Extract-Transform-Load for Video Streams

Ferdi Kossmann
MIT CSAIL
ferdik@csail.mit.edu

Ziniu Wu
MIT CSAIL
ziniuw@csail.mit.edu

Eugenie Lai
MIT CSAIL
eylai@csail.mit.edu

Nesime Tatbul
MIT CSAIL, Intel Labs
tatbul@csail.mit.edu

Lei Cao
MIT CSAIL, University of Arizona
lcao@csail.mit.edu

Tim Kraska
MIT CSAIL, AWS
kraska@csail.mit.edu

Sam Madden
MIT CSAIL
madden@csail.mit.edu

ABSTRACT

Social media, self-driving cars, and traffic cameras produce video streams at large scales and cheap cost. However, storing and querying video at such scales is prohibitively expensive. We propose to treat large-scale video analytics as a data warehousing problem: Video is a format that is easy to produce but needs to be transformed into an application-specific format that is easy to query. Analogously, we define the problem of Video Extract-Transform-Load (*V-ETL*). *V-ETL* systems need to reduce the cost of running a user-defined *V-ETL* job while also giving throughput guarantees to keep up with the rate at which data is produced. We find that no current system sufficiently fulfills both needs and therefore propose *Skyscraper*, a system tailored to *V-ETL*. *Skyscraper* can execute arbitrary video ingestion pipelines and adaptively tunes them to reduce cost at minimal or no quality degradation, e.g., by adjusting sampling rates and resolutions to the ingested content. *Skyscraper* can hereby be provisioned with cheap on-premises compute and uses a combination of buffering and cloud bursting to deal with peaks in workload caused by expensive processing configurations. In our experiments, we find that *Skyscraper* significantly reduces the cost of *V-ETL* ingestion compared to adaptions of current SOTA systems, while at the same time giving robustness guarantees that these systems are lacking.

PVLDB Reference Format:
Ferdi Kossmann, Ziniu Wu, Eugenie Lai, Nesime Tatbul, Lei Cao, Tim Kraska, and Sam Madden. Extract-Transform-Load for Video Streams. PVLDB, 16(9): 2302 - 2315, 2023.
doi:10.14778/3598581.3598600

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ferdiko/vetl.

1 INTRODUCTION

Every day, millions of video streams are produced by smartphones, TV stations, self-driving cars, dashcams, and CCTV cameras deployed in cities and office buildings. These video streams can offer great insights and enormous value in fields such as city planning, marketing, advertisement, smart retail, or autonomous driving. For example, city planners around Vancouver are currently facing the challenge of deciding where to place electric vehicle (EV) chargers. For that, they want to obtain data that tells them which points in the city are most commonly traversed by EVs. Most cities like Vancouver already installed hundreds to thousands of traffic cameras, which could be used to obtain such EV counts.

The naive way of counting how many EVs pass by each camera is to store the video from all cameras and then run an object detection algorithm[1] on the recorded video at query time. However, this approach has major disadvantages. First, storing the video requires outrageously large storage volumes. For example, one thousand traffic cameras roughly produce 230 TB of data every month.[2] Storing one month's data on Amazon S3 would therefore cost $60,000 per year. Second, querying for trends or averages usually requires analyzing months to years of data, which leads to large query latencies. Even on modern GPUs, state-of-the-art computer vision (CV) models can only process a few frames per second. For example, processing one year of video with the YOLO object detector [47] takes six months on an AWS p3.2xlarge instance (with an NVIDIA Tesla V100 GPU). Third, naively applying CV techniques at such scales is prohibitively expensive for many applications. For example, naively running the YOLO object detector [47] to analyze a month of traffic data from 100 cameras costs $110,000 on AWS.[3]

To address the limitations of the naive approach, we propose to manage live video streams like in a data warehouse. Video is a format that is easy to produce but hard to query. A *video warehouse* allows for efficient querying by converting incoming video into an intermediate format that is easy to query. This intermediate format is application-specific and contains the extracted entities of interest. In the EV example, it would contain car counts and types. Analogous to traditional data warehouses, we refer to the process of preparing the data for querying as Video Extract-Transform-Load (*V-ETL*). Video is *extracted* from the cameras, *transformed* into the intermediate format using CV, and *loaded* into a query engine like a relational database system. This lets the user issue queries in SQL against tables with the extracted entities (e.g., obtaining the EV counts is a simple count query on a `Detections` table, where the detected car is an EV, grouped by the camera id).

Video warehouses eliminate the storage problem since users may throw video away after extracting all entities of interest during

Proceedings of the VLDB Endowment, Vol. 16, No. 9 ISSN 2150-8097.
doi:10.14778/3598581.3598600

[1] In Canada (as in many other countries), EVs are especially easy to distinguish from other cars since they have green license plates.

[2] One traffic camera feed in our experiments produces 7.8GB of data per day.

[3] E.g., using 50 p3.2xlarge instances, each of which currently costs 3.06 USD/h.

ingestion. They also solve the query latency issue, since users can issue queries against the intermediate format and no expensive CV algorithm needs to be run at query time. However, video warehouses do not magically solve the cost problem, as the video still needs to be processed during the *V-ETL* Transform step. Furthermore, video processing must happen at the rate at which the video is produced in order to achieve continuous ingestion.

To address the challenges imposed by *V-ETL*, we built *Skyscraper* which allows for cheap video ingestion while also adhering to throughput requirements. *Skyscraper*'s goal is to make the *V-ETL transform* step more practical. It allows users to provision hardware resources according to their monetary budget and optimizes the quality of the extracted video entities on the given resources.

Depending on the provisioned hardware, *Skyscraper* reduces the work imposed by the *V-ETL* job while degrading the result quality as little as possible. *Skyscraper* does this by dynamically configuring knobs that are inherent to CV workloads. Examples of such knobs include the frame rate or the image resolution at which the video is processed, as well as further, application-specific knobs. Each of these knob represents a trade-off between work and result quality: Expensive knob configurations can reliably deliver good results, even for difficult inputs (e.g., many object occlusions); cheap configurations, on the other hand, only deliver good results on easy inputs (e.g. few occlusions, good lighting conditions etc.) but are prone to mistakes on difficult inputs. The content of real-world video streams is highly variable with frequent changes in how difficult it is to analyze the content (i.e., every few 10s of seconds). *Skyscraper* saves work by using expensive knob configurations on difficult video segments and cheap configurations otherwise.

Since *Skyscraper* needs to process data on constrained hardware at a required throughput, *Skyscraper* must configure the knobs not only based on the video content but also on the available hardware resources. Industrial deployments for live video processing are typically provisioned with three types of resources [26]: a local compute cluster with a high-bandwidth connection to the video source, a video buffer, and cloud resources that may be used to rent on-demand cloud compute (to limit cloud costs, users typically want to set a cloud budget.) *Skyscraper* leverages all three of these resource types: *Skyscraper* itself runs on the local cluster and uses it to process video. To keep costs low, the local cluster is typically not provisioned to process the most expensive knob configurations in real-time. When it falls behind, *Skyscraper* sets aside video in the buffer and, as the buffer starts to fill, offloads work to on-demand cloud workers to keep up with processing.

Skyscraper must avoid prematurely using up buffer space and cloud credits in order to not run out of them when expensive knob configurations would have the greatest impact. *Skyscraper* therefore forecasts the workload and rations compute resources with regard to future demand. To still be robust to unavoidable inaccuracies in the forecast, we propose to combine a predictive planning component with a reactive execution component, which lets *Skyscraper* make tuning decisions while considering both, the future demand and the content that is actually streamed in the moment.

Despite the need for predictive knob tuning, *Skyscraper*'s knob tuning decisions must impose a low overhead — this is especially important in low-budget regimes, where large decision overheads would consume a significant portion of the compute resources. While prior content-adaptive knob tuners run additional CV operators to make tuning decisions [12, 31], *Skyscraper* adapts to the video content only based on a user-defined quality metric (e.g., certainties commonly reported by CV models) that are extracted anyways when running the *V-ETL* job. This allows *Skyscraper* to make tuning decisions in under 0.5 ms on a single CPU core.

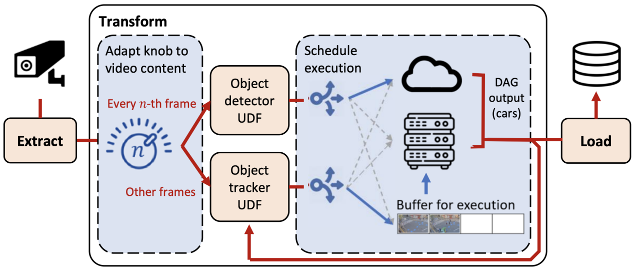


Figure 1: *Skyscraper* optimizing the expensive *V-ETL* Transform step of the EV counting example job. The blue components are provided by *Skyscraper*, the red ones by the user.

Figure 1 shows an overview of how *Skyscraper* processes the EV example workload. The user specifies user-defined functions (UDFs) that transform the video into the application-specific query format. In Figure 1, the user only defines two UDFs. The object detector UDF is responsible for detecting new cars, while the object tracker UDF is responsible for tracking cars as they move across the frame to avoid double counting them. Finally, the user registers the workload's tunable knobs. In the simple example, the user only defines a single knob that controls how frequently the object detector should be run. *Skyscraper* optimizes the costly Transform step while the user code performs the Extract and Load steps.

Prior work. While *Skyscraper* is the first system to specifically address the challenge of *V-ETL*, there are several lines of work that are relevant to *Skyscraper*. We briefly highlight two of them here and refer to Section 6 for a detailed discussion on related work.

First, there is prior work on content-adaptive knob tuning, such as Chameleon [31] and Zeus [12]. These systems are designed to reduce the average processing time per frame while assuming that the provisioned hardware can always ingest video in real-time (even during peak workload). However, when ingesting video on cheaper machines that are not peak-provisioned, prior systems do not provide throughput guarantees and are therefore impractical for *V-ETL*. Adapting these systems to fulfill throughput requirements on cheap hardware is challenging, since they are agnostic to lag and the hardware resources they run on.

Second, there is prior work on systems that use knob tuning to adapt to the current query load. VideoStorm [59] and VideoEdge [26] are designed for scenarios where users run a dynamic set of queries over video streams, which causes dynamic changes to the type and number of queries running. At times when many queries are running concurrently, not all queries may be able to run at maximum quality and in real time. VideoStorm and VideoEdge tune the queries' knobs such that the queries fulfill their quality and latency goals as well as possible. However, VideoStorm and VideoEdge only adapt to the query load (i.e., the queries present in the system) and are agnostic to the streamed content. This brings no benefit in scenarios where the query load is static. While we envision most *V-ETL* applications to ingest video using a static set of processing jobs, VideoStorm might still be used if users dynamically redefine how to ingest video.

In summary, our contributions are as follows:

- We define the problem of Video Extract-Transform-Load (*V-ETL*) and identify its importance.
- To make *V-ETL* more practical, we propose *Skyscraper*, the first content-adaptive knob tuning system with throughput guarantees. *Skyscraper* lets users provision compute resources according to their budget and optimizes the result quality on the given resources.
- To effectively ration compute resources over time, we propose a combination of predictive planning and reactive execution.
- We propose a tuning method that only relies on a user-defined quality metric which is extracted anyways when running the *V-ETL* job. We find that this method allows for negligible tuning overheads.
- We conduct experiments on several real-world and synthetic workloads and find that *Skyscraper* can achieve cost reductions up to 8.7× over baselines on various workloads.

2 PROBLEM DEFINITION AND SYSTEM OVERVIEW

2.1 Problem definition

Video Extract-Transform-Load (*V-ETL*) refers to extracting entities of interest from a video stream by processing it according to a user-defined specification and adhering to two constraints. First, *V-ETL* systems must process video at the rate at which it arrives. A *V-ETL* system may lag behind on processing but may only do so by a constant amount. In practice, this means that *V-ETL* systems may use a fixed-size storage medium (i.e., buffer) to set video aside for later processing. Equation 1 states that the size of the buffered frames may not exceed the size of the buffer.

$$out(t) \subseteq in(t) \;\wedge \sum_{F\in\, in(t)\setminus out(t)} size(F) \leq B \qquad \forall t \qquad (1)$$

where t is a timestamp, $in(t)$ is the set of frames that the video source has produced at time t, $out(t)$ is the set of frames that the *V-ETL* system has processed at time t, $size(F)$ is the size of frame F in bytes and B is the buffer size in bytes.

Second, *V-ETL* systems must process video at a budget that is defined by the user. This budget is provided as a dollar cost $budget_T$ that may be spent over a given time interval T. The processing cost over interval T encompasses all costs including average wear of hardware, cloud costs, etc. The summed cost of processing all frames in T must be below $budget_T$: $\sum_{F\in T} cost(F) \leq budget_T$.

The combination of processing video at a required throughput while being constrained on computing resources makes for exciting optimization problems. *Skyscraper* aims to maximize the overall result *quality* by tuning workload-specific *knobs* that are inherent to computer vision workloads (e.g., the frame rate or image resolution). In *Skyscraper*, the quality is user-defined and is measured and returned by the user code — this lets *Skyscraper* generalize to different workloads with different notions of quality.

Users may further register arbitrary knobs together with a corresponding *knob domain*. The knob domain is a user-defined set of values that the knob may take (e.g. the knob domain for the frame rate knob might be {15 FPS, 30 FPS}). *Skyscraper* dynamically configures registered knobs based on the streamed video content and maximizes the quality (e.g. accuracy) of the extracted entities while adhering to the *V-ETL* requirements.

Formally, a knob configuration k refers to an instantiation of each knob to a value in its domain. Some knob configurations induce more work than others. Similarly, some produce more qualitative results than others. However, the result quality of a knob configuration depends not only on the configuration but also on the video content. While a high image resolution may reliably produce good results, it may not always be needed as some content can also be accurately processed at a lower resolution. Let a *video segment* denote a sequence of successive frames of the video (e.g., 2 seconds of video). We denote the quality that a knob configuration k achieves on a video segment s as $qual(k, s)$. The optimization goal of *Skyscraper* is to maximize the overall quality $qual(v)$ of entities extracted from video v, which is given by $qual(v) = \sum_{s\in v} qual(k_s, s)$where k_s is the configuration used to process segment s.

2.2 System overview

The following subsection gives a high-level overview of *Skyscraper*. Section 3 and Section 4 then provide a more detailed discussion of *Skyscraper*'s design.

Design challenges. To explain why *Skyscraper* works the way it does, we present a simplistic, idealized approach to content-adaptive knob tuning with throughput guarantees, and show where this approach fails in practice. We then present the ideas that *Skyscraper* uses to overcome the issues of the idealized approach.

For now, we do not consider buffering or the scheduling of computation between on-premise resources and the cloud. Instead, we simply consider a computation budget $budget_T$ on the number of arithmetic operations that we may use to ingest video produced during time period T. We are further given a small set of knob configurations $\mathcal{K}$ which allows us to process different segments of the video at different costs and qualities (see Section 2.1).

We observe that the knob tuning system must speculate about the future content of the video in order to effectively ration $budget_T$ over time. Otherwise, the system can not assess whether it is sensible to process content with an expensive knob configuration now or to save the budget for the future when expensive knob configurations might have a larger impact. Furthermore, we find that the effectiveness of different knob configurations often changes within seconds — a content-adaptive knob tuning system should therefore reassess which configuration to use every couple of seconds.

Now, suppose we have a forecasting function that can perfectly predict what quality each knob configuration achieves at any given time in the future. In this idealized world, we can easily build a system that achieves optimal performance: Our optimal system would slice time interval T into segments t_i of equal length, where each segment t_i is a few seconds long. The system then forecasts the quality that each knob configuration achieves on each segment $t_i \in T$. Finally, given the forecasted qualities, optimizing the assignment of knob configurations to segments is an instance of the 0-1 knapsack problem, where the overall quality must be maximized under the given budget $budget_T$.

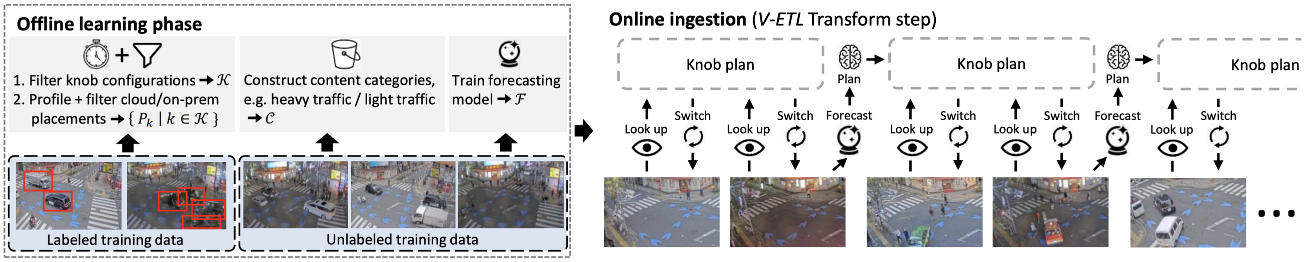

Offline learning phase
1. Filter knob configurations → $\mathcal{K}$
2. Profile + filter cloud/on-prem placements → $\{ P_k \mid k \in \mathcal{K} \}$
Construct content categories, e.g. heavy traffic / light traffic → $\mathcal{C}$
Train forecasting model → $\mathcal{F}$
Labeled training data
Unlabeled training data
Online ingestion (V-ETL Transform step)
Knob plan
Plan
Knob plan
Plan
Knob plan
Look up
Switch
Look up
Switch
Forecast
Look up
Switch
Look up
Switch
Forecast
Look up
Switch
...

Figure 2: Overview over all processing steps of *Skyscraper*.

Unfortunately, we find that achieving good accuracy on this forecasting task is infeasible in the real world. To forecast the knob configurations' qualities for each $t_i \in T$, our forecasting function needs to predict what happens at each second in the video, hours into the future. This is impossible since the precise timing of events is subject to substantial randomness. For example, it is impossible to predict the exact moment in which a large group of pedestrians will pass by a camera, hours into the future. To make our system work in the real world, we design a more practical forecasting task.

We rely on two insights that guide the design of this new forecasting task. First, we observe that there are a few types of video content that characterize any of the videos seen throughout the live stream (e.g., rush hour traffic, normal traffic, low traffic). For the content of the same kind, each knob configuration produces results of similar quality. For example, for content with many occlusions (e.g., rush hour), knob configurations that cannot handle occlusions will always produce low-quality results. Second, we observe that, while it is impossible to predict *when* certain content appears, it is possible to predict *how often* it appears, *assuming the future video is distributed roughly as a recent historical video has been.* For example, while it is impossible to predict the *precise moments* (i.e. the t_i's) at which groups of pedestrian pass by the camera, it is possible to estimate *how often* groups of pedestrians pass by the camera.

We can now design a forecasting task where accurate predictions are feasible in practice. Based on the first insight (content falls into a few categories), we use a simple clustering mechanism to compute *content categories* such that all streamed content falls into one of these categories. We construct them such that all knob configurations achieve a similar quality on the content of the same category (more details in Section 3). Then, based on the second insight (content distribution is predictable), we simply predict how often each content category appears within a time interval T. For example, if our forecasting model thinks that 10% of the video in T shows rush-hour traffic, it would forecast 10% for the rush-hour category. In practice, we can achieve high forecasting accuracy on real-world workloads.

Finally, we need to re-think how to use the forecast for knob tuning. Since we no longer forecast the qualities of individual segments t_i, we cannot assign knob configurations the same way as in our idealized system. Instead, we can only assign knob configurations to content categories. Knowing how often each content category appears allows us precisely determine the overall cost of using a knob configuration to process the content of that category. In Section 4, we describe how this allows us to find the optimal assignment of knob configurations to content categories under a given budget and for a given forecast. Given this assignment, we then need to reactively determine what category the current content belongs to. Once we determine the category, we can simply look up and use the knob configuration we assigned to this category. Section 4.2 describes a simple method for determining the current content category, which runs fast and determines the correct category with high accuracy.

In summary, we took a simplistic, idealized system and made it practical by re-designing the forecasting task. We then built an efficient system around it that can leverage this forecast for predictive knob tuning. *Skyscraper* takes these ideas and implements them for real hardware provisionings.

Skyscraper walk-through. Given these challenges imposed by content-adaptive knob tuning with throughput guarantees, we now give an overview on how *Skyscraper* uses these ideas when provisioned with real hardware (i.e., with a local compute cluster, video buffer and cloud credits). *Skyscraper* is split into an *offline learning phase* and an *online ingestion phase* as shown in Figure 2. Section 3 gives a detailed description of the offline phase and Section 4 gives a detailed description of the online phase.

The offline phase is used to pre-compute invariant properties of the *V-ETL* workload, which allow online ingestion at negligible overheads. To compute these properties, the user provides *Skyscraper* with a small set (e.g. 5 minutes) of labeled data and a larger set (e.g. two weeks) of unlabeled data from the ingested video source. *Skyscraper* uses this data to prepare online ingestion in three steps.

First, *Skyscraper* profiles different knob configurations on the provisioned on-premise hardware and cloud hardware. Each knob configuration corresponds to a directed acyclic graph (DAG) of UDFs. *Skyscraper* profiles the cloud cost and runtime of different UDF placements — executing some UDFs on the cloud may reduce the execution time (due to added parallelism) but increases the cloud cost. *Skyscraper* filters out placements that do not lie on the cost-runtime Pareto frontier. Similarly, *Skyscraper* filters out knob configurations that do not lie on the runtime-quality Pareto frontier.

Second, *Skyscraper* uses the unlabeled data to construct the content categories as discussed under *Design challenges.* The content categories are constructed solely based on a quality metric that is measured and returned by the user code (e.g. certainty or errors commonly reported by CV models). By construction, the content categories discriminate between any content characteristic that affects the quality of at least one knob configuration. Constructing the content categories solely based on a user-defined quality metric lets *Skyscraper* generalize across workloads since *Skyscraper* doesn't need to understand the precise workings of the UDFs and how their performance is affected by pixel-level changes. Furthermore, dealing with low-dimensional quality vectors (e.g., 5-dimensional)

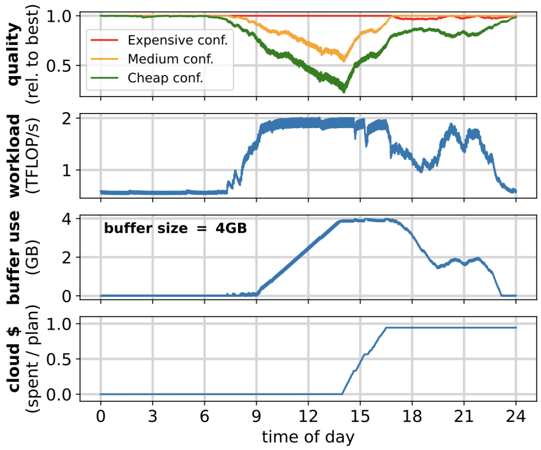


Figure 3: Running the EV workload over a traffic camera.

allows *Skyscraper* to run fast, which is almost impossible when dealing with high-dimensional image data (e.g., 750,000-dimensional).

Third, *Skyscraper* uses the unlabeled data to train the forecasting model. As in under *Design challenges*, the forecasting model forecasts how often each content category appears within a defined future time interval. This forecast is based on how frequently the content categories have appeared in the recent past.

After the offline phase, each knob configuration is characterized by the quality it achieves on different content categories as well as the profiled runtimes and cloud costs when executing the knob configuration using different task placements. When optimizing video ingestion, *Skyscraper* only considers the runtime of knob configurations together with the quality the knob configuration achieves on the current content category. This is sufficient to maximize the quality under throughput constraints and lets *Skyscraper* agnostic to the UDFs. *Skyscraper* periodically performs predictive *knob planning* (e.g. every 2 days) and reactive *knob switching* (e.g. every 2 seconds): Knob planning involves forecasting how often each content category appears in the future (e.g. within the next 2 days) and assigning knob configurations to the content categories based on the forecast. Knob switching involves determining the content category of the current video content and looking up what knob configuration the planning phase assigned to that category. Based on the assigned knob configuration, the available buffer space, and the profiled runtimes, *Skyscraper* then picks a knob configuration and task placement and uses it to process the next segment of video.

Processing example Figure 3 shows how the knob planner and knob switcher use the provisioned resources to achieve high-quality results when running the EV example workload on 24 hours of a traffic camera stream. The uppermost plot in Figure 3 shows how three different knob configurations (expensive, medium, cheap) achieve different result qualities. For the EV workload, the result quality is mainly affected by object occlusions (i.e., one car overlaps with another car). We observe that the expensive configuration reliably produces high-quality results while the cheap one only produces high-quality results at night, when there is little traffic and few occlusions.

The second plot in Figure 3 shows how the dynamic knob switching in *Skyscraper* causes the change in the workload (TFLOP per second). We can see that the workload is low during the night when *Skyscraper* frequently uses the cheap configurations, but high during the day when *Skyscraper* uses the expensive configurations. The data in Figure 3 is smoothed and hides that *Skyscraper* switched 4500 times between knob configurations over the course of the plotted time period. If we would instead always use the most expensive configuration, the workload would be constant at 5.2 TFLOP/s.

The third plot in Figure 3 shows how *Skyscraper* sets video aside into the buffer during the day when frequently running the expensive knob configuration. We can also see how *Skyscraper* catches up on processing the buffered video at 5PM, when the workload decreases. The buffer has a size of 4GB and is full at around 2 PM. When it is full, *Skyscraper* decides to offload some work to the cloud which is reflected by the rising amount of cloud credits spent in the bottom figure (note that the Y axis shows the percentage of the daily cloud budget that has been spent). We can see that *Skyscraper* spent the credits as it had planned for the day.

3 OFFLINE PREPARATION PHASE

In the offline preparation phase, *Skyscraper* is fitted on the historical video data recorded from the same source that will be ingested in the online phase. *Skyscraper* needs a small set of labeled data (i.e., 20 minutes) and a larger set of unlabeled data (e.g., 2 weeks). Based on this data, *Skyscraper* first leverages prior work [2, 59] to create a filtered set of knob configurations and a set of good task placements for them. Then, *Skyscraper* clusters video content into categories allowing *Skyscraper* to reason about video content in the online phase. Furthermore, *Skyscraper* trains a forecasting model to predict the frequency that each content category appears in the near future. We describe these procedures in more detail as follows.

3.1 Filter knob configurations and task placements

In order to optimize video processing while inducing little decision overheads during online ingestion, *Skyscraper* needs to decide the desirable knob configuration k to process the streamed content and the placement TP_k of its task graph G_k. Recall that the placement of G_k specifies which computation components when using knob configuration k to run on the cloud and which ones to run on-premises. The number of all knob configurations is exponential in the number of user-registered knobs. Similarly, the number of all possible placements for a task graph is exponential in the number of tasks. *Skyscraper* leverages prior work [2, 59] to filter the set of knob configurations and task placements down to a smaller set. Thereafter, *Skyscraper* only needs to consider promising candidates in the online phase, reducing the size of the decision problem and therefore online overheads.

We leverage the greedy hill climbing algorithm [50] proposed in VideoStorm [59] to filter the knob configurations. We use PlaceTo [2] to filter the task placements.

3.2 Categorize video dynamics

Skyscraper discretizes video content into *content categories* with the property that knob configurations achieve similar result quality for all video segments belonging to the same content category . In this section, we describe how to identify these content categories and will discuss how to forecast them in Section 3.3 and how the categories allow for efficient video ingestion in Section 4.

Skyscraper categorizes video content using unlabeled training data. *Skyscraper* first samples a set of video segments $\mathcal{S}'$ from the unlabeled data. *Skyscraper* then processes each segment $s \in \mathcal{S}'$ with all configurations $k \in \mathcal{K}$ and records the result quality that each k achieves on the segment s as $qual_s(k)$. The result quality measurement is defined by the user and will be further discussed in Section 4. We group the qualities of all configurations k on a segment s into a $|\mathcal{K}|$-dimensional *quality vector* $qual_s = [qual_s(k_1), ..., qual_s(k_{|\mathcal{K}|})]^T$. We gather the $qual_s$ for all segments $s \in \mathcal{S}'$ to form a set of quality vectors $Q = \{qual_s \,|\, s \in \mathcal{S}'\}$. Then, *Skyscraper* decides the content categories C by running KMeans [40] on Q. Thereafter, the content is clustered according to the quality that the knob configurations achieve on it, ensuring that all knob configurations achieve similar result quality for the content of the same category by the property of KMeans. A content category $c \in C$ is therefore characterized by a $|\mathcal{K}|$-dimensional cluster center, which denotes the average quality that the knob configurations will achieve on content belonging to c. We denote the cluster center as $[\widehat{qual}(k_1, c), ...\widehat{qual}(k_{|\mathcal{K}|}, c)]$, where $\widehat{qual}(k, c)$ is the average quality that k will achieve on videos categorized as c.

We find that *Skyscraper* is not very sensitive to k as long as it is not too small (e.g. $\geq$ 3). Furthermore, it is easy to tune such hyperparameters during the offline phase.

3.3 Train the forecasting model

Skyscraper trains a forecasting model $\mathcal{F}$ to predict how frequently each content category $c \in C$ appears in the near future time interval given their frequency in the most recent history. $\mathcal{F}$ allows *Skyscraper* to effectively ration computational resources and optimally allocate them for different video content categories to come. We denote the forecasted time interval as the *planned interval*.

Skyscraper uses a simple feed-forward neural network as forecasting model $\mathcal{F}$, which we find to be sufficient in Section 5. Let $r^{(T)}$ be $|C|$-dimensional histogram representing the frequency each category $c \in C$ appears over time interval T. The output of $\mathcal{F}$ is thus $r^{(PI)}$ where PI is the planned interval. The input to $\mathcal{F}$ is the content histograms of the most recently ingested data. We split the most recent time interval T_{input} into n equally-sized intervals $T_{input} = [T_1, ..., T_n]$ and provide their category occurring frequency $[r^{(T_1)}, ..., r^{(T_n)}]$ as time-series inputs to $\mathcal{F}$. We find that *Skyscraper* is not very sensitive to T_{input} and n as long as both are reasonably large (i.e. T_{input} is a couple of days and is split into intervals of a couple of hours).

Skyscraper pre-trains $\mathcal{F}$ in the offline phase using the unlabeled data. Furthermore, $\mathcal{F}$ can be fine-tuned in the online phase using the recently ingested data to provide more accurate forecasting.

4 ONLINE VIDEO INGESTION

After completing the offline learning phase, *Skyscraper* is ready to ingest live video streams. During live ingestion, *Skyscraper* uses both a predictive component (*knob planner*) and a reactive component (*knob switcher*) to make knob tuning decisions. The predictive knob planner periodically forecasts trends in the video content and lets *Skyscraper* make knob tuning decisions with the future workload in mind. This allows *Skyscraper* to put the provisioned compute resources to optimal use and prevents premature use of buffer space and cloud credits, making use of expensive knob configurations when they have the greatest impact. However, while it is possible to forecast long-term trends in the content, the exact short-term occurrence of content is subject to substantial noise. Thus, *Skyscraper* also uses a reactive knob switcher that switches between knob configurations based on the current content. The knob switcher presents a way to leverage the forecasted workload trends while being robust to short-term noise. In the following section, we describe the algorithms used for both the knob planner and the knob switcher.

4.1 Knob planner

The knob planner computes a *knob plan* that specifies which knob configurations $k \in \mathcal{K}$ to use for each content categories $c \in C$ to maximize the overall result quality given the available compute resources. Such assignment of knob configurations to c is based on the forecasted *content distribution*, which specifies how frequently each knob configuration will appear over the forecasted interval. Recall from Section 3.3, we refer to this interval as the the *planned interval*. We find that accurate forecasts can be achieved a couple of days into the future and consequently re-compute the knob plan every couple of days using a fresh forecast.

Formally, the knob plan generates a histogram α_c over knob configurations $\mathcal{K}$ for each content category $c \in C$. α_c determines how often a knob configuration $k \in \mathcal{K}$ should be used for processing content of category c - i.e., there is one bucket in the histogram for each knob configuration, indicating the relative frequency with which that configuration should be chosen for the content category. Let $\alpha_{k,c}$ denote the frequency that histogram α_c assigns to knob $k \in \mathcal{K}$ (i.e., how often knob k should be used to process the content of category c). A knob plan $\mathcal{P}$ is thus defined as the set containing the histograms for all content categories: $\mathcal{P} = \{\alpha_c \mid c \in C\}$.

Finding a knob plan that maximizes the result quality under the compute budget involves jointly optimizing the histograms for all content categories. Each category's histogram determines the total resource consumption for processing content of the category, which in turn determines how many resources are available for the remaining categories. *Skyscraper* creates a knob plan in two steps.

First, the knob planner uses the pre-trained model $\mathcal{F}$ from the offline phase to forecast how often (the ratio r_c described in Section 3) each content category will appear over the planned interval.

Second, using the forecasted content ratios r_c, *Skyscraper* formulates the assignment of knobs to content categories as a linear program. This allows *Skyscraper* to find the globally optimal knob plan $\mathcal{P}$. *Skyscraper* maximizes the expected overall result quality using the content category cluster centers computed in the offline phase. As described in Section 3, each content category $c \in C$ is defined by a KMeans cluster center, which is a vector whose i-th element denotes the average quality $\widehat{qual}(k_i, c)$ that knob configuration k_i achieves on the content of category c. Given the average quality of each knob configuration for each content category, the solution of the linear program maximizes the overall expected quality while being constrained by the compute budget $budget$.[4]

[4] The unit of the compute budget is given in $core * s$ using the on-premise server cores. *Skyscraper* internally takes care of converting the user-defined cloud credits budget.

$$\text{maximize} \quad \sum_{k,c} \alpha_{k,c} * r_c * \widehat{qual}(k,c) \tag{2}$$

$$\text{subject to} \quad \sum_{k,c} \alpha_{k,c} * r_c * cost(k) \leq budget \tag{3}$$

$$\sum_{k} \alpha_{k,c} = 1, \quad \alpha_{k,c} \geq 0 \qquad \forall c \tag{4}$$

The decision variables of the linear program are $\alpha_{k,c}$, which determine how often the content of category c should be processed by configuration k and thereby make up the knob plan. The goal of the knob plan is to maximize the overall result quality, which is denoted by Line 2. Line 3 denotes that the total amount of cost should stay below the user-specified budget. Finally, Line 4 enforces that the assigned ratios $\alpha_{k,c}$ add up to 1 for each content category (this is merely for normalization).

We use an off-the-shelf solver [54] which is able to find the solution to this linear program in less than a second for the problem sizes encountered by *Skyscraper*. After finding the optimal value for the decision variables $\alpha_{k,c}$, we have the knob plan $\mathcal{P}$ which tells us how often to use each knob k to process the content of category c in order to achieve maximum quality given the constrained computing resources. In Section 4.2, we show how $\mathcal{P}$ can be leveraged to efficiently switch between knob configurations.

4.2 Knob switcher

Based on the current video content, the knob switcher reactively determines which knob configuration $k_{next} \in \mathcal{K}$ to use and which tasks of k_{next}'s task graph $G_{k_{next}}$ to execute on the cloud and which tasks to execute on-premises. The knob switcher is designed to be lightweight and doesn't induce significant decision overheads, even when run frequently. It decides on the next knob configuration k_{next} and task placement p_{next} in three simple steps: First, it determines the category $c_{cur} \in C$ that the current content belongs to. Second, it looks content category c_{cur} up in the knob plan to obtain the configuration histogram $\alpha_{c_{cur}}$ that the knob plan assigns to c_{cur}. Third, the knob switcher picks knob configuration k_{next} based on $\alpha_{c_{cur}}$ along with a task placement p_{next} — the knob switcher hereby guarantees to never overflow the buffer. In the following, we describe how the knob switcher performs each of these steps in more detail.

In the first step, the knob switcher determines the category c_{cur} of the current content merely using the reported quality $qual^*(k_{cur})$ of the current knob configuration k_{cur}. This allows the knob switcher to select a category in a low overhead way, rather than running an expensive processing step on the video directly. Specifically, given $qual^*(k_{cur})$, the knob switcher selects the current content category c_{cur} as the one whose average quality for k_{cur} ($\widehat{qual}(k_{cur}, c_{cur})$) matches the currently reported quality ($qual(k^*)$) the closest. The average quality $\widehat{qual}(k_{cur}, c)$ of k_{cur} for a category $c \in C$ is given by c's cluster center (see Section 3.1). This is denoted by Equation 5.

$$c_{cur} = \underset{c \in C}{\text{argmin}} \left| \widehat{qual}(k_{cur}, c) - qual^*(k_{cur}) \right| \tag{5}$$

Note that the knob switcher's content classification is analogous to traditional classification with KMeans but only uses one vector dimension since the other dimensions are unattainable. This works well in *Skyscraper*'s case because the content of different categories will induce different result qualities for all knob configurations. As a result, the quality of one knob configuration is sufficient to discriminate between content categories. We experimentally verify this in Section 5.6.

In the second step, the knob switcher then looks up the derived content category c_{cur} in the knob plan $\mathcal{P}$. This yields a histogram $\alpha_{c_{cur}}$ dictating how often each knob configuration $k \in \mathcal{K}$ should be used to process the content of the current category c_{cur}:

In the third step, the knob switcher determines the knob configuration k_{next} that will be used for processing the newly arriving content, together with task placement p_{next} that determines which tasks of k_{next}'s task graph to execute on the cloud and which ones to execute on-premises. The knob switcher tries to adhere as closely to the planned histogram $\alpha_{c_{cur}}$ as possible and therefore keeps a histogram $\widehat{\alpha}_c$ for each $c \in C$, which denotes how frequently each knob configuration has actually been used to process the content of category c. To adhere as closely to the knob plan as possible, the knob switcher picks the knob configuration k_{next} that minimizes the difference between $\widehat{\alpha}_{c_{cur}}$ and $\alpha_{c_{cur}}$. This is denoted by Equation 6. Finally, the knob switcher picks a placement p_{next} for k_{next}. *Skyscraper* picks the cheapest placement of $G_{k_{next}}$ that does not overflow the buffer.

$$k_{next} = k_i \;\; \text{with } i = \underset{1 \leq i \leq |\mathcal{K}|}{\text{argmax}} \left(\alpha_{c_{cur}}[i] - \widehat{\alpha}_{c_{cur}}[i] \right) \tag{6}$$

It is worth noting that there is an edge case when picking the task placement p_{next}: Some knob configurations do not possess task placements that run in real-time, even when heavily adding cloud compute. Reasons for this include limited bandwidth to the cloud, high round trip times to the cloud, and limited opportunities for adding parallelism to the DAG execution. If all placements of k_{next} would make *Skyscraper*'s buffer overflow, the knob switcher will choose a different configuration k'_{next} to be the next one. This knob configuration k'_{next} is the next less qualitative one compared to k_{next}. Like for k_{next}, the knob switcher will pick the cheapest placement of k'_{next} that does not overflow the buffer. If all placements of k'_{next} would overflow the buffer, the knob switcher will recursively apply this procedure of picking the next less qualitative knob configuration until it finds a configuration and task placement that do not overflow the buffer.

In summary, the knob switcher uses three steps to find a knob configuration $k_{next} \in \mathcal{K}$ along with a task placement p_{next} while adding little runtime overheads to the ingestion process. The knob switcher tries to adhere as closely to the knob plan $\mathcal{P}$ as possible, only deviating from the knob plan when this is required to avoid a buffer overflow. This ensures that the knob switcher maximizes the result quality with the given resources.

5 EVALUATION

We evaluate *Skyscraper* on several real-world applications, covering public health monitoring, traffic planning, and social media analysis. We describe these workloads in subsection 5.2. Then, we evaluate *Skyscraper* on the following aspects:

§5.3 What cost savings does *Skyscraper* achieve versus using a static knob configuration?

§5.4 How much do cloud bursting and buffering individually contribute to cost savings in different quality regimes? When do they perform well and when don't they?

§5.5 How much decision overhead does *Skyscraper* impose at different scales?

§5.6 How accurate are knob planner and knob switcher, and what effect do inaccuracies have on *Skyscraper*'s end-to-end performance?

When evaluating different hyperparameter choices of *Skyscraper* (e.g., number of content categories (KMeans clusters), we find that *Skyscraper*'s end-to-end performance is insensitive to many of the hyperparameters as long as they are chosen from reasonable ranges.

5.1 Implementation

We implement *Skyscraper* in Python on top of Ray [46]. We instantiate several Ray actors for both the on-premise and the cloud version of each UDF. The number of duplicate actors is based on the number of logical cores of the machine. We only map UDFs to Ray actors; all of *Skyscraper*'s components run in the parent process and synchronize the calls to the actors.

We use AWS Lambda [51] to run UDFs in the cloud and provision 3GB of memory for each cloud function. To simulate incoming video streams in real time, we read video frames from the disk and pause appropriately between frames to guarantee 30 fps streaming rate. All workloads are compute-bound and we find that in our experiments decode only amounted to 5% of the overall runtime. The streamed video is encoded in H.264 [49] and has a resolution of 1280 × 720 (HD). In our experiments, each frame is decoded when arriving in the system (as part of the user code).

When sending full or partial frames to the cloud, we compress them to JPEG-1 format [10]. We then serialize the JPEG using Base64 [29] and send the string as part of an HTTPS request. The overhead for encoding and decoding is negligible compared to the transfer time saved through compression.

5.2 Workloads

We evaluate *Skyscraper* using three workloads on public health monitoring, traffic planning, and social media analysis. They cover a diverse set of computer vision primitives including object detectors, trackers, and classifiers, as described below.

COVID-19 safety measures (COVID) During the coronavirus pandemic, decision-makers have executed several safety measures to slow down the spread of the virus. Such measures include wearing facial masks and social distancing. Measuring where and how strictly people adhere to these measures can be used for decision-making and informing people at risk. The COVID workload consists of a YOLOv5 object detector [47] to detect pedestrians and a KCF tracker [23] to track the detected pedestrians ("detect-to-track"). After the detection, for each detected pedestrian, the workload employs homography [14] to measure the pedestrian's distance from others.

This workload contains the following knobs: 1) *frame rate* at which video is processed ({30FPS, 15FPS, 10FPS, 5FPS, 1FPS}), 2) *object detection rate* to run object detector (every {1, 5, 30, 60} frames) and 3) *tiling for object detection* that slices the frames into ({1x1, 2x2}) tiles.

The workload is executed on an 8-day video stream of a busy shopping street in Tokyo.[5]We measure quality in terms of the number of people detected and tracked over time as YOLO has a low false positive rate and KCF trackers reliably report tracking errors.

Multi-object tracking (MOT) Multi-object tracking (MOT) is a key primitive in many video analytical pipelines. In this workload, we adopt the recent state-of-the-art TransMOT [11] tracker on MOT benchmark [17] and introduce four tunable knobs: 1) *frame rate* (every {1, 5, 30, 60} frames), 2) *number of tiles* ({1x1, 2x2} tiling), 3) *length of history* denoting the number of historical frames ({1, 2, 3, 5}) as the TransMOT input, and 4) *model size* ({small, medium, large}) that specifies different network sizes of TransMOT.

We run MOT on a stream of a traffic intersection, Shibuya in Tokyo to track pedestrians for 8 days. MOT's processing quality is defined as the sum of tracked pedestrians weighted by the model's reported certainty. With this quality metric, we want to evaluate how *Skyscraper* maximizes model certainty as a proxy for accuracy as proposed in prior work [43, 48].

Multi-modal opinion sentiment and emotion intensity (MOSEI) This workload is synthetic and simulates a video stream analysis application on Twitch. The number of incoming streams varies over time and mimics the number of live Twitch streams over two days.[6] We further introduce two types of spikes to evaluate *Skyscraper* under difficult conditions:

- *MOSEI-HIGH*: We introduce high but short peaks in workload, consisting of 62 concurrently incoming video streams. This makes cloud bursting difficult due to bandwidth limitations.
- *MOSEI-LONG*: We introduce a long peak of continuous workload. In this case, the buffer alone cannot handle all the extra work.

We use the CMU-MOSEI [5] dataset to simulate incoming video streams, as it has ground truth labels that allow us to train the models used in the workload. It contains various talking head videos from YouTube. The task of the MOSEI workload is to classify the opinion sentiment of the speaker using both the audio and the visual content. CMU-MOSEI provides extracted features from the video with ground-truth labels. We trained a neural network on CMU-MOSEI's training set and used its test set to evaluate *Skyscraper*.

MOSEI workload contains the four knobs: 1) *frame rate*, 2) *frequency of sentiment analysis* that we may run sentimental analysis model once every {1, 2, 3, 4, 5, 6, 7} sentences of the spoken audio and video, 3) *model size* of the sentimental analysis model, and 4) the *number of streams* to analyze.

We evaluate the processing quality as the weighted sum over the ingested streams weighted on model's reported certainty.

5.3 Cost efficiency

This section evaluates the end-to-end cost savings that *Skyscraper* achieves on these workloads. We hereby compare *Skyscraper* to two baselines. The Static baseline processes the video streams statically using the same knob configuration throughout the stream. The Chameleon* baseline refers to an adapted version of Chameleon [31]. We equip Chameleon with a buffer and adapt it to set video aside when the provisioned hardware cannot process it in real-time. This

[5] The Koen-Dori street in the Shibuya district: https://youtu.be/gALQR-nsEME

[6] As recorded by Twitch Tracker at https://twitchtracker.com/statistics/active-streamers

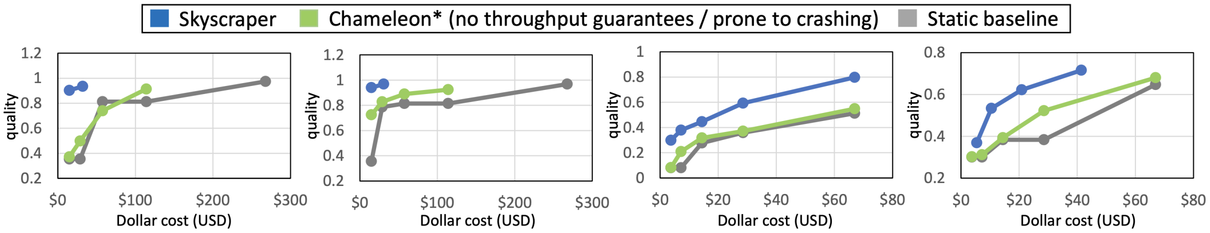


Figure 4: Cost-quality trade-off of *Skyscraper*, Chameleon* and statically using the same knob throughout ingestion.

allows Chameleon to achieve cost savings, since it doesn't need to be provisioned to handle peak workload. However, Chameleon* is not practical and may easily crash, as its lack of throughput guarantees may lead to buffer overflows. We benchmarked Chameleon* on several hardware setups and only report the numbers where it didn't crash during the benchmark.

For each system, we report the overall result quality that the system achieves on different hardware set ups. Since we do not have access to a wide range of compute servers, we use Google Cloud VM instances as the provisioned, always-on hardware ("on-premise servers"). In the case of *Skyscraper*, which additionally uses AWS Lambda, we have verified that the bandwidth and latencies from the Google Cloud VMs to AWS Lambda realistically reflect the ones of commodity on-premise setups. In our experiments, we consider the following Google Cloud machines:

- `e2-standard-4`: 4 vCPUs, 16 GB memory, 0.14 USD/h
- `e2-standard-8`: 8 vCPUs, 32 GB memory, 0.27 USD/h
- `e2-standard-16`: 16 vCPUs, 64 GB memory, 0.54 USD/h
- `e2-standard-32`: 32 vCPUs, 128 GB memory, 1.07 USD/h
- `c2-standard-60`: 60 vCPUs, 240 GB memory, 2.51 USD/h

While these instance types do not possess hardware accelerators (e.g., GPUs), we note that there is nothing fundamental about *Skyscraper* that would prevent users from using hardware different from only CPUs. If a user provisions *Skyscraper* with a server that contains hardware accelerators, the application's UDFs would need to make sure that the hardware accelerators are used when executing the UDF. In the offline phase, *Skyscraper* will then just measure the UDF's runtime and work normally without any modifications.

Figure 4 visualizes the cost of processing the workloads from Section 5.2 with each system. On average, content category changes occured every 42s for COVID, every 43s for MOT, every 30s for MOSEI HIGH, every 24s for MOSEI LONG. However, all workloads had some periods with very frequent category changes and others with few category changes. We pessimistically estimate that the same amount of computing costs 1.8× more when using a Google Cloud VM than when using a provisioned on-premise server (this estimate is high and in favor of the baselines). Thus, the total cost of all systems is given by the cost of renting the Google Cloud VMs divided by 1.8 plus the cost of the AWS Lambda workers.

Summary. Overall, *Skyscraper* offers significantly better cost-quality trade-offs than current approaches. *Skyscraper*'s performance benefits are especially large on the MOT workload: *Skyscraper* is 8.7× cheaper than the static baseline at a comparable quality. Furthermore, *Skyscraper* is 3.7× cheaper than Chameleon* at a better quality. Chameleon* suffered from large profiling overheads. For the COVID and MOT workload, our results are comparable to what the authors report in the Chameleon paper (2-3× speedup over the static baseline at the highest quality level). For the MOSEI workloads, the profiling overheads were especially large since the expensive knob configurations cause large amounts of work.

5.4 Ablation study

To evaluate how much buffering and cloud bursting individually contribute to the cost savings, we run an ablation study where we independently disable them. Running this ablation study on unsimulated hardware is prohibitively expensive (i.e., we need to conduct dozens of measurements as the one in Figure 4), so we can only afford to analyze with simulated results. We use a simple but accurate simulator, that we evaluated on the benchmarked workloads and found to be accurate.

We use two metrics to evaluate the performance of *Skyscraper*:

(1) **The monetary cost** of processing the workload. We hereby also evaluate *Skyscraper* for different cost ratios between the on-premise and the cloud computing. We estimate that a ratio of 1:1.8 between on-premises and AWS Lambda is realistic at the current market prices (this estimate is rather high and in favor of the baselines). We evaluate the monetary cost of four variants of *Skyscraper*:

(1a) *No buffering, no cloud:* We disable both buffering and cloud bursting. Effectively, this corresponds to not switching knob configurations and only using the most qualitative knob configuration that runs in real time on the given on-premise server.

(1b) *Only buffering: Skyscraper* may only use placements that place every task on-premise and can not use the cloud.

(1c) *Only cloud: Skyscraper* may use the cloud but not buffering.

(1d) *Buffering & cloud:* This corresponds to standard *Skyscraper*.

(2) **The amount of work** measured in $core * seconds$ used in the processing. This is independent of whether the computation is buffered or executed on the cloud or on premises. When evaluating the amount of work, we compare *Skyscraper* to two baselines:

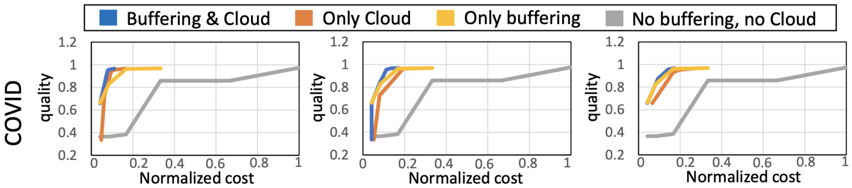
(2a) *Static:* This baseline corresponds to statically using the same knob configuration. It is similar to baseline (1a) where *Skyscraper* also statically uses the same configuration.

(2b) *Skyscraper*: We measure the amount of work that *Skyscraper* performs for processing the workload.

(2c) *Optimum:* The optimum baseline fully leverages the ground truth to always choose the optimal knob configuration. Specifically, given the performance of each knob configuration beforehand, it uses the greedy 0-1 knapsack approximation to choose knob configurations that maximize quality under certain budget.

Figures 6, 8, 10, 12 show the cost-quality trade-off curves for the COVID, MOT, MOSEI-HIGH, and MOSEI-LONG workloads. Figures 7, 9, 11, 13 show the work-quality trade-off curves.

* Chameleon* is an adapted version of Chameleon [31] that uses a buffer. Chameleon* would frequently crash in practice due to overflows of the unmanaged buffer.



a) 1:1 Cloud/on prem cost ratio b) 1.8:1 Cloud/on prem cost ratio c) 5:2 Cloud/on prem cost ratio

Figure 5: Monetary cost comparison for COVID workload

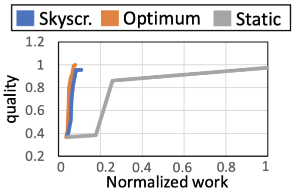
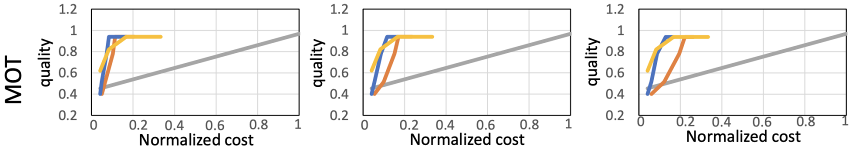


Figure 6: Work (core*s) of COVID workload



a) 1:1 Cloud/on prem cost ratio b) 1.8:1 Cloud/on prem cost ratio c) 5:2 Cloud/on prem cost ratio

Figure 7: Monetary cost comparison for MOT workload

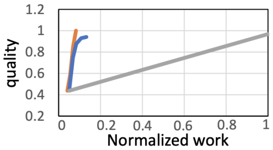
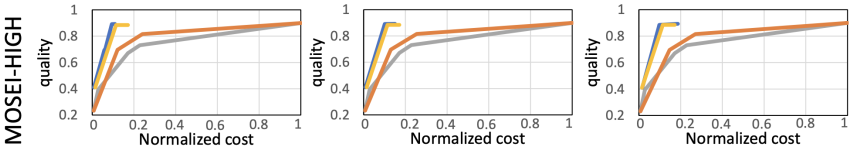


Figure 8: Work (core*s) of MOT workload



a) 1:1 Cloud/on prem cost ratio b) 1.8:1 Cloud/on prem cost ratio c) 5:2 Cloud/on prem cost ratio

Figure 9: Monetary cost comparison for MOSEI-HIGH workload

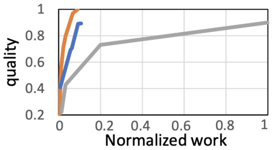
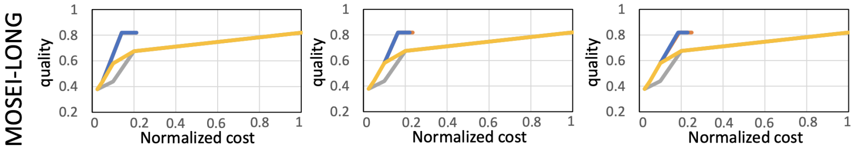


Figure 10: Work (core*s) of MOSEI-HIGH workload



a) 1:1 Cloud/on prem cost ratio b) 1.8:1 Cloud/on prem cost ratio c) 5:2 Cloud/on prem cost ratio

Figure 11: Monetary cost comparison for MOSEI-LONG workload

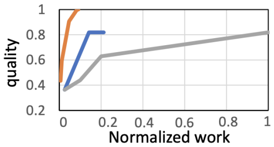


Figure 12: Work (core*s) of MOSEI-LONG workload

For the COVID and MOT workload, *Only cloud* and *Only buffering* alone can achieve significant speed-ups over the baseline. For both workloads, when combining the two (*Buffering & cloud*), peak quality can be roughly reached at 1.5× less cost than when only buffering or only using the cloud for a cost ratio of 1.8:1. For 5:2 cost ratio, *Only cloud* performs significantly worse, because off-loading work off to the cloud incurs a very high cost. For 1:1 cost ratio, *Only cloud* matches the performance of *Buffering & cloud* as using cloud resources has the same cost the on-premises computations.

For the MOSEI workloads, we can see how *Only buffering* and *Only cloud* struggle to deliver good performance for MOSEI-HIGH and MOSEI-LONG, respectively. However, we observe that *Buffering & cloud* delivers good performance on both. The reason for the bad performance of *Only cloud* on MOSEI-HIGH is bandwidth limitations that limit the number of social media streams that can be offloaded to the cloud. The reason for the bad performance of *Only buffering* on MOSEI-LONG is that the buffer gets filled early on, which prevents *Skyscraper* from using expensive knob configurations for the remainder of the long workload peak.

Finally, Figures 7, 9, 11 show that *Skyscraper*'s work reduction method performs close to optimum. *Skyscraper* only leaves large room for improvement for the MOSEI-LONG workload (Figure 13).

Summary. To certain extent, the buffering and cloud bursting optimizations are complementary to each other. Specifically, the performance improvement of using both over using one of them is not as large as performance difference between them. Therefore, cloud bursting lessens the need for buffering and vice versa. However, *Skyscraper* can still achieve 1.5× cost savings in the COVID and MOT workloads over only one of the two methods. Furthermore, the MOSEI workloads show that buffering and cloud bursting struggle for different kinds of workload spiking patterns. By combining the two, *Skyscraper* can achieve good performance for both kinds of patterns.

5.5 Runtime overheads

For the COVID workload, the overall runtime of the offline phase was 1.6 hours on two c2-standard-60 machines. 83% of the time was spent creating the training data for the forecasting model, which is embarrassingly parallel and can be sped up by adding machines.

Skyscraper's *knob planner* and *knob switcher* add overheads to the online execution time. In this section, we evaluate their runtimes for different amounts of placements, content categories, and knob configurations. All runtime measurements are performed on a single core (no parallelization) of the Intel(R) Xeon(R) Gold 6130 CPU with 64 cores at 2.10GHz with 198 GB memory.

The worst-case runtime of the *knob switcher* is linear in the total number of placements (for all knob configurations). This worst case is achieved when the knob switcher needs to iterate through all configuration-placement pairs until it finds one that does not overflow the buffer (see Section 4). The left plot in Figure 13 shows the worst-case runtime as the dashed line and the average runtimes of the *knob switcher* for the COVID, MOT, and MOSEI experiments.

The *knob planner* conducts an inference pass through a small neural network and solves a linear program. For the linear program, the number of variables is $|C| * |\mathcal{K}|$ and the number of constraints is $1 + 2 * |C|$, where C denotes the number of content categories and $\mathcal{K}$ is the number of knob configurations. The right image in

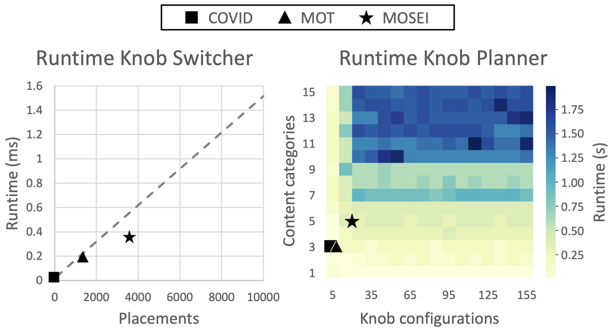


Figure 13: Overheads of knob switcher and planner

Figure 13 uses the heat map to visualize the overheads caused by the *knob planner* for different amounts of content categories and knob configurations. This image also shows the actual runtime of *knob planner* on the three workloads.

Summary. For common problem such as the COVID, MOT, and MOSEI workloads, the overheads of both the *knob switcher* and *knob planner* are negligible. While the *knob switcher* runs every few seconds, its runtime is typically below a millisecond. Similarly, the *knob planner* typically runs every few days but with a runtime below a second. We also show that the runtime overhead of our optimization is reasonable for more complicated workloads.

5.6 Microbenchmarks

This subsection evaluates how accurately *Skyscraper*'s forecasting model $\mathcal{F}$ can predict the future content distribution and how sensitive *Skyscraper*'s performance is to inaccuracies in the forecast. Similarly, the subsection evaluates the accuracy at which the knob switcher classifies the video content into a content category $c \in C$ and how sensitive *Skyscraper*'s performance is to misclassifications. In our evaluation, we focus on the real-world workloads COVID and MOT. The MOSEI workloads are synthetically created by inducing workload spiking patterns as described in Section 5.2. While these workloads present especially difficult spiking patterns for buffering and cloud bursting, the forecasting model achieves 100% accuracy and the knob switcher particularly high performance due to the regularity and smoothness of their workload peaks. We therefore do not evaluate them in terms of accuracy in this subsection.

Forecasting model We evaluate the forecasting model on 8 days of test data after training it on 16 days of unlabeled training data. We train and evaluate the forecasting model on four different lengths of the planned interval: {1, 2, 4, 8} days. As described in Section 4.1, the length of the planned interval determines the frequency of running knob planner and how long $\mathcal{F}$ needs to forecast into the future.

We find that for both workloads, *Skyscraper*'s forecasting method achieves a low Mean Absolute Error (MAE) when forecasting 1 to 4 days into the future. For both workloads, the lowest MAE was achieved when forecasting 2 days into the future, while the largest MAE was incurred when doing so for 8 days.

There is a sweet spot on how far to forecast into the future but this sweet spot is unrelated to the frequency of content category changes. Forecasting over very large time intervals is hard because events far in the future become increasingly uncorrelated to the current events, which the forecast is based on. On the other hand, forecasting over too short time periods is also hard: The streamed video content is always subject to a certain amount of randomness (e.g. a large group of people randomly walking past a camera). Over large enough time intervals, this randomness is smoothed out, which makes the forecast more precise. When this smoothing effect is not achieved, errors due to unforeseen randomness will be noticeable in the MAE of the predictions. The high MAE when forecasting 8 days into the future shows that forecasting far into the future is difficult as events become increasingly uncorrelated to the current events, which the forecast is based on. On the other hand, forecasting over too short time periods also leads to higher MAEs: Streamed video content is always subject to a certain amount of randomness (e.g. a large group of people randomly walking past a camera). Over large enough time intervals, this randomness is smoothed out and therefore doesn't show in the MAE, which doesn't occur for forecasts over short periods.

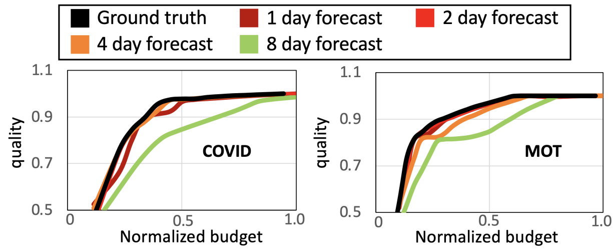


Figure 14: The effect of different planned interval lengths on *Skyscraper*'s end-to-end performance

Figure 14 shows the impact of the prediction errors in terms of end-to-end performance. For comparison, we additionally run *Skyscraper* using the ground truth content distributions (perfect forecast). For planned interval lengths between 1 and 4, *Skyscraper*'s performance is very close to the optimal performance using the ground truth predictions. However, for both workloads *Skyscraper* performs significantly worse for a planned interval length of 8.

Knob switcher As described in Section 4.2, it is possible that the knob switcher misclassifies video content into the wrong content category. We identify two reasons for such misclassifications. First, the knob switcher classifies content based on the quality of one knob configuration. This corresponds to KMeans classification, where a vector is classified using only one dimension instead of all. We denote misclassifications, that occur because of this as *Type-A errors*. Second, the knob switcher determines the current content category based on the past couple of seconds of the video. It will then switch to a knob configuration that is used for processing the next couple of seconds of video, which creates a time mismatch. The last couple of seconds might belong to a different content category than the next couple of seconds. We denote errors caused by this timing mismatch as *Type-B errors*. Distinguishing between these two errors lets us gain insight into where performance losses come from, which could be used for further enhancements of *Skyscraper*.

In Figure 15, we denote the standard knob switcher as described in Section 4.2 as *Standard* and compare it against two baselines: *Ground truth* denoting *Skyscraper* using the ground truth content categories and *No Type-B errors* denotes a baseline that partially uses the ground truth to eliminate errors of Type-B. Specifically, it determines the content category using *Skyscraper*'s standard approach but on the data of a future couple of seconds (i.e., it knows how the current knob configuration would perform in the

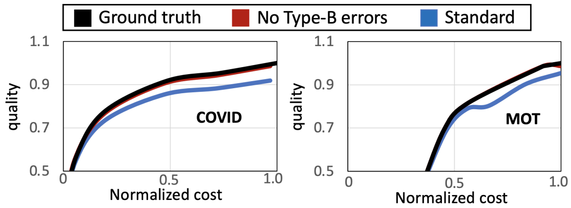


Figure 15: End-to-end performance of knob switcher against baselines that leverage ground truth for content classification

next couple of seconds without executing it). Like this, only errors of Type-A impede the performance of the *No Type-B errors* baseline, which shows their impact on *Skyscraper*'s end-to-end performance.

Figure 15 shows that the knob switcher's misclassifications have a negative impact on *Skyscraper*'s end-to-end performance when using the *Standard*. The misclassification rate of *Standard* is 2.1% on COVID and 6.6% on the MOT workload. However, the performance of the *No Type-B errors* baseline almost matches the *optimum*. This suggests that the remaining Type-A errors barely impede the overall performance. These errors constitute 0.5% of the knob switcher's error rate on COVID and 3.7% on the MOT workload.

Summary The microbenchmarks provide two insights. First, when forecasting between 1 to 4 days into the future, *Skyscraper*'s forecasting method is accurate and does not significantly harm end-to-end performance when compared to using the ground truth as forecast. However, when forecasting further into the future (e.g., 8 days), the forecasts become less accurate, which shows an effect on *Skyscraper*'s end-to-end performance. Second, misclassifications of the knob switcher negatively impact *Skyscraper*'s performance. We hereby identify a time mismatch as the sole driver for the performance losses. This timing mismatch occurs because the knob configuration to process the next couple of seconds with is based on the content of the last couple of seconds.

6 RELATED WORK

While we are not aware of past research which manages video streams like in a data warehouse, several systems propose end-to-end solutions for managing archived collections of video like in a relational database system [16, 21, 41, 56, 57]. Likewise, we are not aware of past work that directly addresses the *V-ETL* problem, but there are several lines of work on efficient video processing that are relevant to *Skyscraper*. We summarize them below.

Content-adaptive knob tuning systems. Content-adaptive knob tuning systems aim at saving computational work by dynamically adjusting knobs that are inherent to CV workloads to the video stream's content. Chameleon performs content-adaptive knob tuning for general CV workloads [31]. However, Chameleon assumes that each knob configuration can be run in real-time on the provisioned hardware resources ("peak provisioning"). Chameleon then minimizes the average processing time per frame. As discussed in Section 1, such systems cannot deliver cost savings while also adhering to throughput guarantees, which is required in the *V-ETL* problem. Zeus is another content-adaptive knob tuning system [12], but cannot be used for general-purpose *V-ETL*, as it is specific to action detection (e.g., detect someone crossing the street).

Query-load-adaptive knob tuning systems. Instead of adapting to the streamed content, some systems tune the knobs of a CV workload solely based on the concurrently running queries (while being agnostic to the streamed content). These systems are useful in scenarios where users issue dynamic queries over video streams, which require the system to dynamically multiplex compute resources among the queries. VideoStorm [59] and VideoEdge [26] go beyond dynamic resource allocation and also tune the queries' knobs based on the other queries that are concurrently running. However, in scenarios where the query load remains static, there is no benefit in dynamically adapting to the query load. In *V-ETL*, a constant set of jobs is used to ingest the video streams. In contrast to VideoStorm and VideoEdge, *Skyscraper* therefore dynamically adapts to changes in the video content instead of the query load.

Streaming ETL. Treating data warehouse ingestion as a stateful stream processing problem is an established approach [18], which is successfully used in many big data applications [44]. Like *Skyscraper*, traditional streaming ETL is also concerned with maintaining data quality while handling fluctuating workloads without peak provisioning. This is typically achieved through methods like back pressure or load shedding, which mitigate workload peaks arising from fluctuating volumes of arriving data [52]. However, in *V-ETL*, data often arrives at constant volume, and only the content of the data changes. In contrast to traditional streaming systems, *Skyscraper* 's optimizations therefore focus on adapting to the content of the streamed data and not to its volume.

General-purpose cloud offloading. Several works have previously explored the idea of offloading work from an on-premise server to on-demand cloud workers [1, 13, 15, 19, 28, 36, 38, 60]. These works assume that jobs occasionally arrive and these jobs may be executed locally or offloaded to the cloud. However, these works only optimize the placement of work and do not reduce work by means like knob tuning as *Skyscraper* does.

Task-specific computer vision optimizations. Several works optimize the application of CV for specific tasks and queries. While these methods cannot be used to optimize arbitrary *V-ETL* jobs, they can be used inside *Skyscraper*'s UDFs to further reduce cost. General methods to improve the efficiency of neural networks include model compression [20, 37], compact neural architectures, [27, 39, 47], and knowledge distillation [4, 24, 33, 53]. Further works propose efficient CV primitives that are query-aware or content-adaptive [3, 7–9, 30, 33, 55]. Finally, some works reduce processing costs of certain video queries by intelligently skipping frames [6, 22, 25, 32, 34, 35, 42, 45, 58].

7 CONCLUSION

In this paper, we defined the problem of *V-ETL* for transforming video streams to a queryable format through expensive ML-based video processing DAGs. In response, we introduced *Skyscraper*, which uses content-adaptive knob tuning to reduce the cost of the *V-ETL* Transform step while adhering to *V-ETL*'s throughput requirements on constrained hardware resources. *Skyscraper* supports conversions to arbitrary query formats.

ACKNOWLEDGMENTS

We thank the Data Systems and Artificial Intelligence Lab (DSAIL) for supporting this work.

REFERENCES

[1] Sándor Ács, Miklós Kozlovszky, and Péter Kacsuk. 2014. A novel cloud bursting technique. *2014 IEEE 9th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)* (2014), 135–138.

[2] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2019. Placeto: Learning generalizable device placement algorithms for distributed machine learning. *arXiv preprint arXiv:1906.08879* (2019).

[3] Michael R. Anderson, Michael Cafarella, German Ros, and Thomas F. Wenisch. 2019. Physical Representation-Based Predicate Optimization for a Visual Analytics Database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE. https://doi.org/10.1109/icde.2019.00132

[4] Rohan Anil, Gabriel Pereyra, Alexandre Passos, Róbert Ormándi, George E. Dahl, and Geoffrey E. Hinton. 2018. Large scale distributed neural network training through online distillation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rkr1UDeC-

[5] AmirAli Bagher Zadeh, Paul Pu Liang, Soujanya Poria, Erik Cambria, and Louis-Philippe Morency. 2018. Multimodal Language Analysis in the Wild: CMU-MOSEI Dataset and Interpretable Dynamic Fusion Graph. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Melbourne, Australia, 2236–2246. https://doi.org/10.18653/v1/P18-1208

[6] Jaeho Bang, Pramod Chunduri, and Joy Arulraj. 2021. EKO: Adaptive Sampling of Compressed Video Data. https://doi.org/10.48550/ARXIV.2104.01671

[7] Favyen Bastani, Songtao He, Arjun Balasingam, Karthik Gopalakrishnan, Mohammad Alizadeh, Hari Balakrishnan, Michael Cafarella, Tim Kraska, and Sam Madden. 2020. MIRIS: Fast Object Track Queries in Video. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1907–1921. https://doi.org/10.1145/3318464.3389692

[8] Favyen Bastani and Samuel Madden. 2022. OTIF: Efficient Tracker Pre-Processing over Large Video Datasets. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2091–2104. https://doi.org/10.1145/3514221.3517835

[9] Jiashen Cao, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. 2021. THIA: Accelerating Video Analytics using Early Inference and Fine-Grained Query Planning. https://doi.org/10.48550/ARXIV.2102.08481

[10] CCITT. 1992. Digital compression and coding of continuous-tone still images - requirements and guidelines. https://www.w3.org/Graphics/JPEG/itu-t81.pdf.

[11] Peng Chu, Jiang Wang, Quanzeng You, Haibin Ling, and Zicheng Liu. 2021. TransMOT: Spatial-Temporal Graph Transformer for Multiple Object Tracking. https://doi.org/10.48550/ARXIV.2104.00194

[12] Pramod Chunduri, Jaeho Bang, Yao Lu, and Joy Arulraj. 2022. Zeus: Efficiently Localizing Actions in Videos Using Reinforcement Learning. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 545–558. https://doi.org/10.1145/3514221.3526181

[13] Li Chunlin, Tang Jianhang, and Luo Youlong. 2019. Hybrid Cloud Adaptive Scheduling Strategy for Heterogeneous Workloads. *Journal of Grid Computing* 17, 3 (01 Sep 2019), 419–446. https://doi.org/10.1007/s10723-019-09481-3

[14] A. Criminisi, I. Reid, and A. Zisserman. 1999. A plane measuring device. *Image and Vision Computing* 17, 8 (1999), 625–634. https://doi.org/10.1016/S0262-8856(98)00183-8

[15] A. Das, A. Leaf, C. A. Varela, and S. Patterson. 2020. Skedulix: Hybrid Cloud Scheduling for Cost-Efficient Execution of Serverless Applications. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE Computer Society, Los Alamitos, CA, USA, 609–618. https://doi.org/10.1109/CLOUD49709.2020.00090

[16] Maureen Daum, Brandon Haynes, Dong He, Amrita Mazumdar, and Magdalena Balazinska. 2021. TASM: A tile-based storage manager for video analytics. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1775–1786.

[17] P. Dendorfer, H. Rezatofighi, A. Milan, J. Shi, D. Cremers, I. Reid, S. Roth, K. Schindler, and L. Leal-Taixé. 2020. MOT20: A benchmark for multi object tracking in crowded scenes. *arXiv:2003.09003[cs]* (March 2020). http://arxiv.org/abs/1906.04567 arXiv: 2003.09003.

[18] Lukasz Golab and Theodore Johnson. 2013. Data stream warehousing. In *ACM SIGMOD Conference*. 949–952.

[19] Tian Guo, Upendra Sharma, Timothy Wood, Sambit Sahu, and Prashant Shenoy. 2012. Seagull: Intelligent Cloud Bursting for Enterprise Applications. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 361–366. https://www.usenix.org/conference/atc12/technical-sessions/presentation/guo

[20] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1510.00149

[21] Brandon Haynes, Maureen Daum, Dong He, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. 2021. Vss: A storage system for video analytics. In *Proceedings of the 2021 International Conference on Management of Data*. 685–696.

[22] Wenjia He, Michael R. Anderson, Maxwell Strome, and Michael Cafarella. 2020. A Method for Optimizing Opaque Filter Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1257–1272. https://doi.org/10.1145/3318464.3389766

[23] João F. Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. 2015. High-Speed Tracking with Kernelized Correlation Filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37, 3 (2015), 583–596. https://doi.org/10.1109/TPAMI.2014.2345390

[24] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the Knowledge in a Neural Network. *CoRR* abs/1503.02531 (2015). arXiv:1503.02531 http://arxiv.org/abs/1503.02531

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

[26] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodík, Leana Golubchik, Minlan Yu, Victor Bahl, and Matthai Philipose. 2018. VideoEdge: Processing Camera Streams using Hierarchical Clusters. In *ACM/IEEE Symposium on Edge Computing (SEC)* (acm/ieee symposium on edge computing (sec) ed.). https://www.microsoft.com/en-us/research/publication/videoedge-processing-camera-streams-using-hierarchical-clusters/

[27] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and lt;0.5MB model size. https://doi.org/10.48550/ARXIV.1602.07360

[28] Mohammad A. Ibrahim, Gamal A. Ebrahim, and Hoda K. Mohamed. 2017. A modern cloud bursting framework. In *2017 12th International Conference on Computer Engineering and Systems (ICCES)*. 148–153. https://doi.org/10.1109/ICCES.2017.8275294

[29] IETF. 2006. The Base16, Base32, and Base64 Data Encodings. https://datatracker.ietf.org/doc/html/rfc4648 (accessed on 7 March 2023).

[30] Samvit Jain, Xun Zhang, Yuhao Zhou, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Victor Bahl, and Joseph Gonzalez. 2020. Spatula: Efficient cross-camera video analytics on large camera networks. In *ACM/IEEE Symposium on Edge Computing (SEC 2020)*. https://www.microsoft.com/en-us/research/publication/spatula-efficient-cross-camera-video-analytics-on-large-camera-networks/

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

[32] Daniel Kang, Peter Bailis, and Matei Zaharia. 2018. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. (2018). https://doi.org/10.48550/ARXIV.1805.01046

[33] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Neural Network Queries over Video at Scale. https://doi.org/10.48550/ARXIV.1703.02529

[34] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, Yi Sun, and Matei Zaharia. 2021. Accelerating Approximate Aggregation Queries with Expensive Predicates. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2341–2354. https://doi.org/10.14778/3476249.3476285

[35] Daniel Kang, John Guibas, Peter D. Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2022. TASTI: Semantic Indexes for Machine Learning-Based Queries over Unstructured Data. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1934–1947. https://doi.org/10.1145/3514221.3517897

[36] Young Choon Lee and Bing Lian. 2017. Cloud Bursting Scheduler for Cost Efficiency. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. 774–777. https://doi.org/10.1109/CLOUD.2017.112

[37] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient ConvNets. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJqFGTslg

[38] Rui Li, Zhi Zhou, Xu Chen, and Qing Ling. 2022. Resource Price-Aware Offloading for Edge-Cloud Collaboration: A Two-Timescale Online Control Approach. *IEEE Transactions on Cloud Computing* 10, 1 (2022), 648–661. https://doi.org/10.1109/TCC.2019.2937928

[39] Min Lin, Qiang Chen, and Shuicheng Yan. 2014. Network In Network. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada,*

April 14-16, 2014, Conference Track Proceedings, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1312.4400

[40] S. Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (1982), 129–137. https://doi.org/10.1109/TIT.1982.1056489

[41] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. 2016. Optasia: A relational platform for efficient large-scale video analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 57–70.

[42] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/3183713.3183751

[43] M D McKay. 1995. Evaluating prediction uncertainty. (3 1995). https://doi.org/10.2172/29432

[44] John Meehan, Cansu Aslantas, Stan Zdonik, Nesime Tatbul, and Jiang Du. 2017. Data Ingestion for the Connected World. In *CIDR*.

[45] Oscar Moll, Favyen Bastani, Sam Madden, Mike Stonebraker, Vijay Gadepally, and Tim Kraska. 2020. ExSample: Efficient Searches on Video Repositories through Adaptive Sampling. https://doi.org/10.48550/ARXIV.2005.09141

[46] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 561–577. https://www.usenix.org/conference/osdi18/presentation/moritz

[47] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 779–788. https://doi.org/10.1109/CVPR.2016.91

[48] Robert Rich and Joseph Tracy. 2003. Modeling Uncertainty: Predictive Accuracy as a Proxy for Predictive Confidence. *SSRN Electronic Journal* (02 2003). https://doi.org/10.2139/ssrn.377462

[49] Iain E. G. Richardson. 2003. *H.264 and MPEG-4 video compression : video coding for next generation multimedia*. Chichester; Hoboken, NJ: Wiley.

[50] Stuart Russell and Peter Norvig. 2003. Artificial Intelligence: A Modern Approach, 2nd Edition. *Pearson* (2003).

[51] Amazon Web Services. 2023. AWS Lambda. https://aws.amazon.com/lambda/ (accessed on 24 Jan 2023).

[52] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load Shedding in a Data Stream Manager. In *VLDB Conference*. 309–320.

[53] Gregor Urban, Krzysztof J. Geras, Samira Ebrahimi Kahou, Özlem Aslan, Shengjie Wang, Abdelrahman Mohamed, Matthai Philipose, Matthew Richardson, and Rich Caruana. 2017. Do Deep Convolutional Nets Really Need to be Deep and Convolutional?. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=r10FA8Kxg

[54] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[55] Li Wang, Yao Lu, Hong Wang, Yingbin Zheng, Hao Ye, and Xiangyang Xue. 2017. Evolving boxes for fast vehicle detection. In *2017 IEEE International Conference on Multimedia and Expo (ICME)*. 1135–1140. https://doi.org/10.1109/ICME.2017.8019461

[56] Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. 2019. Vstore: A data store for analytics on large videos. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[57] Zhuangdi Xu, Gaurav Tarlok Kakkar, Joy Arulraj, and Umakishore Ramachandran. 2022. EVA: A Symbolic Approach to Accelerating Exploratory Video Analytics with Materialized Views. In *Proceedings of the 2022 International Conference on Management of Data*. 602–616.

[58] Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X. Sean Wang. 2022. Optimizing Machine Learning Inference Queries with Correlative Proxy Models. *Proc. VLDB Endow.* 15, 10 (sep 2022), 2032–2044. https://doi.org/10.14778/3547305.3547310

[59] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. 2017. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 377–392. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang

[60] Andrii Zhygmanovskyi and Norihiko Yoshida. 2015. Distributed Cloud Bursting Model Based on Peer-to-Peer Overlay. In *2015 3rd International Conference on Future Internet of Things and Cloud*. 823–828. https://doi.org/10.1109/FiCloud.2015.74