Human-efficient Discovery of Edge-based Training Data for Visual Machine Learning

Ziqiang Feng

CMU-CS-21-120

August 2021

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by National Science Foundation award number: CNS1518865; by United States Defense Advanced Research Projects Agency award number: HR001117C0051; and by Intel Corporation award number: A018540296213611; and by a Croucher Foundation Scholarship for Doctoral Study. Additional support was provided by Vodafone, Deutsche Telekom, Crown Castle, InterDigital, Seagate, Microsoft, VMware, and the Conklin Kistler family fund. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Edge Computing, Cloudlet, Video Analytics, Training Data

To the bond between humans and machines.

Abstract

Deep learning enables effective computer vision without hand crafting feature extractors. It has great potential if applied to specialized domains such as ecology, military, and medical science. However, the laborious task of creating *labeled* training sets of rare targets is a major deterrent to achieving its goal. A domain expert's time and attention is precious. We address this problem by designing, implementing, and evaluating Eureka, a system for human-efficient discovery of rare phenomena from unlabeled visual data. Eureka's central idea is interactive content-based search of visual data based on early-discard and machine learning. We first demonstrate its effectiveness for curating training sets of rare objects. By analyzing contributing factors to human efficiency, we identify and evaluate important system-level optimizations that utilize edge computing and intelligent storage. Lastly, we extend Eureka to the task of discovering temporal events from video data.

Acknowledgments

It has been a long journey filled with many moments of excitement, enlightenment, and discovery, as well as, inevitably, moments of challenge, struggle, and confusion. The work of this dissertation would not be possible without the support of a group of talented and caring people.

I am deeply indebted to my advisor Professor Mahadev Satyanarayanan (Satya). It has been a great honor to be advised by Satya for the past five years. Satya taught me many qualities of a good researcher. His multi-decade research experience in broad areas helps me see connections between things that I may otherwise ignore. He is a visionary who can acutely discover research opportunities driven by emerging and futuristic technologies. In addition to being a great scientist, Satya is also a good leader, a caring mentor, and a nice friend. He has offered invaluable guidance and help in my career decision. I am also fortunate to have researchers with diverse backgrounds in my thesis committee. Padmanabhan (Babu) Pillai is an exemplar system researcher who has broad and deep knowledge of almost every aspect of computer systems ranging from low level hardware components to application software. I also appreciate his patience and kindness in answering my questions and staying with me late night for paper deadlines. I would like to thank Martial Hebert for his guidance and critiques from the computer vision perspective, which is a corner stone of this dissertation. I also thank Roberta (Bobby) Klatzky, who brings in novel and sometime counter-intuitive viewpoints as an expert in psychology and human-computer interaction.

Many ideas and experimental results presented in this document come from discussion and collaborative work with talened individuals of Satya's research group. Their day-to-day technical and emotional support has been an irreplaceable part of my PhD life. Especially, I would like to thank, in alphabetical order, Mihir Bala, Zhuo Chen, Jason Choi, Kevin Christensen, Thomas Eiszler, Shilpa George, Kiryong Ha, Jan Harkes, Wenlu Hu, Roger Iyengar, Natalie Janosik, Haithem Turki, Junjue Wang. I would also thank Chase Klingensmith for his outstanding administrative assistance for the group.

My research also benefited from discussion and collaboration with many talented individuals beyond Satya's group and beyond CMU. Thank you, Ragaad Altarawneh, Jim Blakley, Pedro Cuadra Chamorro, Eduardo Cuervo, Jason Feist, Guoyao (Freddie) Feng, Erin Foley, Chris Fulkerson, Greg Ganger, Wei Gao, Philip Gibbons, James Gross, Vishakha Gupta, Michael Kozuch, Grace Lewis, Tan Li, Lin Ma, Manuel Olguín, Andy Pavlo, Deva Ramanan, Thomas Rausch, Luis Remis, Dan Siewiorek, Asim Smailagic, Christina Strong, Jinliang Wei, Shao-Wen Yang, Canbo (Albert) Ye, Huanchen Zhang.

Finally, I am grateful for the support of my parents. While they had not been in a graduate school, they have shown enormous trust, understanding, and consideration through the ups and downs of my PhD study. I could not have finished the PhD without knowing I will always have their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The general goal of computer vision is to understand, like a human does, what is happening in image and video data. What objects are present in the picture? Where are they? What activity is the person doing in the video? Recently, methods based on *supervised deep learning* have become the standard approach to solving these problems by demonstrating near-human accuracy on certain tasks. This inspires interest in adopting deep learning in specialized domains such as sciences, military, and business. However, deep learning's success is based on the premise of *The Unreasonable Effectiveness of Data* [54], which involves laborious manual annotation of thousands to millions of training examples. In specialized domains, this task often falls on the shoulder of a single domain expert. It consumes a lot of precious time and attention of an expert, hindering practical use of deep learning in those domains. Can we remove the hindrance by improving an expert's efficiency in this painstaking task? What can a computer system offer to achieve this goal? We address these questions in this dissertation.

1.1 Deep Learning for Computer Vision

Until the past decade, the popular approaches in computer vision could be broadly described as the use of manually designed visual feature descriptors (e.g., HOG [32], SIFT [95]) and classical machine learning (e.g., SVM). These descriptors capture certain aspects of visual objects (e.g., color, shape, scale) and each has its limitations. Machine learning is used to properly weigh the features by learning from examples.

In 2012, a deep learning solution called AlexNet [87] scored an overwhelming victory against those traditional methods in that year's ImageNet classification competition. Since then, there has been avid effort to push the envelope of deep learning. In 2015, ResNet [59] marked another milestone by surpassing human accuracy on ImageNet. Meanwhile, deep learning has been extended beyond image classification to other computer vision tasks, including object detection [119], semantic segmentation [102], and activity recognition [47], with various degrees of success. Compared to traditional approaches, deep learning avoids the need for manually designed feature extractors. *Deep neural networks (DNNs)* are believed to be able to *learn* a richer and more complete set of features, given a sufficient amount of example data.

Today, deep learning has reached such a level of accuracy that companies have rolled out services to end consumers. Mobile Apps such as Google Lens use deep learning to recognize objects in users' photos. Flickr uses deep learning to auto-tag millions of images [105].

DNNs can embody near-human discriminative power. This inspires interests in adopting deep learning in specialized domains such as sciences, military, and business. Imagine creating DNNs that can answer questions hard for non-experts to answer. Examples include: "Is this a caterpillar of the pest moth *Cactoblastis cactorum* or of a benign moth?"; "Is this a MQ-9 Reaper drone or a common drone?"; "Does this video clip show a legit car accident or an insurance fraud?". Once created, these DNNs can be deployed at scale in the field to detect targets in real time, or be used to scan many years of archival data in just hours or days.

1.2 Creating and Deploying DNNs

Conceptually, a DNN is a parameterized mathematical function (also called a model) that transforms the high-dimensional input signals (e.g., millions of RGB pixel values) into low-dimensional semantic information (e.g., a handful of object labels and bounding boxes). DNNs are huge, involving millions of parameters and millions of FLOPs to compute. AlexNet [123] has 60 million parameters and takes 720 MFLOPs to compute, while ResNet-50 [119] has 25 million parameters and takes 4,000 MFLOPs to compute. Furthermore, it proves to be crucial to exploit accelerators such as GPUs to achieve acceptable run time performance. Great care needs to be taken to turn such conceptions into computer programs. To this end, many deep learning frameworks have been proposed. The most notable ones among others are Caffe [76], TensorFlow [2], PyTorch [111], and MXNet [22]. These frameworks provide deep learning abstractions such as models, layers, and operators; and encapsulate details of memory allocation, multi-threading, scheduling, model serialization, and interfacing with GPUs. This allows machine learning researchers to focus on algorithmic design and be able to quickly prototype new ideas and conduct experimental evaluation.

Enabled by the convenience of these frameworks, many innovations have been made to DNN architectures ("shape" of the mathematical function) to make them run faster, be more accurate, or both. To name a few, VGG [136], Inception [141], ResNet [59], MobileNet [63], CFNet [149], and Faster R-CNN [119]. For example, ResNet uses a novel technique called residual connection that makes the network significantly smaller and faster than AlexNet, and yet much more accurate.

Deploying DNNs for practical use cases often requires them to run on devices with limited compute, memory, and battery. Resource constraints on these devices make it challenging to meet latency requirements, or even be able to execute at all. To address these challenges, various techniques of model compression [23, 24, 57, 70] have been proposed to reduce the size of the DNN models at the expense of minimal accuracy loss. In terms of hardware, vendors produce specialized accelerators such as Intel Movidius [30] and Google TPU [80] dedicated for deep learning workloads. These chips often include specialized hardware implementation for the most commonly-used DNN layers, such as the convolution layer, thus achieving high run time efficiency and energy efficiency.

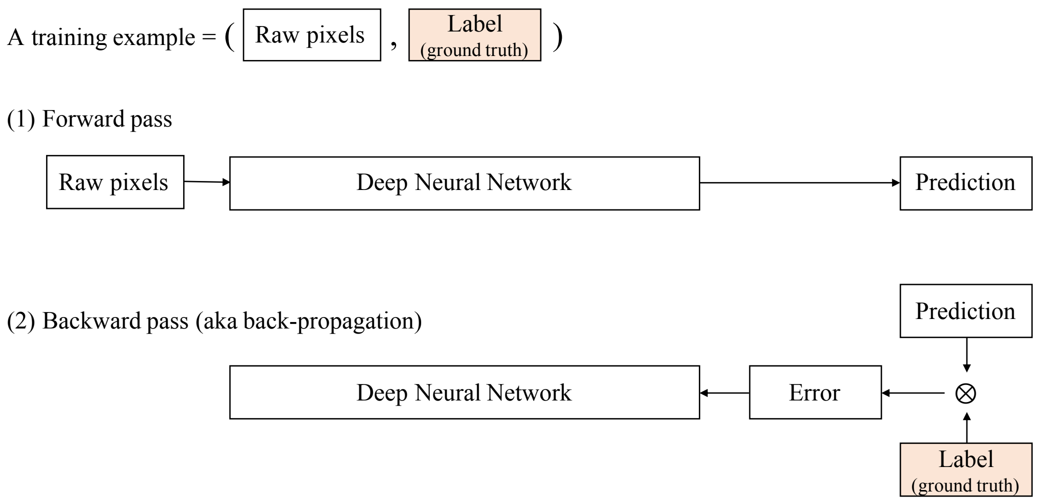


Figure 1.1: The inference and training (learning) steps of DNNs

1.3 Creation of Training Data Sets

DNNs need to be *trained* on a task-specific training set to be useful. Figure 1.1 illustrates the two-pass concept of training. The forward pass runs the model on the input pixels and receives a prediction, as it would when the model is deployed; the backward pass compares this prediction against the ground truth label and adjusts the model parameters in the direction of reducing that error. The ground truth is what the DNN should ideally output, and is often created by manual annotation. For binary classification tasks, this means assigning positive/negative to each example. A training example is thus a pair of raw pixels and label. A training set is a collection of such examples. The forward and backward passes are repeated many times, on many training examples, until the model converges to a good average accuracy over the training set. Work mentioned earlier in Section 1.2 assumes such a training set is available. While the forward/backward computation is automated by systems such as TensorFlow, preparation of the training data is left outside the system.

Labeled training data is the key here. Where do the labels come from? Indeed, the success of deep learning is made possible thanks to public training sets curated by many diligent researchers. Table 1.1 lists the statistics of several data sets widely used in academia. It also contrasts their sizes to the data "in the wild." Note that even at moderate sizes, curation of labeled data sets is a painstaking effort. ImageNet [123] took 19 person-year to curate. The VIRAT video data set [109] was introduced in 2011 and was under annotation until 2016. To date, the prevailing solution to accelerating this process is to utilize crowd-sourcing tools such as Amazon Mechanical Turk. This essentially leverages human-level parallelism offered by many crowd workers. Unfortunately, as we discuss next, this approach is infeasible in specialized domains.

Data set	Size	Example labels
Labeled: object recognition/detection (image)		
PASCAL VOC [40]	12,000 images	person, chair, dining table, dog
COCO [93]	330,000 images	person, bicycle, cat, dog, knife
ImageNet [123]	1,200,000 images	car wheel, goldfish, Irish terrier, kite
Labeled: activity recognition (video)		
HMDB51 [88]	7,000 files; 3.9 hours	clap, kick ball, push, sit, wave
Charades [135]	10,000 files; 83.3 hours	eat sandwich, hold broom, open fridge
UCF101 [138]	13,000 files; 27.0 hours	basketball dunk, jump rope, play guitar
VIRAT [109]	329 files; 8.5 hours	enter building, get into vehicle
Unlabeled		
YFCC100M [145]	99,200,000 images 800,000 video files; 8,000 hours	
In the wild		
Smartphone [104]	1 trillion (10^{12}) photos captured per year (2017)	
YouTube [29]	500 hours' videos uploaded per minute (2019)	

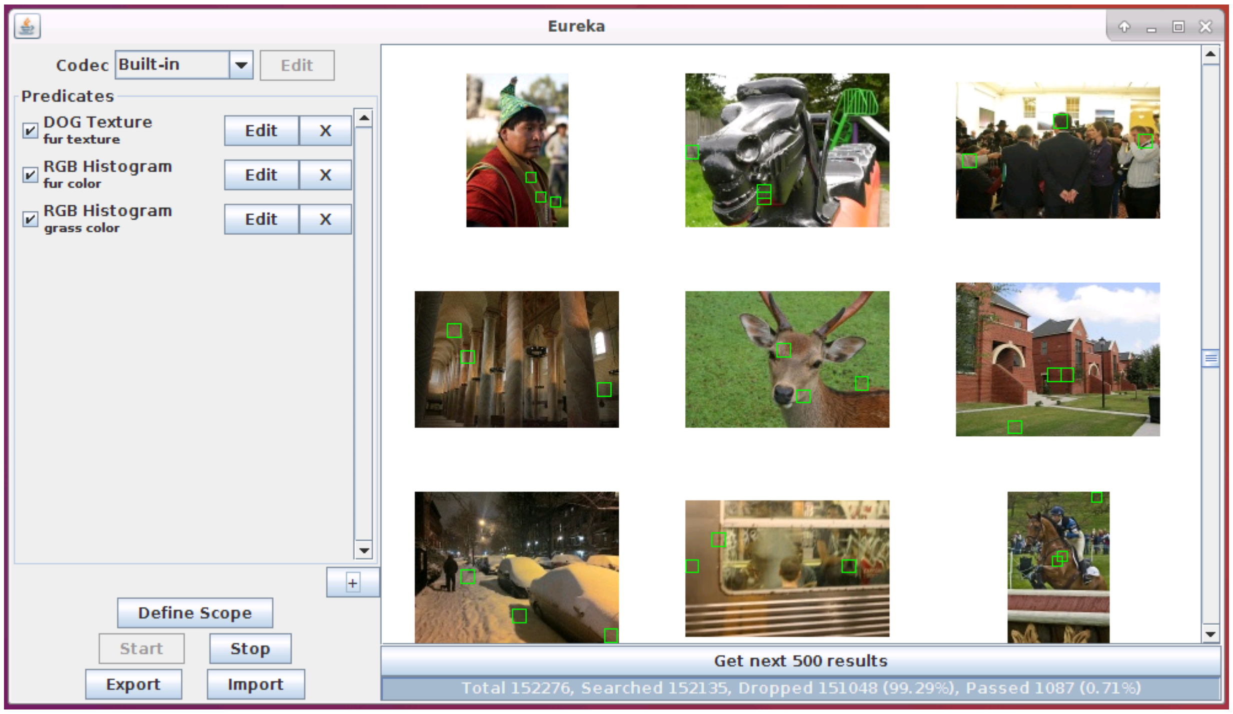
Table 1.1: Quantities of labeled and unlabeled visual data, as well as data in the wild

1.4 Human-efficiency of Domain Experts

For experts in specialized domains, the creation of training sets remains a major challenge, despite the many software and hardware tools introduced earlier. Three factors complicate the challenges compared to those in Section 1.3. First, targets in specialized domains are usually rare, due to a low *base rate* [97]. The target may be a rare animal species, a secret weapon recently developed by the enemy, etc. Negative examples are abundant. It is the discovery of *positive* examples that is difficult and amplifies the labeling work. Second, correctly identifying the target requires domain-specific knowledge only possessed by an expert, for example, telling the pest *Cactoblastis cactorum* from a benign moth. This task cannot be offloaded to a non-expert crowd worker. Third, patient privacy, national security, or business concerns may prevent the data from being accessed by a larger labeler group. As a real world example, the MURA data set of 40,561 images for abnormality detection in radiology took 11 years to create [116]. In the worst case, a single expert needs to create an entire training set by sifting through lots of data and discovering the rare positive instances.

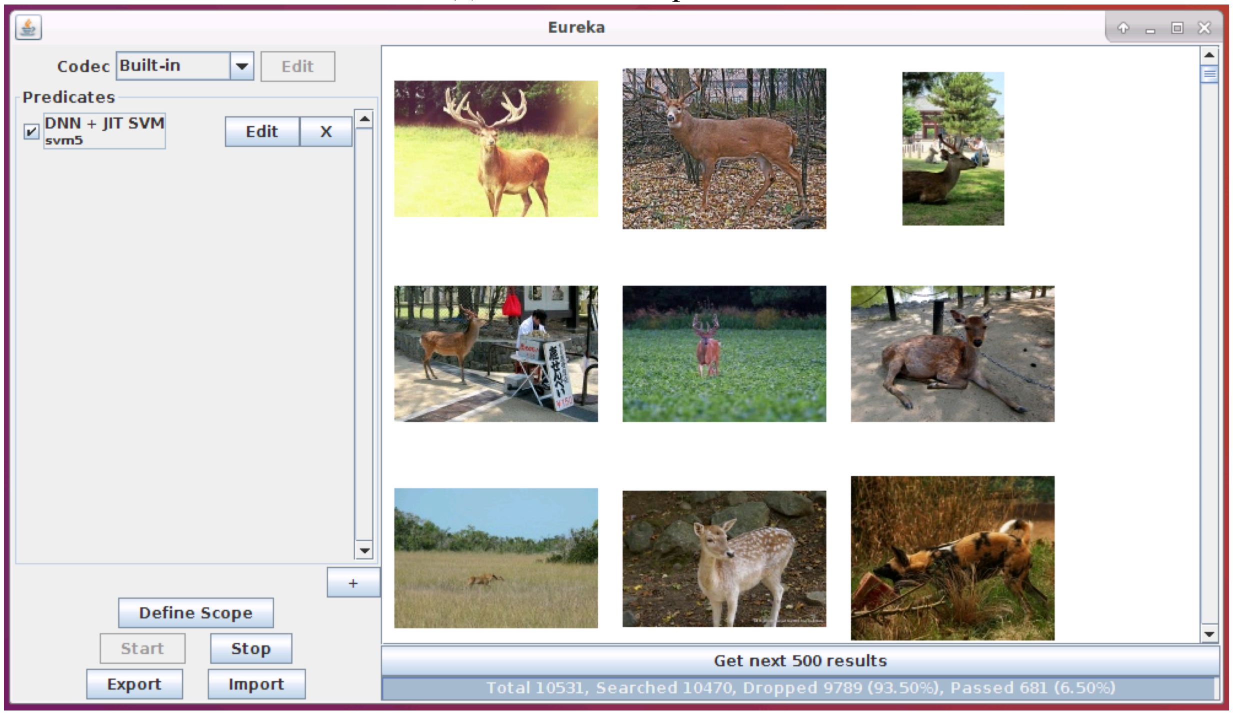
Consider a computer system that scans the raw data and provides potentially positive examples for an domain expert to check and confirm. In this setting, the most precious resource in the system is the expert's time and attention. The expert's time is well spent if she is continuously confirming positive examples presented by the system. It is poorly spent if she is: (a) dismissing a flood of frivolous negative examples; (b) waiting for the system to return a candidate for examination. We collectively call these aspects *human-efficiency*.

Figure 1.2 shows two possible scenarios when an expert is searching for deer images in the YFCC100M [145] data set. Here, each screenshot shows a page from a long result stream. In Figure 1.2(a), only 1 out of 9 displayed candidates is truly a deer (the one in the center). The expert needs to dismiss the other 8 as negatives. This is a poorer use of expert time when com-



Total 152276, Searched 152135, Dropped 151048 (99.29%), Passed 1087 (0.71%)

(a) Poor use of expert time



Total 10531, Searched 10470, Dropped 9789 (93.50%), Passed 681 (6.50%)

(b) Good use of expert time

Figure 1.2: Examples of poor and good use of expert time in finding images of deer

pared to Figure 1.2(b), where 8 out of 9 are true positives (the one at bottom right is a negative). Note the primary metric here is the expert's productivity in discovering positive (deer) examples. This "finding a needle in a haystack" setting is different form the ImageNet competition's task that classifies evenly-distributed classes. Given the same budget of labeling time, an expert in Figure 1.2(b) will likely discover a greater number of deer images than (a). In other words, a system that returns results like (b) is more human-efficient than one that returns results like (a).

1.5 Prior Work in Training Set Creation By Experts

Training set creation is a less-visited topic compared to the design and deployment of DNNs. Nguyen et al. [107] explored the mixing of crowd and experts for the task of literature screening. For reasons discussed earlier, data access by the crowd in specialized domains can be completely precluded. More recently, Snorkel [118] proposed asking the experts to write a handful of "weak labeling functions" to accomplish the task. Each labeling function reflects a simple heuristic to classify the data. A labeling function alone is supposed to be far from good, but still better than a random guess. Multiple labeling functions can then be aggregated under some statistical models to produce more accurate labels. This approach requires domain experts to write code, raising the barrier to access of the system. Moreover, Snorkel relies on some statistical assumptions of the data and the labeling functions that are hard to verify in the real world. Lastly, neither of the above focused on visual data.

The Diamond system [69] explored the concept of *early discard filters* for interactive search of non-indexed image data. In Diamond, a user creates a chain of simple filters based on heuristics, each of which tries to discard an image as early as possible. Note these filter chains are not a replacement of the DNN to be trained eventually. Their purpose is to prune the data space that is likely to be negative, and thus to reduce the amount of data that an expert must sift through. For example, to find images of deer, one may use a color histogram filter to discard images without a green patch, which is indicative of the vegetation background often found in a deer photo.

While Diamond demonstrated the potential of the early discard paradigm, it has major limitations in face of new technologies and new tasks. First, Diamond was conceived around 2004, predating the advent of deep learning. The only way to create more accurate filters in Diamond is to write new pieces of computer vision code. Similar to Snorkel, this requires programming skills of the experts, raises the barrier to access, and slows down the data discovery cycle. We observe that with techniques enabled by deep learning – more concretely, a class of methods called *transfer learning* [110] – an expert may be able to create better filters by simply supplying more labeled examples. How transfer learning can be utilized and how it affects the discovery workflow is unclear. Second, Diamond has solely focused on searching of image data. Discovering events in video data requires the expression of complex multi-object relations and temporal-spatial relations. How to let a non/minimal-programmer expert express her heuristics of such relations and how to execute it on data efficiently remains to be answered. Third, with the emergence of new compute and storage architectures such as *edge computing* [127], there are new opportunities to optimize for machine efficiency, which is critical for reducing system stall time and improving human efficiency. Doing so requires rethinking system-level implementation and optimization.

1.6 Thesis Statement

In this dissertation, we address the problem of *human-efficient discovery of training data for visual machine learning*. We claim that:

The manual effort of discovering a large training set for visual machine learning can be reduced by a system combining: (a) efficient early discard made possible by edge computing; (b) just-in-time machine learning; and (c) the ability to create more accurate filters immediately without writing new code. This approach is effective for different compute/storage architectures and different vision tasks.

Laborious manual discovery of training data for rare phenomena is a major deterrent to adopting deep learning in specialized domains. The thesis, if validated, will significantly reduce this deterrent and enable the creation of scalable computer programs that embody domain-specific expertise. This challenge is unaddressed by existing deep learning software and hardware tools, which have focused on other steps of the deep learning workflow. Addressing this challenge requires taking into account state-of-the-art of machine learning, characterization of system performance, and the way an expert interacts with the system.

1.7 Thesis Validation

We validate the above thesis by building *Eureka*, an interactive system that lets domain experts efficiently discover rare phenomena from a large volume of visual data. Eureka treats the improvement of an expert's productivity as its end goal.

The design of Eureka faces three top-level concerns. First, visual data (image and video) is big, consuming a lot of storage bandwidth and network bandwidth to transmit, and taking a lot of memory and CPU cycles to process. Large-scale data sources are often distributed and distant from the expert. An efficient distributed architecture is needed to achieve good performance. Second, in specialized domains, the use of proprietary or legacy software is common. Eureka needs to be easily adapted for different domains without much deployment effort. Third, domain experts are typically not computer scientists and at best elementary programmers. Eureka needs to provide interfaces that are intuitive and easy to use by experts. We discuss how we design and implement Eureka to address these concerns in Chapter 3.

We explore how Eureka helps to reduce human labeling effort for the task of discovering rare objects in image in Chapter 4. We propose an *iterative early-discard workflow* that is made possible with transfer learning [110]. We show this new approach significantly improves human efficiency compared to exhaustive labeling and Diamond's single-stage early discard approach [69]. A model is developed for analyzing how system performance impacts human efficiency. The model shows that a slow system creates stalls and thus waste of expert time.

Following the model in Chapter 4, we investigate performance bottlenecks of Eureka in Chapter 5. We focus on the edge computing setting as new visual data is intrinsically collected on the edge. We show how Eureka's distributed design alleviates the wide area network (WAN) bandwidth bottleneck. We then study Eureka's single-node performance and observe an opportunity to greatly improve system efficiency by redesigning the storage subsystem on edge nodes. This opportunity is reminiscent of the decades-old *active disk* [84] idea, but is driven by new technology trends such as NVMe on hard disks. We elaborate our new design and present experimental evidence of its potential gain.

We claim that Eureka's effectiveness is not limited to static images. We validate this by considering a more challenging task – discovering instances of novel temporal events in video. To this end, we extend Eureka's programming abstraction to treat time, space, and content as three pillar concepts. Handling video data poses additional challenges in terms of video decoding, frame buffering, temporal cross-reference, and so on. In Chapter 6, we describe the extended programming abstraction and system optimization, and then demonstrate how we can search for novel events in videos using Eureka.

Chapter 2

Background

2.1 Edge Computing

Edge computing traces its root to Weiser's vision of *ubiquitous computing* [158], also referred to as *pervasive computing* [126]. It outlines the dream where a user can access computing services and personal data anywhere and anytime. The proliferation of smart phones circa 2000 marked a big step towards this goal in a large-scale and commercially feasible manner. Smart phones allow users to access most Internet services they can access on PCs, such as email, web browsing, and even video games. Mobile devices, by definition, encompass great potential to deliver pervasive computing, as their users can carry them almost anywhere and anytime. However, as Satya pointed out [125], mobile elements are always resource-poor compared to their static counterparts. This resource gap results from the intrinsic penalty of mobility, rather than the artifacts of current technology. Size, weight, ergonomics, and heat dissipation are always prioritized. This observation proves to be valid through generations of mobile hardware technologies.

A natural solution to addressing the resource limitation of mobile devices is to offload computation to more resource-rich computing infrastructure. This approach was first demonstrated by Noble et al [108], who implemented speech recognition on mobile devices via offloading. It was later generalized and termed *cyber foraging* in 2001 [126]. Many aspects of cyber foraging have since been studied. For example, Flinn et al [45] exploited cyber foraging to adaptively conserve energy on mobile devices. A comprehensive survey of this body of work is given in [44].

The rapid growth of public *cloud computing* services such as Amazon Web Services and Microsoft Azure in the 2010s made the above concepts widely available. Exascale data centers house large pools of compute, memory, and storage resources. The economies of scale greatly improve the efficiency of provision, utilization, maintenance, and SW/HW upgrade. Today, cloud computing is serving many intelligent mobile applications to billions of users. For example, when a user asks the Amazon assistant "Alexa, what is the weather today?", the captured audio is transmitted across the Internet into the cloud for speech recognition, natural language processing, and query from weather services; the result is returned to the mobile devices and displayed or voiced to the user. Millions of users synchronize photos taken on their smart phones to Apple's iCloud, so that they can access them on all of their PCs, tablets, and phones without carrying the original capture device around.

Cloud Location	Bandwidth (Mb/s)		Latency (ms)	
	Average	90%-tile	Average	90%-tile
US East	170.7	181.7	48	41
US West	13.7	14.2	342	336
EU West	18.0	24.9	442	431
EU Central	5.5	5.9	478	464
Asia Pacific	1.5	1.6	897	887

(Measured from Pittsburgh, PA in March 2021 via CloudHarmony)

Table 2.1: Bandwidth and Latency from US East to Different Amazon EC2 Locations

Nonetheless, cloud computing is not without its limitations. The tremendous scale and concentration of resources makes cloud data centers inevitably scarce. As a result, the cloud is far away from most mobile users. Meanwhile, offloading computation requires transmitting input data and output result across the Internet between the mobile and the cloud. Table 2.1 reports the network bandwidth and latency of different Amazon EC2 cloud locations measured from US East. As the cloud gets farther away, both network metrics deteriorate quickly. Using Netflix's recommendation of 25 Mbps for 4K videos [106], even the closest cloud (US East) can only support no more than a few camera streams, let alone sharing the precious network with multiple applications. Akamai reported that the US national average bandwidth in 2017 was merely 18.7 Mbps [6]. This poses severe constraints on bandwidth-hungry applications such as intelligent surveillance. Other genre of applications, such as wearable cognitive assistance [25, 52, 152] and mobile gaming [92], require crisp response time in order to deliver interactive and immersive experience to the users. The high latency incurred by offloading to the cloud is intolerable for these applications. Hu et al [67] quantified this issue in their study.

Edge computing emerges as an answer to the above limitations of mobile-cloud computing. Satya et al [130] present edge computing in a generalized three-tier model of modern computing landscape, as depicted in Figure 2.1. In this model, each tier represents a distinct set of properties and design constraints shared by all elements in that tier, regardless of the protocol used and the hardware vendor. While technology will certainly evolve in absolute terms over time, the relative strength and weakness between these tiers remains unchanged.

Tier-3 devices, such as smart phones, VR/AR headsets, and drones, feature sensing capability and mobility. The main purpose of these devices is to capture raw data from the physical world. For example, modern smart phones include a rich set of sensors, such as camera, microphone, accelerometer, gyroscope and GPS. Internet of Things (IoT), "smart homes," and "smart factories" are collecting data of a wide range of modalities, e.g., smart thermostats. The penalty of mobility makes Tier-3 devices always more resource-constrained than Tier-2 and Tier-1 devices. Many valuable services, such as intelligent surveillance, language translation, and high-resolution VR gaming, require considerably more compute power than that Tier-3 devices offer.

Tier-1 the cloud, by contrast, embodies almost infinite compute resources and power supply. Scalability, elasticity, and reliability are Tier-1's defining features. The success of public cloud service in the last decade has led to it being thought as the default "offload destination" for mobile applications. However, as discussed above, Tier-1 falls short for emerging applications that are

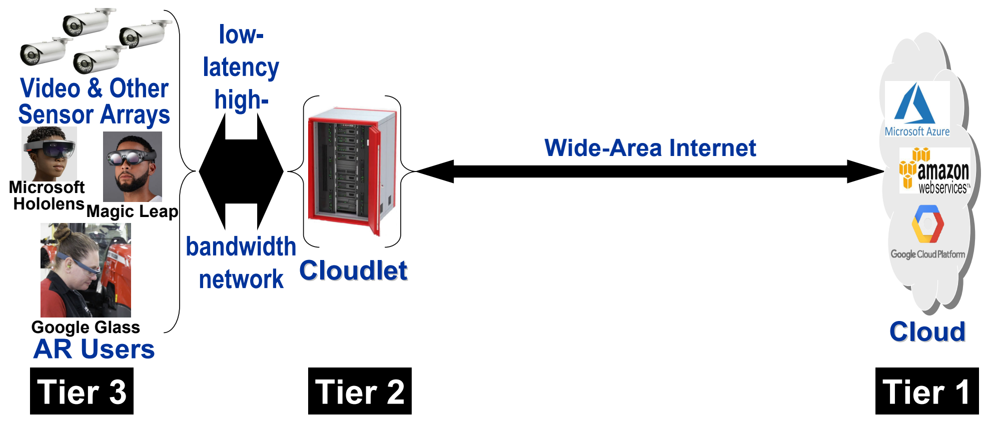


Figure 2.1: A Three-Tier Model of Modern Computing

bandwidth-hungry and latency-sensitive.

Tier-2 bridges the gap by "bringing the cloud closer." It deploys substantial compute and storage infrastructure in proximity to Tier-3, or "the edge of the Internet." The term "proximity" here refers to network rather than physical distance, indicating high-bandwidth and low-latency connection between mobile devices and the edge. Such proximity can be achieved with LAN connection or one-hop away wireless connection. Wifi, 4G LTE, and emerging 5G technologies are enabling such high-quality connections.

Tier-2 devices, also referred to as the edge [134], cloudlets [128], micro data centers [11], or the fog [16], may be in different form factors. Their common characteristics are decent compute and storage resources, reliable power supply and high-bandwidth low-latency network connection to Tier-3. Although a single cloudlet, by its name, cannot compare with the cloud in terms of hardware resources, there can be a large number of them dispersed geographically, effectively amortizing the network, storage, and compute cost. Satya et al [128] highlighted the role of virtual machines in edge computing for the advantages of rapid provisioning, portability, and mobility. Ha et al [53] studied agile VM handoff in this context. Wang et al [152] described the concept of *edge-native application* which cannot be supported with Tier-1 and Tier-3 alone. There has been increasing interest in exploiting the edge to analyze video data collected from Tier-3 sources [19, 77, 78, 153]. Applications running on Tier-2 may pre-filter raw input data so that only meta-data and a tiny fraction of raw data is transmitted over the scarce wide-area network. Our work described in this dissertation also aligns with this principle.

2.2 OpenDiamond: Interactive Search of Non-Indexed Image

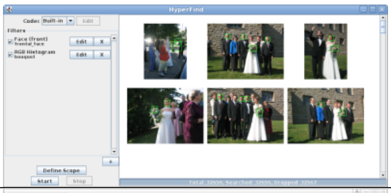
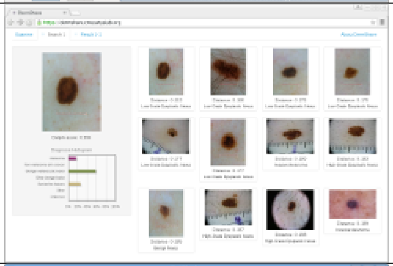
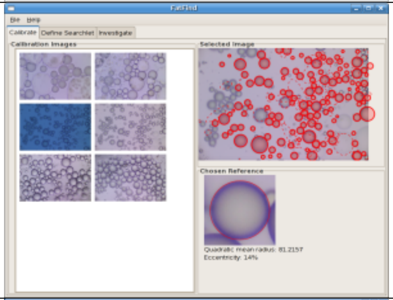
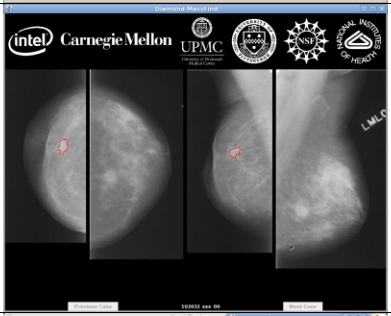
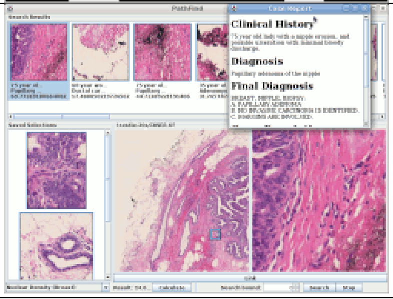
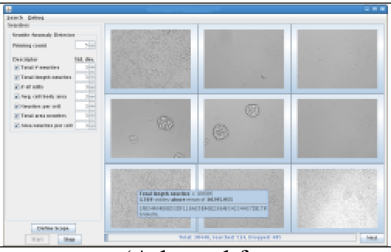
The search of *indexed* image data is traditionally studied under the theme of *image retrieval* [33, 38]. Typically, this task involves computing certain feature descriptors from the RGB content of a large corpus of images and organizing these features in storage in a way that is efficient for future querying and retrieval. Thus, most of its query time is spent on processing the pre-computed index data. This methodology requires heavy-weight pre-processing to index all images. Moreover, it assumes that the selected features are relevant to all queries that may be posed to the system. Unfortunately, the search for a novel target is by definition ad-hoc and hard to predict. The feature descriptors chosen to index the images may not be relevant for a new target. Expanding the set of indexed features incurs significantly wasteful computation and can be prohibitively expensive. In this case, processing of the RGB content at query time is necessary.

Diamond [69] is one of the earliest research efforts to address interactive search of non-indexed image data. It was originally considered in the context of active disk, where part of the application-specific processing pipeline, called "searchlet," is downloaded and executed on the storage devices in order to "discard a bulk of irrelevant data" before shipping it to the host computer. Since a host may be connected to multiple storage devices, this process may run on multiple storage nodes (backends) in parallel. The authors of Diamond [69] referred to such processing based on application logic as *early discard.*

OpenDiamond [129] is an open-source implementation of the Diamond concept in the form of a Linux middleware. OpenDiamond champions the separation of domain-specific and domain-agnostic aspects of the system. The core of OpenDiamond is a domain-agnostic runtime that manages TCP-based connections between the client and the backends. It is also responsible for triggering download and execution of "searchlets" on the backend. Domain-specific code is encapsulated under a set of domain-agnostic searchlet APIs. On the client side, a frontend GUI may be tailored to support domain-specific features and ways of interaction. Table 2.2 shows a number of domain-specific applications built on top of the OpenDiamond platform.

OpenDiamond and these applications were conceived based on the hardware technology and the status of computer vision around 2004–2010. Most applications in Table 2.2 offer image filters based on low-level features such as RGB color histogram, difference of Gaussians (DoG), histogram of oriented gradients (HOG). Some also include domain-specific hand-tuned feature extractors. There is no use of deep learning in these applications. Lastly, the data examined in all these applications can be considered as a form of static images, though it may range from everyday holiday photos to high-resolution whole-slide images obtained from microscopy. Video data has not been considered in OpenDiamond.

The implementation of Eureka, the system described in the rest of this dissertation, is extended from that of OpenDiamond. We have since adapted OpenDiamond's code base on modern software and hardware stack. In addition, we have added new features to support deep learning, just-in-time machine learning, and video processing.

	**HyperFind** is a sample domain-specific search application built on the OpenDiamond platform. HyperFind enables users to quickly search through collections of unlabeled photographs (such as holiday photos).
	**DermShare** is an easy-to-use web-based tool to assist primary care physicians in more accurately detecting melanoma. The physician provides a dermatoscopic image of an unknown lesion, and the tool discovers and displays similar dermatoscopic images from a large library of samples, along with their pathology reports and diagnoses.
	**FatFind** is a Diamond application developed in collaboration with Merck Research. It targets the time-consuming task of manually counting adipocytes (fat cells) in cell microscopy images and characterizing their size. FatFind exploits the almost perfectly circular shape of adipocytes in solution to efficiently locate fat cells
	**MassFind** is an implementation of Interactive Search-Assisted Diagnosis (ISAD), specifically for mammography. ISAD is a collaborative research effort with the University of Pittsburgh and the University of Pittsburgh Medical Center.
	**PathFind** is developed based on analysis of expected workflow by a typical pathologist. It incorporates a vendor-neutral whole-slide image viewer that allows a pathologist to zoom and navigate a whole slide image just as he does with a microscope and glass slides today. The PathFind interface allows the pathologist to identify regions of interest on the slide at any magnification and then search for similar regions across multiple slide formats. The search results can be viewed and compared with the original image.
	**StrangeFind** is an application for online anomaly detection across different modalities and types of data. It was developed with Merck to assist pharmaceutical researchers in automated cell microscopy, where very high volumes of cell imaging are typical.

(Adapted from `http://diamond.cs.cmu.edu/applications.html`.)

Table 2.2: Example Applications Built on the OpenDiamond Platform

Chapter 3

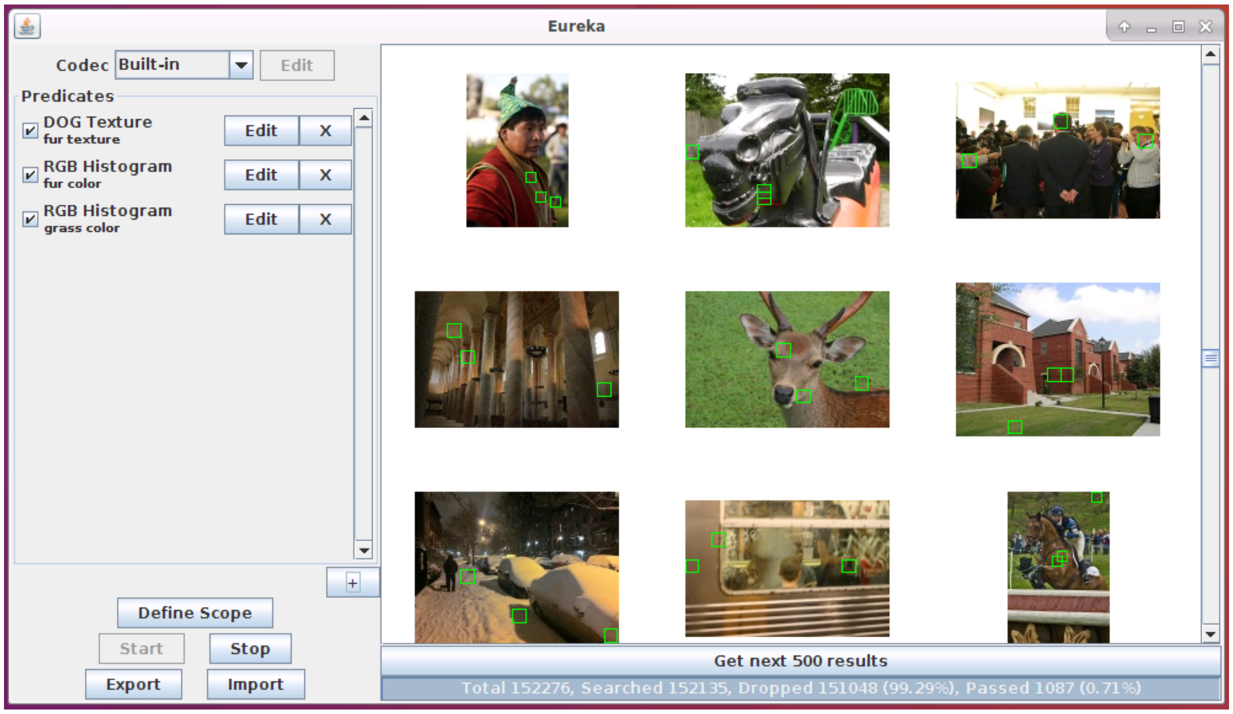
Eureka System Design

We address the research questions in this dissertation by building *Eureka*, a distributed system for interactive content-based discovery of rare phenomena in visual data. Eureka's ultimate goal is to improve an expert's human efficiency in this process. To do so, it needs to address several system design and implementation challenges.

Because we focus on the discovery of *novel* targets, by definition, we cannot have a pre-built index for that target. Hence, *content-based* analysis on the raw pixel data is necessary. This is different from existing multimedia data retrieval tasks that are largely based on indexes. Content-based analysis poses high demand for storage bandwidth, network bandwidth, main memory, and CPU cycles. In particularly, new visual data is usually generated by cameras on the edge. While today's cloud-centric wisdom suggests concentrating all data into the cloud before processing, doing so will likely saturate the scarce wide area network (WAN) bandwidth. Eureka addresses these challenges by using an edge-based distributed architecture described in Section 3.1.

Eureka is *interactive* in two senses. First, it allows a domain expert to express her heuristics through an explicit combination of early-discard filters, namely a query. The expert does so with minimal programming effort. Figure 3.1 shows a screenshot of Eureka's GUI where an expert uses three filters – a Difference of Gaussian (DoG) texture filter and two RGB color histogram filters – to search for deer images. The green color filter, for example, prunes data that does not have a vegetation background. Second, Eureka treats the data source as a stream of data items and present an item to the expert as soon as it passes the query. This gives opportunities for the expert to abort the query before it reaches the end of the stream, refine the query and then restart the search immediately. Because the query is a reflection of expert intuition, it can evolve rapidly as the expert sees results returned by Eureka and comes up with "fixes" to the query.

Domain experts are usually not computer scientists and at best moderate programmers. To facilitate interactive use by those experts, Eureka provides an intuitive programming abstraction that reduces the coding barrier and yet is amenable for system optimization. We describe this programming abstraction in Section 3.2. We further elaborate how the envisioned use cases by domain experts drive several implementation decisions of Eureka in Section 3.3.



Total 152276, Searched 152135, Dropped 151048 (99.29%), Passed 1087 (0.71%)

Figure 3.1: Eureka's GUI for composing a query to search for deer images.

3.1 Edge-based System Architecture

Visual data in real life comes from many difference sources. They can be live (e.g., real time video feed from a drone) or archival (e.g., multi-year patient record on a local hospital's servers); be mobile (e.g., vehicular dash camera) or stationary (e.g., static traffic camera). Data of interest can be coming from a number of different sources which are often distributed. Unless already centralized in the cloud (e.g., Amazon Web Services), these data sources can be broadly classified as the edge. New visual data is always generated at the edge.

A common characteristic of edge data sources is that they connect to the cloud via backbone wide area network (WAN). The popular cloud-centric philosophy suggests we transmit all data into the cloud data centers before processing it. Unfortunately, this places a huge burden on the precious WAN bandwidth. Visual data is large in volume. Unselectively streaming all data from the edge into the cloud is prohibitively costly, if not infeasible.

Eureka addresses this challenge by using an edge-based architecture that unifies processing on the edge and in the cloud. We define a *cloudlet* as a compute infrastructure that can read from a data source at high bandwidth and low latency. Here we interpret cloudlet broadly: in edge computing, a cloudlet may correspond to a micro data center, a vehicular computer, or a smart camera with on-board general-purpose compute capability; in cloud computing, a cloudlet is typically a cloud VM or container (e.g., AWS EC2 instance).

Figure 3.2 illustrates the system architecture of Eureka. A domain-specific frontend GUI runs on a client machine close to the expert. The expert composes a chain of early-discard filters (aka a query) in the GUI. A Eureka backend runs at each cloudlet. As mentioned above, the

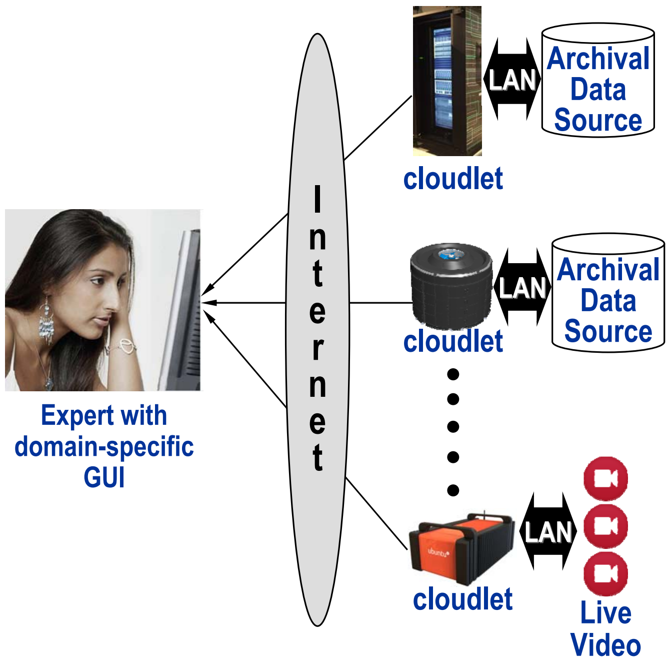


Figure 3.2: Eureka System Architecture

forms of the cloudlets and their associated data sources may be heterogeneous. When the expert starts a search, the query is serialized and pushed to the cloudlets. The backends execute on the cloudlets in parallel, evaluate the query on the data, and transmit thumbnails of undiscarded data to the frontend. Each thumbnail includes back pointers to its origin cloudlet and path to the original data file on that cloudlet. The expert sees a merged stream of thumbnails from all backends. If a thumbnail merits closer examination, a mouse click on it will open a separate window to display the full-fidelity image/video and its associated meta-data. Thumbnails are queued by the frontend, awaiting the expert's attention to examine them. If demand greatly exceeds available attention, queue back pressure throttles cloudlet processing.

Be it on the cloud or on the edge, the expert is usually distant from the data sources and therefore connections between them are across the wide area network (WAN). This is in contrast to the LAN-quality connection between a cloudlet and the data source, which is symbolized by the short and thick "LAN" arrows in Figure 3.2. Since Eureka executes early-discard pipelines on the cloudlets, the bandwidth demand between cloudlet and expert is typically many orders of magnitude smaller than the bandwidth demand between data source and cloudlet. This avoids the WAN being a bottleneck for transmitting unfiltered visual data.

When the expert's search target is scarce, it is possible that each cloudlet yields candidates at a slow rate and thus creates system stalls. As pointed out in Section 1.4, this is a poor use of expert time. To improve this situation, Eureka can harness a larger degree of parallelism by including more cloudlets (and data sources) in the process. Eureka can thus be viewed as an architecture that trades off computing resources (e.g., processing cycles, network bandwidth, and storage bandwidth) for effective use of expert attention. Efficient early-discard close to data is the key to making this trade-off effective.

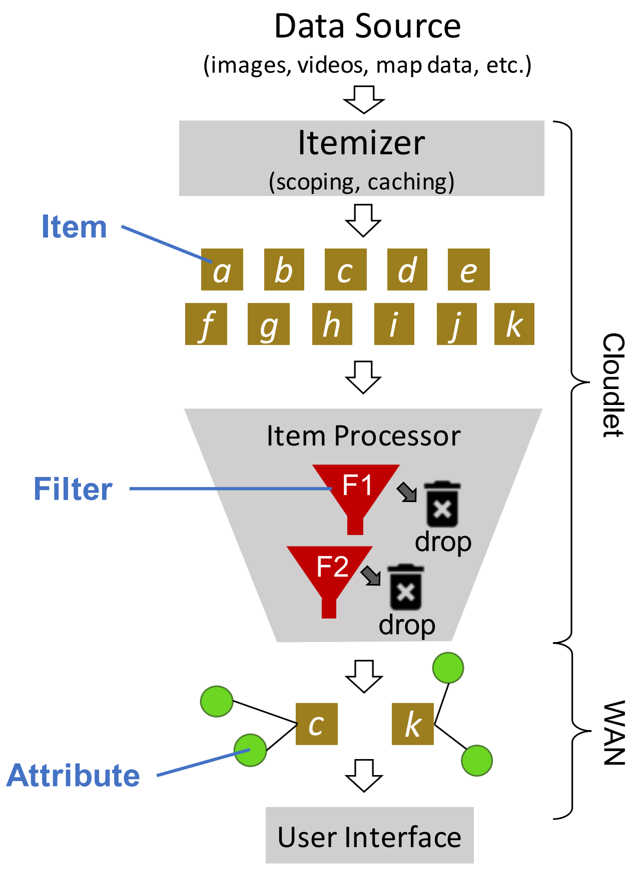


Figure 3.3: Eureka's Programming Abstraction

3.2 Programming Abstraction

Eureka encapsulates from the expert the fact that data processing is remote and distributed. It provides a programming abstraction that: (1) is easy to understand as domain experts; (2) hides the underlying distributed architecture; (3) offers modularity and composability of computer visoin building blocks. An expert considers all data sources as a merged collection of data items and composes a query to discard as many negative examples as possible. Eureka is responsible for sending the query to each cloudlet and executing it on the local data partition. Figure 3.3 depicts Eureka's programming abstraction. The itemizer parses a data source and emits a stream of items (e.g., $a, b, c, \ldots, k$). These items are next processed by the item processor, where each item is evaluated by a cascade of filters (e.g., $F1 \rightarrow F2$). Undiscarded items (e.g, c, k) along with extracted attributes are transmitted and presented to the user interface. We explain the key concepts in below.

3.2.1 Item

Eureka views data sources as unstructured collections of *items*. Items are the top-level elements in the data model and are treated as independent by Eureka. In the simplest case, an item refers to a single image in JPEG, PNG or other formats stored on the disk. An item may also be an MP4 file on the disk or a sliding window from a continuously streaming camera. The appropriate granularity of an item can be task-specific. For example, an object detection task on an image data set may use individual files as items, while an activity recognition task may use 10-second sliding-window video segments as items. The frontend allows the user to configure what constitutes an item before starting a search.

3.2.2 Filter

A *filter* is an abstraction for executing any computer vision code in Eureka. A filter's main function is to inspect items and attempt to discard them. Eureka separates its runtime framework and the filters through a narrow "Eureka Filter API," so that third-party developers can easily create and "plug-and-play" new filters without re-deploying Eureka. A filter uses the `get-parameters` call to get user-supplied parameters (e.g., example texture patches) for a query. A filter is required to implement a scoring function, `score(item)`, where it examines a given item and outputs a numeric score. The user also specifies a *threshold* score of each filter. Eureka applies the filter's scoring function to each item, and if the returned score exceeds the threshold, the item is deemed to pass; otherwise the item is discarded. An early-discard query pipeline consists of multiple filters, with corresponding parameters and passing thresholds. The current system requires an item to pass all of the filters before transmitting and presenting it to the user. This effectively implements the Boolean operator AND. Eureka performs short-circuit evaluation: once an item fails a filter, it is discarded without further evaluation by later filters in a cascade, thus achieving "early" in early discard.

3.2.3 Attribute

Another valuable function of filters is to attach *attributes* to an item as a by-product of scoring it. Attributes are key-value pairs that can represent arbitrary data (including binary), and are read-/written using the `get-attribute(item, key)` and `set-attribute(item, key, val)` Filter API. Attributes track important information extracted by filters that the user may wish to see. They also facilitate communication between different filters, where one filter reads attribute values written by another. Attributes are analogous to columns in relational databases but with significant differences. In Eureka, attributes are rarely complete for all items (rows) in the data, both due to early-discard of items in the query and due to fast-aborted searches. Additionally, unlike most databases, where the schema tends to be stable, new attributes may be created rapidly in each new query as the user applies new filters (e.g., a retrained DNN).

The user can designate a set of interesting attributes to be retrieved along with the items. Unwanted attributes are stripped off before Eureka transmits results back to the user to reduce bandwidth demand over the WAN. Returned attributes can be used for aggregations and joins using other tools such as relational databases in a post-processing step.

Filter	Synopsis
JPEG decoder	jpeg_decode() $\rightarrow$ bool Decodes a JPEG image. Set-attributes: `rgb` Returns true if successful, false otherwise.
SIFT matching	sift_match(distance_ratio: float, example: Image) $\rightarrow$ int Finds matched SIFT keypoints between example and test image. Get-attributes: `rgb` Returns number of matched keypoints.
MobileNet	mobilenet_classify(target_class: string, top_k: int) $\rightarrow$ bool Classifies image into ImageNet classes and test if target_class is in top_k predictions. Get-attributes: `rgb` Set-attributes: `mobilenet_featvec` Returns true if target_class is in top_k predictions of the test image, false otherwise.
SVM	svm(training_data: List$\langle$Image$\rangle$) $\rightarrow$ float Train an SVM with the given training set, using MobileNet's 1024-dimensional feature as SVM input. Then use the SVM to classify the test image. Get-attributes: `mobilenet_featvec` Returns probability of the test image being positive.

Table 3.1: Examples of Eureka Filters

3.2.4 Examples of Filters

Table 3.1 presents several example filters in Eureka. While some filters output a pass/fail Boolean result that evaluates to 0/1 (e.g., JPEG decoder), others output a float score (e.g., SVM confidence). Each filter accepts a set of parameters, and optionally gets/sets some attributes.

Let's take a closer look at the MobileNet filter. This filter wraps around the MobileNet [63] model pre-trained on ImageNet's 1000 classes. At run time, the filter receives two parameters — `target_class` and `top_k` — through the `get-parameters` call. It then executes the DNN over the image item in inference mode. It returns Boolean true (1) if the `target_class` is within the `top_k` predicted labels; otherwise it returns false (0).

In addition to scoring, the MobileNet filter always sets the `mobilenet_featvec` attribute to be the 1024-dimensional feature vector extracted by the penultimate layer of the DNN. Interestingly, this allows the filter to be re-purposed as a feature extractor, rather than a classifier. In this case, the predicted ImageNet labels are ignored and the threshold is set to 0, always passing an item. The feature vector is then used by downstream filters. The SVM filter illustrates such an example. It uses the `mobilenet_featvec` attribute as the input feature to train an SVM to classify the data. This is an example of transfer learning [110] to classify targets that are not in ImageNet's 1000 classes. As can be seen, the attribute mechanism allows decomposition of complex tasks into independent, manageable, and reusable components, while still adhering to the filter chain abstraction.

3.3 Optimization for Domain Experts

Eureka is an extensible architecture that can be flexibly adapted to different specialized domains. Besides, it optimizes for *system efficiency* whenever possible in order to reduce system stalls and avoid underutilizing an expert's time. In the following, we present several implementation details that provide these attributes.

3.3.1 Filter Container: Offering Software Generality

In early versions of Eureka, a filter was encapsulated as an executable program to be downloaded to the Eureka backends. At query time, Eureka's item processor launched the filters by creating sub-processes that ran natively on the same system. Eureka communicated with the filters using the Filter Protocol described earlier via inter-process communication (IPC). This was intended to provide a "plug-and-play" experience for new filters without the need to re-deploy Eureka.

Unfortunately, this goal was not fully achieved due to complexities of software dependencies. Running filters as native processes requires their dependencies be already installed in the cloudlet's OS. This can be challenging for two reasons. First, in specialized domains, the use of legacy or proprietary software is common. It is not always possible to install those dependencies on modern OSes of the cloudlets. For example, Python 2 and Java 7 are no longer supported in OSes after 2019. Second, different filters may depend on conflicting versions of the same software (e.g., of TensorFlow). Managing this complexity on the cloudlets and anticipating frequently changes due to newly-added filters is a non-trivial task.

To address this challenge, we have since added support to encapsulate filters in Docker containers. The software dependencies of a filter are included in its own container, without interfering the cloudlet host OS or other filters' containers. The use of containers also improves reproducibility even if the host OSes on heterogeneous cloudlets are different.

The orchestration of filter processes remains similar, except that filter processes are now created *inside* the container. Figure 3.4 illustrates the system resources allocated when running a three-filter chain: `Jpeg` $\rightarrow$ `MobileNet` $\rightarrow$ `SVM`. The figure depicts several implementation decisions of Eureka: (1) Eureka uses multiple worker threads to harness data-level parallelism. Each worker thread creates its own copy of the filter chain (a set of processes) and runs them on a subset of the data stream. In this way, it can easily scale on multi-core machines. (2) Different filters (e.g, `MobileNet` and `SVM`) may be packaged into the same container snapshot, often when they share software dependencies. This reduces the container snapshot's disk usage and the container's memory usage. (3) At run time, worker threads try to start new filter processes from containers that are already running. For example, two `Jpeg` processes are started in Container I, each used by a different thread. This reduces the number of containers to be created by Eureka. In total, Figure 3.4 shows two containers and six processes being used by two worker threads, each running its own copy of the three-filter query.

We empirically observe no compute or memory penalty for a process that runs inside a container. The only concern is the difference in *process creation time*, which is sub-millisecond for native processes and sub-second for container processes. The reason is because process creation inside containers is managed by the Docker daemon `dockerd`. In order to create a container

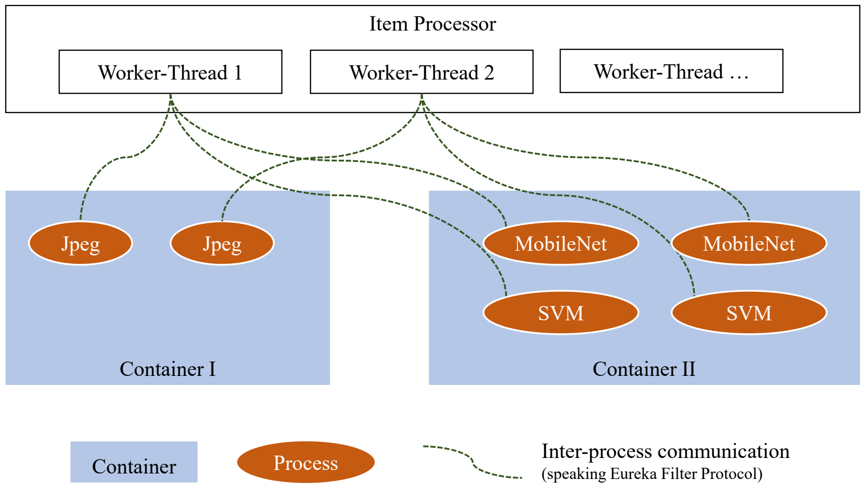


Figure 3.4: An Example of Containers and Processes at Query Time

process, Eureka must communicate with `dockerd` using an HTTP-based Docker Engine API[1]. This adds extra latency. Fortunately, Eureka's Filter Protocol is designed to iterate over many items. Specifically, the `score(item)` call implicitly concludes the evaluation of an item and refreshes the filter's internal states for a new one. In this way, a worker thread needs to create only one process of a filter and use it to evaluate thousands to millions of items. The process creation penalty is thus amortized and negligible.

Filter containers can access multi-core CPU resources as well as specialized hardware (e.g., GPUs). A typical usage of GPU is running DNNs in inference mode. To exploit the efficient mini-batch processing implementation offered by popular deep learning frameworks (e.g., TensorFlow), Eureka provides helper functions to batch items from multiple worker threads. To simplify development of new filters, we have implemented Docker base snapshots for different Linux distributions. These include all of the logic needed to interface with Eureka as well as a skeleton filter. A developer only needs to add code for the computer vision algorithm that corresponds to the filter being implemented.

Figure 3.5 shows a micro-benchmark that evaluates Eureka's single-node scalability. To remove content-dependent variance, we use a simply filter that performs a fixed number of `sqrt` calculations *per item*. Fewer `sqrt` calculations per item leads to higher item throughput. We see the use of Docker containers and Eureka's filter protocol has little impact on the linear scalability when adding more worker processes. Ultimately, the item throughput is limited by how fast Eureka receives items from the data source, which in this case is about 10,000 items per second.

Lastly, containers enables efficient deployment of new filters to a backend by exploiting delta detection and compression. When one installs files, libraries, and software in a container snapshot, Docker creates intermediate *layers* corresponding to incremental changes made to the base OS snapshot. Table 3.2 shows such layers created in the process of creating a set of image filters in Eureka. When a new set of image filters are created and deployed, Docker automatically detects that only the last layer needs to be compressed and transmitted to the cloudlets (569 −

[1]Docker Engine API v1.40. `https://docs.docker.com/engine/api/v1.40/`

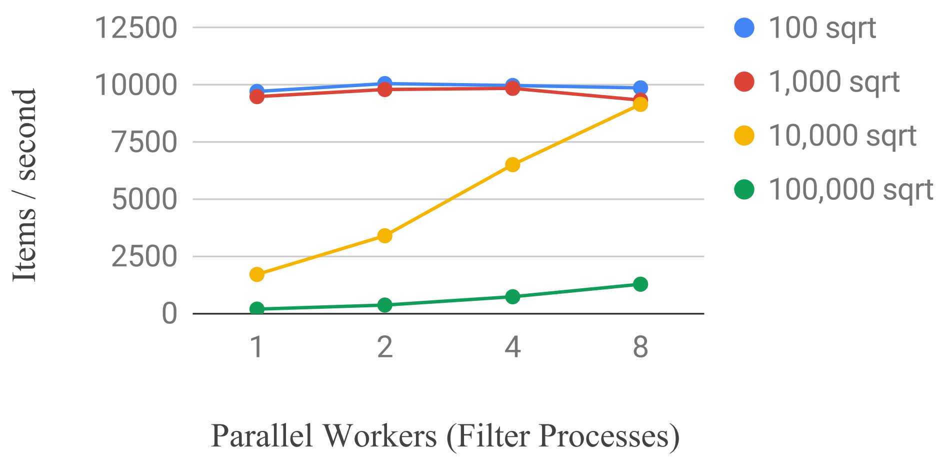


Figure 3.5: Item Throughput with Varying Computation Per Item and Parallel Workers

Docker content	Docker snapshot size
Ubuntu 16.04 base snapshot	124 MB
+ Eureka filter protocol and dependencies (e.g., Python)	245 MB
+ computer vision libraries and filters (e.g., OpenCV)	569 MB

Table 3.2: Docker Container Snapshot Sizes with Incremental Content

$245 = 324$ MB), rather than the whole new Docker snapshot (569 MB). This greatly reduces WAN transmission and deployment time. It can be seen as a variant of VM de-duplication originally described by Ha [53].

3.3.2 Itemizer: Task-specific Data Transformation

The itemizer bridges the gap between the data's raw format (e.g., local file system, multimedia database, or live camera) and Eureka's item stream abstraction, and can be configured by the user to emit items of the desired granularity. Consider a live camera as the data source. While an object detection task may consider individual frames as items, an activity recognition task may require 10-second segments as items. Furthermore, some tasks may desire the segments be overlapping, while other not. To to so properly, the itemizer needs to buffer recent frames, re-encode them into short video clips, and inject the clips into the downstream pipeline.

By default, the itemizer performs these transformations at query time. For data on the disk, this reduces data replication in a different format and saves disk space. Of course, should a certain form of transformation become common, it can be done offline and stored on the disk to reduce future computation.

3.3.3 Scoping: Utilizing Metadata and Indexes

As pointed out earlier, a pre-built index of a novel target is unlikely to exist. However, there may exist indexes of other information that could help to narrow the scope of the data to process, such as timestamp and geo-location of photos. Consider detecting a (novel) human activity. The presence of humans is a pre-condition and can be detected by existing algorithms. If such an index is available, we should utilize it to reduce the data space and thus computation.

Eureka supports utilizing metadata and indexes via *scoping* [124], an optional configurable feature of the itemizer. Before starting a search, the user can configure scoping with pointer to an external index (e.g., a database) and parameters, which is then used by the itemizer to filter the item list it passes along to the item processor.

3.3.4 Result Caching: Accelerating Interactive Search Cycles

Eureka offers an interactive experience where the abort-refine-restart cycle of a search can happen frequently. Consider the example in Figure 3.1. After seeing the initial set of results, the expert may determine that the RGB histogram filter for green vegetation is too restrictive and remove it from the query. More generally, an expert can refine a query by adding or removing filters, changing parameters of existing filters, or changing their thresholds. Relaxing the threshold of a filter may allow the expert to discover missed positive examples (aka false negatives) in the previous query; on the other hand, tightening the threshold may reduce the amount of "junk" (frivolous false positives).

When an expert refines a query, the new query is likely to be partially similar to the previous one. This creates opportunities to reduce redundant computation and thus to improve user experience. Eureka does so via *result caching* and *attribute caching*. Both forms of caching were originally described in OpenDiamond [124]. Eureka renews them for new use cases.

Eureka assumes filters are deterministic — if it re-executes a filter on the same item with the same parameters, the filter should always return the same score, read the same set of attributes, and write the same set of attribute values. When Eureka evaluates a filter on an item, it keeps track of the following information: the set of attributes read (`in_attrs`), the set of attributes written (`out_attrs`), and the final `score`. Then Eureka stores two types of entries in a key-value cache (currently implemented with a Redis database).

In the *result cache*, it writes:

$$\texttt{(item\_id, filter\_id, filter\_params[])} \rightarrow$$
$$\texttt{(score, } \{\texttt{h(a) for a in in\_attrs}\}\texttt{, } \{\texttt{h(b) for b in out\_attrs}\}\texttt{)}$$

In the *attribute cache*, it writes:

$$\texttt{h(b)} \rightarrow \texttt{b} \text{ for all } \texttt{b} \text{ in } \texttt{out\_attrs}.$$

Here, `h(·)` is the hash digest function. When evaluating a new query on an item, Eureka first identifies all filters in the query and retrieves all cache entries with the matching `(item_id, filter_id, filter_params[])` tuple, should they exist. It then validates the cache entries by examining their chains of dependency. A cache entry is deemed valid if and only if all hash digests of its input attributes match the hash digests of the output attributes of: (a) another validated cache entry; or (b) a newly-executed filter. If a valid entry is found, the cached score

Query	Filter 1	Filter 2 (varied)	Use case
Q1	SIFT key point extraction	SIFT key point matching	Object detection
Q2	MobileNet inference	SVM	Image classification

Table 3.3: Experiment query templates

Query	No cache	Cached - refined query	Cached - identical query
Q1	23	69	1535
Q2	172	793	1924

Table 3.4: End-to-end throughput (frames per second)

and output attributes are used in place of re-execution; otherwise, the filter must be re-executed. This approach avoids redundant execution, ensures correctness of cached results, and minimizes re-computations of hash digests.

Compared to query result caching techniques (i.e., hashing the whole query as a single cache key) in some DBMS, Eureka's fine-grained approach provides more opportunities to reuse prior computation even if the query is partially modified. Table 3.3 shows two query templates in Eureka. Each template consists of: (a) Filter 1 that computes a certain type of image features, which can be cached for a refined query; (b) Filter 2 that compares/classifies the feature based on user-provided examples, which may need to be re-computed for a refined query. Table 3.4 shows the query throughput on a cloudlet. Eureka's caching mechanism not only benefits when the same query is issued again ("Cached - identical query"), but also benefits when part of the query is modified ("Cached - refined query").

Chapter 4

Discovering Novel Objects in Image Data

Eureka inherits Diamond's [69] concept of early-discard to reduce human labeling effort. Diamond's approach can be considered as *single-stage early-discard*. The expert creates a query consisting of several filters based on heuristics in her mind, starts the query, and then examines the result stream, from which she discovers the rare positive examples. When the filters perform better than random sampling, the expert's situation is better than exhaustive labeling.

This approach, unfortunately, is not good enough. Diamond includes filters based on classical computer vision feature extractors, such as scale-invariant feature transform (SIFT) and difference of Gaussians (DoG). The accuracy of these filters is at most moderate. When the search target has a very low base rate, the expert still needs to sift through a lot of false positive examples. Table 4.1 shows Diamond's effectiveness in discovering three rare targets (deer, Taj Mahal, fire hydrant) from the YFCC100M [145] data set. Despite of the use of early-discard filters, the expert still needs to label tens of thousands of images just to discover 100 positives.

Can the expert create more effective early-discard filters? In Diamond, the only way to do so is through creating more sophisticated computer vision algorithms. This requires knowledge of computer vision and programming skills, and is often beyond the reach of domain experts.

Recent advancement in deep learning sheds new light on this problem. In particular, the use of *transfer learning* [110] based on DNN-extracted features allows us to create more accurate filters by simply providing more examples and this can be done at interactive speed. We elaborate this idea, which we call *just-in-time machine learning*, in Section 4.1.

The viability of just-in-time machine learning motivates us to propose an *iterative discovery workflow* in Eureka described in Section 4.2. This workflow improves the accuracy of early-discard filter progressively without the need of coding, in contrast to Diamond's single-stage approach. We then evaluate the efficacy of this workflow on a classic task – discovering novel objects in static image data – in Section 4.3. Our result shows Eureka significantly reduces human labeling effort compared to exhaustive labeling and Diamond.

Finally, to generalize our findings, we develop a model in Section 4.4 for analyzing how human-efficiency is impacted by a wide range of factors, including system efficiency, target scarcity, and filter accuracy.

Target	Estimated base rate in YFCC100M	Filter precision in OpenDiamond	Hand-labeled images to discover 100 true positives
Deer	0.07 %	0.27%	37,000
Taj Mahal	0.02 %	0.11%	90,000
Fire hydrant	0.005%	0.09%	111,000

Table 4.1: Diamond's effectiveness in discovering rare objects in YFCC100M.

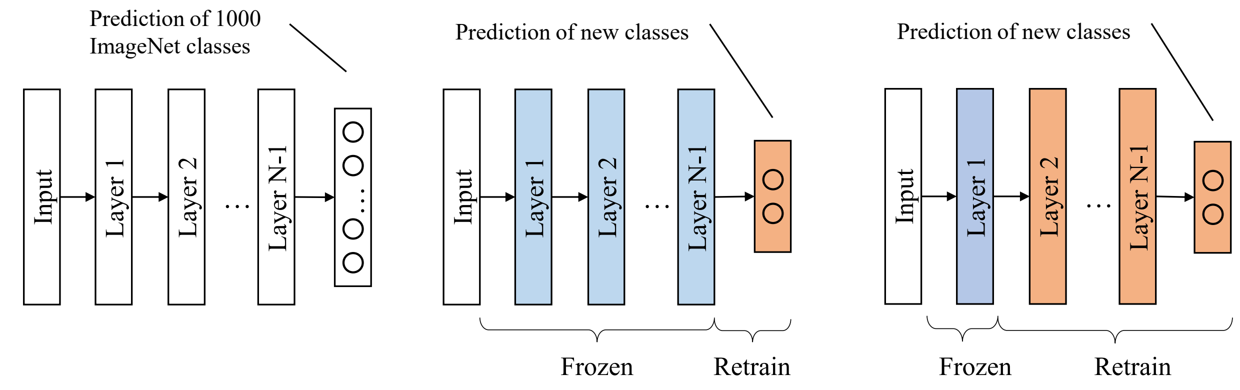
4.1 Just-in-time Machine Learning

A DNN consists of many layers, each of which progressively computes some feature values based on the values in the previous layer. Usually, the last layer outputs the prediction of the task for which it is being trained. In the case of ImageNet, they are probabilities of the 1000 classes in the training set (Figure 4.1(a)). Each layer has parameters (aka weights) to be learned during the training process outlined in Section 1.3. A DNN created from scratch will initialize these weights with random values, and use training examples to adjust the weights until the model converges. This process requires a lot of training examples and compute resources, and may take days or weeks to finish even on a GPU. If there is insufficient data for a large model, the model's output will be largely random and thus unuseful.

For a novel target with a dozen of examples, training a DNN from scratch is not promising. Fortunately, there is hope. It is commonly believed that if a DNN has been trained on a sufficiently large and generic data set (like the ImageNet), it will learn to extract intermediate features (e.g., colors, edges, shapes, body parts, furry patterns) that are relevant beyond the scope of the original task (e.g., classifying unicorns). This concept is called transfer learning [110] as the knowledge learned from an old task is "transferred" to a new task.

More concretely, this is done as follows. In its simplest form, all layers up to the penultimate layers of a *pre-trained* DNN are frozen, when only the last layer is re-initialized randomly (Figure 4.1(b)). The last layer must be changed to match the new task. For example, if the new task is a binary classification task, then it must output a 2-dimensional rather than a 1000-dimensional vector. We then re-train the new model with training examples of the new task. During training, only the final layer's weights are tuned; weights in the frozen layers are unchanged and as a result their outputs can be computed only once and cached. An alternative view to seeing this is to think of the all-but-last-layer part of the pre-trained DNN as a given feature extractor, and we are training a light-weight machine learning model (e.g., an SVM) on top of those features. We call this *shallow transfer learning*. Because the trainable part is small and shallow: (a) it requires fewer training examples to converge; (b) it runs much faster, in particular, at interactive speed. In practice, training an SVM on top of MobileNet's 1024-dimensional features takes a few seconds to minutes.

In a more general form of transfer learning, more layers towards the input are unfrozen and become trainable (Figure 4.1(c)). When training for a new task, the weights of the intermediate layers are initialized from the pre-trained model, rather than randomly. The intuition is that those weights have stored useful knowledge which is a good basis for adaptation. This is also referred to as "finetuning." How many layers to unfreeze is a trade-off. The more layers we unfreeze: (a) the more training examples of the new tasks are needed; (b) the more time it takes to train; (c)



(a) A pre-trained DNN on ImageNet (b) Shallow transfer learning (c) Deep transfer learning

Figure 4.1: Transfer Learning from A Pre-trained DNN to A New Task

the higher accuracy the new model can potentially achieve. We call this *deep transfer learning*. Depending on the model size and the number of layers unfrozen, deep transfer learning can take minutes to hours to finish.

Given a reasonable amount of compute resources (e.g., GPUs), transfer learning can be done approximately at interactive speed. Once configured, all that an expert needs to do in order to retrain a new model is to supply a training set. A more accurate model becomes available after moderate waiting and can be used immediately. We call this concept *just-in-time machine learning*. It motivates Eureka's iterative discovery workflow introduced in the next section.

4.2 Iterative Discovery Workflow

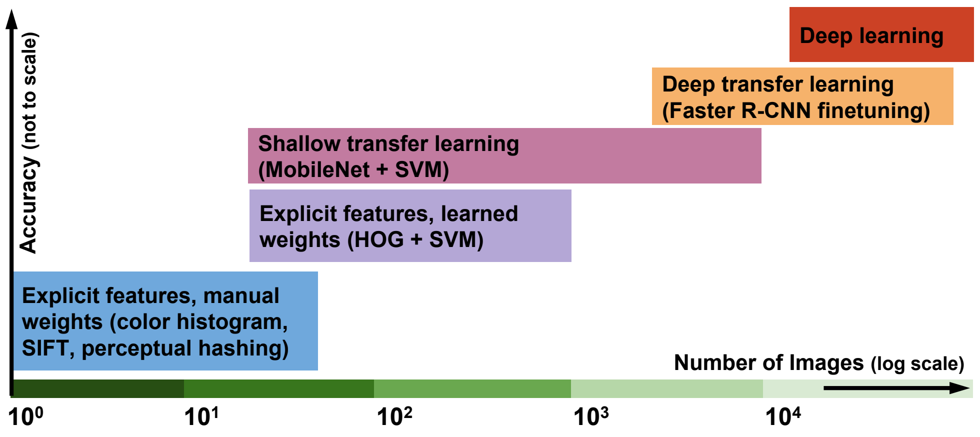
As a working example, consider the scenario described in Figure 4.2. Starting from just a few example images, how can the expert bootstrap her way to the thousands to tens of thousands of images needed for deep learning? The base rate is low — i.e., that the rodent is rarely seen, and hence there are very few images in which it appears. If a good classifier already existed, a single-stage early-discard search could run in parallel on a large number of cloudlets. The number of false positives would be low, and the rate of true positives would be reasonably high. The expert would neither waste her time rejecting obvious false positives, nor would she waste time waiting for the next image to appear. Most of her time would be spent examining results that prove to be true positives. Alas, this state of affairs will only exist at the end of the Eureka workflow. No classifier exists at the beginning. What can we use for early discard at the start of the workflow?

Eureka's discovery workflow is based on the observation depicted in Figure 4.3. A spectrum of different computer vision techniques offer different trade-offs between accuracy (higher is better, but not to scale) and the amount of labeled training data that is available. When there are too few examples for a technique, its output is unreliable; providing more examples often helps improving accuracy up to a certain point; beyond that the technique's performance is saturated and can no longer be improved.

While the ultimate goal of deep learning sits on the extreme right, the expert starts on the

An infectious disease expert has just learned that a shy rodent, long considered benign, may be the transmitter of a new disease. There is a need to create an accurate image classifier for this rodent so that it can be used in public health efforts to detect and eradicate the pest. The expert has only a few images of the rodent, but needs thousands to build an accurate DNN. There are likely to be some untagged occurrences of this rodent in the background of images that were captured for some other purpose in the epidemiological image collections of many countries. A Eureka search can reveal those occurrences. In the worst case, it may be necessary to deploy cloudlets with large arrays of associated cameras in the field to serve as live data sources for Eureka.

Figure 4.2: Example: Infectious Disease Control



Accuracy (not to scale)
Deep learning
Deep transfer learning (Faster R-CNN finetuning)
Shallow transfer learning (MobileNet + SVM)
Explicit features, learned weights (HOG + SVM)
Explicit features, manual weights (color histogram, SIFT, perceptual hashing)
Number of Images (log scale)
10⁰
10¹
10²
10³
10⁴

Figure 4.3: Training Data Set Size vs. Accuracy Trade-off

extreme left. An interactive workflow helps an expert to efficiently climb the stairs and reach the right. In the very beginning, the Eureka GUI allows simple features such as color and texture to be defined by example patches outlined on the few training examples that are available. This defines a very weak and yet better-than-random classifier that can be used as the basis of early discard. Because of the weakness of the classifier, there are likely to be many false positives. Unless the expert restricts her search to just a few data sources, she will be overwhelmed by the flood of false positives. Buried amidst the false positives are likely to be a few true positives. As the expert sees these in the result stream, she labels and adds them to the training set. Over a modest amount of time (tens of minutes to a few hours, depending on the base rate and number of cloudlets), the training set is likely to grow to a few tens of images. At this point, there is a sufficient amount of training data to create a classifier based on more sophisticated features such as HOG (histogram of oriented gradients), and learned weights from a simple machine learning algorithm such as SVM. The resulting classifier is still far from the desired accuracy, but it is significantly improved. Since the improved accuracy reduces the number of false positives, the number of data sources explored in parallel can be increased by recruiting more cloudlets. Once the training set size reaches a few hundreds, shallow transfer learning can be used. This yields an improved classifier that further reduces false positives, and allows further expansion in the number of data sources, thus speeding up the search. Shallow transfer learning may be repeated several times, each time with a larger training set, until it saturates. Once the training set size reaches a few thousands, deep transfer learning can be used and beyond that, deep learning.

Compared to Diamond's single-stage early-discard, Eureka lets a domain expert create more accurate filters progressively without the need to write code. The expert does so by selecting the appropriate filter from the spectrum and providing the training set. Two metrics improve hand-in-hand in this process: (a) the size of the labeled training set collected so far; (b) the accuracy of the filter being deployed. A larger training set allows the creation of more accurate vision filters, which in turn reduces false positives, making it more human-efficient to discover more examples and grow the training set. Hence, an expert's productivity is improved over time.

4.3 Evaluation of Productivity

In this section, we evaluate whether Eureka is effective in improving an expert's productivity. We consider the task of discovering rare objects from the YFCC100M data set [145]. This data set includes 99.2 million (10^6) images uploaded to Flickr between 2004 and 2014. The data is unlabeled and has not been filtered or re-balanced for particular classes. Therefore, we expect it contains a long-tail distribution of rare objects found in mundane photos.

4.3.1 Evaluation Methodology

Two graduate students (including the author) played the role of an expert and conducted case studies by applying the iterative workflow to search for rare targets in the data. We choose three search targets (in descending order of base rate): (1) deer; (2) Taj Mahal; and (3) fire hydrant. Although these are not targets from specialized domains, they are fairly rare in YFCC100M and we do not have (or use) any pre-trained DNN detectors or crowd-sourcing. For each target, five examples are provided for bootstrapping. Figure 4.4 shows example images of those targets.

We keep track of the early-discard filters used in each iteration, the duration of each iteration, the images presented in the GUI, and the positives/negatives labeled by the user. We consider two major criteria:

- the total elapsed time taken to discover a training set of a certain number (100) of positives;
- *productivity:* number of positives discovered per minute, which changes over the iterations.

We compare Eureka against two alternatives: *brute force* and *single-stage early-discard* represented by Diamond. With *brute force,* the user manually inspects and labels every image. For targets with a low base rate this requires the user to plow through many images before collecting even a small training set. Single-stage early-discard improves the situation by allowing the user to create a filter chain in the beginning to perform early-discard. These filters are based on the bootstrapping examples of the target, and are essentially identical to the ones used in the first

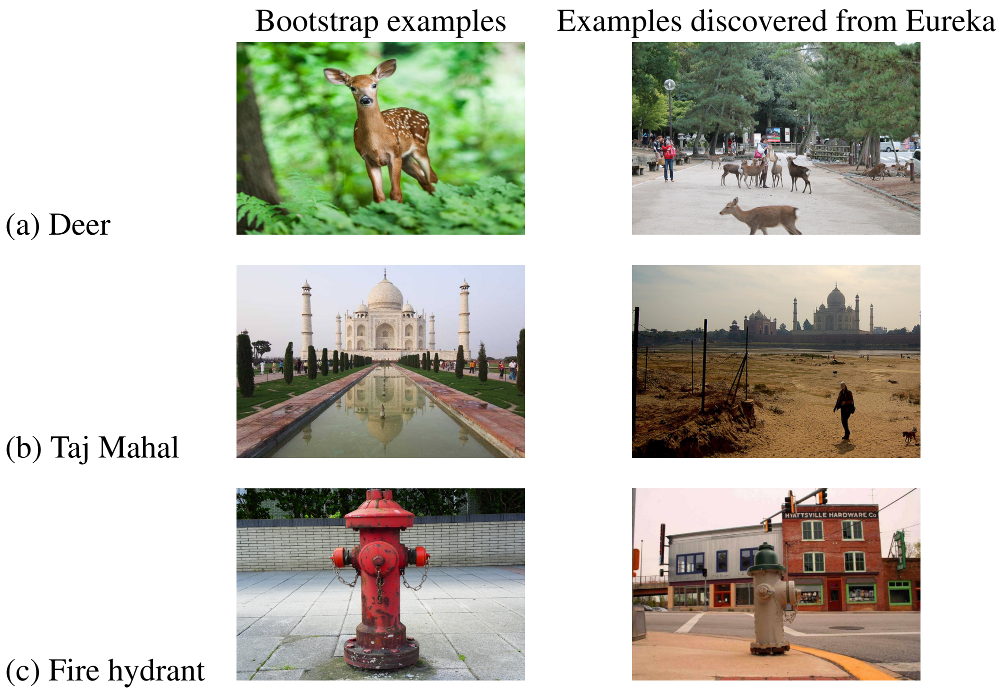


Figure 4.4: Examples of Targets Used in Our Case Studies

iteration of the Eureka experiments. However, there is no use of machine learning or of iterative refinement based on new data.

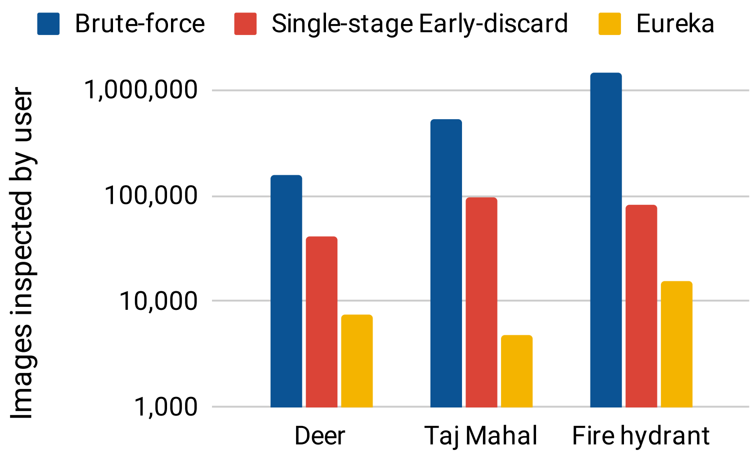
We run experiments on 8 cloudlets. Each cloudlet has a 6-core/12-thread 3.6GHz Intel Xeon E5-1650 v4 processor, 32GB DRAM, a 4 TB SSD for image storage, and an NVIDIA GTX 1060 GPU (6GB RAM). The 99.2 million images in YFCC100M are evenly divided among the cloudlets. Each cloudlet accesses its own subset of the data from its SSD, thus ensuring high bandwidth and low latency. The user's GUI connects to the cloudlets over the Internet.

For each target, we went through five Eureka iterations, reaching a total of approximately 100 positive examples. No coding was needed in this process. The user selected filters, labels results, and supplies training examples through the Eureka GUI. The early iterations usually consist of simple filters based on color (RGB), texture (difference of Gaussians, DoG), etc. Once more than 10 positives are collected, we switched to transfer learning based on MobileNet + SVM. This uses a pre-trained MobileNet to extract 1024-dimensional feature vectors, on top of which we train a new SVM for the novel class. We retrained the SVM (as a new iteration) when the training set was approximately doubled. In general, it is difficult to know *a priori* the optimal point to trigger retraining. We empirically find the double rule works well in practice.

For the purpose of scientific evaluation, we used a disjoint subset of input data for each iteration, so that the filters always run on unseen data. This facilitates unbiased evaluation of the filters and also mimics a streaming setting. Therefore, the following results did not take advantage of result/attribute caching.

Target	Estimated base rate	Images inspected by user	Positive examples discovered
Deer	0.07 %	7,447	111
Taj Mahal	0.02 %	4,791	105
Fire hydrant	0.005%	15,379	74

Figure 4.5: Summary Results for Case Studies



Y-axis in log scale.

Figure 4.6: Images Presented to and Labeled by User (Including Positives and Negatives)

4.3.2 Results

Figure 4.5 summarizes the Eureka's results of discovering approximately 100 examples for each of the three targets. Base rates are estimated based on the metadata of Flickr tags, titles, and descriptions. This metadata is only used in analysis of the results and *not* used in the search process. Although this measure is subject to inclusion error (tag without actual target) and exclusion error (target without tag), it provides at least a crude estimate and still serves as a basis to compare the scarcity of different targets.

Figure 4.6 compares Eureka to alternative approaches in terms of the number of images the user labeled. This measure includes both targets (true positives) and non-targets (false positives) presented to the user and is a direct indicator of human effort. For brute force, the number is extrapolated using the estimated base rate. For single-stage early-discard, this is based on the precision of the filters used in the first Eureka iteration. We see that single-stage early-discard reduces demand for human effort by an order of magnitude over brute force. Eureka reduces demand by a further order of magnitude. In below, we discuss each case study in more details.

	Filters	Examples		Items processed	Items shown	New hits	Pass rate	Precision	Time	Productivity
		+	-							
	Initial set of images	5								
1	RGBh x2 + DoG	5	0	991,814	1,836	5	0.19%	0.27%	12.53	0.40
2	RGB x2 + DoG	10	0	652,357	2,002	5	0.31%	0.25%	13.98	0.36
3	MobileNet + SVM	15	15	90,047	1,704	17	1.89%	1.00%	11.40	1.49
4	MobileNet + SVM	32	32	130,266	1,204	35	0.92%	2.91%	8.25	4.24
5	MobileNet + SVM	67	67	247,039	701	49	0.28%	6.99%	10.27	4.77

Number of initial examples = 5
Examples = positive(+)/negative(-) examples used to create/train the filters
Items processed = Images processed by Eureka on cloudlets (including discarded and undiscarded)
Items shown = Images passing all filters, transmitted and shown to user
New hits = Images labeled as true positives by user in that iteration
Pass rate = Items shown / Items processed
Precision = New hits / Items shown
Time = Wall clock time of that iteration (minutes)
Productivity = New hits / Elapsed time (# per minute)

Table 4.2: Case Study: Building A Training Set of Deer

Case Study: Deer

Table 4.2 reports the detail results of the deer case study. In the first iteration we used an RGB color histogram filter and a DoG (Difference of Gaussians) texture filter. Patches of deer fur from the bootstrapping images were given to the DoG texture filter as reference examples. Patches defining the color of the deer fur and verdure of the scene constituted the two RGB histogram filters. The fact the verdure color is not part of a deer's body and yet a useful filter is, by definition, an expert's heuristics.

Each of the five iterations lasted 8–14 minutes, with the variation reflecting image processing time on cloudlets, filter accuracy, and human labeling time. As with most animals, deer is a highly deformable class of objects. The RGB histogram and DoG texture filters showed no improvement over two successive iterations even when more examples are provided. The MobileNet + SVM filter introduced in iteration 3, by contrast, showed substantial improvements across subsequent iterations in terms of precision. Most importantly, it resulted in greater productivity of the expert (last column). Over the five iterations, the productivity increased from 0.40 new positives per minute to 4.77, an improvement of more than 10x.

Case Study: Taj Mahal

Table 4.3 reports the detail results of the Taj Mahal case study. Since the Taj Mahal is a rigid structure with distinctive features, we used a SIFT (Scale Invariant Feature Transform) filter. We further added a RGB color histogram filter (for the white marble) and a human detector (for tourists). Again, this is based on contextual knowledge that Taj Mahal is a popular tourist destination and is likely to have humans in its images.

On the second iteration, two HOG filters (Histogram of Oriented Gradients) were created

	Filters	Examples		Items	Items	New	Pass	Preci-	Time	Product-
		+	-	processed	shown	hits	rate	sion		ivity
	Initial set of images	5								
1	RGB+SIFT+Human	5	0	850,352	3,741	4	0.44%	0.11%	30.17	0.13
2	HOG x2	9	0	245,315	352	5	0.14%	1.42%	9.88	0.51
3	MobileNet + SVM	14	14	228,266	343	13	0.15%	3.79%	8.07	1.61
4	MobileNet + SVM	27	27	590,187	172	37	0.03%	21.51%	20.50	1.80
5	MobileNet + SVM	64	64	633,560	183	46	0.03%	25.14%	15.63	2.94

Columns have the same meaning detailed in Table 4.3.

Table 4.3: Case Study: Building A Training Set of Taj Mahal

	Filters	Examples		Items	Items	New	Pass	Preci-	Time	Product-
		+	-	processed	shown	hits	rate	sion		ivity
	Initial set of images	5								
1	HOG x2	5	0	524,136	6,643	6	1.27%	0.09%	13.00	0.46
2	HOG x3	11	0	523,008	3,133	5	0.60%	0.16%	15.15	0.33
3	MobileNet + SVM	16	16	210,688	1,775	9	0.84%	0.51%	7.68	1.17
4	MobileNet + SVM	25	25	517,789	2,856	24	0.55%	0.84%	17.52	1.37
5	MobileNet + SVM	49	49	973,828	972	30	0.10%	3.09%	23.18	1.29

Columns have the same meaning detailed in Table 4.2.

Table 4.4: Case Study: Building A Training Set of a Fire Hydrant

based on the 9 available examples, to capture the shape of (1) minarets, and (2) small domes. From the third iteration onwards we used a MobileNet + SVM filter which was improved in each iteration by adding the new positive examples obtained in the prior iteration. The returned false positives were mostly buildings such as Humayun's tomb and Sikandara which closely resemble Taj Mahal's dome and entrance. Similar the deer case study, there is an improvement of productivity over five iteration. In the final iteration, a quarter (25.14%) of the items presented to the user are true positives.

Case Study: Fire Hydrant

Table 4.4 reports the detail results of the fire hydrant case study. The base rate of fire hydrant is much lower than the first two targets chosen. HOG filters were used initially to capture the vertical cylinder shape of the hydrants. From the third iteration onwards, using a MobileNet + SVM filter helped to improve precision. Many of the false positives returned by Eureka include British royal mail boxes and traffic cones that resemble fire hydrants.

In the last iteration, the classifier accuracy improves significantly from 0.84% to 3.09%, but productivity stalls at about 1.3 positives per minute. This is due to: (a) the low inherent base rate of the target; and (b) the limited resources of 8 cloudlets. In particular, we observed user-side wait time in this iteration, a sign of underutilized expert time. In general, a wider range of factors could cause such situations. In the next section, we formally study these factors and propose a model for predicting human efficiency.

		Ground Truth (annotated by expert or confirmed by scientific experiment)	
		Positive	Negative
Prediction (output from a model)	Positive	TP (true positive)	FP (false positive)
	Negative	FN (false negative)	TN (true negative)

$$W = TP + FP + FN + TN$$

Figure 4.7: Notations

$TPR = \dfrac{TP}{TP + FN}$	True positive rate (aka "recall" or "hit rate")
$FPR = \dfrac{FP}{FP + TN}$	False positive rate (aka "false alarm rate")
$BR = \dfrac{TP + FN}{W}$	Base rate (aka "prevalence")

Figure 4.8: Metrics Pertaining to Classifier Accuracy

4.4 Modeling User:System Match

One of Eureka's goal is to best utilize an expert's time without over-committing system resources. We consider this problem by comparing the speed at which an expert can label the candidates and the speed at which Eureka returns candidates. We call the ratio between them *User:System Match*. In the ideal case, the match should be close to 1. That is, the expert is continuously labeling new candidates while not creating a backlog. A match > 1.0 implies the expert is constantly waiting in front of a blank screen for candidates to show up. This is an under-utilization of the precious expert time. A match < 1.0 means Eureka is showing new results faster than the expert can consume them. In this case, some results may never been seen and thus the computation/bandwidth is wasted; unnecessary resources may have been allocated on cloudlets, which could have been released for other edge applications.

An expert's labeling speed is fundamentally subject to human's cognitive ability, and thus typically within small variations depending on the task. The system's speed of returning candidates, however, depends on a wide range of factors, including the intrinsic scarcity of the target, distribution of the data set, accuracy of the filters, and software/hardware resources on the cloudlets. In the following, we model how these factors impact the User:System Match.

4.4.1 Classifier Metrics

Figure 4.7 and 4.8 give the notations and metrics to describe the "goodness" of a classifier and the "scarcity" of a target. Note that we consider TPR (true positive rate) and FPR (false positive rate) separately. Because Eureka is dealing with unbalanced class distributions, the simpler single-number $accuracy = (TP + TN)/W$ provides little insight. A say-no classifier that drops everything would achieve high accuracy but could not discover a single example of the rare object. In that case, TPR will be extremely low (0), which we explicitly account for. Another common used metric is $precision = 1 - FPR$. We use TPR and FPR for the ease of analysis, as they both move in the same direction as the result delivery rate.

4.4.2 Result Delivery Rate

While machines often compute faster than humans, Eureka may not return results fast enough, because only *undiscarded* items are passed and shown to the expert. When the base rate BR of a target is low, a classifier will yield candidate at low frequency even if it is very accurate. This can be captured by the *pass rate* defined below. Note that in a low base rate scenario, BR is small and $(1 - BR)$ is large. Hence, FPR governs the pass rate more than TPR.

$$\begin{aligned} pass\_rate &= BR \cdot TPR + (1 - BR) \cdot FPR \\ &\approx (1 - BR) \cdot FPR \\ &\text{(when } BR \text{ is small)} \end{aligned}$$

Next, we must take into account the computation cost of executing the classifiers. Let t_s be the average time to evaluate a filter chain on a single image on a cloudlet. Typically, this encapsulates the software and hardware configuration of the cloudlets, the availability of accelerators, concurrently-running applications, etc. We can then model the result delivery rate from a single cloudlet with *time to next result (TTNR)*:

$$TTNR = \frac{t_s}{pass\_rate}$$

Scaling out the system to N cloudlets/data sources reduces user-side $TTNR$ (and thus increases delivery rate), assuming the WAN bandwidth is not a bottleneck:

$$TTNR(N) = \frac{t_s}{N \cdot pass\_rate}$$

Finally, let's denote an expert's average time to inspect and label a single image as t_u. The User:System Match can be calculated as

$$\frac{TTNR(N)}{t_u}$$

.

4.4.3 Analysis

The User:System Match depends on the true positive rate TPR, false positive rate FPR, base rate BR of the target, and the number of sources N. It also depends on the time for the classifier to process an item at one cloudlet t_s relative to the time for the expert to evaluate a delivered item t_u. Without loss of generality, in the following, we assume $t_u : t_s = 5$. Other values of this ratio would scale the results differently, but the qualitative pattern would be the same.

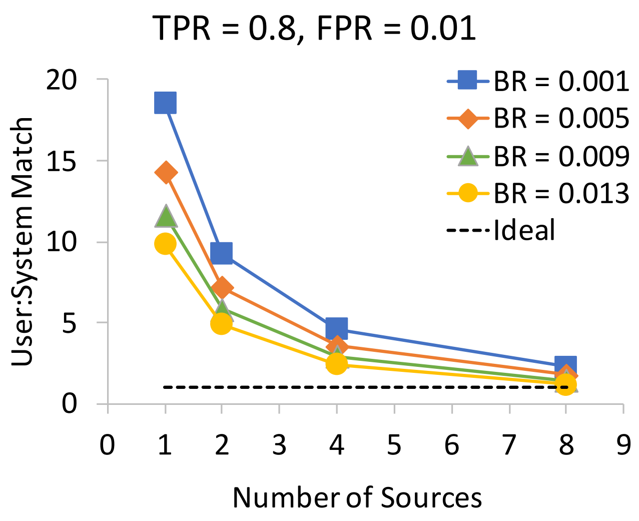
Based on the model described above, Figure 4.9 explores the User:System Match under a wide range of parameter values. The dotted line marks the ideal match of 1.0. Higher than 1.0 implies user wait time and underutilized expert attention. Lower than 1.0 implies wasted system resources. Figure 4.9(a) illustrates the effect of a good classifier ($TPR = 0.8, FPR = 0.01$). Under these conditions, the User:System Match is correlated with the base rate and number of sources/cloudlets. Because target items are scarce, and most of the items that Eureka returns are true positives, it is beneficial to scale out the system to more sources – more so when the base rate is lower. The optimal match is approached here with 8 sources, regardless of base rate.

Figure 4.9(b), by contrast, shows the disastrous effect of a bad classifier with high $FPR = 0.1$. As the number of sources increases, the User:System Match quickly drops below 1. The expert is flooded with mostly false positive results. The system backs up, wasting resources. Base rate is irrelevant here because it is not the sparse targets buried in the results that are consuming most expert attention. Rather it is the majority of false positives. At the lowest base rate $BR = 0.001$, the user needs to reject 125 items for each one accepted.

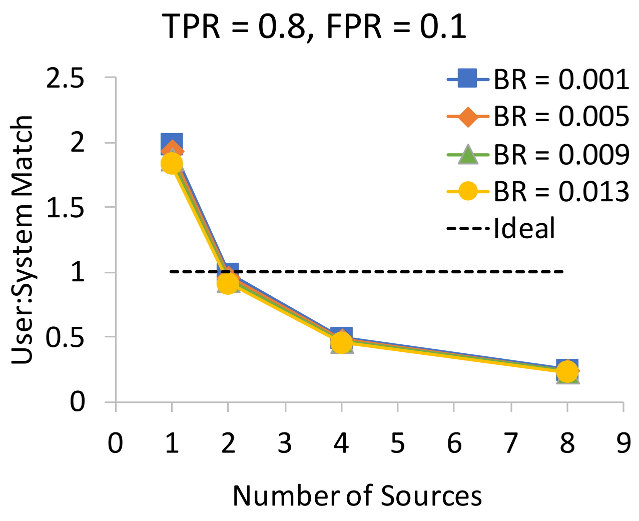
4.4.4 Discussion

In above, we focus on the temporal performance of the system and the expert. Nonetheless, time is not the only concern. It is also desired to keep the expert occupied with meaningful decisions that will ultimately improve the filters. It is easy to let the system send in more "junk" to eliminate user wait time. Imagine a scenario when the User:System Match is high, that is, the expert waits a long time to see the next candidate. It may be tempting to increase the system delivery rate by lowering the thresholds of the filters. Unfortunately, this tends to increase FPR and occupy the expert with obviously bad candidates (false positives) that have little value in improving the filters. Ultimately, the solution is to leverage the iterative workflow to improve the filter accuracy as soon as possible. Ideally, this will reduce false positives without increasing false negatives.

The model is idealized and is intended for studying how the various parameters interact in Eureka. It is not intended to predict numerical outcomes in real-world use cases, where the base rate is generally unknown; the targets are not uniformly distributed; and the ratio of machine to human processing time may be greatly skewed in favor of the machine. Nonetheless, by characterizing the User:System Match, the model provides an observable metric that can be used in Eureka to recruit or omit cloudlets and thus move toward an optimal match.



(a) Low False Positive Rate (Good Classifier)



(b) High False Positive Rate (Bad Classifier)
This graph shows User:System Match under varying number of cloudlets (N), base rates (BR), and different classifier accuracy (TPR, FPR). The ideal match of 1.0 is marked the dotted line.

Figure 4.9: Negative Effect of High False Positive Rate

4.5 Related Work

The practice of using deep learning can be generally arranged in three stages: (1) gathering and annotating training data; (2) designing and training a DNN; (3) deploying the DNN in the field to detect objects of interest. To date, much of stages (2) and (3) have been automated and accelerated by systems such as TensorFlow [2]. Stage (1), however, remains largely manual.

Researchers have taken advantage of millions of images/videos uploaded to the Internet since the inception of social platforms (e.g., Flickr, Youtube). This saves the effort of travelling to a place and taking photos of an object. Nonetheless, the task of annotating the data is not done. The sheer volume of that data entails many person-years' labeling work. Initially, this task was often performed by diligent researchers and in-house paid/volunteer annotators. As the data size grows, *crowd-sourcing* becomes the standard solution [86]. Data sets including ImageNet [123] and Charades [135] crowd-sourced the labeling task on Amazon Mechanical Turk (AMT). This essentially harnesses human-level parallelism. AMT is not free though. Therefore, *active learning* [150, 156] aims to reduce the labeling cost without hurting accuracy of the trained model by selectively requesting human annotations. This is usually done by choosing a subset of most "informative" examples to label based on some criteria.

For reasons discussed in Section 1.4, crowd-sourcing is not viable in specialize domains. Nguyen et al. [107] consider the task of literature screening and use a mix of crowd and experts to reduce total labeling cost. They assume that the crowd are still to some extent competent in labeling the ground truth. Similar to Eureka, Snorkel [117] considers an expert-only setting. However, they request the experts to write "labeling functions" rather than inspecting and annotating examples, which may be a barrier for domain experts. Similar to Nguyen et al., Snorkel only considers text data. We focus on visual data in this dissertation.

Data augmentation is a standard technique in computer vision to create quasi-new training examples from existing ones. Typically, it involves cropping, rotating, flipping, and randomly distorting *existing* training examples. Naturally, the generated examples inherit their ground truth labels from the original one. The purpose of data augmentation is to make the trained DNN more robust to image distortions of the same kinds. Popular deep learning frameworks (e.g., TensorFlow, PyTorch) all support automated data augmentation. *Data synthesis* focuses on synthesizing training images with different backgrounds, clutters, and occlusions [39, 61, 118]. The purpose is to generate more diverse scenarios of the target, without actually setting up the new backgrounds and clutters in the real world, as doing so may be costly. This body of work often borrows techniques from computer graphics (e.g., image smoothing, 3D rendering). Both data augmentation and data synthesis share a fundamental limitation: because the generated examples are based on existing ones, they cannot generate examples of unseen/novel object instances. Therefore, these techniques are orthogonal to Eureka, and can be applied when the expert trains a DNN using examples discovered from Eureka.

Chapter 5

Improving System Efficiency of Eureka on the Edge

Analysis in Section 4.4 highlights the importance of Eureka's system efficiency: a slow system decreases result delivery rate, which increases the User:System Match, resulting in user-side wait time. One way to addressing this problem is to increase the system delivery rate by expanding the system to include more compute power (i.e., cloudlets) and possibly more data sources. However, two limitations need to be considered.

First, although each cloudlet will probably have its own CPU and memory resources, many cloudlets reading from the data sources at the same time may stress the wide area network (WAN) bandwidth. This is particularly the case if the compute is centralized in data centers while the data sources are dispersed on the edge. In 2019, 500 hours of video is uploaded to YouTube per minute [29]. That means an amplification factor of about 30,000. In the meantime, it is estimated there is one public security camera for every eleven citizens in the UK [14]. Although this is a rough estimate, streaming millions of camera feeds into the cloud is expensive and probably beyond the ingress bandwidth of cloud data centers, let alone sharing this precious resource with other applications. Eureka alleviates this bottleneck by using an edge-based architecture (Section 3.1). This architecture executes early-discard filters on compute infrastructure that has high-bandwidth low-latency access to data, and only transmits a small portion of undiscarded data across the WAN. We evaluate the importance of this design in Section 5.1.

Second, compared to the cloud, the edge has limited *elasticity*. The cloud has access to seemingly inexhaustible compute and storage resources that can be spun up dynamically as demand increases. Performance isolation can be easily achieved by launching new cloud VMs for new applications. By contrast, a cloudlet is typically designed with a moderate physical space, electrical power, and thermal envelope, and must be shared by multiple tenants. A smart traffic camera cannot be made into the size of a data center. It is difficult to provide more compute without sacrificing the high-bandwidth and low-latency property (e.g., offloading to a remote cloudlet) or incurring significant cost (e.g., laying direct optical cables). A flash crowd can easily overwhelm cloudlet resources. *How can we improve edge elasticity for Eureka workloads?* We start by recognizing that the very first step of a typical Eureka pipeline, namely *decoding visual data,* is a surprisingly large burden on edge elasticity. As a solution, we propose the well-known mobile computing technique of offloading computation [12, 31, 126], but apply it to a local rather than remote accelerator. Based on thermal, energy density and data copying considerations, we identify cloudlet storage as the optimal location for placement of the accelerator. Synergistically, we also incorporate batch operations and scheduling into the storage system to reduce stalls and disk seek overheads. We collectively called this solution *decode-enabled storage*. We present our findings and solution in Section 5.2.

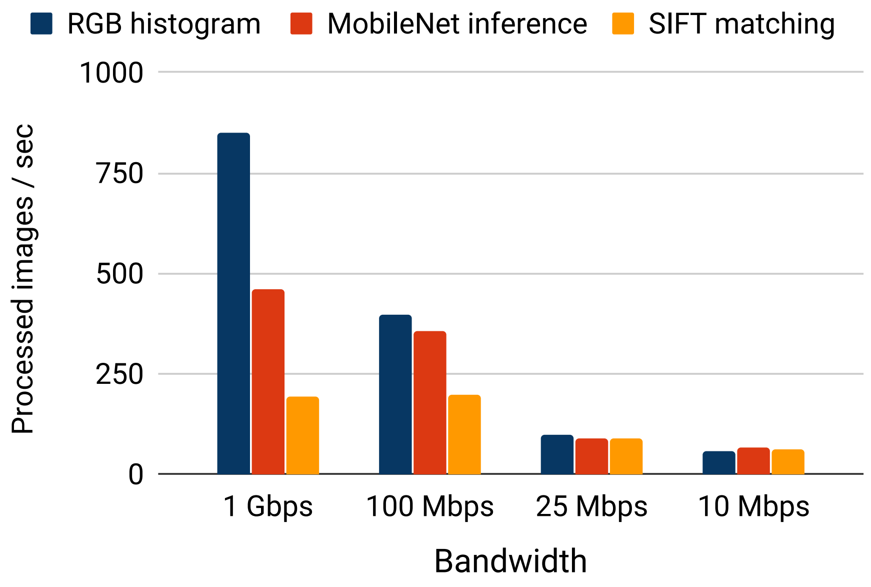
5.1 Alleviating WAN Bottleneck via Edge Computing

In this section, we examine the importance of high-bandwidth and low-latency connections between Eureka's early-discard backends and the data sources. We use the same experimental setting of 8 cloudlets described in Section 4.3. By default, each cloudlet has at least 1 Gbps (Giga bits per second) to its associated data source, representing LAN quality network. We throttle this bandwidth down to 100 Mbps, 25 Mbps, and 10 Mbps respectively using the Linux command line tool `tc qdisc`. The lower bandwidth numbers represent the realistic WAN scenarios — the US national average broadband was 18.7 Mbps in 2017 [6].

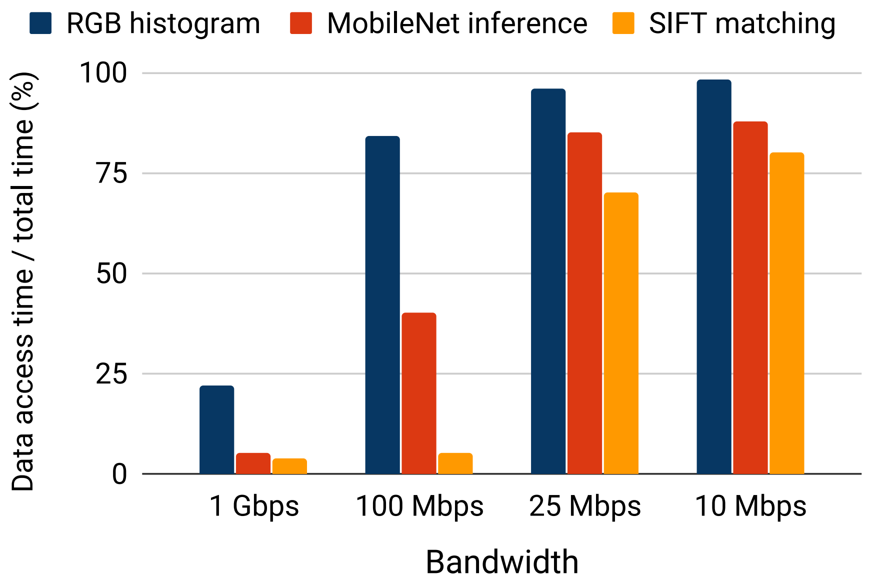
We benchmarked with three early-discard filters: (1) RGB histogram: the least compute-intensive; (2) MobileNet inference: compute-intensive but can be accelerated by using a GPU; (3) SIFT key point matching: compute-intensive and cannot be accelerated by a GPU. Using these filters allows us to gauge the impact of bandwidth for a wide range of computer vision filters in practice.

Figure 5.1(a) measures the processing throughput on the 8 cloudlets (higher is better) as we decrease data access bandwidth from 1 Gbps (LAN) to 10 Mbps (WAN). At 1 Gbps, the RGB histogram filter achieves significantly higher throughput than the other two because it is the least computationally expensive. As the bandwidth decreases, its throughput decreases as the bottleneck quickly shifts from compute to I/O. Decreasing from 1 Gbps to 100 Mbps has little impact on SIFT matching, the most compute-intensive filter. Nonetheless, as we further constrain the bandwidth, its throughput ultimately drops by more than 50%. At 10 Mbps, all three filters show the same throughput as they are bottlenecked by data transfer.

Figure 5.1(b) shows the same experiments from a different perspective. Here we measure the percentage of total run time spent on retrieving data as opposed to computation. As bandwidth becomes more scarce, this percentage tends to increase. At the lowest bandwidth of 10 Mbps, even the most compute-intensive SIFT filter spends 80% of its run time retrieving data, and the RGB filter spends 98%! In summary, the above experiments confirm that high-bandwidth low-latency access is crucial for scaling out Eureka to more cloudlets in order to increase result delivery rate, which in turn is important for achieving human-efficiency.

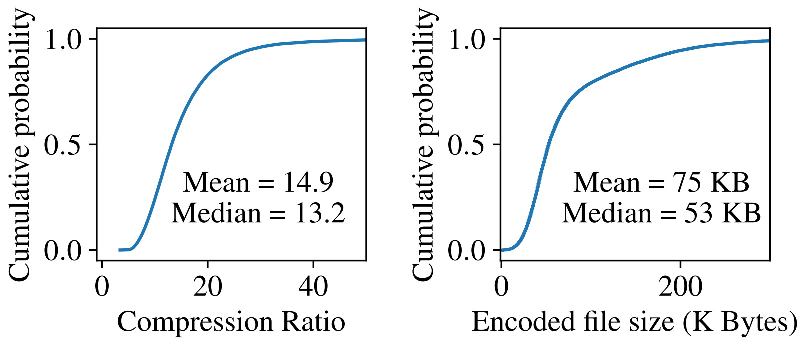


(a) Image Processing Rate



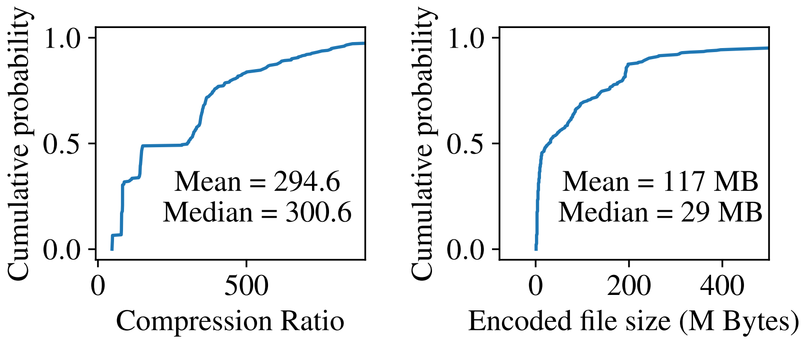
(b) Fraction of Time Spent on Data Access

Figure 5.1: Effect of Bandwidth between Cloudlet and Data Source



99.2 Million JPEG and PNG Images

Figure 5.2: Storage Efficiency of Encoding: YFCC100M



353 MP4/H.264 Videos

Figure 5.3: Storage Efficiency of Encoding: VIRAT

5.2 Enhancing Edge Elasticity via Intelligent Storage

In this section, we will first examine the performance characteristics of Eureka's workload on a cloudlet. Based on our observation, we propose and evaluate an approach that enhances a cloudlet's elasticity through the use of a novel intelligent storage architecture.

5.2.1 Eureka Workload Attributes

Encoded Disk Storage of Visual Data

As discussed, scarce WAN bandwidth precludes transmission of all data into the cloud, making the edge the final destination of most visual data. The search for novel targets requires content-based processing rather than index search. Therefore, re-processing visual data stored on a cloudlet is the norm in Eureka. The quantity of this data is staggering. In 2017 alone, over one trillion (10^{12}) photographs were captured on smartphones [104]. In 2019, estimated 500 hours of new data were uploaded to YouTube every minute [29]. Even more visual data is generated by dense IoT camera deployments. Visual data is large but highly compressible, as shown in Figures 5.2 and 5.3 for the widely-used YFCC100M [145] and VIRAT [109] datasets. Encoded

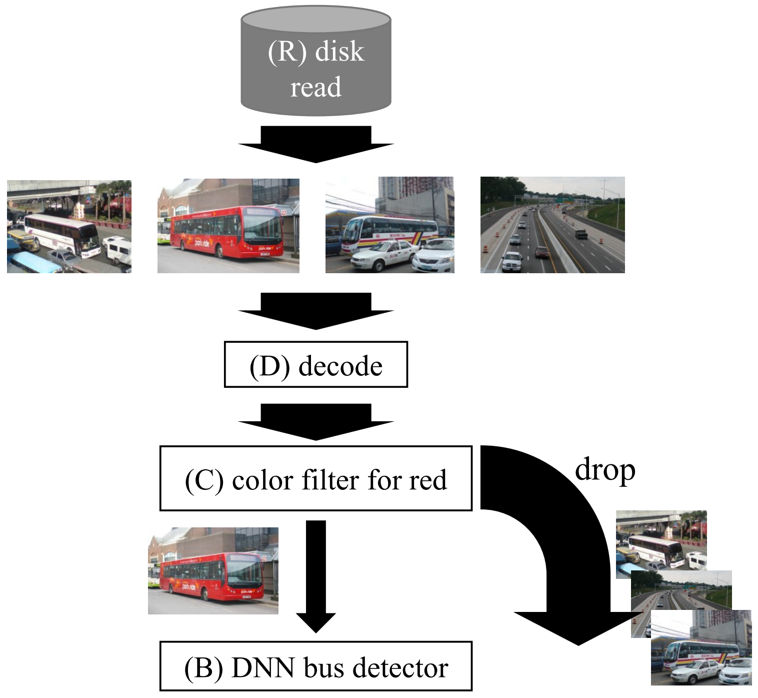


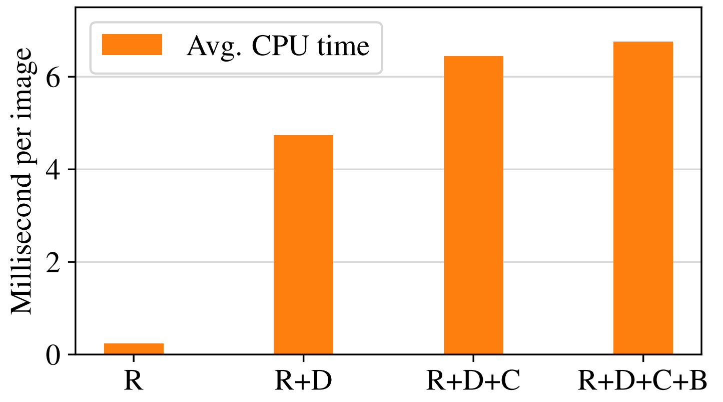
Figure 5.4: An Early-discard Pipeline for "Red Bus"

object sizes will increase over time due to improving camera resolution. As data volumes are high even with compression, the higher capacity and lower cost per bit of spinning disks relative to SSDs make compressed data on disks the only cost-effective storage strategy. Thus, we expect cloudlets to be multi-processor, GPU-enabled machines with multiple direct-attached disks to provide storage and compute capacity to run a Eureka backend.

Short-circuit Execution of Early-discard Pipelines

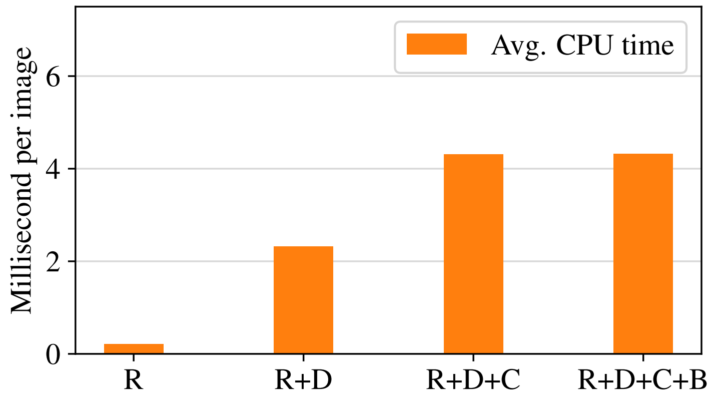
A Eureka query typically consists of a cascade of early-discard filters. These filters may encompass traditional computer vision algorithms (e.g., RGB histogram, perceptual hashing), machine learning (e.g., SVM), and deep learning (e.g., Faster R-CNN). While DNNs are accurate, executing them on all images and video frames can be very expensive. Eureka orders the filters in such a way that cheap and selective filters are executed first; if an data item is dropped, execution of subsequent filters is skipped. In general, this short-circuit evaluation strategy helps Eureka achieve high throughput while maintaining accuracy.

Figure 5.4 depicts a Eureka pipeline for finding red buses. Compressed images (e.g., JPEG files) are read from the disk, decoded into RGB arrays, and then examined by a "redness" color filter, which discards images with a below-threshold number of red pixels. Only a small percentage of images pass the color filter and are processed by the more expensive DNN bus detector. The color filter runs much faster than the DNN. Thus, it is both computationally inexpensive and selective. Unlike "pure" machine learning workloads which run DNN inference/training on huge batches of images, Eureka's early-discard pipelines execute the DNN only occasionally, effectively amortizing its cost over many data items.



R: Read files from disk C: Color filter for redness
D: Decode JPEG to RGB B: Bus detection with DNN

Figure 5.5: High Scalability Cost of Image Decode



R: Read files from disk C: Color filter for redness
D: Decode JPEG to RGB B: Bus detection with DNN

Figure 5.6: Impact of Our Solution on Figure 5.5’s Workload

5.2.2 Problem: High Scalability Cost of Decoding

The Eureka pipeline shown in Figure 5.4 consists of four main operations: (R) reading encoded images from disk, (D) decoding into pixel arrays, (C) execution of color-based filtering, and (B) execution of the bus detection DNN. Figure 5.5 shows the average per-image CPU cost of these four steps in processing 50,000 images from the YFCC100M dataset. The experiments were run on a cloudlet (4-core slice of a server with two Intel® Xeon® E5-2699v3 processors at 2.30 GHz and an NVIDIA GTX 1080 Ti GPU) with a Seagate 4TB hard disk drive (7200 RPM, SATAv3). This configuration reflects the typical per-drive resources of a 2-socket server with 8–12 direct-attached disks.

In Figure 5.5, the four steps are added incrementally from left to right. Initially (label “R”), the process is I/O-bound as the read data is discarded immediately. As soon as decoding of visual data is added (label “R+D”), CPU time jumps dramatically. In fact, image decoding (Step D alone) consumes 70% of the CPU cost of the full pipeline. The third step (label "R+D+C") shows that applying a color detection filter on all data items only increases total CPU usage modestly, relative to the cost of decoding. What this implies is that the color filter (Step C) can operate at much higher throughput than JPEG decoding (Step D). The small difference between the bars labeled "R+D+C" and "R+D+C+B" shows the benefit of early discard. The expensive DNN for bus detection (Step B) only has to be applied to about 3% of the images that pass the color filter. We seek a way to reduce cost (D) that is simple, effective and future-proof. Figure 5.6, which previews our experimental results from Setcion 5.2.8, confirms the effectiveness of the solution. Comparing Figure 5.6 to Figure 5.5, we see that the average per-image CPU cost of decoding is reduced to a half (2.3 ms vs 4.7 ms), indicating potential improvement of scalability.

5.2.3 Solution: Decode-Enabled Storage

Our solution is developed from an application viewpoint, rather than a systems viewpoint. All that a typical visual analytics application desires is to obtain RGB arrays of the visual data in its virtual memory. The application does not care exactly how these arrays materialize. This leads to a simple question: *"Why doesn't the storage subsystem return the decoded data when it is read?"* In below, we propose a *decode-enabled storage API* that embodies this abstraction. The API simultaneously simplifies application development and allows for placement of functionality close to their optimal position (discussed in Section 5.2.4). These APIs let an application obtain the decoded version of a visual data object stored on the disk. Beyond that, it supports a high-value subset of image processing functions and multi-object read optimization. In Section 5.2.4, we consider alternative approaches to implementing the abstraction of decode-enabled storage.

Extended Object Store API to Retrieve Decoded Objects

Our decode-enabled storage abstraction extends the well-known object store concept [101] that allows an application to create, read, write, and delete logical objects. Our extensions let the calling application specify operations and transformations to perform on an object as it is read. This is embodied in the `FetchAndDecodeObject` call as follows:

```
FetchAndDecodeObject(
    int64 object_id,
    int32 opcode,
    void* params,
    iovec* where_to_put_decoded_object,
    iovec* where_to_put_original_object)
```

Each object is addressed by an integer Object ID. The `opcode` field indicates whether to fetch the original compressed object, the decoded version, or both. The last is useful in a server context, where images may need to be transmitted across a network after local filtering: decoded objects improve analytics performance and CPU load, while the original versions can be sent without re-encoding. The final two vectors indicate memory regions into which the fetched data will be placed using a scatter-gather approach.

Other operations may also be requested using `opcode`. In this paper, we focus on image decoding and (content-based) cropping (e.g., face detection), which can be accelerated with ASICs and will remain useful long into the future. Parameters for operations can be provided through the `params` pointer.

Partial read or write of an object is not supported. This enables additional disk optimizations, reduces internal fragmentation, and maximizes sequential reads. These semantics are a natural fit for compressed image data, but work well for video, too. For example, B-frames in H.264 are encoded using information from both past and future frames. Thus, to successfully decode a single frame, the decoder needs to retrieve large amounts of surrounding data, possibly the whole object. We suggest keeping the size of individual objects moderate, on the order of several MBs. Very large videos (GBs to TBs) can be stored as a sequence of objects, for which the mapping can be stored in a separate object.

Multi-Object Batch Iteration

The above single-object API can be seen as offloading computation (e.g., decode) at the granularity of an image. This granularity can be increased to reap additional benefit. Visual data analytics is typically a form of batch processing, applied thousands to millions of images, with no requirements on order. Performance can be improved by re-ordering the objects, for example, to reduce disk seeks or to exploit parallel read heads. Although it is well known that access pattern can have a great impact on I/O performance, the most efficient read order varies from device to device. On traditional disks, it mostly depends on physical block location and disk geometry, but this can be obscured by logical block addressing and remapping. On object store disks such as Seagate's Kinetic HDD [131], it is further complicated by the firmware's approach to storing, fetching, and caching metadata. Access to this information is obscure in the application, but straightforward in the disk. Moreover, a system may use a heterogeneous set of disks that further complicate application-level optimization.

Therefore, it is more intuitive to optimize this within the storage subsystem, behind a unified batch-oriented API call. Note that this is different from request scheduling on current disks, which only re-order requests on short work queues. Our decode-enabled storage API provides an iterator-style batch operation as follows:

```
IterateCollection(
    int64 collection_id,
    int32 opcode,
    void* params,
    int64 logical_index,
    int64* flags,
    int64* returned_object_id,
    iovec* where_to_put_decoded_object,
    iovec* where_to_put_original_object)
```

The list of Object IDs to fetch are created *a priori* in an object of some custom format, referenced by `collection_id`. The application iteratively calls this API to retrieve another object. The `returned_object_id` tells which object was fetched. The API guarantees exactly-once semantics of objects — exhaustive and non-repeating, but makes no promise of the order. The `logical_index` tracks the iterator cursor. A flag `COLLECTION_LAST` is set in `flags` to

Architecture	Efficiency (GFLOP / J)
CPU (Core i7)	1.14
FPGA (Xilinx LX760)	3.62
GPU (NVIDIA GTX285)	6.78
GPU (AMD R5870)	9.87
ASIC	50.73

Source: Table 4 in Chung et al [28]

Table 5.1: Energy Efficiency of Hardware Accelerators

terminate iteration when the last object is returned. The other fields are the same as in the single-object API.

5.2.4 Implementation

The API described in Section 5.2.3 discloses application intent to the system. Using a hardware accelerator to implement the most compute-intensive parts of this API is clearly the way to reduce CPU demand, and thereby improve scalability. On cloudlets without the accelerator, a pure software implementation of the API can provide compatibility. Applications can be written to this API today, and will continue to work unmodified over the long period of time that it typically takes for hardware optimizations to gain market share. This is the same strategy that has been used for GPU acceleration in popular open-source libraries such as OpenGL and DirectDraw.

Two questions follow from this decision. First, what type of accelerator should we use? Second, where should it be placed? We discuss below the design rationale that leads to our recommended solution of ASICs within storage devices.

Energy and Thermal Considerations

As mentioned, energy efficiency is a key consideration for a cloudlet. This is especially true for a hyperconverged multi-tenant cloudlet that must fit into very limited space, yet support many CPU cores and at least one high-end GPU for visual data analytics. The tight thermal envelope and limited electrical power budget of such a cloudlet requires very careful attention to energy and cooling efficiency.

Table 5.1 from Chung et al [28] shows the relative energy efficiency (expressed in GigaFLOPs per Joule) of different hardware accelerators. These specific measurements are for matrix multiplication, but the trend holds across applications [55]. Energy efficiency improves as one moves down the rows of Table 5.1, but comes at the cost of flexibility. When flexibility is paramount, CPUs are optimal; when it is not important, ASICs are optimal. Intermediate points in this spectrum correspond to programmable accelerators such as GPUs.

In our setting, flexibility is not important. The formats used by visual data such as JPEG, PNG, JPEG2000, and MP4 are well standardized today. These will remain unchanged forever, because of vast archives of precious data stored worldwide in these formats. If new formats arise in the future, support for them can be software-only on legacy cloudlets. New accelerators can support the new formats, in addition to all the old formats. Use of the APIs described in

Section 5.2.3 insulates legacy applications from this discontinuity in hardware implementations. They only deal with decoded data, and are thus impervious to lower-level changes. What is unlikely to ever change is the need for decoding data as the first step in visual data processing pipelines.

ASICs, which are fixed-function (i.e., non-programmable), thus emerge as the best type of hardware accelerator to use. Their lack of versatility is not a handicap for our use case, and their superior energy efficiency is a major advantage.

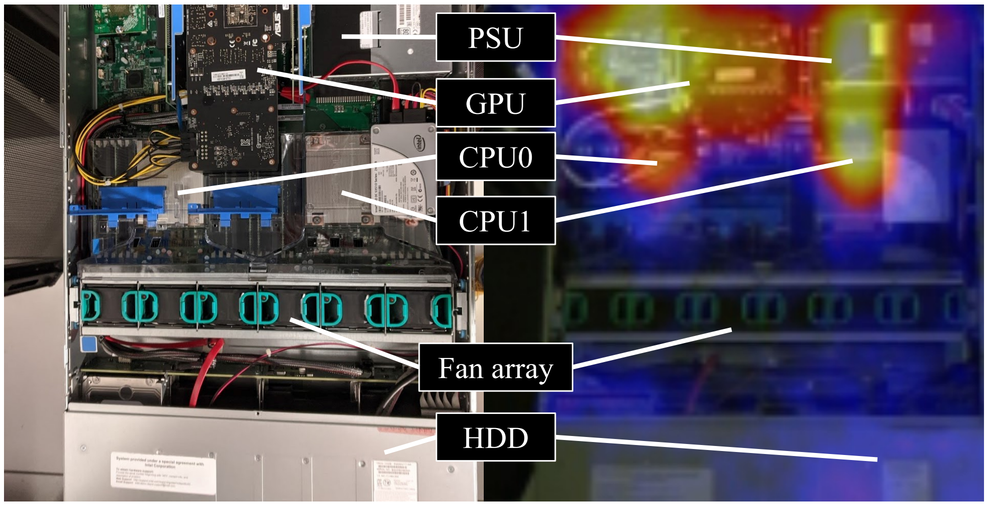
The placement of the ASIC is also guided by energy and thermal considerations. Figure 5.7 shows the thermal heat maps of a typical rackmount cloudlet and a typical standalone cloudlet while they are processing a visual data pipeline. The brightest areas (in white, yellow and orange) represent the current "thermal bottleneck" of the system. Adding new hardware to any of these areas will only worsen this bottleneck, and make cooling the system more difficult. The power density in these areas is already high, and delivering more power there will also be difficult. As long as performance is not compromised, adding new hardware to the coolest parts of this heat map (in blue or black) that are furthest from the current thermal bottleneck is the wise path to follow. Storage devices are among the coolest parts of Figure 5.7, and located furthest from the thermal bottleneck. They also represent the starting points of all visual data pipelines. They are thus the logical choice for placement of the ASIC.

Minimizing Data Copying

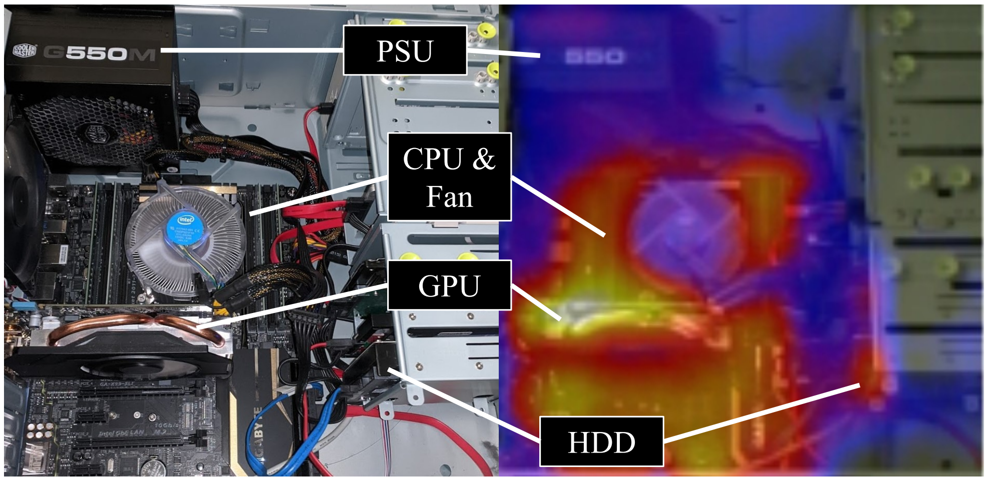
A well-known design principle for scalability from "big data" systems is to minimize data copying [71, 74, 91]. Locating the decode-acceleration ASIC in a storage device aligns well with this principle. No new data copies are needed; rather, data is decoded in a streaming operation as it is read off the disk surface. This may seem to be a counter-intuitive optimization at first glance, because early decoding increases the bandwidth demand on the SATA interconnect from disk. However, this can be overcome by replacing SATA by the modern NVMe host-storage interconnect [1]. NVMe was originally created to support the much higher bandwidth demand of SSDs. There is industry speculatation NVMe will become the unified interface for all storage types in the near future, including SSD and disks [99]. By fortunate coincidence, this trend aligns well with our proposal to place decoding functionality on disks.

Figure 5.8(a) illustrates the placement of a decode ASIC directly on disk. A cloudlet can be attached to multiple such disks to exploit parallel storage bandwidth and decode throughput. In other words, compute and storage scale together when more disks are added to a system.

An alternative approach, shown in Figure 5.8(b), is to separate the ASIC from storage and place it on the I/O bus (typically PCIe). Similar to (a), this strategy makes it easy to add more accelerators to the system. However, it has at least two deficiencies relative to Figure 5.8(a). First, it requires some host mediation of the decoding process, incurring context-switching overheads. Specifically, the host must initiate reads of encoded objects from disk into DRAM; once complete, it triggers the decoding of these objects and sends them to the accelerator. While the overhead of this mediation can be reduced by batching over many objects, it can never be reduced to zero. Second, it incurs two more round trips of the encoded data over the system bus. Assuming a compression ratio of 15, this amounts to $2/15 = 13\%$ additional cost in terms of bus data transfer and energy. Decode in GPU (e.g., NVIDIA's NVDEC) is an instantiation of this



(a) Rackmount Cloudlet



(b) Standalone Cloudlet

Figure 5.7: Thermal Heatmap of Typical Cloudlets

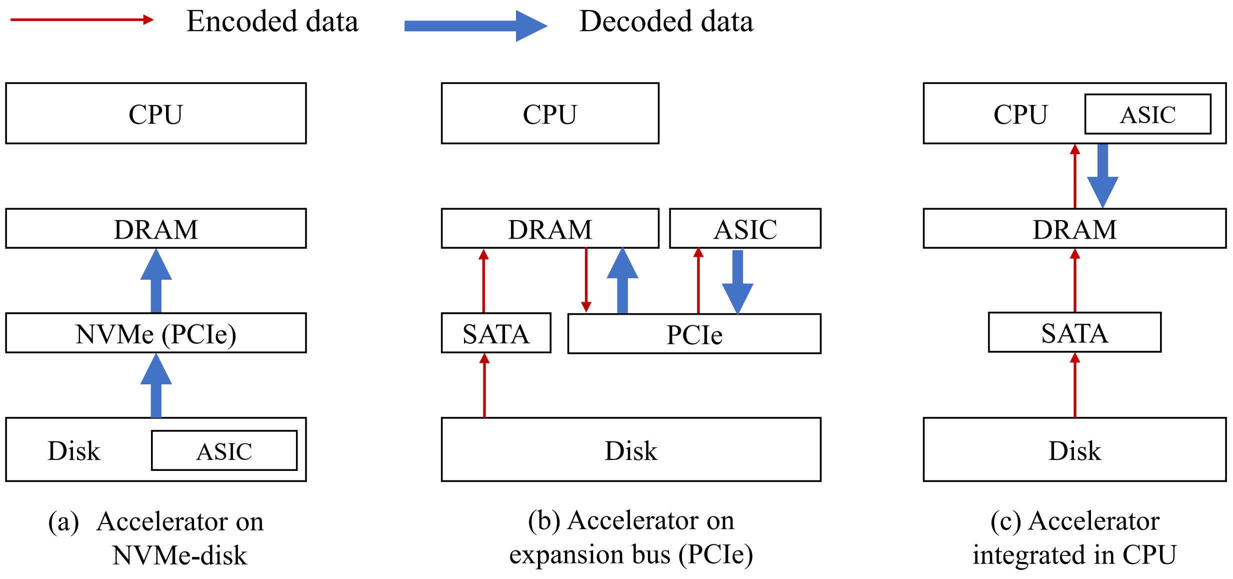

Encoded data
Decoded data
CPU
DRAM
NVMe (PCIe)
Disk
ASIC
(a) Accelerator on NVMe-disk
CPU
DRAM
ASIC
SATA
PCIe
Disk
(b) Accelerator on expansion bus (PCIe)
CPU
ASIC
DRAM
SATA
Disk
(c) Accelerator integrated in CPU

Figure 5.8: Alternative Placement of Decode Accelerator (ASIC)

strategy, but GPUs are more expensive and less energy-efficient than an ASIC, and are already a thermal bottleneck (Figure 5.7).

A third alternative, shown in Figure 5.8(c), avoids data copying by integrating the ASIC directly on a CPU die. An example of this approach is Intel's Quick Sync Video (QSV) feature. With this approach, decoded data completely bypasses the system bus and possibly even DRAM (if it can be written directly to CPU cache). However, as discussed in Section 5.2.4, there are strong thermal considerations that argue against adding an accelerator to the already-hot CPU die if an alternative placement can perform just as well. In addition, it is difficult to scale the CPU-integrated decoder. Most processors have many CPU cores, but just one (if any) hardware decoder block. Such a decoder typically can handle hundreds of frames or images per second, more than sufficient for the vast majority of real-world use cases. Thus, there is little incentive to add more than one such decoder to a general-purpose processor. However, as our experiments in Section 5.2.7 show, we may need many times the decode capability to fully utilize all of the compute cores and disk bandwidth in a reasonably-sized cloudlet. Finally, a CPU-integrated solution provides a fixed decode capability that cannot be scaled up easily – adding processors or additional servers may not be feasible on the edge – in contrast, a solution with one appropriately-sized ASIC decoder per disk will naturally scale up decode capability as more storage is added to a system.

Technical Feasibility

Decoding on disk critically depends on the use of NVMe host-storage interconnects. Table 5.2 lists reference speeds of SATA and NVMe, as well as the internal transfer speeds of disks and SSDs. With an average compression ratio of 15x for JPEG, a disk that delivers 200 MB/s from its platters will produce 3000 MB/s of decoded data. This is well above what SATA can sustain, but well within the bandwidth supported by current NVMe products [142].

SATA	500 – 700 MByte/s
NVMe	1,000 – 6,000 MByte/s
HDD “internal”	100 – 300 MByte/s
SSD “internal”	> 500 MByte/s

Table 5.2: Comparison of Storage Bus Technologies and Storage Devices Internal Throughput

Device	JPEG Decode Speed
Disk CPU (ARM-based, 1.0 GHz)	15 MPixel/s
Host CPU (Intel-based, 2.3 GHz)	60 MPixel/s
FPGA 1 [72]	73 MPixel/s
FPGA 2 [163]	140 MPixel/s
Intel Quick Sync Video	600 – 1,000 MPixel/s

Table 5.3: JPEG Decode: Software vs HW Acceleration

Fixed-function hardware and FPGAs have already been adopted on hard disks for other functions [132, 144, 146, 162]. Low-power, low-cost hardware accelerator or FPGA for image decoding has been studied, prototyped, and validated in both academia and industry (e.g., [5, 72, 89, 163]). Table 5.3 compares the published performance of two FPGA-based JPEG decoders with experimental measurements of software decode on a single host CPU core and an embedded CPU core of an active disk (ARM Cortex A53 on Seagate Kinetic HDD). For reference, we also benchmarked with Intel Quick Sync Video (QSV), a hardware-accelerated decoder integrated on certain Intel processors. We see large performance gains with accelerators compared to software decode on CPUs.

Unlike many previous designs of *intelligent storage* or *active disk* [3, 84, 98, 100, 120, 122, 159], our design precludes the execution of arbitrary code within a disk. This greatly reduces the disk’s cost, complexity, and security risk. In addition, it avoids putting the cost-, power-, and memory-limited on-disk computational capabilities in competition against well-provisioned host processors, which are designed for computational throughput. Any advantage of on-disk general computation is often rendered obsolete by rapid improvements of the host system driven by Moore’s Law. Decoding, in contrast, represents a very constrained form of application-level logic that is well standardized, and is unlikely to become obsolete even if the workload evolves in the long term.

Another advantage of on-drive decoders is that they can be co-designed with the disk’s specs in order to fully utilize its internal throughput and outbound bandwidth. A decode-enabled disk can be equipped with multiple hardware decoders. Under a simple system model, we can bound the number of decoders by the minimum between its internal (encoded) object throughput and outbound (decoded) object throughput:

$$\frac{\min\left(\frac{\text{Disk Internal Read Speed}}{\text{Avg Encoded Object Size}}, \frac{\text{NVMe Speed}}{\text{Avg Decoded Object Size}}\right)}{\text{Object Throughput Per Decoder}} \tag{5.1}$$

The arguments for images apply similarly to video decoding. IP cores exist that can decode 4K resolution H.264 video at 60 – 120 frames per second (FPS) [113, 143]. We will experimentally explore how much on-disk decoding capability is needed to deliver performance gains.

Beyond Decoding on Drive

Beyond hardware-assisted decode, it is difficult to justify running computer vision operations on a disk CPU. Computer vision algorithms tend to be both compute- and memory-intensive, and have been changing at a rapid pace. They are more suited to execution on powerful host CPUs or GPUs. General compute capacity in drives is typically modest, and it is usually not worthwhile to implement new hardware for rapidly evolving workloads.

The one exception we consider here is image cropping on disk. Concretely, we consider two types of cropping. In the first type, the application provides a bounding box. Cropping based on pixel coordinates incurs data movement, but only trivial computation, and can be performed efficiently on disk. This is particularly useful for static cameras where the application has *a priori* knowledge about regions of interest, e.g., portion of a traffic camera view covering a crosswalk [19].

The second type uses dynamic coordinates based on image content analysis. While this can be expensive in general, we consider operations that can be accelerated through hardware, for example, face detection. Prior research on face detection accelerators achieved between 30 and 600 FPS [21, 27, 58, 62, 79]. These accelerators output the coordinates of the faces, which the disk processor uses to perform cropping.

Cropping is possible only after decoding, but offers the opportunity to reduce the amount of data transferred on the bus. Only the cropped patches are returned, and the other bytes are discarded. This has the potential to reduce the bandwidth requirement, depending on the crop size or sparsity of faces in the data set.

Although we only consider cropping here, as other computer vision algorithms become standardized and implemented in low-cost hardware, they can be added to the list supported by a decode-enabled storage device.

5.2.5 Timing-Accurate Emulated Prototype

We implement *timing-accurate emulation* of an NVMe-attached decode-enabled disk that allows execution of real application code and measurement of wall-clock time and real OS-level statistics (e.g., CPU time, bandwidth). The emulation allows us to experiment with a wide range of parameters that are otherwise unavailable in existing hardware products (Section 5.2.7 and 5.2.8). This section describes our discrete event-driven simulation that uses pre-computation and modeling to emulate HW decoders, while disk timing is based on a real HDD.

HW/SW-based Emulation Framework

Figure 5.9 depicts the architecture of our emulator, implemented in a similar way to DiskSim [18] and Memulator [49]. Application programs are largely unchanged except for the use of the new APIs to fetch (decoded) data. It includes critical components of a decode-enabled storage device – the host-disk interconnect bus, potentially parallel HW accelerators for decoding, and mechanical disk timings. The controller implements the logic to control data flow and coordinate different components in order to service a request from applications.

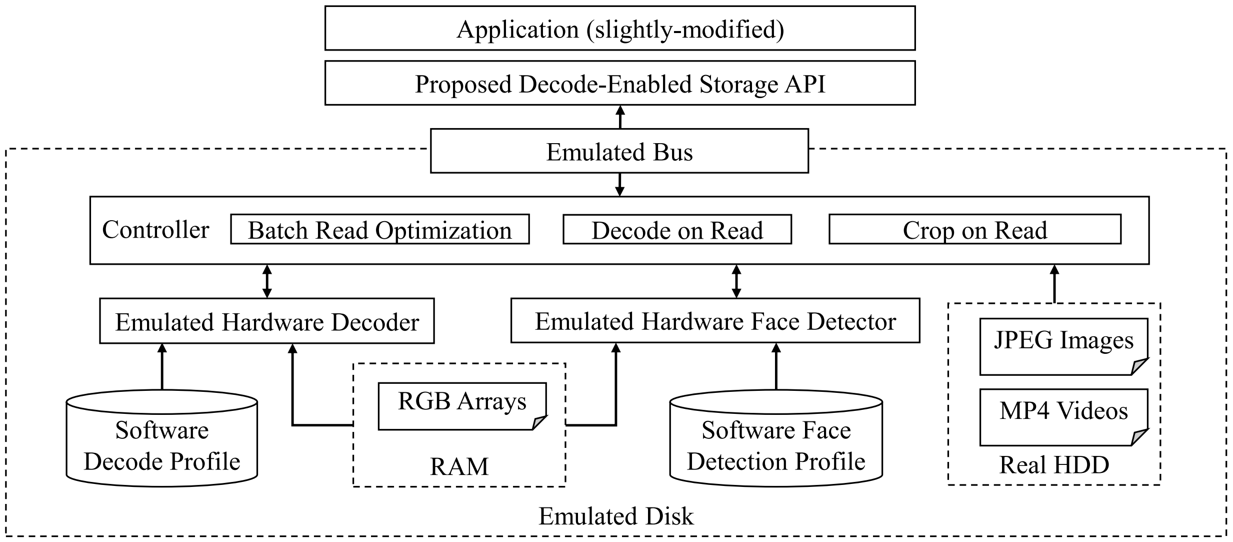

Application (slightly-modified)
Proposed Decode-Enabled Storage API
Emulated Bus
Controller
Batch Read Optimization
Decode on Read
Crop on Read
Emulated Hardware Decoder
Emulated Hardware Face Detector
JPEG Images
MP4 Videos
Real HDD
RGB Arrays
RAM
Software Decode Profile
Software Face Detection Profile
Emulated Disk

Figure 5.9: Decode-Enabled Storage Emulator used in Experimental Evaluation

For each software-emulated component, we construct (1) a model to calculate the (content-dependent) completion time for operations, and (2) a mechanism to generate the actual results produced (e.g., the actual decoded pixel arrays). The latter mechanism must execute faster than the modeled time; this is achieved by serving pre-computed results from memory. As this generally runs faster than needed, we insert high resolution sleep to achieve the modeled latency.

Following DiskSim, we use a discrete event-driven approach to model complex interactions between components. For example, requests to the decode ASICs are serialized through a priority queue sorted by the simulation timestamp. Thus, requests are ordered "correctly" even if generated out of order by the concurrently executing components. The simulation clock is continually updated to match the real world time. When a request's computed completion time is reached by the simulation, we send the response back to the application.

Emulating Specialized Hardware

Prototypes ASIC- and FPGA-based image decoders [5, 72, 89, 163] have been characterized by a metric specified in MegaPixels per second (MPixel/s). Hence, we parameterize our emulated decoder with a targeted MPixel/s. In practice, decoding time may vary based on content and compression level of images. Our emulator accounts for such variability, while maintaining a target speed. We compute a global scaling factor that scales the average software decode time for an entire image dataset to the target rate; we apply this factor to the software decode time of each image to obtian its simulated HW decode time. More concretely, suppose the data set has N images, the software decode time of image i is t_i^{sw}, and its decoded size is r_i MPixel. When simulating an image decoder parameterized at M^{sim} MPixel/s, the simulated decode time of image i is calculated as:

$$t_i^{sim} = \frac{\sum_{k=0}^{N} r_k}{\sum_{k=0}^{N} t_k^{sw}} \cdot \frac{t_i^{sw}}{M^{sim}}$$

To emulate real-time hardware decode, we pre-decode all images and store them in a ram-disk. At run time, the decoded data is rapidly returned to the application.

We emulate video decoding and face detection hardware in a similar fashion. For simplicity, we parameterize them using a target frame per second (FPS), and scale individual elapsed times based on profiled software times.

Emulating Disk Hardware Timing

We emulate mechanical disk timing factors, such as seeks, platter rotations, and block cache by reading files from a real hard disk. Although this will include overheads of the OS, filesystem, and bus, we find that these are small by performing similar tests on a fast SSD. We were careful to clear the OS page cache before each experiment.

Emulating the Bus

We parameterize the host-disk bus by a maximum bandwidth (MByte/s). Each (decoded) object exclusively occupies the bus during its transmission and other objects must wait for the bus to be relinquished. We assume the bus operates at the maximum rate when in use and is idle otherwise. This model is sufficient to estimate transfer rates for different interconnect technologies.

5.2.6 Evaluation Methodology

We chose a set of early-discard filters that prove to be valuable both in Eureka and in other visual analytics systems [81, 82] (Figure 5.10). These filters are typically able to discard a significant portion of data and are less computationally expensive than a DNN, though some of the filters are more compute-intensive than others.

We ran experiments on a workstation with two Intel® Xeon® E5-2699v3 processors (total of 36 cores/ 72 threads @ 2.3 GHz), 128 GB DRAM, and an NVIDIA GTX 1080 Ti. Because our emulator pre-stages decoded data in DRAM to emulate fast HW-accelerated decode, we randomly sampled 50,000 images from the YFCC100M data set [145]. The corresponding decoded data totals 51 GB, fitting comfortably in DRAM. Likewise, we randomly sampled 6 videos from the VIRAT Release 2.0 Ground data set [109], which are encoded in H.264 format @ 30 FPS.

Table 5.4 summarizes the default parameters used in experiments. To saturate a 2,000 MB/s bus, we used Formula 5.1 to calculate that we need about 5 JPEG decoders at 140 MPixel/s each (ref. Table 5.3). We first report results under these default settings, and then study the effect of varying different host and disk parameters. The preview results presented earlier in Figure 5.6 hinted at the potential for greatly improved scalability based on CPU utilization, here we delve into a more detailed study based on wall-clock time, disks supported per server, and end-to-end throughput, by addressing the following questions:

- Can decode-enabled storage improve application-level performance?
- Can decode-enabled storage reduce processing load on the cloudlet CPU?
- Is NVMe necessary and sufficient for transmitting decoded data from the disk?
- How much processing is needed on the disk?

Frame sampling. Useful when event of interest takes some time [81], e.g., pedestrian crossing street.

Color filter. Counts pixels in given RGB range; can cheaply detect sky (blue), vegetation (green), etc.

Face detection. Finds faces in images. Computationally expensive, but is an effective early-discard filter when finding human activities or recognizing individuals. Can be HW-accelerated.

Image difference. Computes mean square error (MSE) between current and prior image; if MSE is small, can assume identical results of later processing stages.

Perceptual hashing. Like image difference, but more robust to pixel noise, minor lighting differences, etc.

Tiny DNNs. Much smaller, faster, but less accurate versions of standard DNNs [81, 82]. Useful as early discard filters prior to running an expensive DNN.

Figure 5.10: Early Discard Filters Used in Experiments

Host	
CPU (`cgroup`)	4 cores/8 threads, 2.30 GHz
DRAM (`cgroup`)	64 GB
GPU	NVIDIA GTX 1080 Ti
Decode-Enabled Disk (Emulated)	
Host-Disk Bus	2,000 MB/s
HW JPEG Decoder	140 MPixel/s $\times 5$
HW Face Detector	30 FPS $\times 1$
HW Video Decoder	480 FPS @ 720p
Standard SATA Disk (Real, baseline)	
Specs	3.6 TB, 7200 RPM, SATAv3
Throughput (Bulk)	187 MB/s
Throughput (JPEG)	98 MB/s

Table 5.4: Default Experiment Setup and Parameters

- How many disks and accelerators can be connected to one server before saturating the host system's resource?
- How does decode-enabled storage compare to alternative solutions?

5.2.7 Micro-benchmark Evaluation

We first evaluate a series of micro benchmarks by running the following early-discard filters on YFCC100M images.

- `Color` finds images with many red pixels.
- `PHash` calculates an image's perceptual hash value.
- `ResNet10`[82] is a tiny DNN based on ResNet [59], with 65×65 input and 10 layers (reduced from 224×224 input, 50–100 layers).
- `Face` detects and crops faces in an image, and drops images with no faces.

`Color` and `PHash` offload image decode operations to the disk. `ResNet10` offloads image decoding to the disk, runs resizing and normalization on the CPU, and runs the neural network on the GPU. `Face` offloads both image decode, face detection, and cropping to the on-disk accelerators. Only the cropped patches containing faces, or a null list if there are none, is returned.

Effects on Application Throughput

Figure 5.11 reports the application-level throughputs (image/s) for three systems:

- Baseline: uses standard SATA-connected disk and software decode;
- Batch Iteration Only: optimizes multi-object batch read order on a standard disk;
- Decode-enabled Storage: combines batch iteration, on-disk decode, and NVMe. The data labels show improvement factors relative to Baseline.

Batch Iteration Only is approximated by accessing files sorted by the starting blocks of the file extents returned by Linux system call `FIEMAP`. This does not fully account for a hardware implementation, but provides a partial estimate of the potential gain.

We see batch iteration alone is effective (up to 2.5x improvement), but is not responsible for all performance gains. Adding on-disk decode HW and NVMe achieves up to 4.9x improvement over Baseline. `Face` is much slower than the others, because face detection is computationally expensive. However, even a single face detection chip at 30 FPS delivers 2x gain over software detection on the 4 CPU cores used in this experiment.

Is NVMe Necessary and Sufficient?

Figure 5.12 measures the data transfer rate on the bus for Baseline and Decode-enabled Storage. With decode-enabled storage, up to 1,400 MB/s is transferred to the host, clearly exceeding SATA bandwidth. This increase reflects two factors: (1) the transfer of decoded, rather than compressed images; (2) the host CPU, freed from decode tasks, can process images at higher throughput (Figure 5.11).

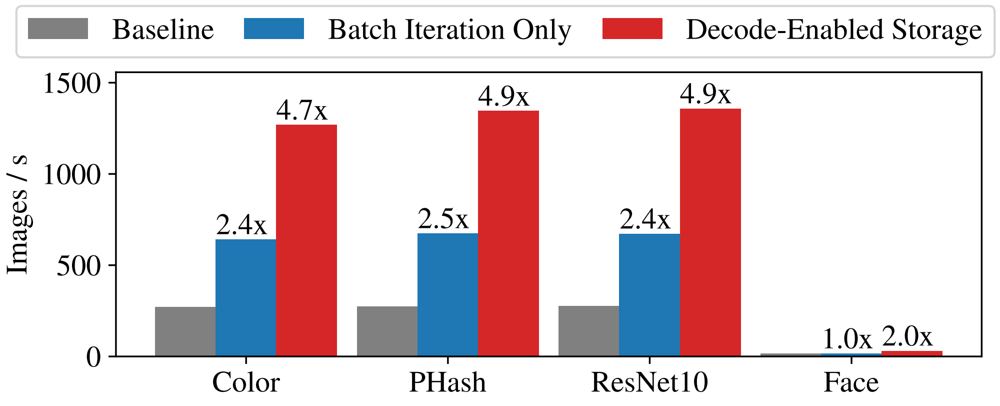


Figure 5.11: Application Throughput of Micro Benchmarks

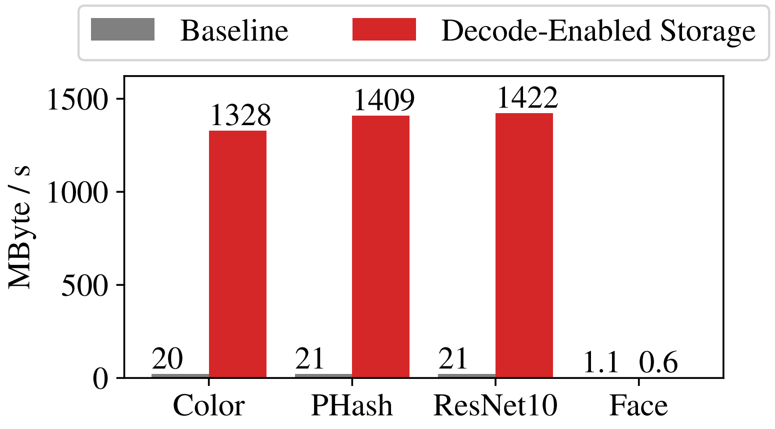


Figure 5.12: Data Transfer Rate on Disk Bus

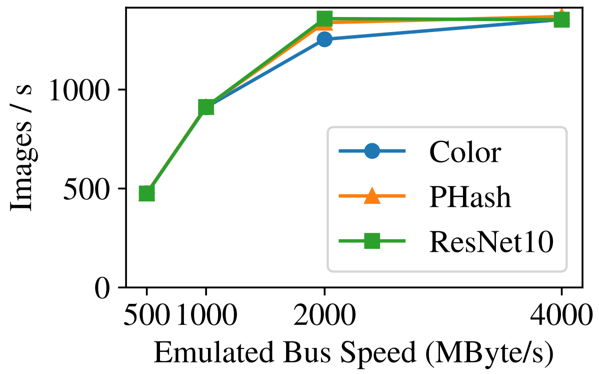
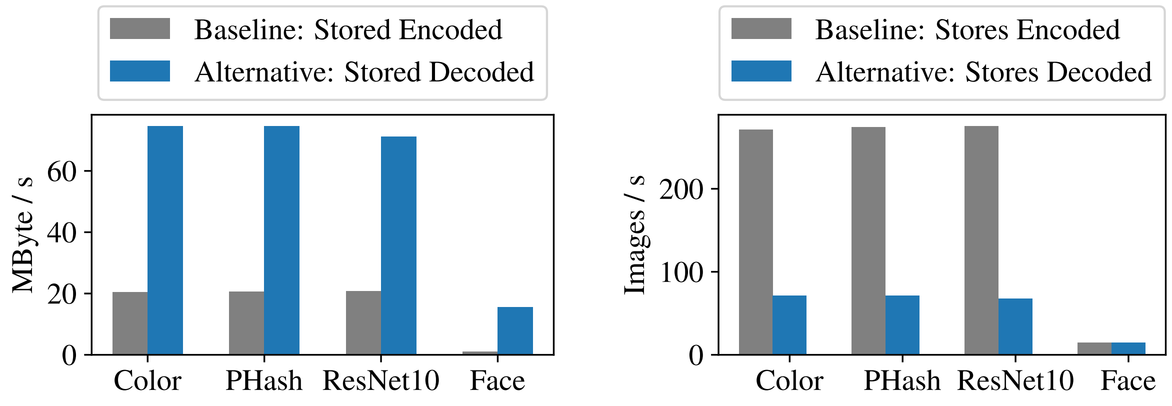


Figure 5.13: Effect of Bus Speed



(a) Transfer rate on Bus (b) Application Throughput

Figure 5.14: Effect of Storing Decoded Images

With `Face`, decode-enabled storage actually consumes less bandwidth than baseline (0.6 vs. 1.1), despite running 2x faster. This is due to cropping-on-disk that only sends cropped faces. In YFCC100M, only 23% of images contain human faces, with an average size of 97.5×126.9 pixels. Hence, even though they are sent as uncompressed pixels, the face crops require less bandwidth than whole-image JPEG files for the entire dataset.

To study how the bus speed impacts performance, we throttle the emulated bus's bandwidth between 500 MB/s (SATA speed) and 4,000 MB/s (high-end NVMe speed), and measure throughput for `Color`, `PHash` and `ResNet10` (Figure 5.13). We see that bus bandwidth has almost linear impact on application performance up to 2,000 MB/s, again confirming the importance of NVMe for our design.

Alternative Solution: Storing Decoded Data

We evaluate the alternative approach of storing decoded images (RGB arrays) instead of compressed ones (JPEG) on a standard disk. This completely eliminates image decoding, but reads more data from the disk platters. Figure 5.14 reports the host-disk transfer rate and application throughput. As expected the data transfer rate increases significantly by up to 3.7x over baseline, limited by the internal read speed of the disk. This increase is due to larger sequentially accessed files, resulting in fewer seeks, and the fact that the decode bottleneck is removed from the CPU. Unfortunately, this increase is offset by the 15x inflation in object sizes, resulting in a net decrease in application-level throughput. Furthermore, this in effect reduces the disk's capacity by 15x as well. Overall, storing decoded images on the disk is a losing proposition due to poor performance and poor cost-efficiency.

Scaling on A Large Cloudlet

In a realistic edge deployment, a cloudlet typically has dozens of CPU cores that process data from a dozen disks in parallel. Among others, *general-purpose* compute cycles are a precious resource. Utilizing decode-enabled storage is a more economic way to improve elasticity than

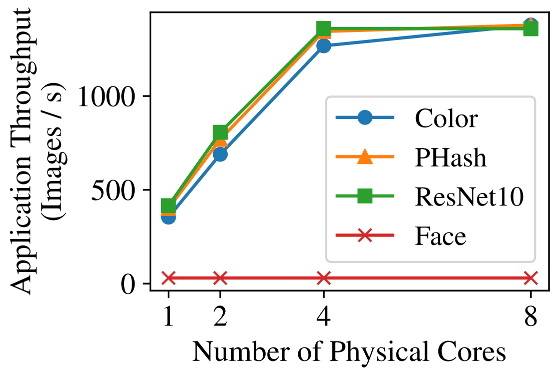


Figure 5.15: Effect of CPU Cores on Throughput

adding more CPU hosts. This leads to a question: *"For a given cloudlet, how many disks should be used?"*

We first investigate how many CPU cores the application utilizes when saturating a single disk. Figure 5.15 shows the application throughput on a single decode-enabled disk, as we vary the number of physical cores allocated to the processes. The "kinks" of the curves indicate when the CPU cores start being underutilized, as the bottleneck shifts to I/O. For `Color`, `PHash`, and `ResNet10`, the sweet spot appears to be 4 cores / 8 threads. `Face` is not limited by CPU, as the processing running on CPU is negligible.

Based on measured CPU/GPU utilization at that point of saturation, we extrapolate the scaled-out performance as more standard or decode-enabled disks are connected to our 36-core cloudlet. This machine has 136 GB/s DRAM bandwidth, well above the bandwidth demands calculated in this scenario. Besides, we benchmarked the GTX 1080 Ti GPU can run `ResNet10` at 40,000 images/sec, and should not be a bottleneck in our scaling range. Figure 5.16 shows the extrapolated application throughput as the number of disks is increased. For all cases, decode-enabled storage achieves more than 2x higher throughput than standard disks. It also suggests connecting up to 20 decode-enabled disks to a cloudlet, requiring up to $20 \times 1.5 = 30$ GB/s of data transfer rate. Can a modern machine support this required I/O bandwidth? With 40 lanes of PCIe 3.0 per socket, and two sockets, the benchmark machine has a theoretical peak I/O bandwidth of 80 GB/s, so it is feasible to support more than 20 decode-enabled disks in one system.

Figure 5.16 also shows that the general-purpose CPU cores, when freed from the decode task, can execute the early-discard filters at more than 15,000 image/s. This is approximately 20x higher than the JPEG decode throughput we can get from the Intel Quick Sync Video accelerator (Table 5.3). In other words, on-die accelerators need to be scaled up by 20x to meet the whole CPU's processing speed, of which the challenges were discussed in Section 5.2.4.

Micro Benchmarks on Video

We run experiments on the VIRAT video data set using a "frame sampling + image difference" pipeline. Here, we sample video frames at fixed intervals; then compute the mean squared error (MSE) between the sampled frame and the frame one second (30 frames) earlier. If MSE is

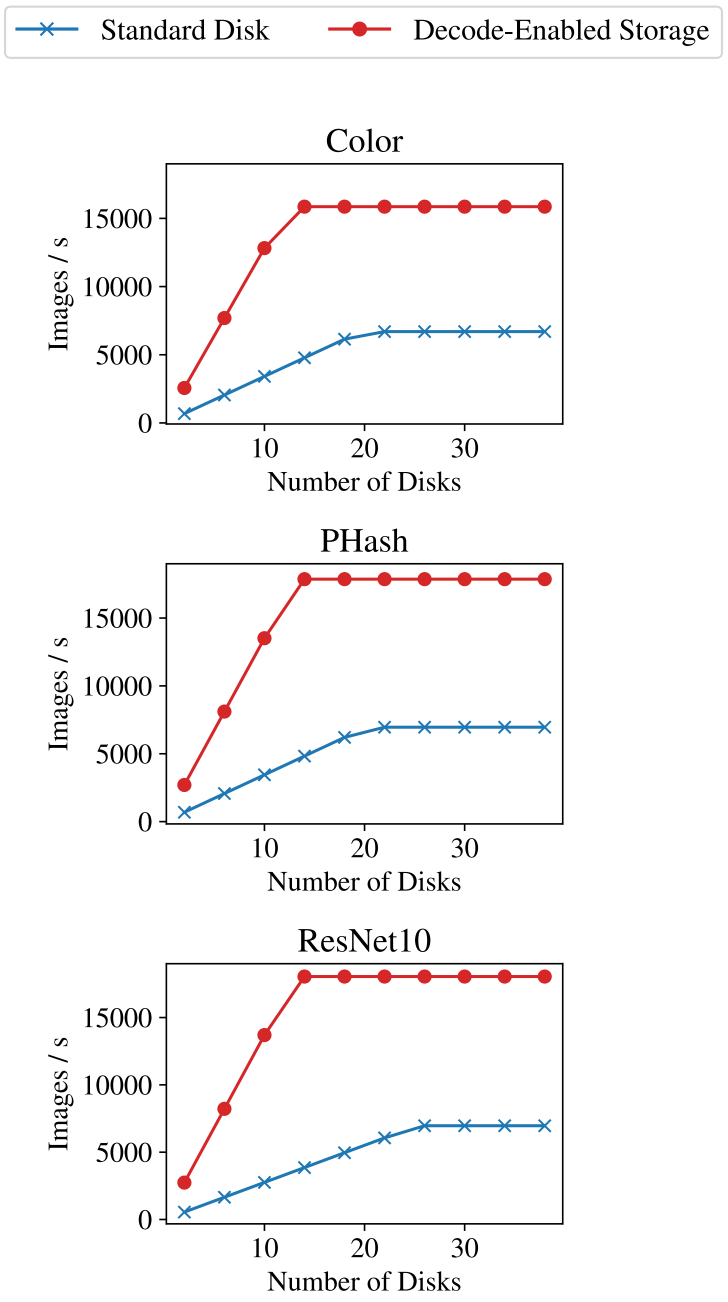
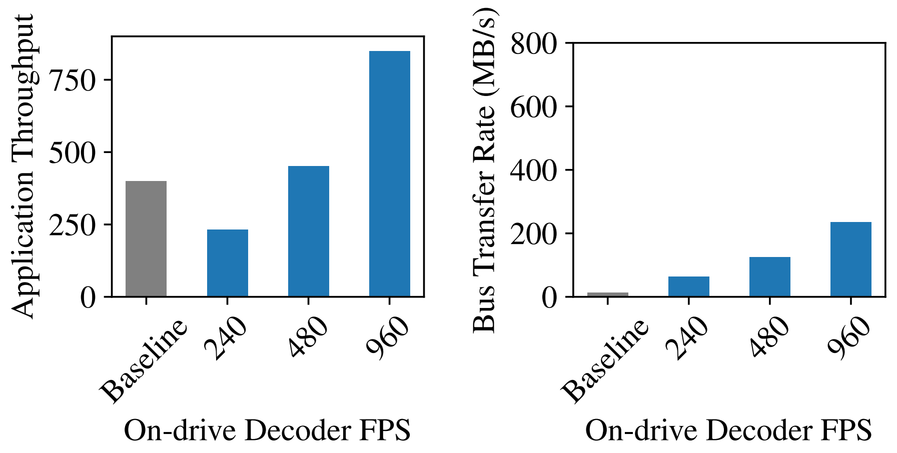
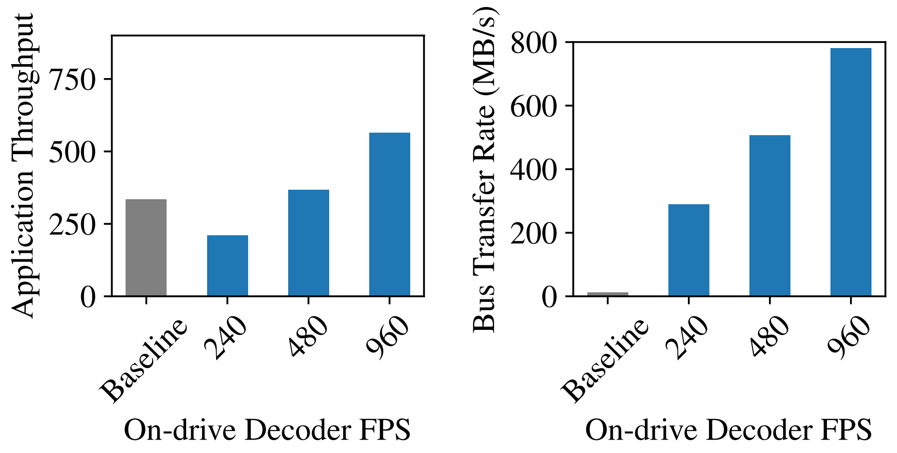


Figure 5.16: Extrapolated Application Throughput with Varying Number of Disks



(a) Sample 10% of Frames (every 10 frames)



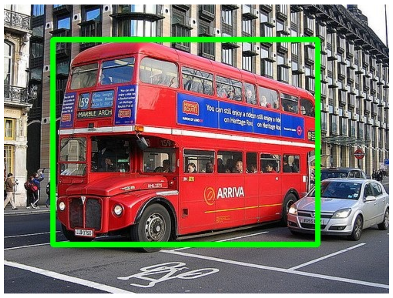
(b) Sample 50% of Frames (every 2 frames)

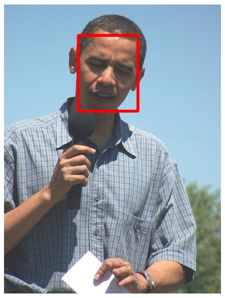
Figure 5.17: Effect of Video Decoding on CPU (Baseline) vs ASIC

lower than a threshold, we consider the frame to have the same content as the prior one and suppress further processing. With decode-enabled storage, decoding is performed on the disks and skipped frames are not transmitted. MSE computation always runs on the host.

Processing video shows two key differences from processing image. First, because of the very high compression ratios and many frames per video, disk read is not as much of a bottleneck as with images, and the use of batch iteration makes little difference. Second, the frame sampling rate greatly affects the relative cost of decode. When frames are sampled more frequently, an increased fraction of CPU cycles need to be devoted to MSE calculation, while the decode costs remain constant since most frames need to be decoded due to the sequential nature of modern video encoders like H.264.

Figure 5.17 reports the application-level throughput and host-disk data transfer rate under two sampling rates: 10% and 50%. We varied the video decoding throughput of decode-enabled storage from 240 to 960 FPS. Higher video decoding capacity in the disk leads to higher application-


YFCC100M/RedBus


YFCC100M/Obama

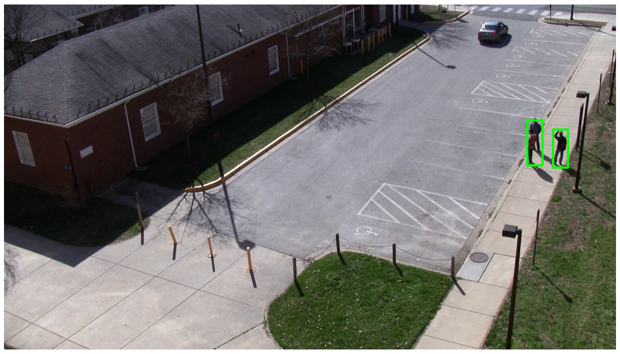

VIRAT/Pedestrian

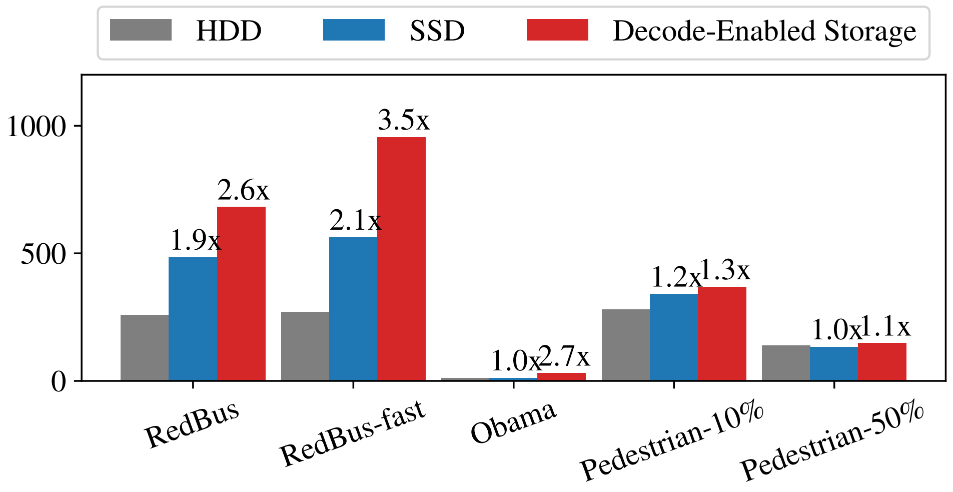
Figure 5.18: Example Results from Full Eureka Pipelines

level throughput, but not proportionally, due to overheads of MSE computation. When sampling rate is high (Figure 5.17(b)), application throughput is lower (600 vs. 800), because more MSE computation is needed. Meanwhile, host-disk transfer rate will be higher (800 vs. 300 MB/s), as more decoded frames are sent over the bus. Overall, we observe that decode-enabled storage must have decode capability of over 480 FPS (at 720p) to outperform baseline on this task.

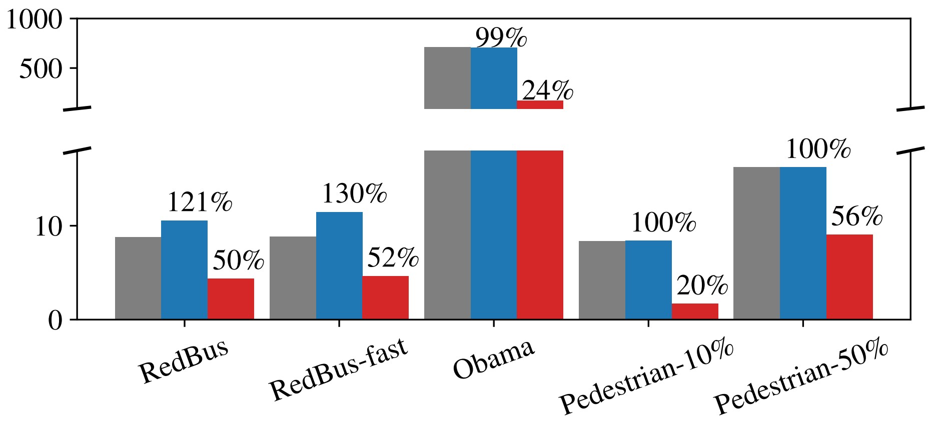
5.2.8 End-to-end Evaluation

Finally, we evaluate the following complete Eureka pipelines for real-world search tasks:

- `RedBus` runs the pipeline depicted in Figure 5.4 to find red buses in YFCC100M. It first runs a redness color filter, and then passes the candidates to an SSD-MobileNet [94] running on the GPU to detect the presence of buses.
- `RedBus-fast` trades off accuracy for speed by replacing object detection with image classification (MobileNet [63]). Classification is faster, but may miss images where the bus is not the dominant object.
- `Obama` searches for Barack Obama in YFCC100M. It first runs face detection to discard images without faces, and then runs face recognition on the face patches.
- `Pedestrian` detects humans in VIRAT videos. It performs frame sampling and uses image difference to filter sampled frames. Because the VIRAT videos are captured from wide angles and far distances, the candidate frames are passed to Faster R-CNN ResNet101 to detect humans, a more expensive but accurate DNN than SSD-MobileNet for this kind



(a) Application Throughput (Images / sec)



(b) CPU Time (Milliseconds) per Image

Figure 5.19: Full End-to-End Visual Pipeline Performance

of task. We evaluate this with two frame-sampling rates: 10% and 50%.

Figure 5.18 gives a search result example of each application. The selectivity — fraction of images/frames that contain the search target — is 0.01% for `RedBus`, 0.004% for `Obama`, and 2.45% for `Pedestrian`.

We compare the performance of these visual search tasks on (1) a standard SATA HDD; (2) a standard SATA SSD — today's go-to choice for fast but expensive storage; and (3) our proposed NVMe-connected Decode-enabled Storage. Figure 5.19 reports the application throughput and CPU cost per image for each application and storage type. The annotated numbers are improvements relative to the standard SATA HDD (gray bar).

Overall, decode-enabled storage shows greater throughput with lower host CPU cost than standard HDD and standard SSD. Comparing `RedBus` and `RedBus-fast`, we observe that this commonly used classification-detection tradeoff is effective only when the disk and decode overhead is removed. With `Obama`, throughput is limited by face detection either in software or hardware. Similarly, `Pedestrian` is largely limited by video decode speed, which is similar in software and hardware. Nonetheless, offloading decode reduces the host CPU cost by 50–80%, allowing more apps to run in parallel on an edge node.

5.3 Related Work

Edge computing features miniature cloud-like compute infrastructure that is deployed close to mobile and IoT devices [127, 134]. This compute infrastructure is exploited either to provide latency-sensitive interactive user experience, such as in wearable cognitive assistant [52, 155] and mobile gaming [133], or to execute pre-filtering of data, thus reducing the demand for WAN bandwidth, such as in distributed video analytics [19, 77, 78]. Similar to the latter, Eureka utilizes edge computing to execute early-discard filters that reduce the volume of data to transmit over the Internet by orders of magnitude.

As mentioned, unlike the cloud, elasticity of the edge is limited. However, it can be improved by coordinated execution between the cloudlet and mobile devices or between different applications [60, 66, 154, 155]. Our observation of image decoding and disk I/O being the local bottleneck stems from the unique characteristics of Eureka workload. In contrast to Eureka, edge applications that process real-time video feeds (e.g., wearable cognitive assistant) obtain frames from cameras directly, without encoding and storing on disks. Likewise, many video analytic systems (e.g., [77]) are GPU-bound as they need to execute DNNs on many frames. Most relevant to Eureka are systems such as NoScope [81] and BlazeIt [82], which focus on the selection and creation of efficient early-discard filters. The use of effective early-discard filters reduces the load on GPUs and shifts the bottleneck towards the early steps (I/O and decoding) of the processing pipeline. Nonetheless, that body of work generally considers target objects for which there exist off-the-shelf DNN detectors. For example, NoScope [81] uses a standard “teacher” DNN to train a light-weight “student” DNN as an early-discard filters. Such “teacher” models are not available in Eureka’s use cases.

Our proposal of decode-enabled storage can be considered as a revisit of *intelligent storage* or *active disks* [120], which refers to the execution of application logic inside storage devices. An excellent account of the origin and history of the active disk concept is provided by Riedel [121]. That work attributes the roots of this concept to research on database machines from the mid-1970s through the early 1990s [17, 34, 35, 36, 64, 114, 140]. Riedel’s work confirmed the significant performance benefits of this approach for systems of the late 1990s to early 2000s.

Various other researchers have also investigated this concept, including Acharya et al [3], Keeton et al [84], Ma et al [98], Memik et al [100], Rubio et al [122], and Wickremesinghe et al [159]. Closely-related work on optimal function placement at different levels of the memory hierarchy include Abacus [8], Coign [68], River [9], and Eddies [10]. By the mid-2000s, interest in active disks had faded. Their predicted wins were muted by the onward march of commodity hardware performance through Moore’s Law. By the late 2000s, active disks appeared to be an idea whose time had come and gone. There has recently been renewed interest in intelligent storage driven by the emergence of SSDs [15, 26, 37, 83, 147, 151]. Most of this work focuses on file system and database workloads rather than multimedia processing.

Decode-enabled storage complements improvements in other layers of multimedia systems, such as visual data encoding [85, 115, 139], hardware accelerators [5, 21, 27, 58, 62, 72, 79, 89, 163], computer vision algorithms [56, 59, 63, 119], and distributed data processing systems, including some optimized for multimedia data [4, 41, 51, 112, 137, 165].

Chapter 6

Extending Eureka to Detect Temporal Events in Video Data

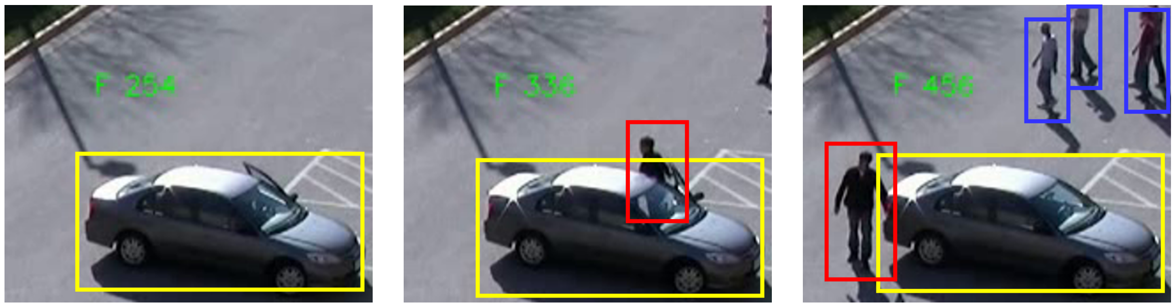
In the previous chapters, we have focused on discovering objects from static images and have treated video data as "a collection of image frames," so that we can adapt techniques designed for static images to videos. While this methodology suffices for a limited set of use cases, it does not fully realize the value of video data. Consider the task of finding examples of the event "a person getting out of a vehicle" (abbreviated get-out), one of the annotated events from the VIRAT data set [109]. If we have hundreds of *labeled* video clips of this event, recent work on *activity recognition* [47, 75, 148, 164] can be applied to train a DNN from those examples. How shall we obtain this training data? Traditionally, these examples are found by human labelers (e.g., through crowd-sourcing) who spend many hours watching possibly boring street surveillance feeds and annotate all "get-out" events they see. Clearly, this cannot scale out to millions of cameras and to specialized domains. Can an expert do better?

By definition, an accurate "get-out" event detector does not exist yet. What an expert can rely on is a discard-based search that hopefully prunes much of the irrelevant video segment. If early-discard is effective, the expert will be able to find a modest number of the event instances without plowing through *all* video record. What early-discard conditions should an expert use?

Notably, the early-discard condition is not only determined by an expert's intuition, but also constrained by visual information that can be reliably extracted using state-of-the-art computer vision techniques. In this "get-out" example, the objects "person" and "vehicle" can be reliably detected in images, but the minuscule gesture of opening a car door may not. Thus, one solution is to filter the video frames by the presence of vehicles and persons in the scene. This makes sense — a scene without a vehicle or a person cannot have the target event. Standard object detection methods for image analysis (e.g., Faster R-CNN [119]) can be applied on individual video frames with little change.

Alas, on an urban road, there are probably some persons and some vehicles at any point of time, even though the "get-out" event is infrequent. Using the above early-discard approach, an expert may still have to watch and label a significant portion of the video data, if not all of it. In other words, improvement of *productivity* is marginal.

The key to this problem is to consider a temporal sequence of frames, instead of filtering frames independently. Figure 6.1 shows a frame sequence of the "get-out" event from VIRAT.



Frame 254 (a) | Frame 336 (b) | Frame 456 (c)

Yellow box: vehicle;
Red box: the person getting out of the vehicle;
Blue box: person irrelevant to the event

Figure 6.1: Examples Frames of "Get-Out-of-Vehicle" Event in A VIRAT Video Clip

One can easily conclude that a person is getting out of a car by looking at the sequence. However, it is not obvious by looking at each frame separately. Reasoning on a sequence of frames, an expert may use the following heuristics: (a) a vehicle that is *stopped* at time t; (b) a person that is not in the scene before time t; (c) the person referred to in (b) *appears* in the scene at time t, *from* the vehicle referred to in (a).

How should an expert convey these heuristics to Eureka? How should Eureka convert these conceptions into computable programs? These are the questions addressed in this chapter. We assume individual objects involved in an event (e.g., person, vehicle, suitcase) can be reliably detected on the frame level. We introduce an approach to modeling events in terms of *inter-object spatial-temporal relationships.* We then describe how Eureka is extended to support and execute these models efficiently.

6.1 A 4-Level Taxonomy of Video Analytics

To start, we first consider the challenge in the broader context of a large body of recent work on the theme of *video analytics systems.* Since AlexNet's victory in the 2012 ImageNet competition — a task on analyzing *static images*, there has been growing interest in applying deep learning to analyzing video data. Two questions follow naturally: *"What can we do with video data?"* and *"How should we do it?"*

Figure 6.2 presents a 4-level taxonomy that can be used to classify current and future research on video analytics systems. As the levels progress, it allows us to extract "deeper" knowledge from video data. Each level poses unique challenges in terms of system design and algorithms

Level 0 (L0) represents the simplest and almost trivial way of handling video data. It treats video as nothing but a bunch of independent static images. Standard image analytics tasks, such as classification and object detection, can be applied to video data with little change.

Level 1 (L1) exploits certain attributes of video data in order to reduce processing cost and

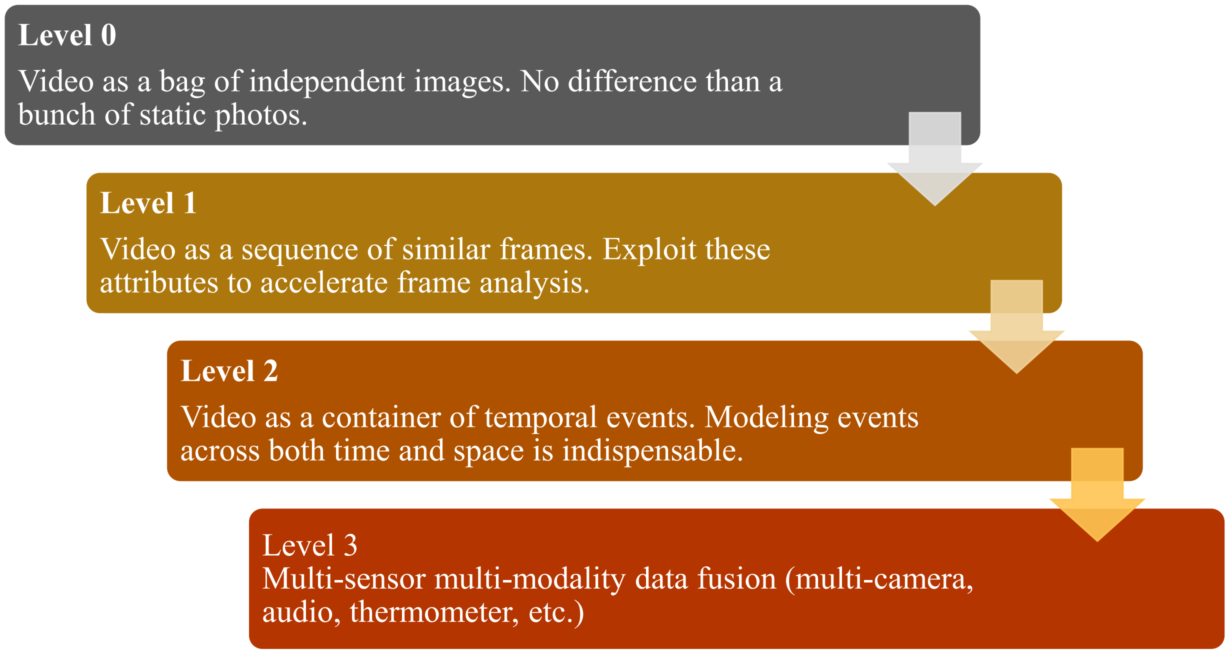

Level 0
Video as a bag of independent images. No difference than a bunch of static photos.
Level 1
Video as a sequence of similar frames. Exploit these attributes to accelerate frame analysis.
Level 2
Video as a container of temporal events. Modeling events across both time and space is indispensable.
Level 3
Multi-sensor multi-modality data fusion (multi-camera, audio, thermometer, etc.)

Figure 6.2: A 4-Level Taxonomy for Classifying Video Analytics Systems

improve application-level utility. The majority of recent work falls in this category. Table 6.1 lists some of the most important recently-published systems. Typically, L1 systems seek to reduce two types of system costs when running video analytics workloads: (1) execution of compute-intensive DNNs on scarce GPU device (e.g., [19, 77]); (2) transmission of frames from wireless devices over scarce 4G/5G bandwidth (e.g., [154, 155]). They take advantage of the fact that frames from a video stream are highly correlated in terms of background, scale, motion speed, appearing objects, etc.; rather than being independent. Commonly used optimizations include:

- Sampling frames at a lower frame rate than the camera source to avoid analyzing every raw frame.
- Reduce redundant computation via inexpensive motion/change detection, such as background subtraction, pixel-level delta, perceptual hashing.
- Training light-weight machine learning models that are effective only for the target camera stream, but not in general settings.

Nonetheless, as can be seen in Table 6.1, L1 systems mostly focus on frame-based analysis. Hence, they are able to answer questions such as “Are there more than 5 cars in this frame?”; and simple aggregations based on those results, such as “What is the 95%-tile of car counts in all frames?”

Level 2 (L2) represents a significant semantics leap from L1, as frame-based analysis alone is insufficient for tasks at this level, as shown in Figure 6.1. L2 systems treat video as a spatial-temporal container of events. Detecting and modeling events in both space and time is indispensable. Because of the need to maintain temporal information, L2 analytics poses additional challenges in terms of algorithmic design and system optimization. L2 analytics is the focus of this chapter.

System from recent work	Search targets considered
NoScope [81]	Person, bus, car
BlazeIt [82]	Bus, car, boat
Chameleon [78]	Targets from the COCO data set [93]
Focus [65]	Targets from the COCO data set
Mainstream [77]	Pedestrian, bus, car, train
FilterForward [19]	Pedestrian, pedestrian wearing red
Live drone video analytics [154]	Person, car, raft, elephant
Wearable cognitive assistance [155]	Lego blocks, ping-pong table, human face

Table 6.1: Search Targets in Recent Work of Video Analytics Systems

Activity recognition from the computer vision literature [47, 75, 148, 164] can be seen as a form of L2 analytics. However, this body of work follows the standard *supervised learning* approach which requires sufficient annotated training data. We give a more detailed account of this related work in Section 6.7. *Object tracking* [157] is another heavily-studied problem that falls in the L2 category. Tracking is a very specific task, though we will show that it is an important building block in Eureka.

Finally, Level 3 (L3) video analytics involves fusion of multi-sensor multi-modality data. This includes video data from multiple cameras, as well as other sensor types such as audio sensors and thermometers. L3 analytics enables new tasks that cannot be accomplished with a single video camera. Some researchers have started to consider this problem [73], but overall it is still a nascent area.

In the above 4-level taxonomy, higher levels subsume lower levels. Tasks of a lower level can be formulated as a degenerated case of a higher level. For example, L1 frame-based analysis can be seen as finding one-frame L2 events (e.g., "object X appears in frame t"); L2 event analysis in a video stream can be seen as n-camera fusion where $n = 1$.

6.2 Modeling Spatial-Temporal Events in Video

As discussed earlier, the collection-of-frames view is insufficient for modeling many events that can be discovered from video data. In this section, we describe a programming abstraction that is more suitable for modeling those events. In particular, we focus on modeling events that can be described by *coarse-grained inter-object spatial-temporal relationships.* We assume the individual objects participating in an event can be reliably detected on per-frame basis. This approach complements computer vision methods that focus on low-level fine-grained features (discussed in Section 6.7).

Our approach is based on the observation that complex events can be specified recursively in terms of simpler events that conform to a set of relationships. For example, object detection on a single frame can discover the following simple event:

"Presence of a person at position (x, y) in frame t"

Grouping all presence of the same person across consecutive frames — typically formulated as a *tracking* problem — produces the person's motion trajectory over time. This can be interpreted as the complex event:

"A person appears in the video at time-space sequence
$(t, x_t, y_t), (t+1, x_{t+1}, y_{t+1}), \ldots, (t+n, x_{t+n}, y_{t+n})$."

On top of this, more complex events can be specified, for example, by comparing a person's trajectory with a vehicle's position, which may calculate the following event:

"A person first appears from a vehicle located at (x, y) in frame t."

In real-world video data, there are likely many persons and many vehicles in the scene, though not every pair of them constitutes the above event. Finding the matches requires comparing them under a set of constrains. These constrains may be not only on their time and space, but also on their visual features. To the best of our knowledge, Eureka is the first system that integrates operations on time-space information (typically considered as "metadata") and operations on visual content in a unified framework.

6.2.1 Spatial-Temporal Interval

We extend Eureka's programming abstraction to explicitly represent the following of an object or event: (a) its spatial position; (b) its temporal position; (c) its visual content, extracted features, and other information. These are encapsulated in the *spatial-temporal interval* data structure (*"interval"* in short):

```
class Interval {
    struct {
        float x1, x2, y1, y2, t1, t2;
    } bounds;

    KeyValue<String, Anything> attributes;
};
```

Figure 6.3 shows how this interval abstraction unifies visual content of different granularities. The `bounds` coordinates define an axis-aligned bounding box. By default, the x-y coordinates are between 0 and 1, and are *relative* to the screen space. This allows for simple calculation of spatial predicates (e.g., overlap) even if the visual content (pixel arrays) are down-sampled. The temporal duration of a single frame is assumed to be the reciprocal of the the frame rate (FPS).

The `attributes` key-value dictionary is similar to the attributes described in Section 3.2 and supports values of any type. Most notably, a dictionary value can be:

- Raw pixels arrays representing the visual content within the interval's `bounds`
- Feature vectors computed on the pixel content
- Meta-data extracted by computer vision algorithms (e.g., labels, confidence scores)
- Pointers to another `Interval` instance
- Nested structure of the above, for example, a list of `Interval` objects.

The `Interval` data structure can seen as the "container" of an event. Hence, the `bounds`

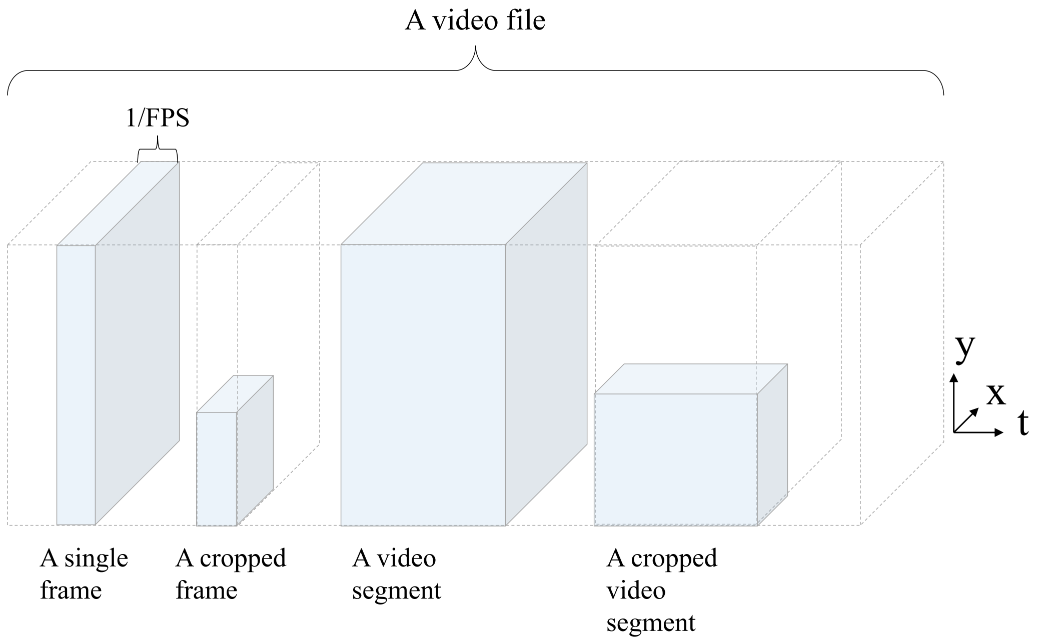


Figure 6.3: Spatial-Temporal Interval Unifies Different Granularities of Visual Content

should be *tight and complete* with regard to the `attributes`. In other words, the information carried in the `attributes`, let it be RGB arrays or meta-data, must span the full extent of the `bounds`, no less and no more. Manipulation of intervals should maintain this semantic invariant. For example, making a crop of a frame requires: (1) shrinking the `bounds` to reflect the spatial position of the crop; (2) changing the RGB array maintained in `attributes`.

Compared to the collection-of-frames abstraction, the use of intervals provides a finer-grained notation of the space-time of objects/events. It also allows us to model events that cannot be modeled on per frame basis. Besides, it fits with the "entry-points" of most computer vision primitives. For example, most image classification and object detection algorithms take rectangular-shaped images as input; activity recognition algorithms take sequences of temporally-consecutive frames as input. Both of them are trivially represented by intervals. This unified representation allows flexible *composition* of computer vision primitives even if they receive different forms of data as input. For example, detecting the "get-out" event involves operators that work on different granularity of data: (1) object detection (e.g., by Faster R-CNN) that works on individual frames; (2) object tracking (e.g., by optical flow) that works on a fixed-length sequence of W frames; and (3) generating sliding window of W frames from an endless stream.

There exist more complicated and finer-grained abstractions from computer vision literature, such as polygon mesh and point cloud. Although these abstractions can be more powerful, they are less intuitive and more difficult to reason with for a domain expert who does not specialize in computer vision. Nonetheless, Eureka does not preclude extensions to support those abstractions. In fact, the interval abstraction provides a good basis for such extensions. Consider computing whether two arbitrary polygons overlap, which is $O(n)$ in computational complexity. If built on top of the interval abstraction, it can be accelerated by first computing whether their respective axis-aligned bounding boxes overlap, which takes only $O(1)$ time.

Category	Example Operators and Predicates
Generic	map, reduce, filter, sort, join, flatten
Data transformation	VideoToFrames, FramesToVideo, ImageCrop, VideoCrop
Spatial	IoU (intersection over union), merge_span, above, below
Temporal	during, before, after, coalesce
Visual features	RGB color histogram, perceptual hashing, SIFT key points, image classification, object detection

Table 6.2: Example Operators and Predicates in Eureka

6.2.2 Interval Stream

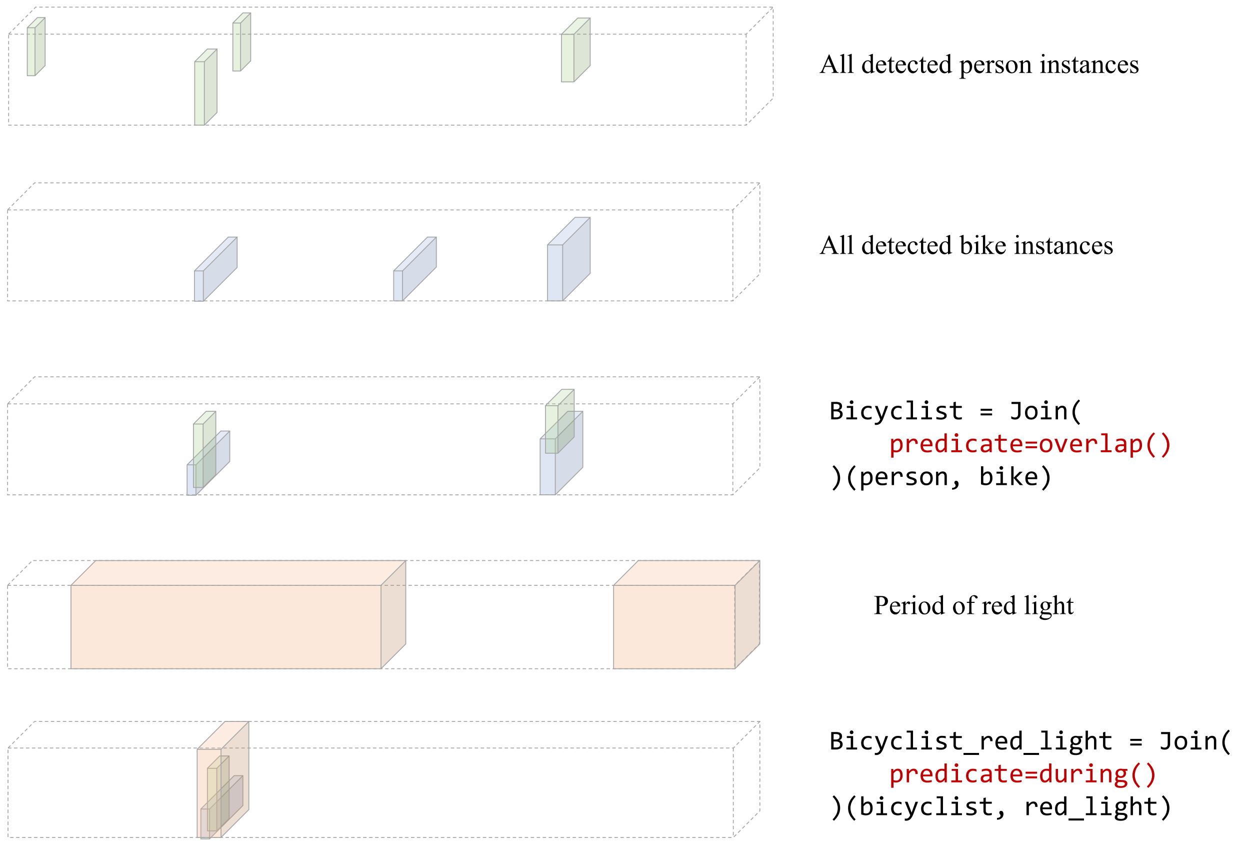
An *interval stream* is a stream of intervals that are in ascending order of `(t1, t2)`. Typically, an interval stream is used to represent all instances of the same type of event (e.g., presence of a vehicle), which are considered as the "candidates" used to compose a more complex event. The choice of the stream abstraction, rather than the set abstraction (i.e., not necessarily ordered in time), offers two benefits: (1) it maps naturally to live video feeds so that Eureka's implementation can be applied to live cameras and archival video records without change; (2) it enables optimization of the query engine and certain operators, which we will discuss later.

6.2.3 Operator on Interval Streams

Finally, Eureka enables the discovery of novel events through the use of *operators*. An operator takes interval stream(s) as input and outputs interval stream(s). Since both the input and output are interval streams, one can pipeline the output of an operator into the input of another and therefore compose a complex query in the form of a directed acyclic graph (DAG) of operators. An operator examines the spatial/temporal/visual information of its input "candidates" and emits those satisfying certain constraints. A user specializes the behavior of operators by passing in different parameters. Eureka offers a set of operators found in standard "big data" systems, such as `map, filter, reduce, flatten, sort, join`. Of course, its power stems from specializations of these generic operators that encompass a rich set of spatial, temporal, and, visual predicates. Table 6.2 lists some commonly used operators and predicates. We will elaborate these operators through more concrete examples in the rest of the chapter. The use of DAGs, in principle, allows for open-ended composition that can detect very complicated events.

6.2.4 Example

Figure 6.4 illustrates the concepts of interval, interval stream, and operator in a simple example that detects "a person riding a bike during red traffic light." Here, each row represents an interval stream where each colored cuboid represents an interval. Assume *object detection* on individual video frames produces the atomic events "person instances" and "bike instances." A `Join` operator with the spatial `overlap()` predicate finds matching person-bike pairs that indicate the event "bicyclist." On top of that, joining these "bicyclist" events with the "red light" events



```
Bicyclist = Join(
    predicate=overlap()
)(person, bike)
```

```
Bicyclist_red_light = Join(
    predicate=during()
)(bicyclist, red_light)
```

Figure 6.4: Example: “A Person Riding A Bike During Red Traffic Light”

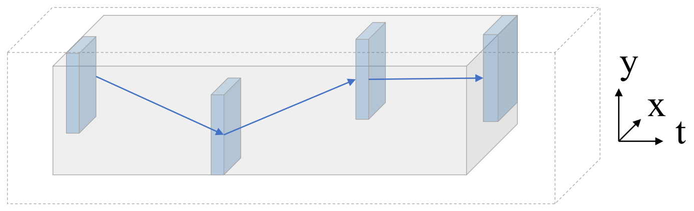


Figure 6.5: Representing An Object's Trajectory in Eureka

using the temporal `during()` predicate selects only those bicyclists who appear simultaneously with a red traffic light.

6.3 Event Discovery Idioms

In this section, we introduce several event discovery "idioms" that we empirically find to be useful in a wide range of use cases. These idioms embody certain ways of reasoning about multiple events' relationships, and more technically, recurring query (sub-)graph patterns. We do so by walking through several concrete examples. Throughout these examples, we will also introduce the most important operators in Eureka as well as several implementation details.

6.3.1 Representing and Computing Object Trajectories

One cornerstone of modeling temporal events in video data is to represent an object's motion trajectory across space and time. This takes more than object detection on individual frames — we must maintain the identity of the object at different timestamps.

In Eureka, a trajectory is represented as *a temporal list of intervals belonging to the same entity*, indicated by the blue cuboids in Figure 6.5. Here, each blue box represents the atomic event "object entity X appears at time-space (t, x, y)." Bundling them together creates a higher-level event, that is, the object's trajectory. The trajectory, considered as an interval by itself, has its `bounds` corresponding to the gray box in Figure 6.5. Essentially, the gray box "spans" all of the blue boxes. The blue boxes, in turn, are stored under the `attributes` of the gray box.

The task of computing object trajectories has been traditionally considered as "tracking" in computer vision. In particular, when there are multiple objects to track in the scene, it creates challenges in terms of computation and ambiguity. In the following, we describe a Eureka query to accomplish this task. Although there have been many specialized solutions to the problem [160, 161], our approach has the advantage of simplicity. It is composed only of off-the-shelf object detector and single-object trackers, and produces reasonable results in many simple cases.

Figure 6.6 shows the query graph of the query. This is a relatively simple linear DAG. Figure 6.7 shows its line-by-line Python code and visualization of intermediate results. Each line uses an operator (name in blue font) to compute an output interval stream (LHS of =), which is used as input to the next operator. On the right shows the output stream of each line, where every colored cuboid represents an `Interval` object in the stream. The horizontal dimension is time.

Line 1 parses a local video file and emits intervals representing all frames. Note that the RGB pixels of the frames, which can be accessed through the reserved attribute name `rgb`, are

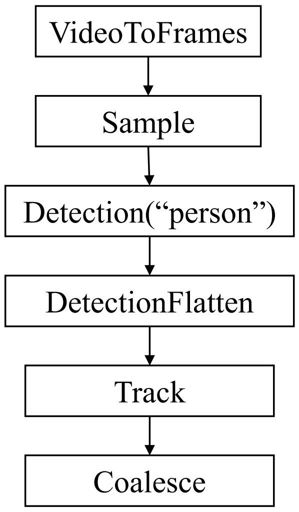


Figure 6.6: Query Graph: Finding Person Trajectory

not materialized at this point. Each interval simply has different `(t1, t2)` values and has a pointer to a video decoder. The RGB pixels are materialized only when the attribute `rgb` is accessed for the first time — in this case, Line 3. This *late materialization* strategy reduces memory footprint to hold decoded data and avoids unnecessary decoding.

Line 2 samples the frames with a step size of 5. The input stream to the `Sample` operator is the output stream from Line 1.

Line 3 runs DNN-based object detection on each (sampled) frame to detect persons. Here, the operator `Detection` is a specialization of `Map` and treats each frame as an independent image. RGB pixels are needed to run object detection, so the `rgb` attribute is materialized during execution. The JSON-style detection result is stored as an attribute in the output intervals. Typically, the result includes x-y coordinates and confidence scores of multiple detected instances (black-edge hollow boxes).

Line 4 parses the JSON-style detection result extracted in Line 3, and emits smaller intervals that tightly bound individual persons. Here, each input interval may generate multiple output intervals. This is done by the operator `DetectionFlatten`, a specialization of `Flatten`. Each output interval essentially represents a crop of a video frame. Likewise, the RGB pixels of these crops are not materialized at this point. In total, the operator receives 4 inputs and emits 6 outputs.

Line 5 uses a cross-frame single-object tracking algorithm to track each input interval forward in time. The tracking is done for up to five frames in the original video but may lose track earlier. We use a small tracking window because most tracking algorithms tend to lose track over time, and we want to re-initialize the trackers with object detection. The window size is tunable to trade off between expensive object detection and accuracy. The `Track` operator generates a short trajectory (black curved arrows). As mentioned, a trajectory is programmatically a temporal list of (nested) intervals (not shown here). Again, these short trajectories are stored as an attribute in each output interval.

Finally, Line 6 attempts to merge short trajectories found in Line 5 to form complete tra-

```
all_frames = VideoToFrames("input.mp4")()

sampled_frames = Sample(step=5)(all_frames)

detections = Detection(
  class="person")(sampled_frames)

persons = DetectionFlatten()(detections)

short_trajectories = Track(
  method="optical_flow")(persons)

full_trajectories = Coalesce(
  predicate=lambda i1, i2:iou(
    i1.trajectory.last,i2.trajectory.first)>0.5,
)(short_trajectories)
```

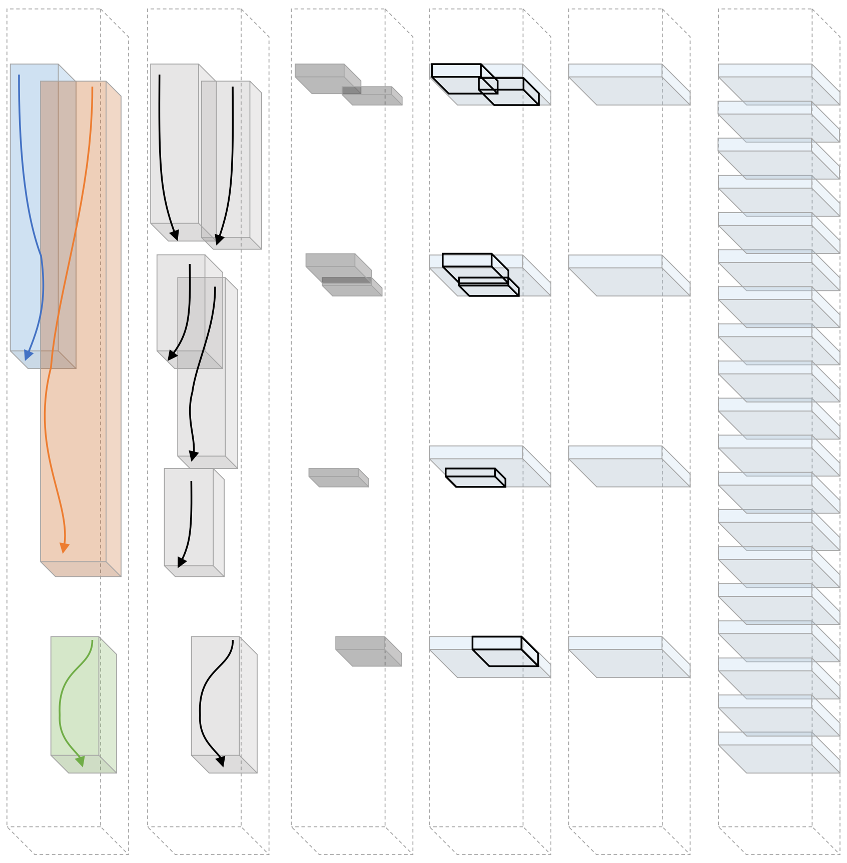


Figure 6.7: Step-by-step Visualization of the Person Trajectories Query

jectories of unique persons. We do so with the `Coalesce` operator, which tries to merge two temporally-adjacent intervals into a larger one, if they satisfy a predicate. Eureka provides a library of auxiliary functions to interpolate, smooth, and compare trajectories, which can be passed in as the predicate. For illustration purpose, Figure 6.7 includes the function definition of a relatively simple predicate. Conceptually, this predicate tests whether the "end" of a short trajectory is spatially overlapped with the "start" of another that immediately follows in time. A low IoU means they are likely of two different persons and should not be merged. More sophisticated predicates can be used in place of IoU, though, such as color histogram, perceptual hashing, and deep feature extractors [161]. The output intervals, colored differently, ideally represent the complete trajectories of three unique persons. The trajectory in green is not coalesced with any other because it is not temporally adjacent to any, indicating a person who newly entered the scene. Also note that the coalesced trajectories (red and blue) have bigger `bounds` than their constituents in Line 5, following the principles that the bounds must be complete and tight.

6.3.2 Predicating on Trajectory Relationship

The notion of trajectory gives us a powerful tool to express intuition of temporal events. Consider the "a person getting out of a vehicle" (get-out) event mentioned earlier. How shall we distinguish the get-out event from the following events, when all of them involve persons and vehicles?

- A person getting into a vehicle
- A person walking past a vehicle from elsewhere
- A vehicle moving past a person from elsewhere

We propose the following heuristics:

"When a person first appears in the video, she appears from a vehicle."

More technically:

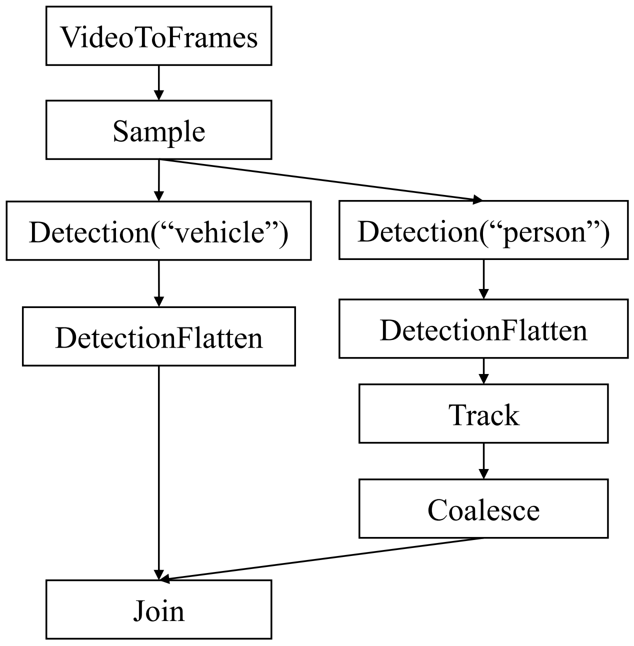
"The first position of a person's trajectory overlaps with a vehicle."

We can see that, if this condition can be evaluated accurately, it can successfully resolve the ambiguity mentioned above. A person that walks past a vehicle from elsewhere will likely have her trajectory originated from elsewhere in the scene, not the vehicle. Obviously, this condition cannot be evaluated on per-frame basis.

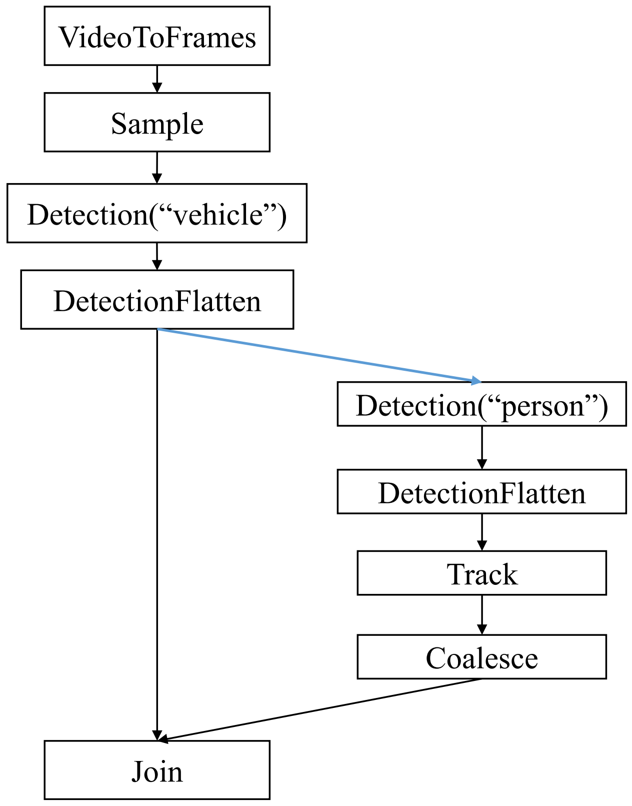
Figure 6.8(a) shows a straightforward translation of the above heuristics in a Eureka query. This query DAG has a branch. The left branch detects vehicles in individual frames, while the right branch constructs person trajectories using the same approach described in Figure 6.7. The `Join` operator receives two interval streams as input and finds vehicle-person pairs that satisfy our condition. The behavior of `Join` is controlled by passing in a join predicate:

```
lambda i_vehicle, i_person:
    during(i_person.t1, i_vehicle) and
    iou(i_person.trajectory.first, i_vehicle) > 0.5
```

This tests whether a person trajectory originates from a vehicle — both spatially (via the `IoU` condition) and temporally (via the `during` condition). The `.trajectory.first` attribute



(a) Simple version



(b) Improved version

Figure 6.8: Query Graph: Person Getting Out of Vehicle (Get-Out)

represents the location of that person's first appearance. In other words, the person is not in the scene before that point of time.

6.3.3 Content-based Hierarchical Detection

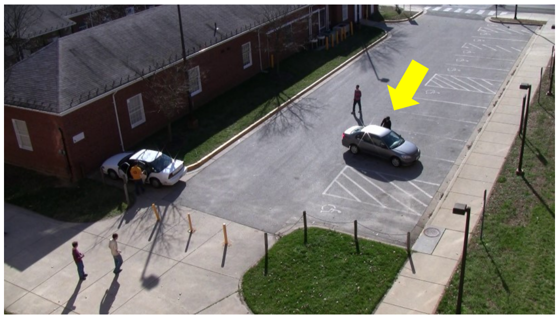
The query expressed in Figure 6.8(a) depends critically on accurate *object detection* on video frames to detect persons and vehicles. Unfortunately, modern DNN-based object detection algorithms (e.g., Faster R-CNN [119]) are well known to have difficulty detecting small-scale objects, because they always resize input images to a fixed size (e.g., 600x800). For the "get-out" event, we find that two factors exacerbate the issue. First, public surveillance cameras are often installed from a far distance and cover a large area, rendering each person relatively small in the screen space. Second, intrinsic to the event, a person who is getting out of a vehicle is likely occluded by the vehicle itself, making detection even harder. As a result, we observe that even state-of-the-art object detectors often fail to detect the person involved in the get-out event, as shown in Figure 6.8(a).

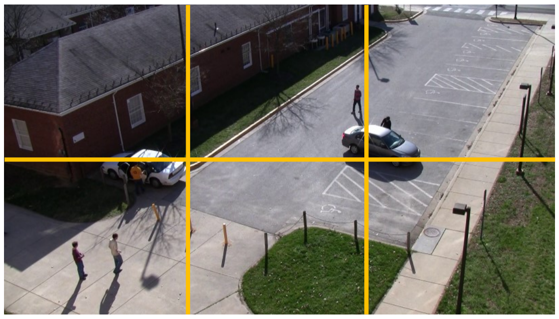
The traditional way to addressing this problem is to divide the screen into fixed-size tiles (Figure 6.9(b)) and run object detection on each tile separately, so that the relative scale of the objects become larger, which in turn increases the chance of successful detection. However, this approach has three drawbacks: first, it increases cost of GPU-bound DNN-based object detection linearly with the number of tiles; second, it does not account for scale variance – objects farther from the camera may require more zoom-in than closer ones; lastly, it is non-trivial to handle objects crossing tile boundaries.

Figure 6.9(c) shows a better approach called *content-based hierarchical detection*. We observe that vehicles are usually larger and less prone to the small object issue. Hence, we first detect vehicles in the full frame. After that, we make a crop surrounding each detected vehicles and run the DNN again on the crops to detect persons. Similar to tiling, this greatly increases the relative scale of the persons in the second detection step and therefore increases the chance of success. In contrast to tiling, the size of the crop adapts naturally to the size of the vehicles. Besides, it avoids wasting DNN computation on frames where no vehicles are detected.

Figure 6.8(b) shows the query DAG that applies content-based hierarchical detection. Interestingly, compared to Figure 6.8(a), Eureka's programming abstraction makes it almost trivial to implement the change — all it takes is to move one edge in the DAG (colored in blue), or changing one line of Python code. The new query pipelines the individual vehicle patches coming out from the vehicle detector, instead of the full frames, into the person detector. Because `Interval` unifies the representation of both full frames and image patches, no change of the operators is needed.

Note that the query in Figure 6.8(b) also ignores persons that are not nearby a vehicle. Thus, the applicability of this optimization is specific to the search target. Without *a priori* knowledge of the the task, the system cannot apply this idiom during its pre-processing/indexing stage. This highlights the importance of the human-in-the-loop nature of Eureka.


(a) Small-object issue: the pointed person is not detected by Faster R-CNN


(b) Tiling increases input resolution to DNNs about also increases computation cost linearly

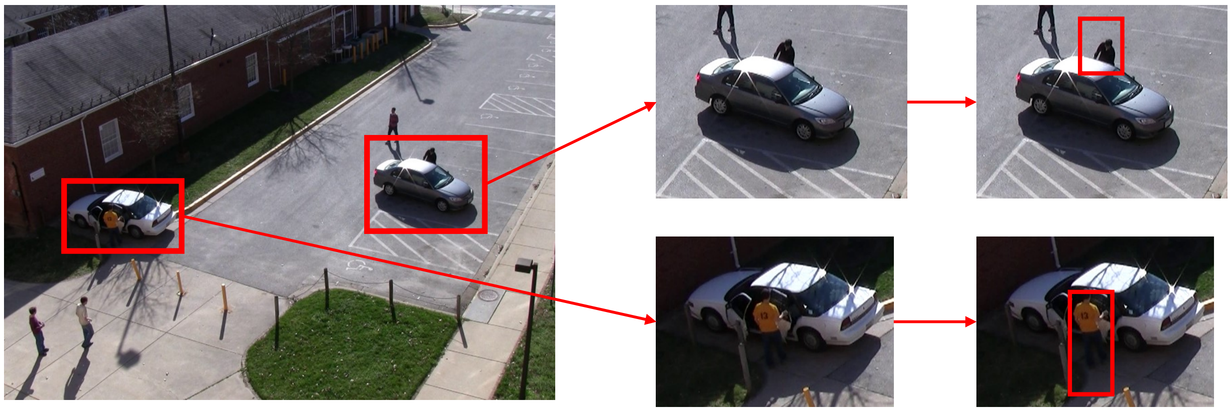
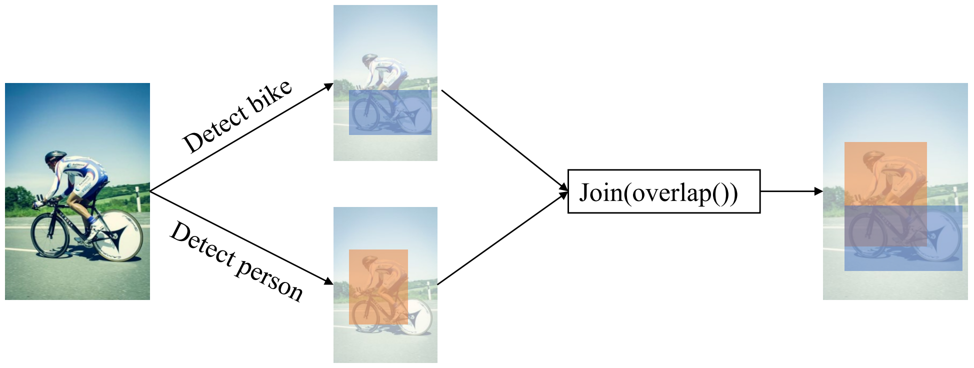
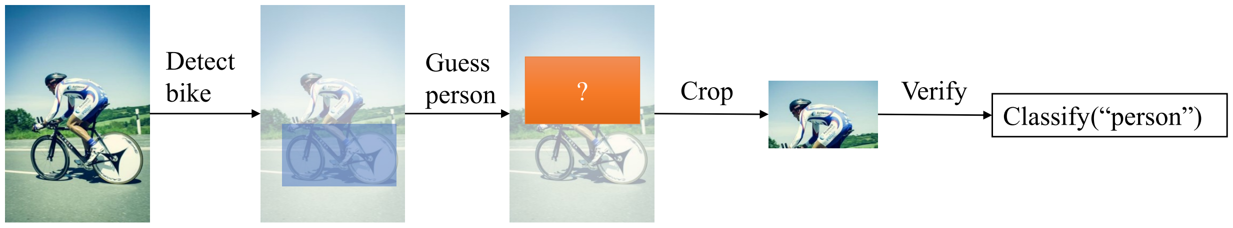

(c) Content-based hierarchical detection

Figure 6.9: Detecting Small Objects in Large-view Cameras



(a) Detect bicyclist by construction



(b) Detect bicyclist by guess-and-verify

Figure 6.10: Alternative Approaches to Detecting Bicyclist

6.3.4 Guess and Verify

So far, we have described several examples where we detect events *by construction.* That is, we start by detecting individual objects that involve in an event, and then we examine whether they relate to each other in a certain manner. Figure 6.10(a) visualizes this approach for detecting bicyclist: we first try to detect the bike and the person in the frame, respectively; we then check whether whether they overlap with each other.

Figure 6.10(b) shows an alternative guess-and-verify approach for the same task. In the beginning, we only try to detect the bike. Using the bike's position as an anchor, we *guess* the position of a person (orange box). This guess is based on our real-life knowledge: if this frame includes a bicyclist, the rider will probably be found on top of the bike. Finally, we crop our guessed area and *verify* whether there is a person in it. Again, without a large amount of training data to learn from, an expert's knowledge is the only source of the guessing heuristics.

Compared to Figure 6.10(a), this guess-and-verify approach has three advantages. First, we may use *image classification* instead of *object detection* in the verify step. As a result, it only incurs 1 detection call + 1 classification call per frame, rather than 2 detection calls. This greatly reduces computation cost as classification methods generally execute an order of magnitude faster than detection. Second, the verify step only needs to focus on a small region of interest rather than the whole frame, making it less prone to clutter in the background. Third, in case we do not have access to an effective person detector but just a person classifier, this is the only method that enables us to detect bicyclist.

```
def guess(i_anchor: Interval) -> Interval:
    # custom logic to guess target's location
    # based on anchor's location and visual information
    ...

def verify(i_target: Interval) -> bool:
    # custom logic to verify target, returns True or False
    ...

def postprocess(
    i_anchor: Interval, i_target: Interval) -> Interval:
    # custom logic to postprocess a matching pair
    ...

anchors = DetectionFlatten(anchor_label)(input)
guessed_targets = Map(map_fn=guess)(anchors)
verified_targets = Filter(filter_fn=verify)(guessed_targets)
final_results = Join(merge_fn=postprocess)(
    anchors, verified_targets)
```

Figure 6.11: Eureka Query Template of the “Guess-and-Verify” Idiom

In general, implementing the “guess-and-verify” idiom in Eureka is simple, using the the query template given in Figure 6.11. The expert only needs to provide the task-specific `guess`, `verify` functions and optionally the `postprocess` function.

6.4 Implementation and Optimization for Video Data

Eureka features the separation of *expression* and *execution*. A domain expert expresses *what* to compute in a succinct, declarative programming abstraction (Figure 6.7); the Eureka system takes care of *how* to compute it efficiently. Efficiency is the key here, due to the high bit rate of visual data and the compute- and memory-intensive nature of computer vision algorithms. On receiving a query, Eureka compiles it into a physical execution plan that optimally utilizes multi-core CPUs, main memory, GPUs, and parallel cloudlets.

On the high level, Eureka exploits distributed cloudlets and data sources to increase result delivery rate, as already described in previous chapters. In this section, we focus on implementation and optimization that is specialized for video processing. Many of these optimizations are harnessed through the use of special attribute names, for which Eureka provides special treatments and auxiliary libraries. Table 6.3 summarizes these reserved attributes.

Attr name	Type & Format	Semantics & Eureka Support
`parent`	`Ref<Interval>`	The upper-level interval from which the current interval is (temporally-spatially) cropped.
`rgb`	`RGBArray`	The decoded RGB data that is "tight and complete" with regard to the bounds. Together with `parent` enables late materialization.
`detection`	`JSON`	Labels, coordinates, and scores of multiple detected objects. Eureka provides operators for futher manipulation of these results (Table 6.4).
`trajectory`	`List<Interval>` (temporally ordered)	Eureka provides auxiliary functions and operators to interpolate, smooth, manipulate, and compare trajectories.
`v_decoder`	`LRUVideoDecoder`	Thread-safe wrapper of a video decoder and a LRU frame cache.

Table 6.3: Reserved Attribute Names with Specialized Optimization and Support

`Detection (string model_name)` A specialization of `Map` that runs an object detection model on every frame and stores the JSON-formatted result under the `.detection` attribute. The JSON result contains bounding boxes, object classes, and scores of multiple detected objects.
`DetectionFilter (string class_name, float score)` A specialization of `Filter`. It parses the `.detection` attribute of input, typically generated by the `Detection` operator, and only passes on frames with certain object classes detected.
`DetectionFlatten (string class_name, float score)` A specialization of both `Filter` and `Flatten`. Similar to `DetectionFilter`, it filters inputs by object classes. Moreover, it emits tight crops around the bounding boxes as individual outputs. Thus this is a 1-to-N operator.

Table 6.4: Operators Related to Object Detection

6.4.1 Maintaining the Stream Invariant

The interval stream model in Eureka maps natively to live video processing, as discovered events arrive in temporal order and may never end. Moreover, maintaining the stream assumption throughout the system allows for optimized implementation of certain operators.

For example, the `Join` operator, in theory, may produce the Cartesian product of its inputs in the worst case. Nonetheless, we observe that the more constrained `WindowedJoin` operator is sufficient in most use cases, as one typically wants to relate an event to other events that happen in proximate time. The `WindowedJoin` operator only considers input pairs that are within a small temporal window of each other. As Algorithm 1 shows, the stream assumption of the join inputs allows an efficient implementation that only needs to maintain a small input buffer, and releases obsolete buffered inputs as soon as new ones run out of their window.

Algorithm 1: Windowed Join of Two Interval Streams

```
Input: IntervalStream $leftIn$, IntervalStream $rightIn$
Input: int $window$, function $predicateFn$, function $mergeFn$
$leftBuffer \leftarrow List()$;
$rightBuffer \leftarrow List()$;
while True do
    // Get a new input from the left upstream
    $iL \leftarrow leftIn.getNext()$;
    for iR in rightRuffer do
        if $iR.t2 < iL.t1 - window$ then
            Remove $iR$ from $rightBuffer$ ;
        else if $iR.t1 < iL.t2 + window$ then
            if $predicateFn(iL, iR) = True$ then
                Output($mergeFn(iL, iR)$);
    $leftBuffer.append(iL)$;
    // Get a new input from the right upstream
    $iR = rightIn.getNext()$;
    ... // Mirror operations above
```

6.4.2 Exploiting Parallelism

Under the collection-of-frames model, it is straightforward to exploit *inter-frame parallelism* in processing video data, as the frames are treated just as independent image files. However, exploiting such parallelism is a challenge when we must account for the sequential dependency between the frames. Instead, Eureka uses following three levels of parallelism on a cloudlet.

Eureka exploits *inter-file/inter-stream parallelism*, by creating separate instances of the user query for each input file or live stream, and executes them in parallel. Nonetheless, a high degree of inter-stream parallelism may not be available, for example, when only a single camera is attached to a cloudlet. On the other hand, it may not be desirable, because a complicated query requires a lot of DRAM to run. In those cases, the other forms of parallelism are used.

Eureka exploits *inter-operator parallelism* within a query. This is done by running each operator in its own thread. Take Figure 6.8(a) as an example. Eureka creates a thread for each node (box) and a thread-safe queue for each edge in the query graph. In other words, it adopts the producer-consumer execution model. The operator threads run in parallel, possibly overlapping CPU-bound, GPU-bound, and memory-bound processing. Temporally-ordered interval streams flow through the queues. Queue sizes are bounded and queue pressure throttles the pace of individual operators. This implementation offers a large degree of parallelism to utilize many-core CPUs, while preventing the in-memory states from growing indefinitely.

Finally, Eureka exploits *intra-operator parallelism* offered by two major sources: (1) multi-threaded implementation of computer visions algorithms (e.g., those offered by OpenCV and OpenBLAS); (2) data-parallel versions of generic operators (e.g., `Map` $\rightarrow$ `ParallelMap`, `Filter` $\rightarrow$ `ParallelFilter`). These data-parallel operators typically use a thread pool to process multiple inputs simultaneously. In the specific context here, it can seen as a form of *inter-interval parallelism*.

6.4.3 Provenance and Late Materialization

Eureka queries often involves deriving visual data from the input, e.g., by (spatial-temporal) cropping. Such derivation may be recursive, e.g., by making crops inside a crop. Eureka tracks provenance of such derivations via a reserved attribute name `parent`. It allows an operator to access the upper-level element from which the crop is created, much like the ".." (double-dot) directory in Linux file system.

Tracking provenance enables *late materialization* of decoded visual data. When an operator makes a crop of a frame, the crop is *logical* — a new `Interval` object is created with new (reduced) `bounds` and its `parent` pointing to the source. The RGB array is not materialized immediately. Instead, the reserved attribute name `rgb` is replaced by a callback function that performs the actual cropping using its source's RGB data. The source itself may not have been materialized, either. In that case, callback and cropping is perform recursively towards upper-level data.

6.4.4 Video Decoder and LRU Frame Cache

During the execution of a query, many operators require decoding video frames from a file. Because of the high frame rate and high compression ratio of video, decoding and staging all frames in the DRAM is infeasible. Neither is pre-decoding them on disk, as discussed in Chapter 5. The only feasible path is to read compressed files from the disk, decode specific frames on demand, and release the memory as soon as they are no longer needed.

With this approach, there are still several concerns. First, the runtime cost of video decoding is high. Second, for a large query graph, it requires a lot of DRAM to hold "on-the-fly" decoded frames that awaits to be processed by downstream operators. Third, video decoders are stateful and have tape-like performance: decoding forward in time is efficient, while winding-back or random seek is costly.

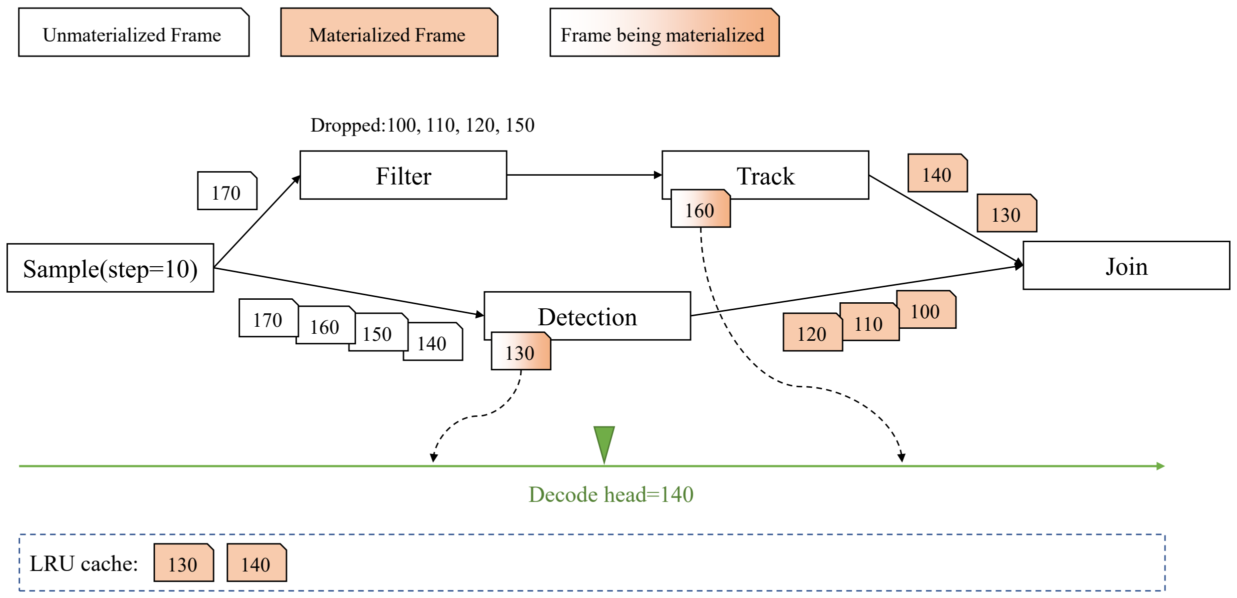


Figure 6.12: LRU Frame Cache Reduces Redundant Decoding and "Decoder Seeks"

We address these issues by sharing a single video decoder between multiple operators. Furthermore, we introduce an LRU frame cache. Figure 6.12 visualizes a snapshot of the system state during the execution of a simple query. The `Track` operator and the `Detection` operator are trying to decode frame 130 and 160, respectively, while the decode head is at frame 140. Without the frame cache, an expensive "wind-back" is necessary and frame 130 needs to be decoded again. With the frame cache, cached frame 130 can be returned to `Detection` without moving the decode head. Ditto for the subsequent frame 140. The frame cache not only reduces redundant decoding of the same frame, but also reduces DRAM used by "on-the-fly" intervals by passing cached frames by reference, instead of by value.

The above optimization is encapsulated in a class `LRUVideoDecoder`. An "unmaterialized" interval has its `v_decoder` attribute pointing to an `LRUVideoDecoder`, and its `(t1, t2)` indicating its time in the video. Again, the `rgb` attribute in this case is a callback function that materializes the frame's RGB data on demand.

6.5 Evaluation

6.5.1 Metrics

Before presenting our evaluation results, we first introduce the evaluation metrics. We focus on *event discovery from open-ended video*. We consider the following scenario: the system being evaluated, regardless which method is used, processes the raw video data and returns a variable number of video segments, each of variable length; a domain expert goes through these video segments and labels all event instances found in them. As a result, the expert's labeling effort is measured by *the total time length of the returned video segments* (= number of frames / frame rate). Essentially, this measures the time (hours) to play back all video segments at 1.0x speed

Data set	Size	Description
VIRAT [109]	8.6 hours, 329 files	Captured from real-life public surveillance cameras at parking lots, school campus, streets, etc.
Okutama [13]	0.55 hours, 33 files	Captured from drones. Aerial view of actors performing scripted scenes in a playground.

Table 6.5: Data Set Used in Evaluation

and watch them. Furthermore, as we focus on detecting temporal events, we should count the number of *event instances*, rather than the number of video frames. Some instances may last for longer duration, but do not necessarily offer more value from the perspective of discovering events. This is significantly different from classification of static images. It is also different from video classification in data sets such as HMDB51 [88] and UCF101 [138]. In those data sets, video clips are universally trimmed to 2–3 seconds and a label is assigned to the whole clip. In that case, an expert's labeling effort can be simply counted by the number of video clips. By contrast, in our setting, a clip's length may range between seconds to minutes, demanding different amount of expert attention.

Our most important metrics is an expert's *productivity*:

$$\text{Productivity [\#/hr]} = \frac{\text{\# Event instances discovered}}{\text{Total length of returned video}}$$

The above definition of productivity makes two simplifying assumptions. First, we assume system performance is not the bottleneck, so that we can focus on evaluating the expressive power of our approach. Second, we ignore confounding factors that may affect an expert's labeling time in a more realistic setting. For example, certain events may require an expert to wind back and re-watch the video in order to confirm her opinion. This is more likely to happen when the target event has subtle details or the camera scene is crowded. An expert may also choose to play back video clips at faster or slower speed, while still being able to correctly label events. In this chapter, we ignore such task-dependent and content-dependent complications and assume that labeling time is a flat rate in terms of total video length in its recording speed.

In case the ground truth (i.e., *all* event instances in the raw data) is available, it also allows us to calculate the *event recall*:

$$\text{Event recall [\%]} = \frac{\text{\# Event instances discovered}}{\text{\# Ground Truth Event instances}}$$

Note that the canonical concept of *precision* is ill-defined here. Imagine the system returns a 1-minute "negative" clip and a 1-hour "negative clip, " both of which do not include any event. It would be unfair to simply count them as "2 negative examples." The different amount of time it takes an expert to label them needs to be accounted. Our productivity metric serves this purpose.

6.5.2 Data Sets and Tasks

We evaluate using the VIRAT [109] and Okutama [13] video data sets. Table 6.5 summarizes their characteristics. They are two of the very few public data sets that can be considered as open-

Data set	Event (abbreviation)	Frame-based early-discard heuristics	Event-based early-discard heuristics
VIRAT	Person getting out of a vehicle (get-out)	Presence of person and vehicle	A person's trajectory *emerges* from a *stopped* vehicle
VIRAT	Person getting into a vehicle (get-in)	Presence of person and vehicle	A person's trajectory *disappears* into a *stopped* vehicle
VIRAT	Person loading or unloading objects into/from a vehicle (loading)	Presence of person and vehicle	A person stays still next to a vehicle for > 2 seconds
VIRAT	Person carrying bag (carry-bag)	Presence of person	A person attached to a bag for > 2 seconds.
Okutama	Person pushing or pulling object (pushing)	Presence of person and non-person object	A person attached to a non-person object and move with it for > 2 seconds
Okutama	Person shaking hands (hand-shake)	Presence of ≥ 2 persons	Two persons' trajectories "meet" for > 2 seconds

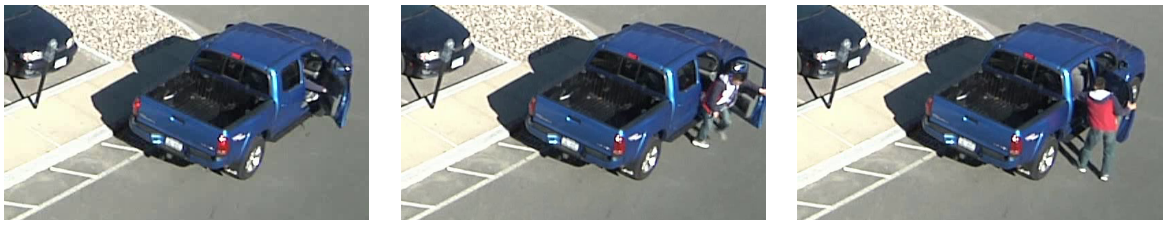
Table 6.6: Search Targets and Conditions in Evaluation

ended and have annotated multi-object events. That is, unlike HMDB51 [88] and UCF101 [138], video files in these data sets are *not trimmed* to singleton actions. Instead, multiple actions can be happening simultaneously in a scene, while there are also "boring" non-event periods in the long video streams. We choose 6 person-object and person-person events as our search targets, for which the data sets provide ground truth annotations. Table 6.6 lists these targets, as well as the frame-based early-discard heuristics (as discussed in previous chapters) and the event-based early-discard heuristics (as discussed in this chapter) used to search for the targets. These heuristics conditions, such as the choice of constants like 2 seconds, are based on human intuition about these actions in real life. Figure 6.13 shows example frame sequences of each event discovered in our evaluation.

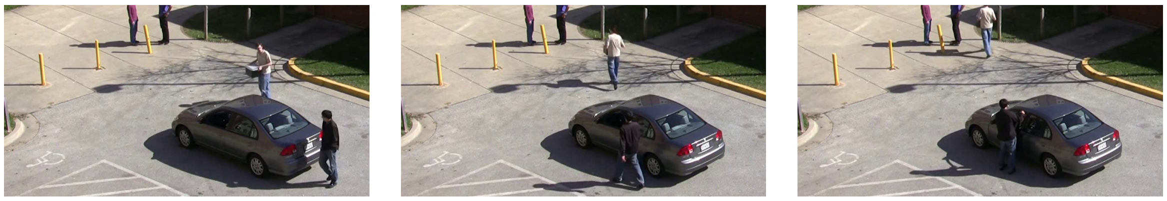
6.5.3 Result

Using the 6 search tasks from Table 6.6, we evaluate the effectiveness of the following labeling methods:

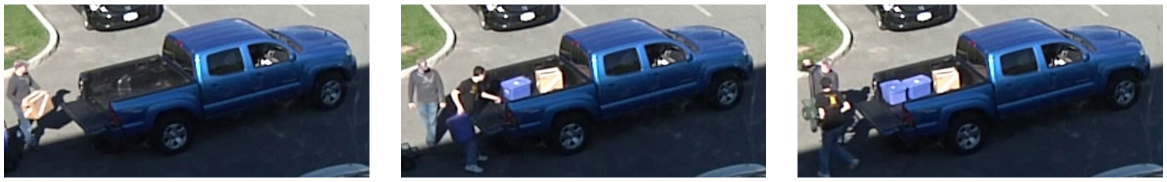
- Brute force: An expert goes through all video data and labels all event instances in it, without the use of early discard. This result is calculated based on ground truth (GT) annotations — after all, that is how the GT was created in the first place. By definition, the expert will discover all events, effectively achieving 100% event recall. However, it comes at the cost of plowing through hours of video data.



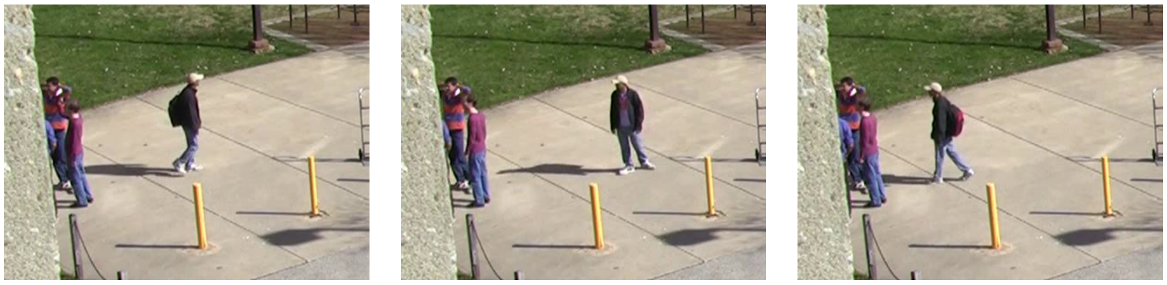
(a) VIRAT Get-Out Example



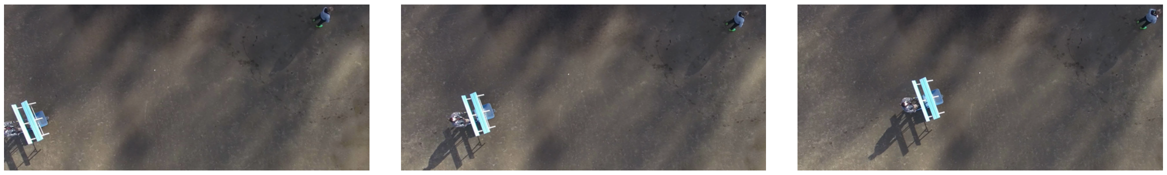
(b) VIRAT Get-In Example



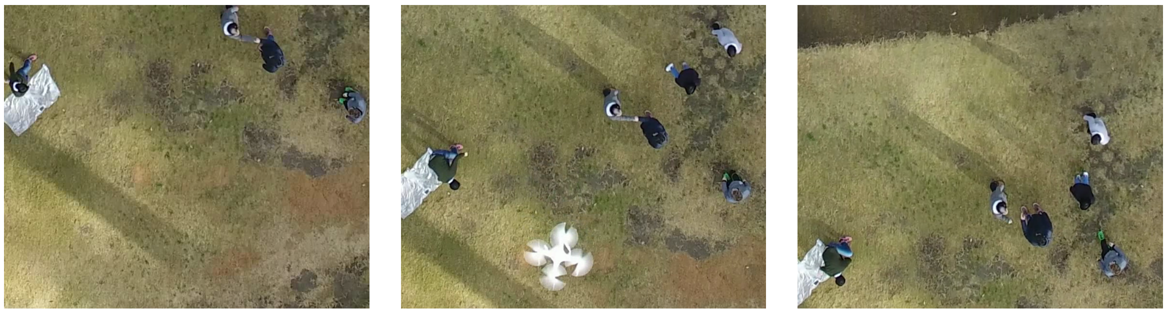
(c) VIRAT Loading Example



(d) VIRAT Carry-Bag Example



(e) Okutama Pushing Example



(f) Okutama Handshake Example

Figure 6.13: Event Examples Discovered in Our Evaluation

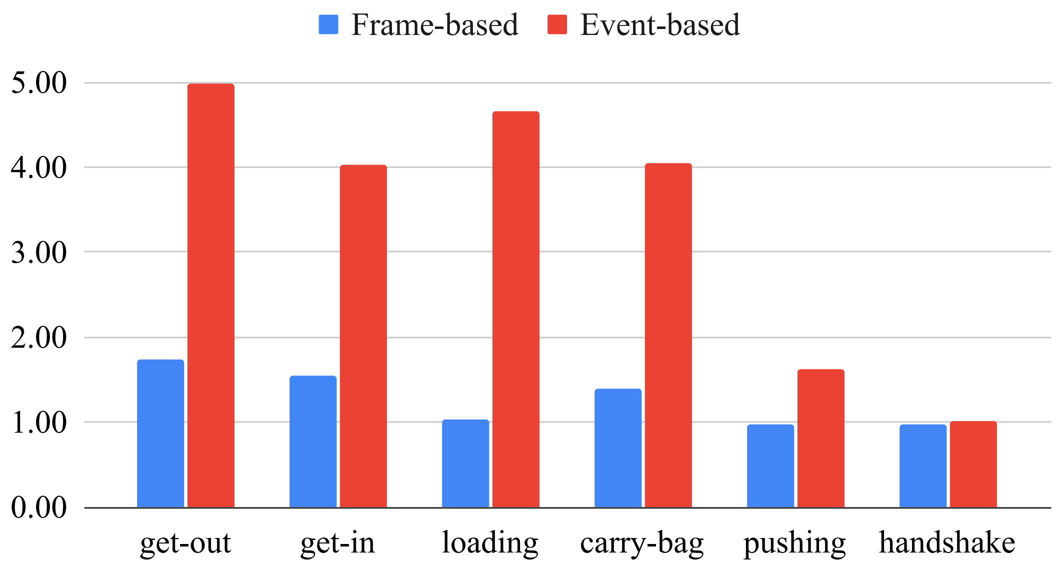


Figure 6.14: Productivity Improvement Factor over Brute-force Labeling

- Eureka (frame-based): The expert uses a frame-based early-discard approach to filter video frames independently. Consecutive undiscarded frames are encoded back to video clips and returned to the expert. The frame-based early-discard conditions used are given in Table 6.6 column 3.
- Eureka (event-based): The expert uses an event-based early-discard approach as described in this chapter. The final output spatial-temporal intervals are materialized as video files and returned to the expert. The event-based early-discard conditions used are given in Table 6.6 column 4.

Figure 6.14 summarizes the improvement factors over brute force labeling of the two early-discard approaches. Tables 6.7–6.12 report the details of each task. Note that the “Productivity” column of the “Brute force” row in these tables may also be re-interpreted as the “base rate” of an event — it measures how many event instances can be found per hour of raw data. We see that our search targets have a diverse range of base rate of 9 – 242 instances per hour.

The VIRAT data set includes long surveillance recordings from real-life scenes. Therefore, there are often “boring” periods when the objects involved in an event are not present in a frame, for example, when nobody is in a parking lot. As a result, frame-based early-discard based on presence of these objects is able to improve productivity *modestly* ($\leq$1.8x, Figure 6.14) by dropping those non-event frames. Nonetheless, as discussed in the beginning of this chapter, those frame-based heuristics tend to be under-specifying. For example, a pedestrian walking by a vehicle may be returned under the frame-based approach for the get-out event. The event-based approach, by contrast, is more descriptive and is able to eliminate those candidates, resulting in up to 5.0x productivity improvement. In either case, imperfection of DNN-based object detection, especially *false negatives* (i.e., failure to detect an object in a scene), causes the system to miss events, therefore reducing their event recall. The tradeoff, however, is effective in terms of improving an expert’s productivity of discovering event instances.

	Labeling effort [hr]	Discovered events [#]	Event recall [#/GT]	Productivity [#/hr]
Brute force	8.60	97	100%	11
Eureka (frame-based)	3.86	76	78%	20
Eureka (event-based)	1.05	59	61%	56

Ground truth events (GT) = 97; Base rate = 11/hr

Table 6.7: Event Discovery VIRAT: Person Getting Out of Vehicle (get-out)

	Labeling effort [hr]	Discovered events [#]	Event recall [#/GT]	Productivity [#/hr]
Brute force	8.60	111	100%	13
Eureka (frame-based)	3.86	77	69%	20
Eureka (event-based)	1.06	55	50%	52

Ground truth events (GT) = 111; Base rate = 13/hr

Table 6.8: Event Discovery VIRAT: Person Getting into Vehicle (get-in)

For the get-out and get-in events, the event-based predicates are based on the start/end point of a person's trajectory. These require object tracking. Because tracking is not perfect, it causes an additional 17% (get-out, Table 6.7) and 19% (get-in, Table 6.8) drop of event recall, compared to frame-based. Interestingly, we observe that some of those missed instances are discovered by the frame-based approach, because a person irrelevant to the event appears in the scene at the same time. Nonetheless, event-based is effective in most cases and thus is more than 2x productive than frame-based.

For the loading event, the object being loaded/unloaded is often occluded by the person or the vehicle. Therefore, we exclude it from our predicate. Likewise, for the carry-bag event, the bag being carried is often so small that it cannot be detected on the frame level without using the "content-based hierarchical detection" idiom introduced in Section 6.3. Therefore, with frame-based early-discard, we only predicate on the presence of persons. Unsurprisingly, this leads to its high event recall but low productivity (Table 6.10).

The last two tasks on the Okutama data set (pushing and handshake) is more difficult than VIRAT for two reasons. First, because the content of the videos is performed by actors according to a script, there are much less "boring" moment than in real-life video. In other words, there are almost always some actions happening in the video. This can be seen from the Okutama events' relatively high "base rate" (Table 6.11 242/hour for pushing; Table 6.12 128/hour for handshake). Second, the videos are captured from a drone that flies at an altitude and performs abrupt turnings. The aerial view of objects and the rapid camera motion makes them very difficult to detect and track even with state-of-the-art algorithms (Figure 6.13(e) and (f)). With frame-based early-discard, the false negatives issue offsets the benefit gained from effective pruning of non-event frames, resulting in little productivity improvement — in fact, it hurts productivity slightly. Nonetheless, the event-based approach is able to achieve 1.6x improvement for the pushing event, despite suffering from the same issues.

The results above are measured by writing Eureka queries that embody the heuristics shown in Table 6.6. These queries are deterministic and thus the results can be seen as the effect achieved by one expert. For broader evaluation in the future, it would be valuable to let multiple experts perform the same search task and possibly describe the same event in different ways. This will allow us to gauge Eureka's applicability and benefit at a larger scale.

	Labeling effort [hr]	Discovered events [#]	Event recall [#/GT]	Productivity [#/hr]
Brute force	8.60	80	100%	9
Eureka (frame-based)	3.86	37	46%	10
Eureka (event-based)	0.76	33	41%	43

Ground truth events (GT) = 80; Base rate = 9/hr

Table 6.9: Event Discovery VIRAT: Person Loading/Unloading Object to/from Vehicle (loading)

	Labeling effort [hr]	Discovered events [#]	Event recall [#/GT]	Productivity [#/hr]
Brute force	5.80	822	100%	141
Eureka (frame-based)	4.08	809	98%	198
Eureka (interval-based)	0.97	556	68%	573

Ground truth events (GT) = 822; Base rate = 141/hr

Table 6.10: Event Discovery VIRAT: Person Carrying Bag (carry-bag)

	Labeling effort [hr]	Discovered events [#]	Event recall [#/GT]	Productivity [#/hr]
Brute force	0.55	134	100%	242
Eureka (frame-based)	0.43	100	75%	234
Eureka (event-based)	0.20	77	57%	395

Ground truth events (GT) = 134; Base rate = 242/hr

Table 6.11: Event Discovery in Okutama: Person Pushing An Object (pushing)

	Labeling effort [hr]	Discovered events [#]	Event recall [#/GT]	Productivity [#/hr]
Brute force	0.55	71	100%	128
Eureka (frame-based)	0.39	48	68%	124
Eureka (event-based)	0.37	48	68%	130

Ground truth events (GT) = 71; Base rate = 128/hr

Table 6.12: Event Discovery in Okutama: Person Shaking Hands (handshake)

6.6 Discussion

In the earlier chapters, we described Eureka's discard-based methodology that lets a domain expert discover rare objects based on human knowledge. In this chapter, we extended that approach to discovering events from video data. The primary goal of Eureka is to let a domain expert effectively express her knowledge of an event and let the system compute the heuristics efficiently. With event discovery, the challenge is to support the expression of temporal-spatial constraints of events. Our interval-based abstraction provides a tool for an expert to convert an intuitive idea in mind into a computable program

In all examples above, the lowest-level events are location of an object in a single frame. The term "object" here refers to entities such as persons, vehicles, suitcases, chairs, dogs. The methods introduced in this chapter are suitable for describing events in terms of *coarse-grained inter-object relationship.* The relationship can be spatial ("is A located to the left of B?"), temporal ("did A happen before B?"), or visual ("how many SIFT key points of A match with B?"). Such relationships can be specified recursively and hierarchically. Nonetheless, there are some limitations to this approach.

Eureka $\neq$ Activity Recognition. Certain events are defined by *intra-object states* rather than *inter-object relationships*, for example, whether person is clapping or waving her hands. An object detector can only output the location of the person without knowing her gesture. Furthermore, some inter-object relationships are difficult to compute. Consider the "get-out" query described in Figure 6.8. This search predicate, in fact, does not distinguish between "person getting out of a vehicle from driver seat" (get-out) and "person crawling out from under a vehicle" (crawl-out), because both satisfy the condition that a person's trajectory "emerges" from a vehicle. Making this subtle distinction requires examining finer-grained intra-object features. Those kinds of tasks are more suitable to be solved by standard activity recognition methods which learn sophisticated mathematical functions from large amount of training examples. Eureka is not intended to replace those methods. Instead, Eureka's goal is to accelerate the process of finding those training examples. Although the query in Figure 6.8 does not differentiate get-out from crawl-out, it is able to significantly improve an expert's productivity in finding get-out examples in hours long video data.

Eureka $\neq$ Deep Learning System. Eureka's implementation (Section 6.4) is a form of *data flow graph* where each node represents a unit of computation and each edge represents a producer-consumer data flow. This model is widely adopted by deep learning systems such as TensorFlow [2]. However, those systems mainly focus on mathematical operations on numeric matrices and do not have explicit notions of event, time, space, etc. By contrast, Eureka must deal with hybrid forms of data, including JSON-style metadata, RGB arrays, and feature vectors. On the other hand, deep learning systems usually provide important functions for learning, such as automatic differentiation and back-propagation, which are beyond the scope of Eureka.

Eureka $\rightarrow$ Declarative Video Query Language. This chapter represents an important step towards the vision of a declarative video query language. With this vision, a user can express a search target via (nearly) natural language (e.g., "a person on top of a bike"), rather than writing low-level, imperative code that computes over pixel values. Eureka advances state-of-the-art by encapsulating low-level computer vision code into operators and providing relational operators on time, space, and features. This makes it possible to analyze video data without taking a PhD in computer vision. Nonetheless, the domain expert still needs to have some basic understanding of computer vision in order to use Eureka effectively. One such example is to figure out the content-based hierarchical detection improvement presented in Section 6.3.3, by knowing the small-object issue faced by DNN-based object detectors. Moving forward, it will be highly valuable if the system can employ this optimization *without* user guidance. However, it is a research challenge to automatically detect and materialize such opportunities, given the stochastic and content-dependent nature of computer vision algorithms. We leave this exciting direction to future work.

6.7 Related Work

Edge-based Video Analytics Systems

There is growing interest recently in building edge-based video analytics systems [19, 65, 77, 78, 81, 82, 154, 155]. This interest is driven by deep learning's success in image analysis and increasingly pervasive deployment of video cameras in the forms of surveillance, autonomous cars, drones, smart phones, and wearable devices. These two converging factors enable valuable applications such as wearable cognitive assistance and urban traffic analysis. DNNs are accurate, but demand a lot of compute resources to run. It is particularly challenging to execute DNNs at full frame rate in edge computing environments such as drones, smart glasses, and multi-tenant edge nodes. Other resources, such as 4G/5G wireless bandwidth, may also limit the transmission of frames for DNN inference on a cloudlet.

As elaborated in Section 6.1, this body of work focuses on exploiting certain aspects of video data to reduce *frame-based* analytics cost, and thus can be classified as L1 in our 4-level taxonomy (Figure 6.2). Modeling events across the frame boundary has not been studied in this context.

Spatial Databases

Spatial databases are a class of database systems that specialize in managing and processing spatial data and spatial relationships [50]. Traditionally, it mainly refers to geographical data obtained from GPS sensors. Obtaining this data is considered to be reliable and inexpensive. Thus, the main focus has been developing algorithms to efficiently compute certain spatial relationships, such as detecting "stay points" and "flock" patterns. Zheng [166] gives a comprehensive survey of related work in the context of trajectory data mining.

Eureka focuses on information extracted from analyzing image and video. In this setting, "spatial" refers to the screen space rather than geo-location in the physical world. Closest to our work is the Rekall system [46], which describes a set of spatial and temporal operators that extend from Allen's interval operations [7]. However, Rekall only deals with operations on spatial-temporal "metadata" that is extracted outside the system. The extraction of this data relies on compute-intensive deep learning methods, which consumes most of the compute time in the whole process. Besides, the separation of space-time processing and content processing makes it impossible to implement all of the idioms we introduce in Section 6.3. To the best of our knowledge, Eureka is the first system to integrate both in a unified framework.

Activity Recognition

In computer vision, *activity recognition* is one of the most considered problems where examining a sequence of video frames is indispensable. Indeed, it can be classified as L2 analytics according to our 4-level taxonomy (Section 6.1).

Some prior work shares our idea of using *object detection* on individual frames as a preliminary step. Zhu et al [167] use techniques pre-dating deep learning to detect persons and vehicles (e.g., part based models [42]) in VIRAT; and then extract histogram of oriented gradients (HOG) and histogram of optical flow (HOF) features from the relevant regions, which are later used to learn a classifier. Gleason et al [48] follow a similar methodology but with more advanced building blocks. They use Faster R-CNN to detect objects of interest in frames. Based the detections, they generate multiple spatial-temporal region proposals which are then classified using a DNN. Another body for work called *skeleton-based* activity recognition focuses on learning classifiers on top of positions of human body joints [164]. These joint positions may be annotated or computed by human pose estimation algorithms such as OpenPose [20]. This class of methods are suitable for distinguishing actions that are solely dependent on a person's posture, such as sitting versus standing. However, it may ignore useful contextual information for predicting an activity, for example, the presence of a basketball for "basketball dunk." More recently, there is growing interest in creating DNNs that can be trained end-to-end directly from RGB pixel data [47, 75, 148] without an intermediate step of object detection or human pose estimation. These methods can learn human and non-human information from the scene, but may suffer from noise and clutter in the background.

Nevertheless, all the above solutions are considered in the standard setting of *supervised learning*. Typically, a model needs to be trained on at least tens to hundreds of example clips of the actions. Creating labeled data sets of activities is extremely expensive, given the large (almost infinite) vocabulary of objects and interactions between them. For example, it took the VIRAT data set's authors [109] at least four years to annotate the data and the authors had to seek additional funding in order to complete the project. Eureka can be seen as a tool to accelerate this process of curating training data for activity recognition.

In addition to the requirement of training data, Eureka is complementary to activity recognition in two aspects. First, Eureka focuses on events defined by *coarse-grained inter-object relationships*, while activity recognition focuses on activities defined by *fine-grained intra-object features*. For example, determining whether a person is clapping or waving hands requires examining her fine-grained limbs configuration. Second, some activity recognition solutions only work on video that is *trimmed both to space and to time* of the actions. In popular data sets such as HMDB51 [88] and UCF101 [138], all video clips are trimmed to 2–3 seconds long and there is only one person performing one activity in a clip. When applying those methods to open-ended video such as VIRAT, one first needs to find out *when and where* to look. Eureka can be used to bridge this gap: it finds the space-time span of probable events based on coarse-grained analysis, makes the 2-second single-person crops, and then passes those cropped segments to the aforementioned activity recognition algorithms.

Part-based Models

The idea of specifying an object detector as a set of geometric relationships between parts dates back to Fischler and Elschlager's pictorial structures [43], with the deformable part models (DPM) [42] being a more modern realization. It can be seen as a top-down approach where the parts are detected by geometric templates. This methodology was later muted by the outstanding performance achieved by deep learning that learns visual features bottom-up. In a way, Eureka can be seen as a system that nicely combines top-down relationships and bottom-up learned DNNs to detect the parts. Besides, part-based models only focus on detecting objects, not events in video.

Zero-shot Learning and Compositional Models

Eureka lets an expert discover events that do not have large-scale training sets. In computer vision, this is typically considered as the problem of *zero-shot learning* (ZSL). Here is a classic ZSL problem: given that there is no example of "black swan" in the training set, can we build a model that is able to detect black swans in new data?

An important class of ZSL methods to address this problem is *compositional model* [90, 96, 103]. Composition models treat the novel target "black swan" as a *composition* of two atomic *attributes* — "black" and "swan." They further assume that those attributes exist in the training data, separately. For example, there are training examples of "black sheep" and "white swan." Then, compositional models are able to learn from the training data: (1) to distill classifiers of the atomic attributes; (2) to compose these attributes into a compound target organically. In other words, although the training set might only have "black sheep" and "white swan," the compositional model is able to detect "black swan."

In a similar way, a novel target "monkey riding horse" may be seen as a composition of three attributes — "monkey," "riding," and "horse." It can be detected by a compositional model if the training set has the following labels: "monkey riding bike," "monkey eating banana," and "person riding horse."

Although compositional models are powerful in some cases, they are highly constrained in two senses:

First, the form of composition is constrained. In the above example, "black swan" is in the form of "Adjective-Noun" (A-N) composition; while "monkey riding horse" is the form of "Subject-Verb-Object" (S-V-O) composition. A model that is trained on one form: (a) cannot be applied to detect targets in another form; (b) requires training set that is *labeled in the same form*. To train an A-N compositional model, it requires training labels such as "black sheep," "white swan," "gray rhino," etc. This data cannot be used to train an S-V-O model. To do so, one has to carrying out additional work to collect data and label it in the S-V-O format, which in turn can be very costly.

For real-life events, the form of composition can be quite complicated, such as "Adjective_1-Subject-Verb-Adjective_2-Object." Creating data sets for every possible way of composition is infeasible. By contrast, Eureka offers a way to specify events hierarchically and recursively. Composition in Eureka, at least in principle, can be open-ended and infinite.

Second, the attribute vocabulary is constrained. That is, the attributes used in the composition are limited by the model's vocabulary. For example, an A-N compositional model is unable to detect "cyan swan" if there is no cyan things in the training set. Likewise, a S-V-O model is unable to detect "monkey pushing suitcase" if there is no "pushing" examples in the training set. In other words, although composition models tolerate the lack of examples of a compound target, they do not tolerate the lack of examples of all attributes. When these examples are not available, we see Eureka's approach of letting an expert conduct early-discard based on intuition as the only path forward.

Finally, most recent work on ZSL focus on classifying static image data. We are not aware of any ZSL techniques for temporal event detection in video data.

Chapter 7

Conclusion and Future Work

This dissertation addresses the problem of *human-efficient discovery of training data for visual machine learning*. It tackles a major hindrance to common adoption of deep learning in specialized domains. We propose Eureka, a system that supports interactive search of scarce targets in a large volume of unindexed visual data. We describe system architecture, programming abstraction, modelling of human productivity, and various optimizations that improve an expert's efficiency in discovering training examples. We show Eureka is effective and efficient for discovering rare phenomena from both static image data and temporal video data. In this chapter, we summarize the contributions made this dissertation and outline future directions.

7.1 Contributions

In this dissertation, we claim that

The manual effort of discovering a large training set for visual machine learning can be reduced by a system combining: (a) efficient early discard made possible by edge computing; (b) just-in-time machine learning; and (c) the ability to create more accurate filters immediately without writing new code. This approach is effective for different compute/storage architectures and different vision tasks.

We validated this thesis statement from multiple aspects. We started by recognizing the unique challenges faced by domain experts that interesting phenomena in those domains are often scarce and crowd-sourced labeling is not feasible. We designed Eureka to support *human-efficient* interactive discovery of scarce targets by an expert. Eureka treats the expert's attention as a critical resource in the system. Eureka is built on OpenDiamond's [69] idea of early-discard, but extends it with iterative just-in-time machine learning. Using the task of object detection in images, we demonstrated this approach is able improve an expert's productivity progressively.

Based on image analysis, we developed a model of the Use:System Match — a metric for gauging human efficiency. Our model depends on a wide range of factors pertaining to the user, the system resources, the data, and the search target. Among others, it highlights the importance of Eureka's system efficiency in reducing an expert's wait time.

We studied methods to improve Eureka's system efficiency. Eureka utilizes edge computing to execute early-discard filters on cloudlets with high-bandwidth access to data sources, and thus avoids transmitting large amounts of visual data across the WAN. Nonetheless, *elasticity* of a cloudlet on the edge is limited compared to their counterparts in the cloud. We further identified that decoding is a scalability bottleneck in Eureka workload. To address this bottleneck, we proposed a novel storage architecture on the edge which encompasses hardware accelerators for decoding in disks and high-speed NVMe interconnect for transmitting decoded data.

Finally, we demonstrated that Eureka is not only effective for discovering objects in static images, but also effective for discovering temporal events in videos. We described extensions of Eureka for specifying spatial-temporal relationships and multi-object relationships. In addition, we described system optimizations specific to expensive video processing. We showed that Eureka enables human-efficient discovery of multi-object events with little coding effort.

7.2 Future Directions

7.2.1 Integrating Labeling, Learning, and Inference

In this dissertation we validated the effectiveness of an iterative discovery workflow, where an expert frequently repeats labeling, (re-)training, and inference with new classifiers. Nonetheless, in the current implementation, labeling and learning is performed "out-of-band" via a separate tool chain. For example, to train a new object detector, the user needs to: (1) download the positive and negative examples from Eureka; (2) upload them to the training directory, either to the local machine or to a remote service such as OpenTPOD [152]; (3) configure several training parameters and launch the training process, again, either locally or remotely; (4) download the trained model and upload it to Eureka for new queries.

Integrating labeling and learning within the system can offer additional usability and efficiency benefits. It can lower user friction by reducing moving parts of the system, ensuring compatibility of APIs, and simplifying deployment. It may also overlap model training, user labeling, and inference time on the edge, which further improves user productivity.

Training DNNs involves choosing a set of hyper-parameters (e.g., batch size, number of epochs, learning rate) which we currently leave to the user's decision. Although many third-party training services automate this to some extent, the optimal choice of hyper-parameters are often task-dependent. Hence, the expert needs to at least know a set of machine learning rule-of-thumbs. Integrating labeling and learning into Eureka takes off this burden from the expert. Moreover, it can expose run-time statistics (e.g., inference speed, accuracy, AUC) to the training component so that it can make better-informed decisions about configuring the hyper-parameters. How that information should be used to derive effective training process needs to be explored.

7.2.2 Adaptive Workload Sharing Between Edge and Cloud

So far, we have assumed the expert interacts with the Eureka backends on a computer that is quite wimpy and only sufficient for running simple GUI programs (aka a think client). As a result, we push all the heavy weight computation towards the edge, which both harvests parallel edge-based computation and reduces data traffic across the WAN. In a broader setting, an expert may have access to a public or private cloud which can share the load from the edge; the edge may be hierarchical, with some cloudlets being slightly "farther away" from the data sources but still having higher bandwidth than the cloud. At first glance, offloading to the cloud or deeper cloudlets seems to be defeating the purpose of edge computing. However, edge infrastructure is typically shared by multiple tenants and multiple applications. A transient surge of demand for edge resource may occur, for example, due to a flash crowd of mobile VR gamers. In that case, we may forego a certain degree of WAN frugality and move part of the processing pipeline into the cloud. Hopefully, the *filter* and *operator* concepts introduced in this dissertation provide a starting point of structuring a workload into offload-able units. More work needs to be done in monitoring cloudlet resources, identifying profitable offload options, and developing work sharing protocols and fault tolerance mechanisms.

7.2.3 Advanced Computer Vision for Video Analysis

Although deep learning has proven tremendous success on static image analysis, it is not until very recently that applying it on video understanding starts to gain traction. To date, state-of-the-art video analysis methods are still very compute-intensive and lagging behind image analysis in terms of quality. For example, one of the most recent methods for *multi-object tracking* [157] reports 65% accuracy and runs at near real-time (30 FPS) on a single video stream, while image classification DNNs easily achieve human-level accuracy and super real-time speed. Although one can usually gain better speed by sacrificing accuracy, that tradeoff may not always be desirable. There is still large room for pushing the Pareto frontier of the speed-accuracy space (i.e., improving one without sacrificing another). In addition to more advanced algorithms, specialized hardware for certain tasks (e.g., multi-object tracking, human pose recognition) is also an attractive path, although it requires significant incentive to drive its development.

7.2.4 User Study with Domain Experts

The experiments in this dissertation were conducted by the author and other graduate students, and used data sets curated by the computer vision community. This allows us to conduct reproducible experiments and evaluate metrics such as recall that otherwise cannot be measured without ground truth annotations. Throughout, we chose low base-rate targets for which there do not exist off-the-shelf models. In this regard, it has the same attributes as we envision in Eureka's use cases by domain experts.

It would be valuable to invite experts from specialized domains (e.g., medical, military) to use Eureka for discovering domain-specific targets. This may confirm our observation or offer new insights. Domain experts may have different user habits; domain-specific targets may have novel visual characteristics that we have not yet accounted for. These can help us significantly improve Eureka's usability and effectiveness towards its real-world deployment.

7.2.5 Eureka in Non-Visual Domains

The value of Eureka hinges on two fundamental premises: (1) the use of *supervised learning* that requires a significant amount of accurately annotated examples to train effective machine learning models; (2) the demand for *domain expert knowledge* to label a rare phenomenon, where the expertise is generally not available in the crowd. While this dissertation has focused on visual data (image and video), these premises generalize to non-visual domains. Recently, there has been success in applying (supervised) deep learning to areas beyond computer vision, such as natural language processing and time series analysis. Thus it will be interesting to extend Eureka's approach to these areas as well. Interesting questions include: identifying tasks that require domain expertise to label; designing proper ways for an expert to express heuristics with those data types; quantifying the benefit of Eureka in those domains.

Bibliography

[1] NVM Express. `https://nvmexpress.org/`. 5.2.4

[2] TensorFlow. `https://www.tensorflow.org/`, 2019. 1.2, 4.5, 6.6

[3] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, 1998. 5.2.4, 5.3

[4] Harsh Agrawal, Clint Solomon Mathialagan, Yash Goyal, Neelima Chavali, Prakriti Banik, Akrit Mohapatra, Ahmed Osman, and Dhruv Batra. Cloudcv: Large-scale distributed computer vision as a cloud service. In *Mobile cloud visual media computing*. Springer, 2015. 5.3

[5] Jahanzeb Ahmad, Kamran Raza, Mansoor Ebrahim, and Umar Talha. FPGA Based Implementation of Baseline JPEG Decoder. In *Proceedings of the 7th International Conference on Frontiers of Information Technology*, 2009. URL `http://doi.acm.org/10.1145/1838002.1838035`. 5.2.4, 5.2.5, 5.3

[6] Akamai. Q1 2017 state of the Internet / connectivity report. 2017. 2.1, 5.1

[7] James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983. 6.7

[8] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic Function Placement for Data-Intensive Cluster Computing. In *Proceedings of USENIX Annual Technical Conference*, 2000. 5.3

[9] R. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. Culler, J. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *Proceedings of Input/Output for Parallel and Distributed Systems*, 1999. 5.3

[10] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of ACM SIG on Management of Data*, 2000. 5.3

[11] Victor Bahl. Emergence of micro datacenter (cloudlets/edges) for mobile computing. *Microsoft Devices & Networking Summit 2015*, 2015. 2.1

[12] Rajesh Balan, Jason Flinn, Mahadev Satyanarayanan, Shafeeq Sinnamohideen, and Hen-I Yang. The Case for Cyber Foraging. In *Proceedings of the 10th ACM SIGOPS European Workshop*, 2002. 5

[13] Mohammadamin Barekatain and et al. Okutama-action: An aerial view video dataset for concurrent human action detection. In *IEEE Computer Vision and Pattern Recognition*

Workshops, pages 2153–2160, 2017. 6.5.2, 6.5.2

[14] David Barrett. One surveillance camera for every 11 people in Britain, says CCTV survey. *Daily Telegraph*, July 10, 2013. Last accessed: December 2019. 5

[15] Simona Boboila, Youngjae Kim, Sudharshan S. Vazhkudai, Peter Desnoyers, and Galen M. Shipman. Active Flash: Out-of-core Data Analytics on Flash Storage. In *Proceedings of the 28th IEEE Mass Storage Symposium*, 2012. 5.3

[16] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012. 2.1

[17] H. Boral and D.J. DeWitt. Database machines: An idea whose time has passed? In *International Workshop on Database Machines*, September 1983. 5.3

[18] John S Bucy, Jiri Schindler, Steven W Schlosser, and Gregory R Ganger. The DiskSim simulation environment version 4.0 reference manual, 2008. 5.2.5

[19] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G. Andersen, Michael Kaminsky, and Subramanya R. Dulloor. Scaling video analytics on constrained edge nodes. In *SysML*, 2019. 2.1, 5.2.4, 5.3, 6.1, 6.7

[20] Z. Cao, G. Hidalgo Martinez, T. Simon, S. Wei, and Y. A. Sheikh. Openpose: Realtime multi-person 2d pose estimation using part affinity fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019. 6.7

[21] Changjian Gao and Shih-Lien Lu. Novel fpga based haar classifier face detection algorithm acceleration. In *2008 International Conference on Field Programmable Logic and Applications*, 2008. doi: 10.1109/FPL.2008.4629966. 5.2.4, 5.3

[22] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015. 1.2

[23] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices. In *Proceedings of ACM SenSys*, 2015. 1.2

[24] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015. 1.2

[25] Zhuo Chen. *An Application Framework for Wearable Cognitive Assistance*. PhD thesis, Intel, 2016. 2.1

[26] Benjamin Y Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro. XSD: Accelerating mapreduce by harnessing the gpu inside an ssd. In *Proceedings of the 1st Workshop on Near-Data Processing*, 2013. 5.3

[27] Junguk Cho, Shahnam Mirzaei, Jason Oberg, and Ryan Kastner. Fpga-based face detection system using haar classifiers. In *ACM/SIGDA international symposium on Field programmable gate arrays*, 2009. 5.2.4, 5.3

[28] Eric Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *Proc. of the 43rd Annual IEEE/ACM Intl. Symp. on Microarchitecture (MICRO-43)*, 2010. 5.2.4, 5.2.4

[29] J. Clement. Hours of video uploaded to youtube every minute as of may 2019. `https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/`. Last accessed December 11, 2020. 1.3, 5, 5.2.1

[30] Intel Corp. `https://www.movidius.com/myriadx`, 2019. 1.2

[31] Eduardo Cuervo, Aruna Balasubramanian, Dae ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, 2010. 5

[32] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. Ieee, 2005. 1.1

[33] Ritendra Datta, Dhiraj Joshi, Jia Li, and James Z Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Surveys (Csur)*, 40(2):1–60, 2008. 2.2

[34] D. J. Dewitt, S. Ghandeharizadehand, D. A. Schneider, A. Bricker, H-I Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990. 5.3

[35] D.J. DeWitt. DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems. *IEEE Transactions on Computers*, 28(6), June 1979. 5.3

[36] D.J. DeWitt and P.B. Hawthorn. A Performance Evaluation of Database Machine Architectures. In *Proceedings of VLDB*, September 1981. 5.3

[37] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD*. ACM, 2013. 5.3

[38] Shiv Ram Dubey. A decade survey of content based image retrieval using deep learning, 2020. 2.2

[39] Debidatta Dwibedi, Ishan Misra, and Martial Hebert. Cut, paste and learn: Surprisingly easy synthesis for instance detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1301–1310, 2017. 4.5

[40] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2): 303–338, June 2010. 1.3

[41] Zhou Fang, Dezhi Hong, and Rajesh K. Gupta. Serving deep neural networks at the cloud edge for vision applications on mobile platforms. In *Proceedings of the 10th ACM Multimedia Systems Conference*, 2019. doi: 10.1145/3304109.3306221. URL `https:`

//doi.org/10.1145/3304109.3306221. 5.3

[42] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE transactions on pattern analysis and machine intelligence*, 2009. 6.7, 6.7

[43] Martin A Fischler and Robert A Elschlager. The representation and matching of pictorial structures. *IEEE Transactions on computers*, 100(1):67–92, 1973. 6.7

[44] Jason Flinn. *Cyber Foraging: Bridging Mobile and Cloud Computing via Opportunistic Offload*. Morgan & Claypool Publishers, 2012. 2.1

[45] Jason Flinn and Mahadev Satyanarayanan. Energy-aware Adaptation for Mobile Applications. In *Proceedings of the Seventeenth ACM Symposium on Operating systems Principles*, Charleston, SC, 1999. 2.1

[46] Daniel Y. Fu, Will Crichton, James Hong, Xinwei Yao, Haotian Zhang, Anh Truong, Avanika Narayan, Maneesh Agrawala, Christopher Ré, and Kayvon Fatahalian. Rekall: Specifying video events using compositions of spatiotemporal labels. 2019. URL http://arxiv.org/abs/1910.02993. 6.7

[47] Rohit Ghosh. Deep learning for videos: A 2018 guide to action recognition. https://blog.qure.ai/notes/deep-learning-for-videos-action-recognition-review, 2018. Last Accessed January 23, 2021. 1.1, 6, 6.1, 6.7

[48] Joshua Gleason, Rajeev Ranjan, Steven Schwarcz, Carlos Castillo, Jun-Cheng Chen, and Rama Chellappa. A proposal-based solution to spatio-temporal action detection in untrimmed videos. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 141–150. IEEE, 2019. 6.7

[49] John Linwood Griffin, Jiri Schindler, Steven W Schlosser, John S Bucy, and Gregory R Ganger. Timing-accurate storage emulation. In *Conference on File and Storage Technologies*, 2002. 5.2.5

[50] Ralf Hartmut Güting. An introduction to spatial database systems. *the VLDB Journal*, 3 (4):357–399, 1994. 6.7

[51] Gylfi Guundefinedmundsson, Laurent Amsaleg, Björn Jónsson, and Michael J. Franklin. Towards engineering a web-scale multimedia service: A case study using spark. In *Proceedings of the 8th ACM on Multimedia Systems Conference*, 2017. doi: 10.1145/3083187.3083200. URL https://doi.org/10.1145/3083187.3083200. 5.3

[52] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014. 2.1, 5.3

[53] Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai, and Mahadev Satyanarayanan. You can teach elephants to dance: Agile vm handoff for edge computing. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–14, 2017. 2.1, 3.3.1

[54] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009. doi: 10.1109/MIS.2009.36. 1

[55] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding Sources of Inefficiency in General-Purpose Chips. In *Proc. of the 37th Annual Intl. Symp. on Computer Architecture*, 2010. 5.2.4

[56] Ben Hamlin, Ryan Feng, and Wu-chi Feng. Isift: Extracting incremental results from sift. In *Proceedings of the 9th ACM Multimedia Systems Conference*, 2018. doi: 10.1145/3204949.3210549. URL `https://doi.org/10.1145/3204949.3210549`. 5.3

[57] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015. 1.2

[58] Chun He, Alexandros Papakonstantinou, and Deming Chen. A novel soc architecture on fpga for ultra fast face detection. In *2009 IEEE International Conference on Computer Design*, 2009. 5.2.4, 5.3

[59] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of IEEE Computer Vision and Pattern Recognition*, pages 770–778, 2016. 1.1, 1.2, 5.2.7, 5.3

[60] Jeffrey Helt, Guoyao Feng, Srinivasan Seshan, and Vyas Sekar. Sandpaper: Mitigating performance interference in cdn edge proxies. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019. doi: 10.1145/3318216.3363313. URL `https://doi.org/10.1145/3318216.3363313`. 5.3

[61] Stefan Hinterstoisser, Olivier Pauly, Hauke Heibel, Martina Marek, and Martin Bokeloh. An annotation saved is an annotation earned: Using fully synthetic training for object instance detection, 2019. 4.5

[62] Yuichi Hori and Tadahiro Kuroda. A 0.79-mm^2 29-mw real-time face detection core. *IEEE Journal of solid-state circuits*, 2007. 5.2.4, 5.3

[63] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 1.2, 3.2.4, 5.2.8, 5.3

[64] D.K. Hsiao. DataBase Machines Are Coming, DataBase Machines Are Coming! *IEEE Computer*, 12(3), March 1979. 5.3

[65] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, 2018. 6.1, 6.7

[66] Bo Hu and Wenjun Hu. Linkshare: Device-centric control for concurrent and continuous mobile-cloud interactions. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019. doi: 10.1145/3318216.3363303. URL `https://doi.org/10.`

1145/3318216.3363303. 5.3

[67] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan. Quantifying the impact of edge computing on mobile applications. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, page 5. ACM, 2016. 2.1

[68] G. Hunt and M. Scott. The Coign Automatic Distributed Partitioning System. In *Proceedings of USENIX Operating Systems Design and Implementation*, 1999. 5.3

[69] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, Mahadev Satyanarayanan, Gregory R Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedins of USENIX Conference on File and Storage Technologies*, 2004. 1.5, 1.7, 2.2, 4, 7.1

[70] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016. 1.2

[71] Dewan Ibtesham. *Improving Large Scale Application Performance via Data Movement Reduction*. PhD thesis, University of New Mexico, December 2017. 5.2.4

[72] Intel. Intel jpeg decoder core. `https://www.intel.com/content/www/us/en/programmable/solutions/partners/partner-profile/a2e-technologies/ip/jpeg-decoder-core.html`, 2019. Last accessed September 12, 2019. 5.2.4, 5.2.5, 5.3

[73] Samvit Jain, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, and Joseph Gonzalez. Scaling video analytics systems to large camera deployments. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 9–14, 2019. 6.1

[74] Peter Jeffcock. Minimize Data Movement. `https://blogs.oracle.com/bigdata/minimize-data-movement`. Last accessed January 10, 2021. 5.2.4

[75] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 2012. 6, 6.1, 6.7

[76] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014. 1.2

[77] Angela H Jiang, Daniel L-K Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A Kozuch, Padmanabhan Pillai, David G Andersen, and Gregory R Ganger. Mainstream: Dynamic stem-sharing for multi-tenant video processing. In *2018 USENIX Annual Technical Conference USENIX ATC 18)*, pages 29–42, 2018. 2.1, 5.3, 6.1, 6.7

[78] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference*

of the ACM Special Interest Group on Data Communication, pages 253–266. ACM, 2018. 2.1, 5.3, 6.1, 6.7

[79] Seunghun Jin, Dongkyun Kim, Thuy Tuong Nguyen, Daijin Kim, Munsang Kim, and Jae Wook Jeon. Design and implementation of a pipelined datapath for high-speed face detection using fpga. *IEEE Transactions on Industrial Informatics*, 2011. 5.2.4, 5.3

[80] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017. 1.2

[81] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the Very Large Data Bases*, 2017. 5.2.6, 5.3, 6.1, 6.7

[82] Daniel Kang, Peter Bailis, and Matei Zaharia. BlazeIt: Fast Exploratory Video Queries using Neural Networks. arXiv:1805.01046v1, 2018. 5.2.6, 5.2.7, 5.3, 6.1, 6.7

[83] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. Enabling cost-effective data processing with smart ssd. In *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*, 2013. 5.3

[84] K. Keeton, D. Patterson, and J. Hellerstein. A Case for Intelligent Disks (IDISKs). *ACM SIG on Management of Data Record*, 1998. 1.7, 5.2.4, 5.3

[85] Aaron Koehl and Haining Wang. Serf: Optimization of socially sourced images using psychovisual enhancements. In *Proceedings of the 7th International Conference on Multimedia Systems*, 2016. doi: 10.1145/2910017.2910609. URL `https://doi.org/10.1145/2910017.2910609`. 5.3

[86] Adriana Kovashka, Olga Russakovsky, Li Fei-Fei, and Kristen Grauman. Crowdsourcing in computer vision. *CoRR*, abs/1611.02145, 2016. URL `http://arxiv.org/abs/1611.02145`. 4.5

[87] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012. 1.1

[88] H. Kuehne, H. Jhuang, E. Garrote, T. Poggio, and T. Serre. HMDB: a large video database for human motion recognition. In *Proceedings of the International Conference on Computer Vision (ICCV)*, 2011. 1.3, 6.5.1, 6.5.2, 6.7

[89] George Kyrtsakas and Roberto Muscedere. An fpga implementation of a custom jpeg image decoder soc module. In *IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 2017. 5.2.4, 5.2.5, 5.3

[90] Jimmy Lei Ba, Kevin Swersky, Sanja Fidler, et al. Predicting deep zero-shot convolutional neural networks using textual descriptions. In *Proceedings of the IEEE International Conference on Computer Vision*, 2015. 6.7

[91] Patrick Michael Leyshock. *Optimizing Data Movement in Hybrid Analytic Systems*. PhD thesis, Portland State University, 2014. 5.2.4

[92] Yong Li and Wei Gao. Muvr: Supporting multi-user mobile virtual reality with resource constrained edge cloud. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 1–16. IEEE, 2018. 2.1

[93] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *European Conference on Computer Vision*. Springer, 2014. 1.3, 6.1

[94] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European Conference on Computer Vision*, pages 21–37. Springer, 2016. 5.2.8

[95] David G Lowe. Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. Ieee, 1999. 1.1

[96] Cewu Lu, Ranjay Krishna, Michael Bernstein, and Li Fei-Fei. Visual relationship detection with language priors. In *European Conference on Computer Vision*, 2016. 6.7

[97] Spencer K Lynn and Lisa Feldman Barrett. 'Utilizing' signal detection theory. *Psychological science*, 2014. 1.4

[98] X. Ma and A. Reddy. MVSS: An Active Storage Architecture. *IEEE Transactions On Parallel and Distributed Systems*, 2003. 5.2.4, 5.3

[99] Chris Mellor. Will NVMe become the universal block storage access protocol? `https://blocksandfiles.com/2020/05/27/nvme-universal-block-storage-access-protocol/`, 2020. Last accessed: August 23, 2020. 5.2.4

[100] G. Memik, M. Kandemir, and A. Choudhary. Design and Evaluation of Smart Disk Architecture for DSS Commercial Workloads. In *Proceedings of the International Conference on Parallel Processing*, 2000. 5.2.4, 5.3

[101] Mike Mesnier, Gregory R Ganger, and Erik Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8), 2003. 5.2.3

[102] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey, 2020. 1.1

[103] Ishan Misra, C Lawrence Zitnick, and Martial Hebert. Shuffle and learn: unsupervised learning using temporal order verification. In *European Conference on Computer Vision*, 2016. 6.7

[104] MyLio.com. Here's How Many Digital Photos Will Be Takein in 2017. `https://focus.mylio.com/tech-today/how-many-digital-photos-will-be-taken-2017-repost`, 2017. 1.3, 5.2.1

[105] BRAD NEMIRE. Nvidia gpus sort through tens of millions of flickr photos — nvidia blog. `https://blogs.nvidia.com/blog/2015/07/15/flickr/`, 2015. (Accessed on 11/05/2020). 1.1

[106] Netflix. Internet connection speed recommendations. `https://help.netflix.`

com/en/node/306, 2017. 2.1

[107] An Thanh Nguyen, Byron C Wallace, and Matthew Lease. Combining crowd and expert labels using decision theoretic active learning. In *Third AAAI conference on human computation and crowdsourcing*, 2015. 1.5, 4.5

[108] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997. 2.1

[109] Sangmin Oh, Anthony Hoogs, Amitha Perera, Naresh Cuntoor, Chia-Chih Chen, Jong Taek Lee, Saurajit Mukherjee, JK Aggarwal, Hyungtae Lee, Larry Davis, et al. A large-scale benchmark dataset for event recognition in surveillance video. In *CVPR 2011*. IEEE, 2011. 1.3, 5.2.1, 5.2.6, 6, 6.5.2, 6.5.2, 6.7

[110] Sinno Jialin Pan, Qiang Yang, et al. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 2010. 1.5, 1.7, 3.2.4, 4, 4.1

[111] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017. 1.2

[112] Konstantin Pogorelov, Sigrun Losada Eskeland, Thomas de Lange, Carsten Griwodz, Kristin Ranheim Randel, Håkon Kvale Stensland, Duc-Tien Dang-Nguyen, Concetto Spampinato, Dag Johansen, Michael Riegler, and et al. A holistic multimedia system for gastrointestinal tract disease detection. In *Proceedings of the 8th ACM on Multimedia Systems Conference*, 2017. doi: 10.1145/3083187.3083189. URL https://doi.org/10.1145/3083187.3083189. 5.3

[113] Cision PRWeb. Introducing the highest performance and most power efficient 4Kp120 HEVC/H.265 decoder. http://www.prweb.com/releases/2014/01/prweb11491436.htm, 2014. Last accessed September 17, 2019. 5.2.4

[114] G. Qadah and K. B. Irani. A Database Machine for Very Large Relational Databases. *IEEE Transactions on Computers*, C-34(11), November 1985. 5.3

[115] Yanyuan Qin, Shuai Hao, Krishna R. Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. Quality-aware strategies for optimizing abr video streaming qoe and reducing data usage. In *Proceedings of the 10th ACM Multimedia Systems Conference*, 2019. doi: 10.1145/3304109.3306231. URL https://doi.org/10.1145/3304109.3306231. 5.3

[116] Pranav Rajpurkar, Jeremy Irvin, Aarti Bagul, Daisy Ding, Tony Duan, Hershel Mehta, Brandon Yang, Kaylie Zhu, Dillon Laird, Robyn L Ball, et al. MURA: Large dataset for abnormality detection in musculoskeletal radiographs. *arXiv preprint arXiv:1712.06957*, 2017. 1.4

[117] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid training data creation with weak supervision. *Proceedings of the VLDB Endowment*, 11(3):269–282, 2017. 4.5

[118] Alexander J Ratner, Henry Ehrenberg, Zeshan Hussain, Jared Dunnmon, and Christopher Ré. Learning to compose domain-specific transformations for data augmentation. In *Advances in neural information processing systems*, pages 3236–3246, 2017. 1.5, 4.5

[119] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, 2015. 1.1, 1.2, 5.3, 6, 6.3.3

[120] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of Very Large Data Bases*, 1998. 5.2.4, 5.3

[121] Erik Riedel. *Active Disks — Remote Execution for Network-Attached Storage*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1999. CMU-CS-99-177. 5.3

[122] J. Rubio, M. Valluri, and L. John. Improving Transaction Processing using a Hierarchical Computing Server. Technical Report TR-020719-01, Laboratory for Computer Architecture, The University of Texas at Austin, July 2002. 5.2.4, 5.3

[123] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 2015. 1.2, 1.3, 4.5

[124] M Satyanarayanan, Rahul Sukthankar, Adam Goode, Larry Huston, Lily Mummert, Adam Wolbach, Jan Harkes, Richard Gass, and Steve Schlosser. The opendiamond platform for discard-based search. In *Tech rep School of Computer Science, Carnegie Mellon University*. 2008. CMU-CS-08-132. 3.3.3, 3.3.4

[125] Mahadev Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Ottawa, Canada, 1996. 2.1

[126] Mahadev Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4), 2001. 2.1, 5

[127] Mahadev Satyanarayanan. The Emergence of Edge Computing. *IEEE Computer*, 50(1), January 2017. 1.5, 5.3

[128] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), October-December 2009. 2.1

[129] Mahadev Satyanarayanan, Rahul Sukthankar, Lily Mummert, Adam Goode, Jan Harkes, and Steve Schlosser. The Unique Strengths and Storage Access Characteristics of Discard-Based Search. *Journal of Internet Services and Applications*, 2010. 2.2

[130] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 45–50, 2019. 2.1

[131] Seagate. Kinetic HDD. `https://www.seagate.com/support/enterprise-servers-storage/nearline-storage/kinetic-hdd/`.

5.2.3

[132] K. Sengchuai, W. Wichakool, N. Jindapetch, and P. Smithmaitrie. Fpga-based hardware-in-the-loop verification of dual-stage hdd head position control. In *IEEE Regional Symposium on Micro and Nanoelectronics (RSM)*, 2015. 5.2.4

[133] Shu Shi, Varun Gupta, Michael Hwang, and Rittwik Jana. Mobile VR on Edge Cloud: A Latency-Driven Design. In *Proceedings of the 10th ACM Multimedia Systems Conference*, 2019. doi: 10.1145/3304109.3306217. URL `https://doi.org/10.1145/3304109.3306217`. 5.3

[134] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5), October 2016. 2.1, 5.3

[135] Gunnar A. Sigurdsson, Gül Varol, Xiaolong Wang, Ali Farhadi, Ivan Laptev, and Abhinav Gupta. Hollywood in homes: Crowdsourcing data collection for activity understanding. In *European Conference on Computer Vision*, 2016. 1.3, 4.5

[136] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 1.2

[137] Chen Song, Jiacheng Chen, Ryan Shea, Andy Sun, Arrvindh Shriraman, and Jiangchuan Liu. Scalable distributed visual computing for line-rate video streams. In *Proceedings of the 9th ACM Multimedia Systems Conference*, New York, NY, USA, 2018. doi: 10.1145/3204949.3204974. URL `https://doi.org/10.1145/3204949.3204974`. 5.3

[138] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012. 1.3, 6.5.1, 6.5.2, 6.7

[139] Hui Su, Alexander Bokov, Urvang Joshi, Debargha Mukherjee, Jingning Han, and Yue Chen. Context-adaptive recursive-filtering-based intra prediction in video coding. In *Proceedings of the 24th ACM Workshop on Packet Video*, New York, NY, USA, 2019. doi: 10.1145/3304114.3325615. URL `https://doi.org/10.1145/3304114.3325615`. 5.3

[140] S.Y.W. Su, L.H. Nguyen, A. Emam, and G.J. Lipovski. The Architectural Features and Implementation Techniques of the Multicell CASSM. *IEEE Transactions on Computers*, 28(6), June 1979. 5.3

[141] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of IEEE Computer Vision and Pattern Recognition*, pages 1–9, 2015. 1.2

[142] Billy Tallis. Western digital launches new wd black nvme ssds and thunderbolt dock. `https://www.anandtech.com/show/16149/western-digital-launches-new-wd-black-nvme-ssds-and-thunderbolt-dock`, 2020. Last accessed January 21, 2021. 5.2.4

[143] SOC Technologies. H.264 4k video decoder chipset. `https://www.soctechnologies.com/chipsets/chipset-h264-4k-decoder`, 2019.

Last accessed September 17, 2019. 5.2.4

[144] J. N. Teoh, W. E. Wong, T. Ould Bachir, Y. Hu, F. Hong, C. Du, and A. Al-Mamun. Fpga implementation of nonlinear control on hard disk drive. In *2009 IEEE International Conference on Control and Automation*, 2009. 5.2.4

[145] Bart Thomee, David A Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. YFCC100M: the new data in multimedia research. *Communications of the ACM*, 2016. 1.3, 1.4, 4, 4.3, 5.2.1, 5.2.6

[146] K. Thongkhome, C. Thanavijitpun, and S. Choomchuay. A fpga design of aes core architecture for portable hard disk. In *International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2011. 5.2.4

[147] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines. In *Proceedings of the File and Storage Technologies Conference*, 2013. 5.3

[148] Arash Vahdat, Kevin Cannons, Greg Mori, Sangmin Oh, and Ilseo Kim. Compositional models for video event detection: A multiple kernel learning latent variable approach. In *Proceedings of the IEEE International Conference on Computer Vision*, 2013. 6, 6.1, 6.7

[149] Jack Valmadre, Luca Bertinetto, João Henriques, Andrea Vedaldi, and Philip HS Torr. End-to-end representation learning for correlation filter based tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2805–2813, 2017. 1.2

[150] Sudheendra Vijayanarasimhan and Kristen Grauman. Large-scale live active learning: Training object detectors with crawled data and crowds. *International Journal of Computer Vision*, 2014. 4.5

[151] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. Ssd in-storage computing for list intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, 2016. 5.3

[152] Junjue Wang. *Scaling Wearable Cognitive Assistance*. PhD thesis, CMU-CS-20-107, CMU School of Computer Science, 2020. 2.1, 2.1, 7.2.1

[153] Junjue Wang, Brandon Amos, Anupam Das, Padmanabhan Pillai, Norman Sadeh, and Mahadev Satyanarayanan. A Scalable and Privacy-Aware IoT Service for Live Video Analytics. In *Proceedings of ACM Multimedia Systems*, Taipei, Taiwan, June 2017. 2.1

[154] Junjue Wang, Ziqiang Feng, Zhuo Chen, Shilpa George, Mihir Bala, Padmanabhan Pillai, Shao-Wen Yang, and Mahadev Satyanarayanan. Bandwidth-efficient live video analytics for drones via edge computing. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 159–173. IEEE, 2018. 5.3, 6.1, 6.7

[155] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards Scalable Edge-Native Applications. In *Proceedings of the Fourth IEEE/ACM Symposium on Edge Computing (SEC 2019)*, Washington, DC, November 2019. 5.3, 6.1, 6.7

[156] K. Wang, D. Zhang, Y. Li, R. Zhang, and L. Lin. Cost-effective active learning for deep image classification. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(12):2591–2600, 2017. doi: 10.1109/TCSVT.2016.2589879. 4.5

[157] Zhongdao Wang, Liang Zheng, Yixuan Liu, and Shengjin Wang. Towards real-time multi-object tracking. *arXiv preprint arXiv:1909.12605*, 2019. 6.1, 7.2.3

[158] Mark Weiser. The computer for the 21 st century. *Scientific american*, 265(3):94–105, 1991. 2.1

[159] R. Wickremisinghe, J. Vitter, and J. Chase. Distributed Computing with Load-Managed Active Storage. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing*, 2002. 5.2.4, 5.3

[160] Nicolai Wojke and Alex Bewley. Deep cosine metric learning for person re-identification. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 748–756. IEEE, 2018. doi: 10.1109/WACV.2018.00087. 6.3.1

[161] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. Simple online and realtime tracking with a deep association metric. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 3645–3649. IEEE, 2017. doi: 10.1109/ICIP.2017.8296962. 6.3.1

[162] W. Wu, H. Su, and Q. Wu. Implementing a serial ata controller base on fpga. In *Second International Symposium on Computational Intelligence and Design*, 2009. 5.2.4

[163] Xilinx. Xilinx jpeg decoder. `https://www.xilinx.com/products/intellectual-property/1-4dcu5s.html`, 2019. Last accessed: September 12, 2019. 5.2.4, 5.2.5, 5.3

[164] Fan Yang, Sakriani Sakti, Yang Wu, and Satoshi Nakamura. Make skeleton-based action recognition model smaller, faster and better. In *ACM International Conference on Multimedia in Asia*, 2019. 6, 6.1, 6.7

[165] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. Video processing with serverless computing: A measurement study. In *ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2019. doi: 10.1145/3304112.3325608. URL `https://doi.org/10.1145/3304112.3325608`. 5.3

[166] Yu Zheng. Trajectory data mining: An overview. 6(3), 2015. ISSN 2157-6904. doi: 10.1145/2743025. URL `https://doi.org/10.1145/2743025`. 6.7

[167] Yingying Zhu, Nandita M Nayak, and Amit K Roy-Chowdhury. Context-aware activity recognition and anomaly detection in video. *IEEE Journal of Selected Topics in Signal Processing*, 7(1):91–101, 2012. 6.7