# OTIF: Efficient Tracker Pre-processing over Large Video Datasets

Favyen Bastani
Massachusetts Institute of Technology
US
favyen@csail.mit.edu

Samuel Madden
Massachusetts Institute of Technology
US
madden@csail.mit.edu

## ABSTRACT

Performing analytics tasks over large-scale video datasets is increasingly common in a wide range of applications, from traffic planning to sports analytics. These tasks generally involve object detection and tracking operations that require pre-processing the video through expensive machine learning models. To address this cost, several video query optimizers have recently been proposed. Broadly, these methods trade large reductions in pre-processing cost for increases in query execution cost: during query execution, they apply query-specific machine learning operations over portions of the video dataset. Although video query optimizers reduce the overall cost of executing a single query over large video datasets compared to naive object tracking methods, executing several queries over the same video remains cost-prohibitive; moreover, the high per-query latency makes these systems unsuitable for exploratory analytics where fast response times are crucial.

In this paper, we present OTIF, a video pre-processor that efficiently extracts all object tracks from large-scale video datasets. By integrating several optimizations under a joint parameter tuning framework, *OTIF is able to extract all object tracks from video as fast as existing video query optimizers can execute just one single query.* In contrast to the outputs of video query optimizers, OTIF's outputs are general-purpose object tracks that can be used to execute many queries with sub-second latencies. We compare OTIF against three recent video query optimizers, as well as several general-purpose object detection and tracking techniques, and find that, across multiple datasets, OTIF provides a 6x to 25x average reduction in the overall cost to execute five queries over the same video.

## CCS CONCEPTS

• **Information systems → Multimedia databases**.

## KEYWORDS

video query, video analytics

**ACM Reference Format:**
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 (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3517835

*SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA*
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3517835

## 1 INTRODUCTION

Over the last decade, improvements in machine learning methods, especially in convolutional neural networks (CNNs), have enabled numerous applications that involve querying large-scale video data. In particular, CNNs have been applied to accurately extract object detections (bounding box positions of objects) and tracks (sequences of bounding boxes over time) from video. Detections and tracks are used in virtually all video analytics tasks, such as in traffic planning to conduct turning movement counts [6] (counting the number of cars turning in each direction in each time interval), in autonomous vehicle development to identify signs [28], and in sports analytics to derive statistics from the motion of players and balls [24].

However, object detection methods are GPU-intensive: for example, on the $10,000 NVIDIA Tesla V100 GPU, the YOLOv3 object detector [20] can process $960 \times 540$ video frames at 100 frames per second (fps). A user with a large volume of video that needs to be processed, say from hundreds of traffic cameras, would require one GPU for every 3-4 video feeds captured at 30 fps. We could obtain some speedup by reducing the resolution and sampling rate at which the detector processes video, e.g., sampling only five $640 \times 480$ frames per second of video. However, this provides a limited speedup: accuracy drops off rapidly once the resolution is reduced far enough that the objects of interest occupy only a few pixels in the frame, or the sampling rate is low enough that objects move long distances between successively sampled frames.

Thus, recent work proposes video query optimizers that incorporate new approximate optimization techniques for efficiently analyzing video [1, 10–12]. However, while these systems substantially reduce pre-processing cost, they incorporate slow query-specific execution phases where portions of video are selected for processing through expensive models. This introduces substantial per-query latency: then, not only are these systems unsuitable for exploratory analytics where fast response times are crucial, but they are also no faster than naively processing every frame when executing several queries over the same video. For example, to optimize frame-level limit queries, TASTI [12] first pre-processes video to build a query-agnostic index, but then conducts a search phase that needs to be repeated per-query and involves repeatedly applying an expensive detector; the query-specific phase may require several minutes or even hours depending on the desired output cardinality and index precision, which is unacceptably long for exploratory queries.

Rather than push computation from pre-processing to query time, we propose instead focusing on minimizing the pre-processing time needed to accurately extract object tracks from video. While prior work has studied fast object tracking [4, 17, 22, 27], these methods optimize execution time on one dimension instead of considering multiple avenues for obtaining large speedups (e.g., reducing resolution, reducing framerate, and using proxy models).

In this paper, we present OTIF, a video pre-processor that efficiently extracts all object tracks from large-scale video datasets. Users can efficiently conduct exploratory analytics tasks by post-processing the tracks computed by OTIF, without requiring further video decoding or ML inference. To provide a superior speed-accuracy curve when extracting tracks from video, OTIF (1) incorporates novel adaptations of two video analytics optimizations proposed in prior work; and (2) integrates these methods into a cohesive system that jointly tunes multiple parameters to provide a greater speedup while introducing less error than prior approaches.

We now detail these two aspects of our method. First, we develop novel variants of two optimizations, *proxy models* [10, 11] and *reduced-rate tracking* [1], to improve their robustness and speed by incorporating recent progress in computer vision techniques. Prior work in *proxy models* (NoScope [11]) trains fast classification models to input low-resolution video and estimate whether or not each frame contains at least one object detection; these models are then employed to skip execution of the slower detector model on frames where the proxy model has high confidence that there are no objects. However, many video datasets consist of busy scenes where there are objects in every frame, and proxy models provide no speedup. We extend the proxy model method to a multi-scale detection context [5], where we use a proxy model to not only determine which frames contain objects, but also which spatial regions of frames contain objects. Then, even in videos of busy scenes, our method can still yield a speedup by only applying the slower detector in small windows of the frame that contain objects.

Prior work in *reduced-rate tracking* (Miris [1]) proposes techniques to process video at substantially reduced sampling rates while still extracting accurate tracks. However, tracking models used in prior work are limited: they only consider matching detections between pairs of frames at a time (the previous frame and a new frame), and form tracks by creating chains of matches. Thus, the model cannot leverage useful cues such as object motion (e.g., velocity) that require analyzing multiple previous detections of an object. We instead employ a recurrent model that is able to account for information in multiple previous frames when matching detections in a new frame, and address challenges to apply such a model in a reduced-rate tracking framework.

Second, we integrate these two novel techniques, along with a simple detector resolution optimization, into a cohesive system by applying a parameter tuning algorithm to choose multiple parameters across the three optimization methods, including the proxy model threshold, tracking sampling rate, and detector resolution.

We evaluate OTIF on 7 diverse video datasets, and compare its performance in terms of speed and accuracy against three video query optimizers (BlazeIt [10], TASTI [12], and Miris [1]) and four object detection and tracking baselines (NoScope [11], Chameleon [9], CaTDet [17], and CenterTrack [26]) on both frame-level and object track queries. Most significantly, we find that OTIF is able to extract all tracks from video as fast as video query optimizers can execute just one query. Since OTIF's outputs are reusable across multiple queries, OTIF provides a 6x to 25x average speedup at executing 5 queries over the same video, at the same level of accuracy. We release the OTIF source code at https://github.com/favyen/otif.

In summary, our contributions are:

- We develop two novel video analytics optimization techniques, segmentation proxy models and recurrent reduced-rate tracking.
- We integrate these and other optimizations in a cohesive system that greedily tunes multiple parameters to select parameters providing high speed at every accuracy level.
- We show that OTIF is able to extract all tracks from video as fast as current video query optimizers can execute a single query. Thus, when executing multiple queries, OTIF provides a substantial speedup.

## 2 RELATED WORK

**Video Analytics.** Several systems have recently been proposed for performing video analytics tasks over large-scale video datasets. NoScope [11] and Probabilistic Predicates [16] propose training a classification proxy model to input low-resolution video frames and quickly determine whether the frame is relevant to the query; processing of expensive object detectors can be skipped on frames where the proxy model is confident it does not match the query.

Video Monitoring Queries [14], Focus [8], and several other works propose various extensions to the proxy model technique. In particular, BlazeIt [10] proposes techniques specialized for efficiently executing limit and aggregate queries by post-processing the proxy model outputs. However, proxy models in BlazeIt are generally specialized for individual queries; thus, repeated per-query video decoding and inference makes BlazeIt costly when executing multiple queries. TASTI [12] proposes splitting the proxy model into two components: a feature extractor that processes video frames into embeddings that describe the objects that likely appear in the frame, and a scoring model that determines how likely a frame matches a query based on its embedding. The embeddings are query-agnostic, so the feature extractor only needs to be applied once over the video; however, since the detector must still be applied on many video frames during query execution for most query types, TASTI remains costly when executing repeated queries over the same video. In Section 4, we show that OTIF's tracker pre-processing outperforms NoScope, BlazeIt, and TASTI even when executing just one query.

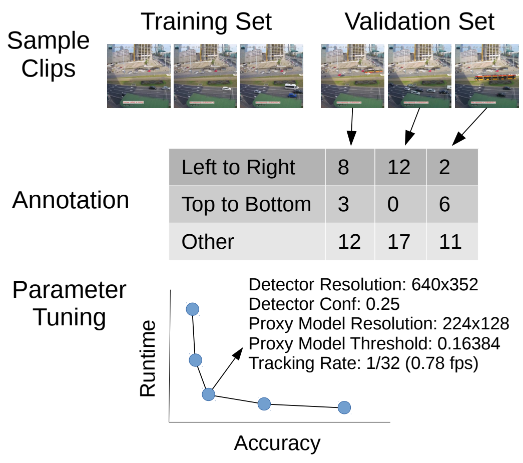
Tuning parameters for video storage and decoding to optimize analytics performance has also been studied in systems such as VStore [23]. Our work complements these methods by additionally tuning parameters in the execution and inference pipeline.

Other techniques for optimizing video analytics tasks have been explored as well. Chameleon [9] proposes optimizing the object detector input resolution and sampling framerate to robustly extract detections from a video dataset. Miris [1] proposes a variable rate tracking method that processes video at substantially reduced framerates when possible to accurately answer object track queries. In Section 4, we show that OTIF outperforms these methods as well.

**Multi-Object Tracking.** Our work is also related to object tracking methods such as Deep Tracklet Association [25] and Bilinear-LSTM [13]. In contrast to these methods, which are designed for high-accuracy but slow tracking in high-resolution video, OTIF focuses on providing a good speed-accuracy tradeoff. More closely related to our work, several methods study efficiently tracking objects through techniques such as updating object positions using

| Symbol | Description |
|---|---|
| $s_i$ | An object track. |
| $d_i$ | An object detection. |
| $I_t$ | Video frame at timestamp $t$. |
| $\theta$ | A configuration (settings for OTIF parameters). |
| $\theta_{\text{best}}$ | Configuration providing highest accuracy. |
| $B_{\text{proxy}}$ | Confidence threshold for segmentation proxy model. |
| $W$ | Window sizes used for detector execution. |
| $R_t$ | Rectangles in which we apply the detector on $I_t$. |
| $S^*$ | Tracks computed using $\theta_{\text{best}}$ in the training set. |
| $g$ | Sampling gap: only process 1 in every $g$ frames. |
| $C$ | Parameter tuning granularity. |

**Table 1: Summary of key notation.**



| | | | |
|---|---|---|---|
| Left to Right | 8 | 12 | 2 |
| Top to Bottom | 3 | 0 | 6 |
| Other | 12 | 17 | 11 |

Detector Resolution: 640x352
Detector Conf: 0.25
Proxy Model Resolution: 224x128
Proxy Model Threshold: 0.16384
Tracking Rate: 1/32 (0.78 fps)

**Figure 1: Overview of the OTIF workflow. 60 one-minute clips are sampled from the dataset to form training and validation sets. The user provides ground truth data in each validation clip, e.g. by annotating counts of objects following each of three spatial patterns. The tuner outputs a sequence of parameter configurations that offer a tradeoff between speed and accuracy. The user selects one configuration (one point along curve) to apply on the entire dataset.**

optical flow estimates [3, 4, 15, 22, 27]. We will show that OTIF outperforms several of these methods, including CaTDet [17] and CenterTrack [26], by integrating novel extensions of several optimization methods in a joint parameter tuning framework.

**Foreground Extraction and Multi-Scale Detection** Our segmentation proxy model method is related to several computer vision techniques. Foreground extraction methods seek to segment the foreground and background of an image for image editing and other tasks [21]. While these methods employ segmentation models similar to our work, they do not address tuning the model to complement a detector for efficient execution. Multi-scale detection methods, including Dynamic Zoom-in Networks [5] and AutoFocus [18], propose to optimize object detection speed with a coarse-to-fine architecture, where objects are first detected in a down-sampled image, and portions of an image are processed at higher resolutions if the model predicts this will yield an accuracy gain. Our method adapts these techniques for video query execution in diverse datasets, incorporating extensions such as dynamic resolution and window size selection.

## 3 OTIF

OTIF is a general-purpose video pre-processor for exploratory video analytics tasks that involve object detections or tracks. Given a video dataset, OTIF efficiently and accurately extracts all object tracks from the video: its output is a set of tracks $\{s_1, \ldots, s_n\}$, where each track $s_i = (C_k, \langle d_1, \ldots, d_{m_i} \rangle)$ is a unique object of some category $C_k$ (e.g., car or pedestrian) represented as a sequence of detections, and each detection $d_i = (t, x, y, w, h)$ specifies a timestamp $t$ and a bounding box where the object appears. After pre-processing video with OTIF, users can rapidly answer queries by post-processing the computed tracks, without needing additional video decoding or ML inference. Example queries that can directly be answered in traffic camera video from extracted object tracks include: (1) find cars that decelerate at $5\text{m/s}^2$ or more (hard braking); (2) find frames with at least three buses and three cars; (3) find the average number of cars visible in the video over time; (4) find the average number of *unique* cars over time (i.e., the traffic volume).

We summarize key notation we use in this section in Table 1.

### 3.1 Workflow

Before detailing OTIF's design, in this section we first describe the workflow of applying OTIF on a new video dataset (Figure 1). Users first uniformly randomly sample training and validation sets from the dataset, which each consist of many sampled clips of a certain length—in our implementation, we sample one hour of video consisting of 60 one-minute clips for each of the training and validation sets. OTIF uses the training set to train proxy models, and uses the validation set to select parameters.

The user then provides a metric and corresponding ground truth for evaluating the accuracy of tracks extracted by various OTIF parameter configurations in each validation clip. The ground truth data may be hand-labeled, or may be automatically computed through an "oracle" pipeline where we apply an object detector and tracker at the native video resolution and framerate.

The OTIF parameter tuner will then experiment with various parameter configurations and evaluate the speed and accuracy when executing the pipeline under each configuration over the validation set. The tuner begins with the slowest possible configuration (which may or may not yield the highest accuracy), and then greedily selects parameters that yield speedups with the smallest reductions in accuracy. The output of this process is a speed-accuracy curve, where each point along the curve corresponds to one parameter configuration. The user can then select a point on the speed-accuracy curve, and OTIF will extract tracks over the entire video dataset using the corresponding parameters.

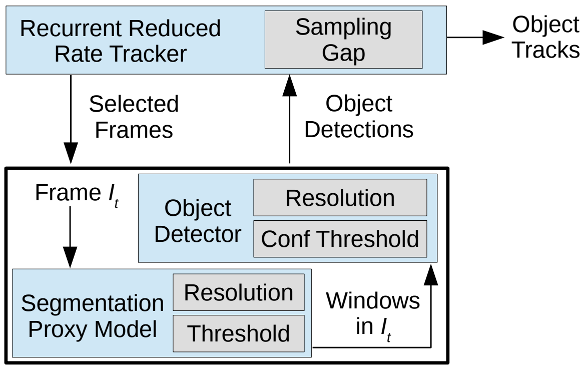


**Figure 2: OTIF execution pipeline architecture. The tracker selects which frames to process. To detect objects in each sampled frame, the segmentation proxy model determines which windows of the frame may contain objects, and the detector runs in those windows. Parameters selected by the tuner are shown in grey.**

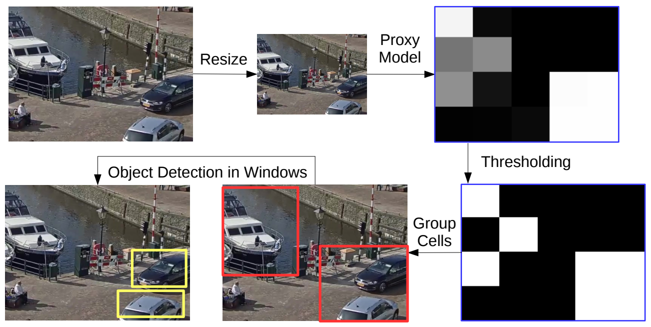


**Figure 3: Summary of our novel segmentation proxy model method. A proxy model inputs a video frame at a low resolution, and scores each cell in the frame with the likelihood that the cells intersects a detection. Positive cells after thresholding are grouped into rectangular windows, and the object detector is applied only in those windows.**

## 3.2 Architecture

We now introduce the OTIF architecture (Figure 2) at a high level. The execution pipeline consists of three modules, where each module exposes several parameters that influence speed and accuracy. First, a segmentation proxy model determines which frames and which parts of frames contain objects, so that a more expensive object detector can be executed only on those regions. This module is configured with the input resolution of the proxy model, and a threshold on the proxy model confidence that determines how confident the model must be before skipping the processing of portions of frames. Second, the detection module applies an object detection model, and is configured with the model architecture (e.g., YOLO [19] or Mask R-CNN [7]), input resolution, and detection confidence threshold. Lastly, sitting on top of the other two modules, the recurrent reduced-rate tracking method decides which frames should be processed for computing object detections, and groups detections of the same object across different frames to produce object tracks. The tracking module is configured with a sampling gap that specifies the rate at which frames should be processed.

Additionally, OTIF includes a parameter tuner that outputs a speed-accuracy curve of parameter configurations using a greedy algorithm. After training and validation sets are sampled, and the evaluation metric for the validation clips is provided by the user, OTIF initializes by training a range of proxy models and a recurrent tracking model, and then executing the tuner.

Below, we introduce our novel segmentation proxy model method in Section 3.3 and our recurrent reduced-rate tracking method in Section 3.4. We then detail the OTIF tuner in Section 3.5.

## 3.3 Segmentation Proxy Model

In prior work, proxy models are applied to determine which frames should be processed to compute object detections. For example, NoScope [11] trains a proxy model to classify whether or not a video frame contains at least one object. Then, NoScope skips object detection processing on frames where the proxy model has sufficiently low confidence. The proxy model is substantially faster than the object detector since it inputs video at a lower resolution, and, to a lesser extent, since it employs a shallower model architecture; thus, this yields a speedup in videos where a large fraction of frames contain zero objects. However, we find that many video datasets contain relevant objects in every frame — for example, video of a traffic junction may continuously contain cars if the junction is busy. Classification proxy models provide no speedup in such videos since no frames can be skipped entirely.

Intuitively, though, proxy models still could provide a benefit by identifying *regions* of frames that contain no objects, and skipping object detection processing on those regions. Then, if the video contains many segments where the camera frame is sparsely populated by objects, this method can provide a substantial speedup: as long as the proxy model can accurately distinguish regions with objects from regions without objects at a lower resolution than that at which the object detector can accurately compute bounding box detections, then although the detector must still be applied on each frame, it can be applied only in small windows where the proxy model determines that objects are present. For example, in Figure 3, the proxy model is confident that only the top-left and bottom-right regions of the frame contain objects, and so the detector can be applied only on those small regions.

In this section, we detail our novel segmentation proxy model method that implements this idea. We employ a segmentation CNN model architecture, which processes an image and outputs a score at each grid cell of pixels in the image (e.g., every $32 \times 32$ cell). We train the model to classify whether each cell intersects at least one detection. Then, during inference, we aim to only apply the detector on "positive" cells where the proxy model has high confidence, i.e., where the score exceeds a threshold parameter $B_{\text{proxy}}$.

However, there are several challenges with applying the detector in this way. First, objects may span multiple adjacent cells, and the object detector can only be efficiently applied on GPUs on rectangular inputs. Thus, after using the proxy model to determine a set of positive cells that may contain detections, we must aggregate these cells together into rectangular groups such that the rectangles cover all of the positive cells; we can then apply the object detector in these rectangles. Second, the object detector is much slower when applied on variable-dimension inputs, since high detection performance on GPUs relies heavily on batching many equal-dimension inputs together. While input padding is often used to ensure all inputs are the same dimension, in our scenario, this would erase the time savings from applying the detector on small regions of a video frame. Instead, we develop an algorithm that determines ahead of time a small number (three in our implementation) of fixed window sizes, and initializes the detector on the GPU to execute at each of those sizes. Then, during inference, for each frame of video where we need to compute detections, we select rectangles sized at one of the pre-selected window sizes to cover the positive cells determined by the proxy model, falling back to detecting objects in the entire frame when doing so would be faster.

**Model Architecture.** We employ a simple, standard segmentation CNN architecture for the proxy model. Our model consists of a five-layer encoder followed by a two-layer decoder. The encoder inputs the video frame, and applies a series of five strided convolutional layers, producing features at 1/32 the resolution of the input. The decoder applies two additional convolutional layers, and its output is a classification score at each $32 \times 32$ cell of the input image indicating the likelihood that the cell intersects an object. We opt for a 32×32 cell size since objects are usually comparable or larger in size, and since this yields few enough cells so that the data does not become unwieldy when we group cells into rectangular windows.

**Training.** As in prior work, we use the object detection outputs of a *best-accuracy* parameter configuration $\theta_{\text{best}}$ as rough labels for training the segmentation proxy model, where the model should output a score close to 1 at cells that intersect a detection, and 0 at other cells. $\theta_{\text{best}}$ is a configuration of parameters in the OTIF pipeline that provides the best accuracy (which may still be far from 100%). We detail the selection of $\theta_{\text{best}}$ in the next sub-section.

Thus, we first compute object detections $D^{(t)} = \{d_1^{(t)}, \dots, d_n^{(t)}\}$ using $\theta_{\text{best}}$ on each frame $I_t$ in the training set of video. Then, we generate input-output training examples for the proxy model by first sampling a frame $I_t$ from the training set where $|D^{(t)}| > 0$, i.e., at least one detection was output by the best-accuracy configuration. We construct classification labels for $I_t$ corresponding to the detections in $D^{(t)}$: at each $32 \times 32$ cell, the label is 1 if there is some detection $d_i^{(t)} \in D^{(t)}$ that intersects the cell, and 0 otherwise.

Prior to training the proxy model, we cannot be certain how accurate the model will be at different resolutions—e.g., inputting $416 \times 256$ versus $224 \times 128$ frames (which yield $13 \times 8$ and $7 \times 4$ output grids, respectively) may provide tradeoffs between speed and accuracy. Thus, we train proxy models at several pre-determined resolutions (5 resolutions in our implementation), and leave the resolution as a parameter for the tuner to set. Although we train 5 models, training requires <10 minutes for all models since all input resolutions are much lower than the native video resolution.

**Best-accuracy Configuration Selection.** As discussed above, OTIF requires the outputs of a best-accuracy configuration $\theta_{\text{best}}$ to train proxy models. $\theta_{\text{best}}$ is simply a selection of parameters for the OTIF pipeline that yields highest accuracy over the validation set on the user-provided metric. The detections and tracks that it computes may contain errors, but nevertheless correspond to the best accuracy we can obtain through automatic labeling. To select these parameters, we begin by evaluating the accuracy of the slowest possible configuration on the validation set (using the metric and ground truth provided by the user), i.e., the configuration with no segmentation proxy model, maximum object detector resolution, and maximum sampling rate. Then, we repeatedly reduce the detector resolution to increase speed by a factor of $C = 30\%$ on each step, and re-evaluate accuracy until the accuracy decreases, keeping the resolution providing the best achieved accuracy. We then reduce the sampling rate in a similar procedure: we repeatedly reduce the rate to increase speed by $C$, and re-evaluate accuracy until accuracy drops. This procedure is crucial because we find that accuracy is oftentimes higher at lower resolutions. We do not consider employing a proxy model for $\theta_{\text{best}}$ since the proxy model never improves accuracy, and because at this stage the proxy models have not yet been trained. Since the tracking model has also not yet been trained, we use the heuristic SORT tracker [2] in $\theta_{\text{best}}$, which tracks objects based on bounding box overlap.

**Inference.** The proxy model inference procedure is configured by two parameters: the model input resolution (which determines which of the several trained proxy models to use) and a confidence threshold $B_{\text{proxy}}$ on the segmentation outputs. We will discuss selecting these parameters in Section 3.5. During inference, on each frame of video, we apply the proxy model to compute classification scores on each $32 \times 32$ cell. After thresholding the scores by $B_{\text{proxy}}$, we derive a binary grid consisting of a (possibly empty) set of positive cells where the output scores exceeded $B_{\text{proxy}}$. These cells are ones in which we must apply the object detector. The final step in the inference procedure is to select rectangular windows of the video frame in which to apply the detector. These windows should cover the positive cells, but should do so tightly to minimize the execution time needed to apply the detector over the windows. We will discuss this final step in two sections below: grouping cells into rectangular windows of certain sizes during inference, and determining ahead of time the fixed set of window sizes at which we will run the object detector.

**Grouping Cells during Execution.** On a frame $I_t$, the proxy model yields a set of positive cells $X^{(t)} = \{x_1^{(t)}, \dots, x_n^{(t)}\}$. We assume we are given a fixed set of window sizes at which the detector will run, $W = \{(w_1, h_1), \dots, (w_k, h_k)\}$, as well as the detector execution time of each size, $T_{w,h}$. We will detail how we decide on $W$ prior to execution in the next section. Our aim is to find a set of rectangular windows $R_t = \{r_1, \dots, r_m\}$ that covers all of the cells in $X^{(t)}$. Here, each $r_i$ is a 4-tuple $(r_i.x, r_i.y, r_i.w, r_i.h)$ that specifies the position and size of the rectangle, where $(r_i.w, r_i.h) \in W$. Let $est(R_t) = \sum_{r_i \in R_t} T_{r_i.w, r_i.h}$ be the estimated execution time of applying the detector in these windows. Then, in particular, we want the optimal set of rectangles $R^*(I_t; W)$ that minimizes $est(R_t)$. Intuitively, in frames sparsely populated with small objects, $R^*$ should include small rectangles covering the positive cells where objects are located, while in densely packed frames, $R^*$ should simply include a single rectangle corresponding to the entire frame.

We find an approximation to $R^*$ using a density-based greedy agglomerative clustering method. At a high level, we initialize a cluster $c_i$ for each connected component of positive cells, and repeatedly check whether two clusters can be combined to form a merged cluster that would be faster to process through the object detector than processing the two original clusters separately (i.e., whether merging would decrease $est(R)$). Once merging offers no further improvement, the clustering procedure terminates, and we create a set of rectangular windows $R$ by constructing one rectangle for each final cluster.

**Determining Fixed Set of Window Sizes.** Prior to execution, we decide the fixed set of window sizes $W = \{(w_1, h_1), \ldots, (w_k, h_k)\}$ (of a predetermined cardinality $k = 3$ set based on available GPU memory) at which to run the object detector, so that we can take advantage of the speed savings from batch detector execution on the GPU. The optimal set of window sizes $W^*$ is the set that yields the lowest runtime in expectation over video segments sampled uniformly from the dataset. When computing $W$, to simplify the problem, we assume that our proxy model performs perfectly (i.e., positive cells correspond directly to the locations of object detections). Then, it follows that $W^* = \arg\min_W \sum_t est(R^*(I_t; W))$. Intuitively, $W^*$ should tightly cover the locations of objects in video frames, so that on most frames, we can apply the detector only in small windows where objects are present.

Similar to our approach for computing $R_t$, we employ a greedy algorithm to pick $W$ so that the resulting runtime is close to that provided by $W^*$. At a high level, we first initialize $W$ to only contain the size corresponding to the entire video frame—we always include this maximum window size in the set so that the option of simply applying the detector on the entire frame is always available. We then repeatedly greedily add the window size to $W$ that provides the greatest decrease to $\sum_t est(R(I_t; W))$, until $|W| = k$.

## 3.4 Recurrent Reduced-Rate Tracking

Prior work (Miris [1]) applies reduced-rate tracking to speed up tracking in a query-driven context, where it is assumed that a query involving object tracks is provided, e.g., finding tracks of cars in traffic camera footage that travel north to south through a junction. Reduced-rate tracking applies an object detector at greatly reduced sampling rates (e.g. 1 fps instead of 25 fps), and attempts to accurately recover groups of detections of the same object.

The method in Miris has two limitations. First, Miris employs a graph neural network (GNN) model that only compares detections in two consecutive frames at a time to produce probabilities that each pair of detections across the frames correspond to the same object. Although this simplifies the training process and the overall tracking architecture, it limits accuracy when tracking at low sampling rates, since, unlike a recurrent model, the GNN model cannot exploit motion and other patterns observed in multiple previous frames when matching detections in a new frame to existing track prefixes. Second, it relies on capturing additional detections at high sampling rates to predict the position and time where a track first becomes visible, along with that where it is no longer visible. This step is needed since, when tracking at reduced rates, the first and last detections observed in the track may be a second before or after the object actually enters or leaves the camera frame; thus, for example, a car that traveled north to south through a junction may first be seen in the middle of the junction, resulting in the track being excluded from the north-to-south query. While the procedure of "refining" tracks by computing additional detections is effective for extracting tracks under predicates that only select small subsets of tracks, it becomes cost-prohibitive when extracting all tracks from video, which is the objective of OTIF.

We propose a recurrent reduced-rate tracking method in OTIF to address these challenges. First, we train a recurrent tracking network similar in design to architectures used in state-of-the-art multi-object tracking methods in the computer vision literature. A recurrent model inputs a sequence of data step-by-step, and produces a corresponding output sequence; here, the recurrent tracker processes video one frame at a time, inputting the new object detections in each successive frame and outputting the associations between those detections and previously observed tracks. Unlike prior work, though, we propose a novel specialized training process to ensure the model is robust when applied at varying reduced sampling rates. Second, rather than apply the track refinement methods introduced in Miris to accurately localize the start and end of each track, we propose estimating the start and end position based on the average path of the most similar tracks computed in the training set. To support efficient lookup of similar tracks, we cluster and index the training set tracks ahead of time.

**Model Architecture.** We use a recurrent tracking model that is comparable to that used in recent work in multi-object tracking [13]. The tracker inputs a video sequence $\langle I_1, \ldots, I_n \rangle$ and object detections computed by the detector over the sequence, where $D^{(t)} = \{d_1^{(t)}, \ldots, d_{n_t}^{(t)}\}$ denotes the detections computed in frame $I_t$. The goal of the tracker is to assign a track ID to each detection so that detections of the same object are all assigned the same track ID. To apply the model, we iterate over the frame sequence, maintaining a set of active track prefixes $A = \{s_1, \ldots, s_m\}$ computed up until the previous frame. Processing begins on $I_1$, where we initialize a new track prefix $s_i = \langle d_i^{(1)} \rangle$ for each detection in $D^{(1)}$. On a successive frame $I_t$, we apply a neural network model to compute a score $p_{i,j}^{(t)}$ indicating the likelihood that each detection $d_j^{(t)} \in D^{(t)}$ corresponds to each track prefix $s_i$. We match detections with tracks based on these scores, and add each detection to the track that it matches with. If a detection $d_j^{(t)}$ does not match with any track, we initialize a new track prefix $s = \langle d_j^{(t)} \rangle$ and add it to $A$.

Our model has three components. First, we compute detection-level features that describe each detection $d = (x, y, w, h)$. This component applies a CNN to process the image content of the detection (i.e., the portion of the frame contained in the bounding box that $d$ specifies) into a compact feature vector representation. Then, the detection-level feature vector for $d$ is the concatenation of the representation computed by the CNN with the 4D spatial bounding box coordinates $(x, y, w, h)$. Second, we compute track-level features that describe each track prefix. A track $s = \langle d_1, \ldots, d_n \rangle$ is a sequence of detections, so it is natural to use an RNN for this component. We compute the detection-level features for each detection in $s$, and apply an RNN over those features. We use the final output of the RNN as the track-level features. Third, we compute the scores $p_{i,j}^{(t)}$ matching the detection $d_j^{(t)}$ with $s_i$. This component applies a matching network consisting of several fully connected layers that inputs the detection-level features of $d_j^{(t)}$ and the track-level features of $s_i$, and outputs a single score.

**Training.** Typically, tracking models are trained on ground truth labels annotated at the full video framerate. However, we do not have ground truth labels, and we require our tracker to be effective at several possible reduced sampling rates. To address the first issue, as with training the segmentation proxy models, we use outputs of the best-accuracy configuration $\theta_{\text{best}}$ from Section 3.3 as a rough ground truth for training. Let $S^* = \{s_1^*, \ldots, s_n^*\}$ be the set of tracks computed by $\theta_{\text{best}}$ over the training set of video.

To train a model that will be robust even when track prefixes are captured at various reduced sampling rates, we construct training examples that consist of detection sequences sampled from some $s_i^* \in S^*$ with diverse spacing. During inference, we will process video at some sampling gap, where each gap is a power of two, and a gap $g$ specifies that one out of every $g$ frames should be processed ($g = 1$ corresponds to processing the video at its native framerate). Let $G = \langle 1, 2, 4, \ldots, 2^n \rangle$ be the maximal gap sequence, i.e., we will not track at sampling gaps higher than $2^n$ during inference. Then, to create a training example, in addition to sampling a track $s \sim S^*$, we sample a gap $g \sim G$. We then iteratively sub-sample detections from $s$, beginning from its first detection, such that each following detection is at least $g$ frames after the previous detection.

We have shown how to generate examples for training the model. However, if we do not provide any temporal information to the model, then when matching a detection $d$ with a track prefix $s$, the model cannot, for example, predict the position of $s$ at the current frame based on its velocity in preceding frames, since doing so would necessitate multiplying this velocity by the time elapsed between $d$ and the last detection in $s$. Thus, we augment the detection-level features of $d$ to include the number of frames $t_{\text{elapsed}}$ between $d$ and some preceding detection, to enable the model to account for temporal information. When computing track-level features of $s = \langle d_1, \ldots, d_n \rangle$, the value of $t_{\text{elapsed}}$ for $d_i$ is the number of frames between $d_{i-1}$ and $d_i$. When computing detection-level features for a detection $d_j^{(t)}$ in the current frame $I_t$, $t_{\text{elapsed}}$ is the number of frames between $I_t$ and the previously processed frame. Providing this additional input enables the model to more robustly compute scores $p_{i,j}^{(t)}$ when using reduced-rate tracking.

**Inference.** Before applying the tracker on a video dataset, we assume that the OTIF tuner (Section 3.5) has selected a gap $g \in G$. We decode video at a framerate corresponding to the gap $g$ to obtain a sequence of frames $\langle I_0, I_g, I_{2g}, \ldots \rangle$. On each frame $I_t$, we first compute a set of object detections $D^{(t)}$ in the frame using the segmentation proxy model and object detector components. We then apply the tracker model to compute scores $p_{i,j}^{(t)}$ between detections $d_j^{(t)} \in D^{(t)}$ and track prefixes $s_i$, and match detections to tracks based on the highest-scoring pairs. After tracker execution, we prune tracks that consist of only a single detection, since such tracks likely correspond to noisy detections.

Note that we do not use the variable rate selection technique in Miris, which tracks objects not at a fixed gap $g$ but at a variable gap that changes based on tracker confidence. In preliminary experiments, when using our recurrent model, we found the accuracy of the variable gap method comparable to simply using a fixed gap.

**Refinement.** The recurrent reduced-rate tracking method that we have introduced improves accuracy when tracking objects at reduced sampling rates. However, one crucial issue with tracks extracted at low sampling rates is that their first and last detection will be offset from the start and end of the object's actual trajectory. This is primarily problematic when extracting tracks in video captured from fixed cameras, because oftentimes, video analytics tasks over such video involve spatial predicates on the first and last detections in tracks: for example, performing a turning movement count in video of a traffic junction necessitates counting the number of cars that start and end at each pair of roads. Simply extrapolating a track to the frame boundary is insufficient since tracks may start or end at the horizon, or anywhere in the frame due to an occlusion. On the other hand, queries involving moving cameras are unlikely to contain such predicates, since the physical location of a particular pixel is dynamic in moving cameras; thus, tracks extracted by OTIF in moving camera footage can be used in such queries without refinement.

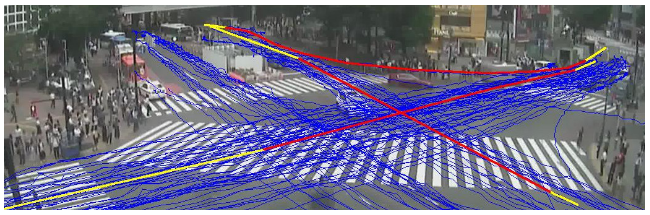


**Figure 4: We refine the start and end position of tracks based on similar tracks seen in the training set. Here, blue lines show clusters computed from the training set tracks, red lines show tracks captured at a high sampling gap during execution, and yellow lines show extensions to the tracks added by our refinement method.**

Miris [1] proposes processing additional frames to refine tracks. However, this is cost-prohibitive when extracting all tracks from video. Instead, we observe that in fixed video (where refinement is most needed), we can generally estimate the start and end of a track by comparing the portion of a track captured at a low sampling rate against known tracks captured at the native framerate. We show an example in Figure 4. Then, a simple algorithm for refining tracks would be to find the k-nearest neighbors $knn(s)$ of a track $s$ among tracks captured using $\theta_{\text{best}}$ in the training set, $S^*$, and estimate the start and end of $s$ as the median start and end of tracks in $knn(s)$.

However, computing pairwise distances between tracks grows more expensive as $S^*$ increases in size, and while we could sample a subset of tracks from $S^*$ to use for k-nearest neighbor computation during inference, this reduces accuracy for infrequently-occurring paths. To speed up k-nearest neighbor lookups, we cluster tracks in $S^*$ prior to inference, and construct a spatial index over cluster centers (which are paths). Clustering reduces the number of distances that must be computed, so that redundant tracks that follow the same path in the video are merged into one cluster.

In our approach, we begin by clustering the tracks in $S^*$ using DBSCAN. To apply DBSCAN, we must define the distance metric between tracks, as well as how to compute the center of a cluster. We use a simple distance metric: given $s_1$ and $s_2$, we compute $N$ points evenly spaced along each track, yielding sequences of points $P(s_1)$ and $P(s_2)$, and define $d(s_1, s_2) = \frac{1}{N}\sum_{i=1}^{N} \text{eucl}(P(s_1)[i], P(s_2)[i])$, i.e., as the average distance between corresponding points. Here, $\text{eucl}(p_1, p_2)$ is the Euclidean distance between 2D points. In our implementation, $N = 20$. We define the center of a cluster of tracks $c = \{s_1, \dots, s_n\}$ as a path $s = \langle p_1, \dots, p_N \rangle$ of $N$ points, where $p_i$ is the average of points in $\{P(s)[i] | s \in c\}$, i.e., the average position of the $i$th evenly spaced point computed among tracks in the cluster.

DBSCAN produces a set of clusters by iteratively grouping together sets of many tracks sharing similar high-density paths. We then build a spatial index over the cluster centers. During inference, we approximate the k-nearest neighbor lookup for a computed track $s = \langle d_1, \dots, d_n \rangle$ as follows. We first use the index to identify several cluster centers that pass close to $d_1$ and $d_n$. We compute the distance between each of those cluster centers and $s$, and keep the $k = 10$ closest clusters. Finally, we extend $s$ with a start and end detection computed by taking the median coordinate on each dimension of the start and end points across the clusters, weighted by cluster sizes.

## 3.5 Joint Parameter Tuning

As detailed above, each module in the OTIF pipeline is governed by several parameters. The *OTIF tuner* is responsible for selecting values for each of these parameters. The six parameters are:

- The detection module is configured by the object detector model architecture (e.g. YOLOv3 or Mask R-CNN), input resolution, and confidence threshold.
- The proxy model module is configured by the input resolution and confidence threshold.
- The tracking module is configured by the sampling gap $g$.

The output of the tuner is a sequence of configurations $\Theta = \langle \theta_1, \dots, \theta_n \rangle$, where each $\theta_i$ represents a setting of the above parameters. $\Theta$ forms a speed-accuracy tradeoff curve, and the goal of the tuner is for it to be as close as possible to the Pareto-optimal frontier of the space of all possible configurations. After the tuning process completes, the user will pick a configuration $\theta_i$ along the output curve to use for execution over the rest of the dataset.

Profiling every possible configuration would be cost-prohibitive since the search space is exponential in the number of parameters. Another naive approach would be to update parameters using a hill climbing procedure, as in Chameleon [9]: starting with some initial configuration, we could iteratively test small parameter updates, selecting the updates that provide the best speed-accuracy tradeoff. However, this too is cost-prohibitive: because of the number of parameters in OTIF's execution pipeline, and the presence of real-valued parameters such as confidence thresholds, it would require evaluating hundreds of configurations.

Instead, we adapt the hill climbing procedure by incorporating a modular framework that reduces the number of updates that must be tested on each iteration. Intuitively, since choices such as the detector model architecture and detector confidence threshold are closely entangled, batching these updates can reduce the number of configurations that we must test. In our approach, we begin with a slow but accurate configuration, and on each iteration, we independently update different subsets of parameters to offer a speedup, test the accuracy of each update on the validation set, and choose the update that offers the best accuracy.

We initialize $\theta_1 = \theta_{\text{best}}$ as the best-accuracy configuration (Section 3.3), i.e., the parameters that yield the best possible accuracy on the validation set, but poor execution speed. On each iteration, given the current configuration $\theta_i$, the tuner queries each module and requests a new configuration where the parameters for that module have been updated to provide an overall speedup of approximately $C$ over $\theta_i$, where $C$ is the parameter tuning coarseness. For example, $C = 30\%$ implies the next configuration should be around 30% faster. This yields three candidate configurations $\theta_{i+1,\text{detection}}, \theta_{i+1,\text{proxy}}, \theta_{i+1,\text{tracking}}$, where each reflects updates to the parameters of one module that provide the desired overall speedup. Then, the tuner executes each candidate configuration over the validation set to measure accuracy. We set $\theta_{i+1}$ to the best-accuracy candidate, and iteratively repeat this procedure. By repeatedly choosing parameter changes that provide the smallest drop in accuracy for a fixed desired speedup, this approach produces an output $\Theta$ that approximates the Pareto frontier. If there are $m$ modules and we seek a curve of $n$ configurations, then the algorithm uses $O(mn)$ trials on the validation set ($n$ iterations where we test $m$ candidates on each iteration).

The parameter tuning coarseness $C$ must be chosen to balance between being sufficiently fine-grained to effectively model the speed-accuracy curve, and being coarse-grained to minimize the number of iterations needed (i.e., parameter tuning time). We use $C = 30\%$ in our implementation, since we find in preliminary experiments that this provides a good tradeoff.

The OTIF tuner executes in two phases. First, during a caching phase, each module caches information that it will need to answer the tuner requests for next candidate configurations (i.e., to compute a configuration $\theta_{i+1,\text{module}}$ that is $C$ faster than $\theta_i$). Then, during the tuning phase, we apply the greedy algorithm detailed above that produces $\Theta$. Below, we detail the operation of each module.

### 3.5.1 *Detection Module.*

Because the object detector is generally the slowest component of the pipeline, increasing object detection speed by $C$ usually corresponds to an overall speedup of almost $C$. Then, at a high level, our strategy in the detection module during the tuning phase is to simply find the object detector configuration that offers the highest accuracy among configurations that are at least $C$ faster than the previous configuration.

Let $A = \{a_1, \dots, a_n\}$ be the set of model architectures (e.g. YOLOv3 [20], Mask R-CNN [7], etc.), and let $R = \{(w_1, h_1), \dots, (w_m, h_m)\}$ be the set of input resolutions. Then, during the caching phase, for each architecture $a_i$ and each input resolution $(w_j, h_j)$, we evaluate the execution time $t_{i,j}$ and validation accuracy $\alpha_{i,j}$ of the corresponding configuration, where parameters for other modules are taken from $\theta_{\text{best}}$. During tuning, given a configuration $\theta_k$ that uses architecture $a_i$ and input resolution $(w_j, h_j)$ for the detection module, we identify the choice of $a_{i'}$ and $(w_{j'}, h_{j'})$ with maximum $\alpha_{i',j'}$ such that $t_{i',j'} \leq (1 - C)t_{i,j}$, i.e., a new architecture and resolution that offers the highest accuracy among the choices that yield a speedup of at least $C$ over the previous selection.

### 3.5.2 *Proxy Model Module.*

In the proxy model module, we need to pick two parameters: the model's input resolution, and the confidence threshold that specifies when the output score at a cell is high enough to mark that cell positive (and require object detector processing over the cell). One complication here is that speed is not directly related to the number of positive cells, since cells must first be grouped into rectangular windows where the object detector will be applied. On the other hand, if we can reduce the area of these windows by $C$, then this would speedup detector runtime by approximately $C$, and thus provide an overall speedup close to $C$.

We follow a similar approach as the detection module: during caching, we estimate the runtime and proxy model recall on the validation set at each resolution and threshold, and during tuning, we use these estimates to update parameters from the previous configuration. We define recall as the fraction of object detections that are covered by rectangular windows; during tuning, we will pick the resolution and threshold that have highest recall among choices with runtime at least $C$ faster than the previous configuration.

We first cache the per-cell proxy model classification scores for each resolution $(w_i, h_i)$ on every frame in the validation set, along with the detections computed by $\theta_{\text{best}}$. Let $T_{\text{proxy},i}$ be the runtime of the proxy model at each resolution. Then, for each threshold $B_j$, we compute rectangular windows using the cell grouping method from Section 3.3 on each frame. Let $R_{i,j} = \{r_1, \ldots, r_n\}$ be the set of rectangles computed across the frames at a given resolution and threshold. Then, our runtime estimate for this resolution and threshold is $T_{\text{proxy},i} + \sum_k T_{r_k.w, r_k.h}$, i.e., the proxy model runtime added to the detector runtime, which we estimate based on the rectangle sizes. The recall is the fraction of detections computed by $\theta_{\text{best}}$ that are covered by rectangles in $R_{i,j}$.

*3.5.3 Tracking Module.* The tracking module exposes a single parameter, the sampling gap $g$. We can obtain an overall $C$ speedup over the previous configuration by adjusting $g$ so that the tracker processes $C$ fewer frames. In particular, during tuning, we compute a new sampling gap by dividing $g$ by $C$ and rounding up to the nearest power of two.

## 4 EVALUATION

We now evaluate OTIF on 7 diverse video datasets against seven baselines, including three video query optimizers (Miris [1], BlazeIt [10], and TASTI [12]), three fast object detection and tracking methods (NoScope [11], Chameleon [9], and CaTDet [17]), and one high-accuracy multi-object tracker (CenterTrack [26]). In Section 4.1, we first evaluate the methods on object track queries that Miris is specifically designed to optimize; these queries select tracks satisfying predicates about the path or speed of a track. Then, in Section 4.2, we evaluate on frame-level queries that BlazeIt and TASTI are specifically designed to optimize; these queries select video frames satisfying predicates about the objects appearing in the frame.

**Datasets.** To robustly evaluate the methods, we employ a highly diverse benchmark consisting of 7 video datasets, including 5 datasets from two prior works as well as 2 new datasets. We use the Tokyo, UAV, and Warsaw datasets from Miris [1], and the Amsterdam and Jackson datasets from BlazeIt [10]. We also evaluate OTIF on two new datasets, Caldot1 and Caldot2, consisting of video captured by California DOT cameras along two highways. UAV consists of video captured from an aerial drone, while the other 6 datasets consist of video from fixed street-level cameras. Amsterdam captures activity at a riverside plaza, Caldot1 and Caldot2 capture highway activity, while the other 4 datasets capture city traffic junctions. By combining datasets from multiple prior works as well as introducing new datasets, we ensure that methods are benchmarked on performance over a wide range of video query scenarios.

We sample three one-hour sets of 60 one-minute clips from each dataset: a training set (for training models), a validation set (for parameter tuning), and a hidden test set (for experimental evaluation). Thus, we allow each method to select Pareto-optimal configurations on the validation set, but report accuracy computed over a test set that the methods do not observe until execution.

In Sections 4.1 and 4.2, we formulate a variety of queries about cars appearing in the video datasets. We opt to focus on cars because, while other objects appear in all 7 datasets, object detectors (specifically, YOLOv3 [20] and Mask R-CNN [7]) trained on COCO are not able to accurately detect these other objects, and other objects are highly ambiguous (e.g., it is unclear whether a pedestrian far in the background near the horizon that occupies only a few pixels should count).

**Baselines.** We compare OTIF to six baselines. Miris [1] applies variable framerate tracking under different error tolerances. BlazeIt [10] applies classification and regression proxy models to optimize query speed. TASTI [12] employs proxy models trained to produce per-frame feature embeddings to build a query-agnostic video index. NoScope [11], Chameleon [9], and CaTDet [17] propose methods to accelerate the extraction of object detections and object tracks from video. CenterTrack is a recent computer vision method for multi-object tracking published in ECCV 2020 that performs well on the MOT17 pedestrian tracking benchmark; we obtain a speed-accuracy tradeoff by tuning resolution and framerate.

We use our implementations of Miris, BlazeIt, NoScope, Chameleon, and CaTDet. Although implementations of Miris, BlazeIt, and NoScope are available, they are not readily adaptable to our dataset; we validate the performance of our implementations in Section 4.6. Chameleon and CaTDet do not have public implementations. We use the TASTI and CenterTrack implementations released by the respective authors.

**Implementation.** We store the training, validation, and test video on a local SSD at a fixed resolution, which is $720 \times 480$ for Caldot1 and Caldot2 and $1280 \times 720$ for the other 5 datasets, encoded in mp4 container with H264. The native video framerate ranges from 5 fps for UAV to 30 fps for Amsterdam and Jackson. We run all methods on a machine with one NVIDIA Tesla V100 GPU and one Intel Xeon Gold 6142 CPU.

The execution pipeline for all methods involves decoding the video using ffmpeg, and processing decoded frames iteratively in a way that is specific to each method. Frames are decoded at the object detector resolution, so reducing the model input resolution may provide a speedup both through faster model execution and through faster video decoding. After the last frame is decoded and processed, each method outputs a set of extracted object tracks. The parameter selection phase for all methods involves repeatedly applying the execution pipeline with different parameter configurations.

### 4.1 Object Track Queries

We now evaluate OTIF against Miris and the three object detection and tracking baselines on 7 object track queries, where we formulate one query over each of the 7 video datasets. We do not compare against BlazeIt and TASTI here, since they are only directly

**Table 2: Runtime in seconds of each method on the test set of each dataset, using the fastest candidate configuration that provides accuracy within 5% of the best achieved accuracy. Runtime for five queries is estimated by scaling the runtime of query-specific phases in each method.**

| # Queries | 1 Query | | | | | | 5 Queries (estimated) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Dataset** | **OTIF** | **Miris** | **Cham** | **NoScope** | **CaTDet** | **CTrack** | **OTIF** | **Miris** | **Cham** | **NoScope** | **CaTDet** | **CTrack** |
| Caldot1 | **40** | 533 | 209 | 990 | 601 | 1420 | **40** | 2665 | 209 | 990 | 601 | 1420 |
| Caldot2 | **38** | 129 | 123 | 803 | 462 | 1593 | **38** | 645 | 123 | 803 | 462 | 1593 |
| Tokyo | **37** | 123 | 84 | 823 | 810 | 751 | **37** | 615 | 84 | 823 | 810 | 751 |
| UAV | **99** | 188 | 188 | 323 | 188 | - | **99** | 940 | 188 | 323 | 188 | - |
| Warsaw | **49** | 422 | 422 | 867 | 961 | 900 | **49** | 2110 | 422 | 867 | 961 | 900 |
| Amsterdam | **25** | 64 | 44 | 666 | 564 | 293 | **25** | 320 | 44 | 666 | 564 | 293 |
| Jackson | 44 | 82 | **41** | 618 | 643 | 672 | 44 | 410 | **41** | 618 | 643 | 672 |

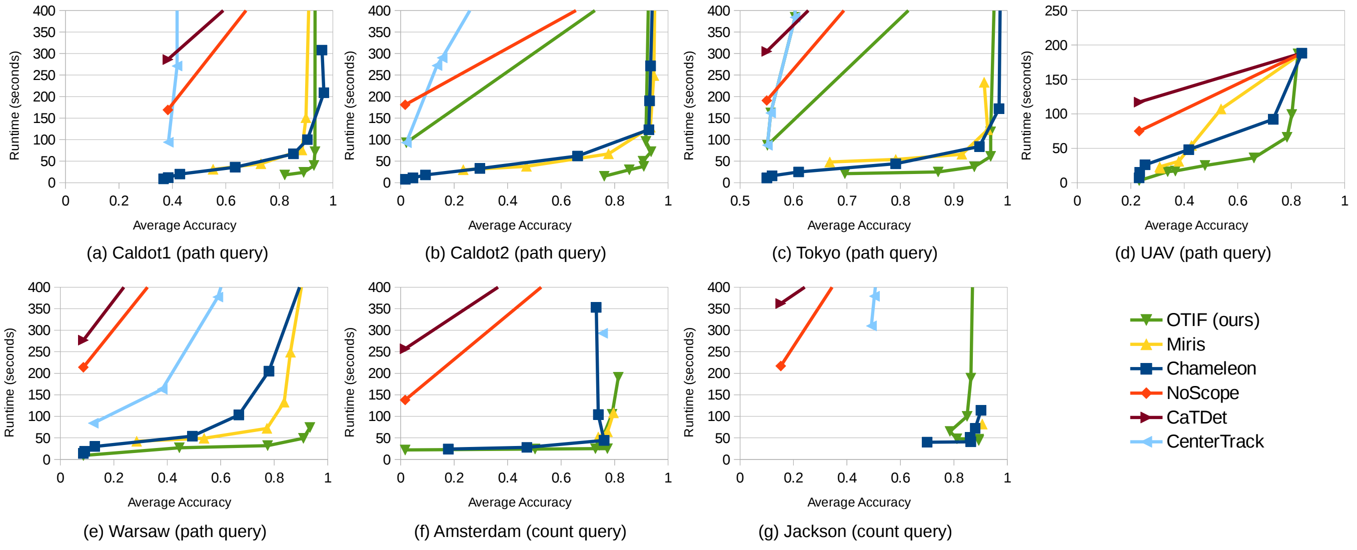


**Figure 5: Runtime-accuracy curves on each query over the test set of the corresponding datasets. Each point is a parameter configuration for the method, which is chosen based on performance in the validation set.**

applicable to frame-level queries. We formulate a variety of queries, including track count queries and path breakdown queries. Below, we first detail the queries and metrics, and then present results.

**Query Types.** On the Amsterdam and Jackson datasets, we employ *track count queries* that report the number of unique cars appearing in each one-minute clip. On the Caldot1, Caldot2, Tokyo, UAV, and Warsaw datasets, we employ *path breakdown queries* that breakdown tracks based on the spatial path each track exhibits. The output of the query consists of counts of tracks exhibiting each path type. For example, in the Tokyo video of a traffic junction, we identify 10 unique turning directions that cars take through the junction, and count the number of objects passing through the junction along each direction.

**Metrics.** We measure accuracy by comparing the counts inferred by a method from the video with hand-labeled ground truth counts, using percent accuracy averaged over clips and, if applicable, path types. If a method infers a count $\hat{x}$, and the ground truth count is $x^*$, then we define accuracy as $1 - |\hat{x} - x^*|/x^*$.

We also evaluate methods on their runtime. We exclude training costs of each method from the runtime, focusing only on costs that grow linearly with the dataset size.

All methods that we compare on object track queries expose parameters that can be tuned to provide a tradeoff between speed and accuracy. Thus, we compare methods on their speed-accuracy curve. We first apply the parameter selection phase of each method on the validation set, which yields a speed-accuracy curve consisting of several Pareto-optimal configurations. We then apply each of these parameter configurations on the unseen test set, yielding a test speed-accuracy curve: the runtime of a configuration is its execution time over the unseen one-hour test set, and its accuracy is the percentage accuracy of its output counts in each one-minute clip, averaged across 60 clips.

**Results.** Table 2 and Figure 5 show the results. In Figure 5, we plot speed-accuracy curves provided by the five methods on each of the 7 queries. In some cases, several methods share the same top-right slowest, highest-accuracy point: this is because Miris, Chameleon, NoScope, and CaTDet fallback to the same naive configuration that processes every video frame. In Table 5, for each dataset and method, we show the runtime of the fastest configuration (among candidates selected using the validation set) providing accuracy within 5% of the best achieved accuracy on the test set (across all methods). We choose the 5% threshold since average accuracy is computed over 60 accuracy samples (the test set consists of 60 clips randomly sampled from the video dataset), and so the variance in the sample mean implies that differences in accuracy less than 5% may not be meaningful. Thus, Table 2 highlights the runtime that each method can achieve while maintaining an accuracy close to the best accuracy achieved by any method.

Although Miris is specifically designed to optimize the execution of object track queries, and leverages knowledge of the query to plan an execution configuration, OTIF provides a 5x speedup on average over Miris at the same accuracy level. By jointly tuning its segmentation proxy model, object detector, and recurrent reduced rate tracker, OTIF is able to extract tracks from video substantially faster than Miris can execute just one query. Since Miris utilizes a per-query execution stage, the speedup balloons to 25x when executing 5 queries over the same video.

OTIF also consistently performs comparably to or better than the next best object detection and tracking baseline across all seven datasets: it provides an average 3.4x speedup over the next best baseline in Table 2. Although Chameleon offers good speed-accuracy tradeoffs on every dataset, OTIF provides better performance (especially on Tokyo, UAV, Warsaw, Caldot1, and Caldot2), both through the novel techniques employed in its segmentation proxy model and recurrent tracker components, and simply by exploring a more diverse range of ways to improve execution speed without impairing accuracy. NoScope provides some degree of a tradeoff between speed and accuracy on Amsterdam, Caldot1, and Caldot2; however, it only yields two candidate configurations for the other datasets because those datasets have objects visible in every frame (one candidate applies the detector on every frame, while the other skips the entire video and simply outputs 0 for all counts). CaTDet offers a slightly better tradeoff than NoScope on most datasets, but similarly exhibits poor performance as it does not optimize framerate or resolution. CenterTrack performs poorly on all datasets except Amsterdam: CenterTrack is designed for high-accuracy but slow multi-object tracking in native-framerate and native-resolution video, so although the method yields state-of-the-art performance on MOT17, it may not be competitive on the speed-accuracy tradeoff; additionally, it may require extensive hyperparameter tuning to improve results on our video datasets. Across all accuracy levels, OTIF is better than all other methods on 5 out of 7 datasets; on Amsterdam and Jackson, it provides a comparable speed-accuracy curve to Chameleon.

## 4.2 Frame-Level Queries

In this section, we evaluate OTIF against BlazeIt and TASTI on 6 frame-level limit queries. We formulate three types of queries: object count queries, region queries, and hot spot queries. BlazeIt and TASTI both incorporate techniques specifically designed to optimize the execution of frame-level limit queries: in particular, after building a query-specific (BlazeIt) or query-agnostic (TASTI) index, they use the index to derive scores on how likely each frame is to satisfy the query, and then process video starting from the highest-scoring frames until the desired number of frames specified in the limit query are found. Below, we first detail the queries and evaluation metrics, and then present results.

**Query Types.** We formulate 6 frame-level limit queries to benchmark the methods, split into three types:

- On the UAV and Tokyo datasets, we apply *count queries* that select frames with at least $N$ objects.
- On the Jackson and Caldot1 datasets, we apply *region queries* that select frames with at least $N$ objects in a fixed spatial region (polygon) of the frame.
- On the Warsaw and Amsterdam datasets, we apply *hot spot queries* that select frames where there are at least $N$ objects in a circular cluster of radius $R$.

Methods must output both the matching frames (i.e., clip filename and timestamp) and the object positions in the frame, region, or hot spot. We require query outputs, which are video frames, to be at least 5 seconds apart. We set the limit (desired output cardinality) of each query to 25 or 50 depending on the total number of matching frames, and set the query parameters (e.g., $N$) so that there are fewer than 250 matching five-second video segments.

**Execution Details.** To apply BlazeIt and TASTI on frame-level limit queries, we train a proxy model to either produce object counts (BlazeIt) or feature embeddings (TASTI) on every frame. Both methods then score frames based on the likelihood that they satisfy the predicate. BlazeIt uses the per-frame counts directly as scores, while TASTI trains an auxiliary model to compute query-specific scores from general-purpose embeddings. BlazeIt and TASTI then optimize query execution by applying the detector (using an input resolution that provides the best accuracy, in our implementation) on frames in order from highest-scoring to lowest-scoring until they encounter a sufficient number of frames that match the predicate to satisfy the desired output cardinality.

Note that BlazeIt and TASTI do not expose parameters to provide a tradeoff between speed and accuracy, and instead optimize speed under the assumption that the object detector is always fully accurate (which is not the case in practice). Thus, they produce one speed and accuracy point rather than a curve. When applying OTIF, we choose one parameter configuration to execute on the test set so that we also yield one speed-accuracy point. In particular, we execute OTIF to extract all tracks from the dataset using the fastest parameter configuration among the candidates that yield within 5% of the best-achieved accuracy on object track queries (i.e., the same configurations as the ones from Table 2). We then post-process the tracks to pick frames that match the query, starting with frames where the visible tracks have the highest minimum duration.

**Metrics.** As with object track queries, we evaluate the methods on speed and accuracy. We define accuracy as the fraction of frames produced by a method that satisfy the query based on hand-labeled ground truth. Note that, while BlazeIt and TASTI are designed to produce outputs with 100% accuracy with respect to a given object detection model, in practice no model is fully accurate, and so their actual accuracy may be much lower.

As before, we ignore training costs in both methods, and focus only on execution runtime that grows linearly with the dataset size.

**Table 3: Evaluation of OTIF, BlazeIt, and TASTI on the six frame-level queries. We show metrics averaged across the queries. Runtime for five queries is estimated by scaling the runtime of query-specific phases in each method.**

| # Queries | 1 Query | | | 5 Queries (estimated) | | |
|---|---|---|---|---|---|---|
| Method | OTIF | BlazeIt | TASTI | OTIF | BlazeIt | TASTI |
| Average Pre-processing Time (sec) | 100 | 98 | 814 | 100 | 490 | 814 |
| Average Query Time (sec) | 1 | 35 | 23 | 1 | 175 | 115 |
| Average Total Time (sec) | **101** | 133 | 837 | **101** | 665 | 929 |
| Average Accuracy | 97% | 86% | 84% | 97% | 86% | 84% |

**Results.** Table 3 shows the results. OTIF's pre-processing time is comparable to BlazeIt's, implying that OTIF is able to extract all tracks from video in the same time that it takes BlazeIt to apply a query-specific proxy model. There are two reasons for this. First, although BlazeIt processes video at a lower resolution (64×64), OTIF combines its segmentation proxy model with its resolution and framerate optimizations to achieve a similar speedup. Second, since both methods substantially reduce the cost of machine learning inference, video decoding begins to become a bottleneck, occupying roughly one-third of CPU time.

Then, since the outputs from OTIF's pre-processing stage (i.e., object tracks) can be efficiently post-processed to execute multiple queries, OTIF provides a 6x speedup over BlazeIt when executing 5 queries over the same video data.

TASTI's pre-processing stage is much slower since it processes every frame of video at a resolution of $224 \times 224$. Then, even though TASTI's query-agnostic feature embeddings can be reused across queries (unlike BlazeIt), its overall runtime on 5 queries is worse than that of BlazeIt.

Note that BlazeIt's query time consists of applying the object detector 1,523 times on average, while TASTI's query time consists of applying the detector 1,890 times on average. We exclude video decoding time in the query time numbers (35 s for BlazeIt and 23 s for TASTI on one query) since we have not optimized the query pipeline for random-access decoding; thus, query time for BlazeIt and TASTI only includes the object detector inference runtime.

OTIF also provides higher average accuracy than BlazeIt and TASTI: several frames produced by the latter methods contain spurious detections where the object detector exhibited errors; OTIF's tracking procedure is able to avoid most of these errors by pruning length-1 tracks. Nevertheless, the main conclusion from this experiment is that, remarkably, OTIF is able to accurately extract all tracks from video in the same time that prior work requires for answering a single limit query, even when the desired output cardinality is small. After tracks are extracted, exploratory queries can be executed in milliseconds with OTIF instead of tens of seconds with BlazeIt and TASTI.

## 4.3 Cost Breakdown

In Figure 6, we show a breakdown of OTIF's pre-processing and execution costs on Caldot1. We categorize costs as execution if they scale linearly with the dataset size, and as pre-processing otherwise. Pre-processing is dominated by object detector training, which is required to get high-accuracy on this dataset. Note that the segmentation proxy model time includes 3 seconds for computing the fixed window sizes. The execution cost breakdown corresponds to the fastest OTIF configuration providing accuracy within 5% of the best achieved accuracy. Because several components run in parallel, and on different hardware (CPU vs GPU), the breakdown does not sum to the overall 40-second runtime. As with pre-processing, object detection dominates the execution cost.

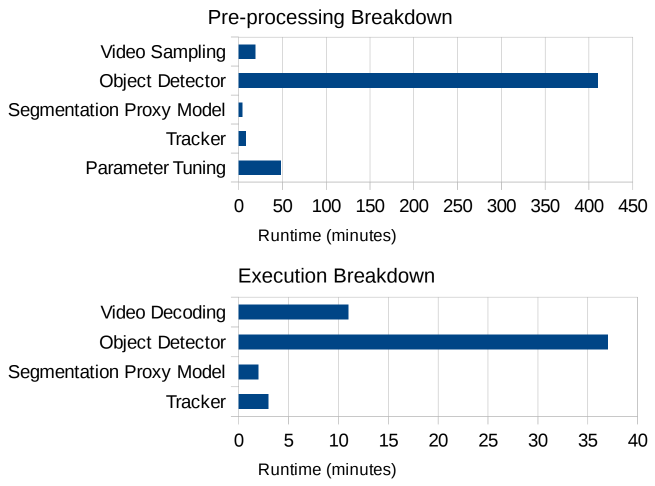


**Figure 6: Cost breakdown of OTIF for the Caldot1 dataset.**

| Method | Caldot1 | Warsaw |
|---|---|---|
| Detector Only | 299 | 951 |
| + Sampling Rate | 208 | 432 |
| + Recurrent Tracker | 42 | 71 |
| + Segmentation Proxy Model | 40 | 49 |

**Table 4: Ablation study of OTIF on Caldot1 and Warsaw, showing runtime (sec) using the fastest configuration providing accuracy within 5% of best achieved accuracy. Rows show increasingly complete implementations of OTIF.**

## 4.4 Ablation Study

In Table 4, we conduct an ablation study of OTIF on the Caldot1 and Warsaw object track queries, testing four successively more complete implementations of OTIF: in Detector Only, we start with our parameter tuning method with only the object detection module; in "+ Sampling Rate", we add a tracking module using the heuristic SORT [2] tracker; in "+ Recurring Tracker", we replace SORT with

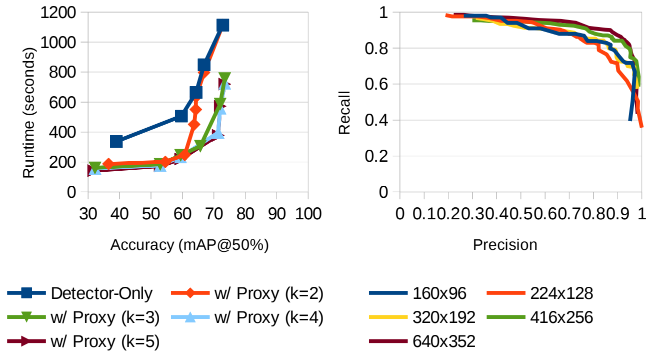


**Figure 7: On the left, object detection speed and accuracy on Caldot1 of YOLOv3 versus integrating our segmentation proxy model, with different numbers of detector window sizes. On the right, precision-recall curves of the proxy model at different input resolutions.**

our recurrent reduced-rate tracking method; and in "+ Segmentation Proxy Model", we add the segmentation proxy model, which corresponds to our full method. Each additional module improves performance in different conditions: for example, while the segmentation proxy model provides little improvement on Caldot1, it provides a 1.5x speedup on Warsaw.

## 4.5 Segmentation Proxy Model

Here, we directly evaluate the performance gain provided by the segmentation proxy model by comparing the speed-accuracy curve obtained when using YOLOv3 alone at varying resolutions, against using it in conjunction with our proxy model approach. We apply both methods on one hour of video in Caldot1, and evaluate accuracy in terms of a standard object detection metric, mean average precision (mAP@50%), on 50 hand-labeled frames. With YOLOv3, we obtain a speed-accuracy curve simply by varying resolution. When adding our method, since values for several parameters must be set, we use the greedy tuning process in Section 3.5.

We show results in Figure 7 (left). We include ablations of the proxy model with different numbers of window sizes $k$; $k = 3$ corresponds to the default setting we use for OTIF in other experiments, while $k = 1$ would be equivalent to using the detector only. Performance increases with more window sizes, but increasing $k$ beyond 3 provides diminishing returns that are not worth the increased GPU memory capacity requirements. At $k = 3$, our segmentation model approach provides a substantial gain in speed at all accuracy levels over using only YOLOv3.

In Figure 7 (right), we show precision-recall curves comparing the per-cell scores produced by the proxy model against cells that intersect ground truth boxes in the 50 hand-labeled frames. We obtain a curve by varying the score threshold $B_{\text{proxy}}$. Performance decreases as we reduce the input resolution, but the model remains highly effective even when inputting $160 \times 96$ frames. Note that overall performance depends not only on precision and recall, but also how tightly we can create rectangles around the positive cells for detector execution.

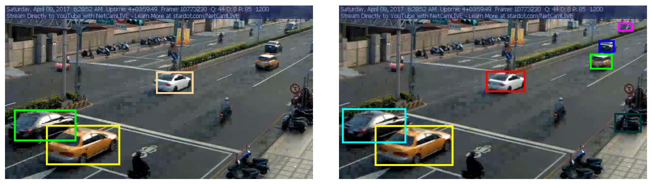


**Figure 8: The original BlazeIt implementation detects cars in the authors' Taipei video with unreasonably poor accuracy. On the left, we show detections taken directly from files published by the authors. Our implementation improves over the original in accuracy and delivers similar speed.**

## 4.6 Implementation Performance Comparison

As discussed above, we re-implement Miris, BlazeIt, and NoScope since the respective authors' implementations are not readily adaptable to our environment. Here, we validate the performance of our implementations by comparing our BlazeIt implementation with the authors' implementation on their Taipei dataset.

On runtime, when excluding video decoding time, our BlazeIt implementation takes 85 seconds to process the 33-hour Taipei video dataset through the proxy model, which is comparable to the 100-second runtime reported by the authors. The authors do not report the overall runtime (i.e., including video decoding).

On accuracy, we find that our implementation, which applies YOLOv3 or Mask R-CNN (whichever yields higher accuracy), captures cars in Taipei far more accurately than the original implementation. For example, in Figure 8, the authors' code only identifies 3 of 6 visible cars, while ours successfully detects all 6 cars with one false positive. Due to the substantially lower detector accuracy in the original implementation, we cannot compare the accuracy of the proxy model in a way consistent with the reported results.

## 5 CONCLUSION

In this paper, we have presented OTIF, a video pre-processor for exploratory video analytics queries. Compared to prior work, OTIF is faster, offering a 6x to 25x average speedup across 7 datasets over video query optimizers at the same accuracy level; is more general, enabling execution of any query that involves object detections and tracks; and substantially reduces query latency after pre-processing, since queries can be answered by processing extracted tracks without additional video decoding and ML inference. More broadly, our results suggest that current video query optimizers face significant challenges in competing with fast object trackers, and that further improving the speed-accuracy tradeoff of object tracking and other computer vision tasks is likely a more promising future direction.

## REFERENCES

[1] 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 *ACM International Conference on Management of Data (SIGMOD)*. 1907–1921.
[2] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, and Ben Upcroft. 2016. Simple Online and Realtime Tracking. In *IEEE International Conference on Image Processing (ICIP)*. IEEE, 3464–3468.
[3] Peng Chu and Haibin Ling. 2019. FAMNet: Joint Learning of Feature, Affinity and Multi-dimensional Assignment for Online Multiple Object Tracking. In *IEEE*

*International Conference on Computer Vision (ICCV)*.

[4] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick Van Der Smagt, Daniel Cremers, and Thomas Brox. 2015. FlowNet: Learning Optical Flow with Convolutional Networks. In *IEEE International Conference on Computer Vision (ICCV)*. 2758–2766.

[5] Mingfei Gao, Ruichi Yu, Ang Li, Vlad I Morariu, and Larry S Davis. 2018. Dynamic zoom-in network for fast object detection in large images. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6926–6935.

[6] Derek Gloudemans and Daniel B Work. 2021. Fast Vehicle Turning-Movement Counting Using Localization-Based Tracking. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4155–4164.

[7] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask R-CNN. In *IEEE International Conference on Computer Vision (ICCV)*. 2961–2969.

[8] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 269–286.

[9] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: Scalable Adaptation of Video Analytics. In *ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*. 253–266.

[10] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. Challenges and Opportunities in DNN-Based Video Analytics: A Demonstration of the BlazeIt Video Query Engine. In *Conference on Innovative Data Systems Research (CIDR)*.

[11] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Neural Network Queries over Video at Scale. In *Proceedings of the VLDB Endowment*.

[12] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2020. Task-agnostic Indexes for Deep Learning-based Queries over Unstructured Data. *arXiv preprint arXiv:2009.04540* (2020).

[13] Chanho Kim, Fuxin Li, and James M Rehg. 2018. Multi-Object Tracking with Neural Gating using Bilinear LSTM. In *European Conference on Computer Vision (ECCV)*. 200–215.

[14] Nick Koudas, Raymond Li, and Ioannis Xarchakos. 2020. Video Monitoring Queries. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 1285–1296.

[15] Qiankun Liu, Qi Chu, Bin Liu, and Nenghai Yu. 2020. GSM: Graph Similarity Model for Multi-Object Tracking. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

[16] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *International Conference on Management of Data (SIGMOD)*. ACM, 1493–1508.

[17] Huizi Mao, Taeyoung Kong, and William J. Dally. 2019. CaTDet: Cascaded Tracked Detector for Efficient Object Detection from Video. In *Conference on Systems and Machine Learning (SysML)*.

[18] Mahyar Najibi, Bharat Singh, and Larry S Davis. 2019. AutoFocus: Efficient Multi-scale Inference. In *IEEE International Conference on Computer Vision (ICCV)*. 9745–9755.

[19] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[20] Joseph Redmon and Ali Farhadi. 2018. *YOLOv3: An Incremental Improvement*. Technical Report. University of Washington.

[21] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. 2004. "GrabCut" interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics (TOG)* 23, 3 (2004), 309–314.

[22] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. 2017. Simple Online and Realtime Tracking with a Deep Association Metric. In *IEEE International Conference on Image Processing (ICIP)*. IEEE, 3645–3649.

[23] Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. 2019. VStore: A Data Store for Analytics on Large Videos. In *European Conference on Computer Systems (EuroSys)*. 1–17.

[24] Dan Zecha, Moritz Einfalt, and Rainer Lienhart. 2019. Refining Joint Locations for Human Pose Tracking in Sports Videos. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

[25] Yang Zhang, Hao Sheng, Yubin Wu, Shuai Wang, Weifeng Lyu, Wei Ke, and Zhang Xiong. 2020. Long-term Tracking with Deep Tracklet Association. *IEEE Transactions on Image Processing* (2020).

[26] Xingyi Zhou, Vladlen Koltun, and Philipp Krähenbühl. 2020. Tracking Objects as Points. In *European Conference on Computer Vision (ECCV)*.

[27] Xizhou Zhu, Yuwen Xiong, Jifeng Dai, Lu Yuan, and Yichen Wei. 2017. Deep Feature Flow for Video Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2349–2358.

[28] Zhe Zhu, Dun Liang, Songhai Zhang, Xiaolei Huang, Baoli Li, and Shimin Hu. 2016. Traffic-Sign Detection and Classification in the Wild. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2110–2118.