, and Ross B. Girshick. 2018. Long-Term Feature Banks for Detailed Video Understanding. *CoRR* abs/1812.05038 (2018). arXiv:1812.05038 http://arxiv.org/abs/1812.05038

[74] Zuxuan Wu, Caiming Xiong, Chih-Yao Ma, Richard Socher, and Larry S. Davis. 2018. AdaFrame: Adaptive Frame Selection for Fast Video Recognition. *CoRR* abs/1811.12432 (2018). arXiv:1811.12432 http://arxiv.org/abs/1811.12432

[75] Wyze. Wyze Camera. https://www.safehome.org/home-security-cameras/wyze/.

[76] Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. 2019. VStore: A Data Store for Analytics on Large Videos. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, Article 16, 17 pages. https://doi.org/10.1145/3302424.3303971

[77] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li. 2017. LAVEA: Latency-Aware Video Analytics on Edge Computing Platform. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2573–2574. https://doi.org/10.1109/ICDCS.2017.182

[78] Yueting Zhuang, Yong Rui, T. S. Huang, and S. Mehrotra. 1998. Adaptive key frame extraction using unsupervised clustering. In *Proceedings 1998 International Conference on Image Processing. ICIP98 (Cat. No.98CB36269)*, Vol. 1. 866–870 vol.1. https://doi.org/10.1109/ICIP.1998.723655

[79] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. 2017. Live Video Analytics at Scale with Approximation and Delay-tolerance. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, Berkeley, CA, USA, 377–392. http://dl.acm.org/citation.cfm?id=3154630.3154661

[80] Tan Zhang, Aakanksha Chowdhery, Paramvir Bahl, Kyle Jamieson, and Suman Banerjee. 2015. The Design and Implementation of a Wireless Video Surveillance System. 426–438. https://doi.org/10.1145/2789168.2790123

[81] Tan Zhang, Aakanksha Chowdhery, Paramvir (Victor) Bahl, Kyle Jamieson, and Suman Banerjee. 2015. The Design and Implementation of a Wireless Video Surveillance System. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom '15)*. ACM, New York, NY, USA, 426–438. https://doi.org/10.1145/2789168.2790123

[82] Alex Zihao Zhu, Nikolay Atanasov, and Kostas Daniilidis. 2017. Event-based feature tracking with probabilistic data association. In *2017 IEEE International Conference on Robotics and Automation, ICRA 2017, Singapore, Singapore, May 29 - June 3, 2017*. 4465–4470. https://doi.org/10.1109/ICRA.2017.7989517

[83] A. Z. Zhu, N. Atanasov, and K. Daniilidis. 2017. Event-Based Visual Inertial Odometry. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 5816–5824.

[84] Zhengxia Zou, Zhenwei Shi, Yuhong Guo, and Jieping Ye. 2019. Object Detection in 20 Years: A Survey. *CoRR* abs/1905.05055 (2019). arXiv:1905.05055 http://arxiv.org/abs/1905.05055

A APPENDIX

Appendices are supporting material that has not been peer-reviewed.

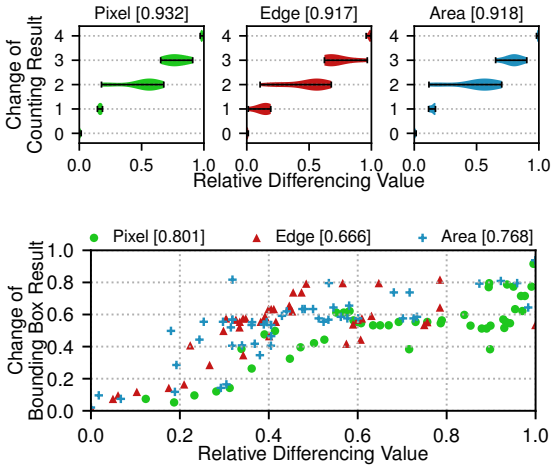


Figure 14: Correlations between differencing values and changes in query results for a 10 seconds clip in Jacksonhole [5].

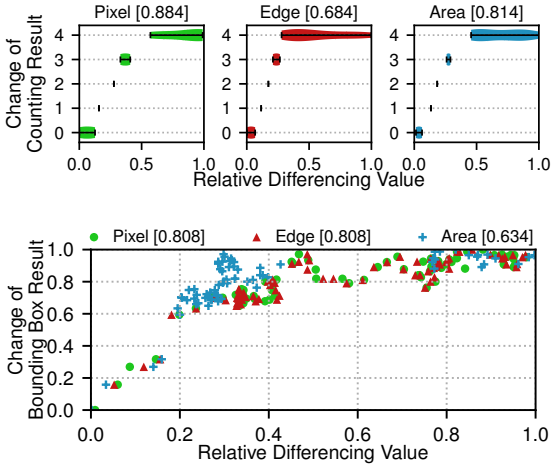


Figure 15: Correlations between differencing values and changes in query results for a 10 seconds clip in Southampton [10].

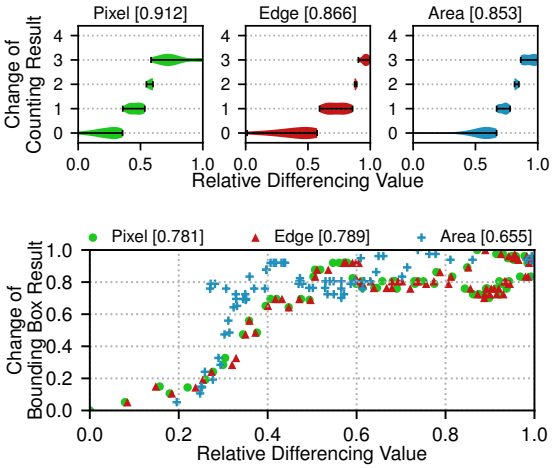


Figure 16: Correlations between differencing values and changes in query results for a 10 seconds clip in Lagrange [7].

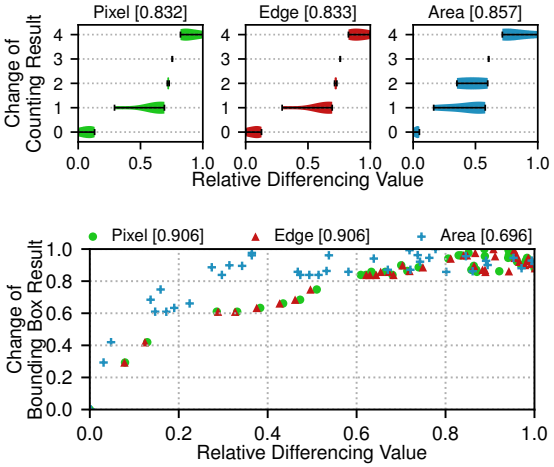


Figure 17: Correlations between differencing values and changes in query results for a 10 seconds clip in Newark [8].

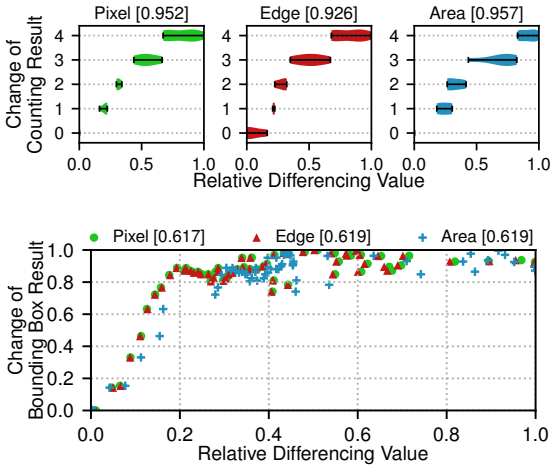


Figure 18: Correlations between differencing values and changes in query results for a 10 seconds clip in Banff [1].

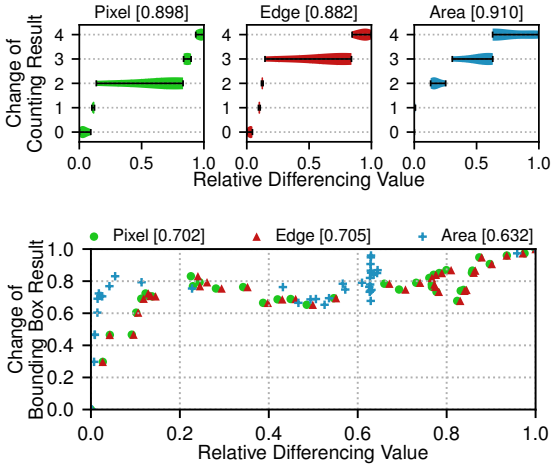
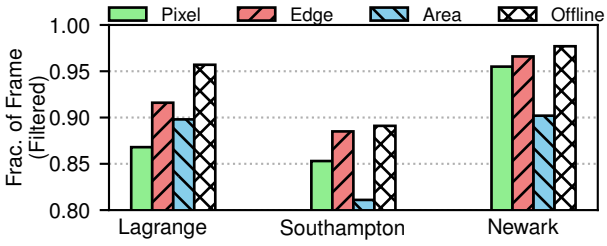
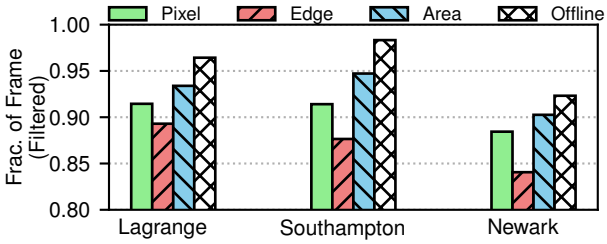


Figure 19: Correlations between differencing values and changes in query results for a 10 seconds clip in Casa Grande [4].

Figure 20: Screenshots from several of the videos in our dataset. Left is Jackson Hole, WY, and right is Newark, NJ.

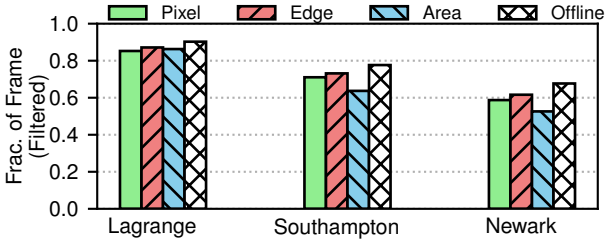


(a) Query: People bounding box detection

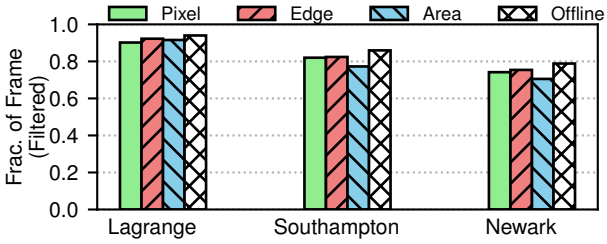


(b) Query: People counting

Figure 21: Filtering efficacy of the 3 low-level features across 3 videos and 2 queries. This figure shows Best feature holds across other objects of interest.

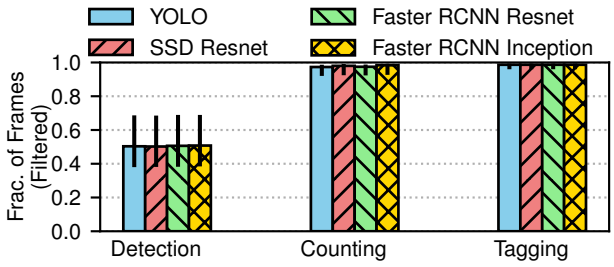


(a) Car bounding box detection with 80% accuracy target

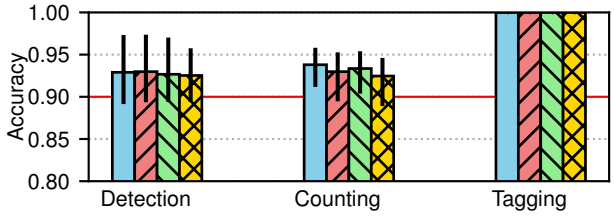


(b) Car bounding box detection with 70% accuracy target

Figure 22: Filtering efficacy of the 3 low-level features across 3 videos and 2 queries. This figure shows Best feature holds across other accuracy targets.



(a) Fraction of filtered



(b) Achieved accuracy

Figure 23: Comparing Reducto with different detection models for three query types and two objects of interest across our entire dataset.

Feature	Description
Area	Calculates the areas of motion in the frame and sends frame if the largest area (as a fraction of total pixels) is above the given threshold.
Pixel	Finds pixels that changed value from the last frame, rounds very small changes to 0, and sends frame if the resulting fraction of changed pixels is above the given threshold.
Edge	Separates the pixels that belong to edges and sends frame if the pixel differences among edge pixels is above the given threshold.
Corner	Detects the pixels that belong to corners and sends frame if the pixel differences among corner pixels is above the given threshold.
SIFT	Detects key points based on contrast, assigns them orientations based on the "neighborhood" of surrounding pixels, and matches them between frames.
SURF	Detects key points using a blob detector, assigns them orientations such that the key points remain if the object is either scaled or rotated, and matches key points between frames.
HOG	Divides frame into small cells, collects a distribution of gradient directions across cells, and compares the distribution across frames.

Table 8: Description of differencing features considered in our survey (§3).