EVALUATING TEMPORAL QUERIES OVER VIDEOS

YUETING CHEN

A DISSERTATION SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
JUNE 2023

Abstract

Videos have been an important part of people's daily lives and are continuously growing in terms of volume, size, and variety of content. Recent advances in Computer Vision (CV) algorithms have improved accuracy and efficiency, making video annotations possible with high accuracy. In this work, we follow a general framework to first obtain annotations utilizing state-of-the-art CV algorithms, and then consider three research problems on evaluating temporal queries with such annotations.

We first investigate temporal queries involving objects and their co-occurrences in video feeds. For example, queries that identify video segments during which the same two red cars and the same two humans appear jointly for five minutes are of interest to many applications ranging from law enforcement to security and safety. We take the first step and define such queries in a way that they incorporate certain physical aspects of video capture such as object occlusion. We present an architecture consisting of three layers, namely object detection/tracking, intermediate data generation, and query evaluation. We propose two techniques, Marked Frame Set (MFS) and Sparse State Graph (SSG), to organize all detected objects in the intermediate data generation layer, which effectively, given the queries, minimizes the number of objects and frames that have to be considered during query evaluation. We also introduce an algorithm called SSG-CM that processes incoming frames against the SSG and efficiently prunes objects and frames unrelated to query evaluation, while maintaining all states required for succinct query evaluation.

Then, we consider the query with a ranking mechanism that aims to retrieve clips from large video repositories in which objects co-occur in a query-specified fashion. For example, ranked window queries allow retrieval of clips of a set duration (e.g., 10 seconds) with the highest score from a long video, where at least the same 3 cars (with matching conditions

based on suitably defined metadata) appear jointly. To answer such queries, we propose a two-phased approach, which builds indexes for all desired objects of the given videos during an Ingestion Phase and evaluates query answers efficiently in the Query Phase. During the Ingestion Phase, the proposed Partition-Based Index Construction (PBIC) algorithm builds indexes on partitions obtained by splitting each given video. Leveraging such indexes, queries are answered in the Query Phase using the Partition-Based Query Processing (PBQP) algorithm, which efficiently produces the desired (query-specified) number of results with the highest scores.

Finally, we further consider both spatial and temporal information with graph representations and define the problem of Spatial and Temporal Constrained Ranked Retrieval (STAR Retrieval) over videos. Based on the graph representation, we propose a two-phase approach, consisting of the ingestion phase, where we construct and materialize the Graph Index (GI), and the query phase, where we compute the top-ranked windows (video clips) according to the window matching score efficiently. We propose two algorithms to perform Spatial Matching (SMA) and Temporal Matching (TM) separately with an early-stopping mechanism.

We present the details of the above three research problems and our proposed methods. Via experiments conducted on various datasets, we show the effectiveness of our proposed methods.

Acknowledgements

Completing this dissertation and earning the degree has been a long and challenging journey of growth, uncertainties, and moments of self-doubt. I am filled with gratitude for all the help, support, and encouragement I have received in the past years.

This endeavor would not have been possible without all the support I received from my supervisor, Prof. Xiaohui Yu, whose unwavering dedication and expert guidance have illuminated my path throughout this research journey, and I am truly grateful for the profound impact he has had on my academic and personal growth. I would also like to express my deepest gratitude to Prof. Nick Koudas for his exceptional mentorship and support throughout these years of academic growth, whose guidance, wisdom, and encouragement I have received have been a constant source of inspiration.

I am deeply grateful to my supervisor committee, Prof. Parke Godfrey and Prof. Jimmy Huang, and the examination committee, Prof. Alex Thomo, Prof. Arik Senderovich, and Prof. Ling Jiang, for their meticulous review and insightful suggestions to improve the dissertation. I am also grateful for all the help, knowledge, and fruitful discussions I received from my colleagues, Daren Chao, Dr. Yifan Li, and Prof. Ziqiang Yu, which have been pivotal in broadening my perspectives and making the journey more enjoyable.

I would also like to extend my sincere thanks to my parents, whose unwavering love and support have been the bedrock of my academic journey. Moreover, I am immensely grateful to my beloved wife, Guannan, whose patience, understanding, and unwavering support have been the driving force throughout these years. Without their support, this journey would not have been as rewarding and meaningful.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Background and Motivation

Videos have been widely produced and consumed over recent decades, thanks to the prevalence of smartphones, cameras, and mobile networks. Due to the complexity and richness of video content, video processing and understanding have become a very crucial topic. However, the way of querying video streams is still primitive and requires lots of human intervention. In scenarios such as surveillance applications, humans often have to visually inspect a large amount of video to identify persons or objects of interest. It is crucial to develop systems that could support the extraction of meaningful information from videos, utilizing declarative queries, in a way akin to how people are interacting with RDBMS today.

Some early works in the multimedia community focused on indexing and retrieving videos based on keywords or low-level features [69, 134], due to the limitation of computational resources and image representation. Similarly, early algorithms for computer vision tasks such as object detection [178] require handcrafted features and usually suffer from accuracy and efficiency. Recent advances in Deep Learning [50] have addressed such issues and improved both accuracy and efficiency for many computer vision tasks, including object detection [125], object tracking [18, 152], etc., enabling the opportunities for deploying such algorithms on a much larger scale at lower costs.

Based on such Deep Learning models, there are two main streams of works that aim to address issues regarding video understanding and retrieval. One of them is to develop end-to-end complex networks for specific video understanding tasks. For example, by considering a sequence of images as input, end-to-end networks are designed for action detection and recognition [165, 176, 39]. Recent advances also utilize end-to-end models to retrieve videos

using a natural language interface [100, 104, 113, 38]. However, such works rely heavily on annotated datasets [158] and are tailored to specific query forms, such as natural languages, making it cumbersome to support new videos or new queries.

To provide efficient query answering on videos, there is another line of work that adopts CV models and algorithms as building blocks and leverage them to answer complex queries [67, 82, 87, 106, 10]. The basic ideas are: (1) train cheaper specialized models (proxies) on the specific video and query domain [82, 67] and use them to reduce model inference cost; (2) by optimizing query plans consists of simple predicates [87, 106, 10], the efficiency of query answering can be improved. However, such systems answer queries based on approximations and mainly focus on queries that can be answered by collecting results from single frames. Although there are some works that consider aggregation queries and limit queries [80, 84], the tracking information is usually not involved to guarantee the uniqueness of objects, thus failing to support queries with strict temporal constraints. Other works focus on specific queries have also been studied, for example, queries with spatial constraints [155] or interactions [24], which also demonstrate the potential of video queries.

In this dissertation, we focus on processing videos with structured data representations. Specifically, we obtain annotations for each frame from videos and treat them as structured data. Instead of operating on the raw videos directly, we answer queries based on the annotations only. This decoupling brings us two main benefits: (1) We treat CV algorithms as black boxes, such that new algorithms with higher accuracy could be adopted as soon as they are available. The quality of the annotations can be improved orthogonal to query evaluation. (2) By describing objects with structured data, we could further support strict query semantics such as window semantics and consider the uniqueness of objects that appear across frames. Moreover, new types of queries can also be supported as long as new annotations are available, which provides the ability to open to other query extensions.

In the dissertation, I am going to focus on the problem of evaluating temporal queries over videos with the above setting. The research outline is as follows: we first consider simple temporal queries, where the main focus is to evaluate object co-occurrence relationships with given query conditions (research problem 1). Then, we wish to further optimize such queries with archived videos and ranking mechanisms (research problem 2). Next, we further

consider complex queries involving both spatial and temporal constraints on archived videos with ranked queries (research problem 3). To summarize, We focus on three main research problems, listed as follows:

- Research Problem 1: Evaluating temporal queries over video feeds.
- Research Problem 2: Ranked window query retrieval over video repositories.
- Research Problem 3: Spatial and temporal constrained ranked retrieval over videos.

1.2 Research Problems

In this section, we first present the overall framework that we would like to adopt, which serves as the main foundation of our research. Then, we discuss several interesting topics that we focused on. Finally, we summarize the research problems and discuss future work plans along the research direction.

1.2.1 The Overall Framework

We adopt a two-phase framework to perform query evaluation over videos, where in the first phase, we utilize state-of-the-art CV algorithms to extract structured data from videos, while in the second phase, efficient algorithms are designed to answer queries based on structured annotations. In this framework, we consider CV algorithms as black boxes and only use them to extract structured annotations. This brings us two main benefits: (1) decoupling CV algorithms and video query evaluation allow us to focus on the annotations only, the quality of the annotation can be further improved by either utilizing more accurate algorithms or by humans directly; (2) the type of information included in the annotation could also be enriched by utilizing specialized CV algorithms. For example, by training a DNN model which classifies the color of objects, we could obtain the color annotation for each identified object from the given video.

A video feed V is considered as a bounded sequence of frames, i.e., $V = \langle f_0, \ldots, f_i, \ldots, f_{N-1} \rangle$, where N is the total length of the video. From each frame $f_i \in V$, we can obtain a set of objects O_i. To begin with, we consider two main types of information that can be extracted

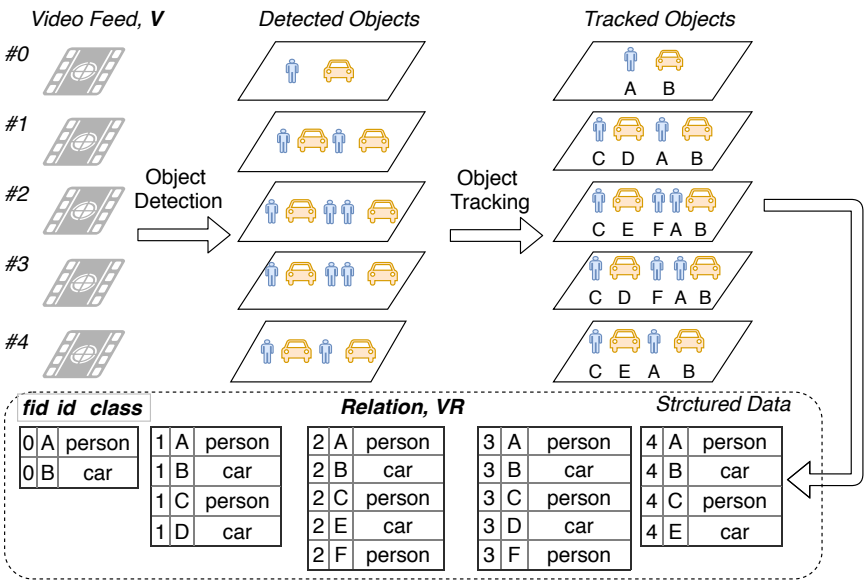


Figure 1.1: Converting a Video into a Relation

for each object $o_j \in O_i$, namely: (1) The type (class) of the object. We use $\mathbb{L}$ to denote the set of all possible classes for all objects, and use $label(o_j) \in \mathbb{L}$ to denote the class of object o_j. Class labels (e.g., person, car) are extracted by applying object detection models on each frame. (2) The unique identifier of the object. We use $\mathbb{ID}$ to denote the set of possible object identifiers (id) that can appear in the video, and use $id(o_j)$ to denote the unique identifier of the object o_j. Such identifiers can be produced by object tracking models [18, 152].

Figure 1.1 presents an example of obtaining a relation from a video feed consisting of five frames. Utilizing object detection algorithms, we can extract a set of bounding boxes that may contain objects for a given frame f_i. Most object detection algorithms also produce a confidence score [149] to assess the probability of the object class appearing in the bounding box. In accordance to prior work, we specify a cut-off threshold, which is applied to the detection results directly. Each result with a score above the threshold is considered a positive detection, and only such results are processed further. Based on the detection results, tracking algorithms [153, 152, 18] then assign unique identifiers to all detected objects. Object tracking algorithms ensure that an object has the same id for all frames in which it appears, even under the presence of occlusions [109, 153, 154]. As such, we obtain a structured relation R_V from the video feed V, with schema $(fid, id, class)$, where fid is the frame id i ($i \in [0, N)$), $id \in \mathbb{ID}$ is the object id, and $class \in \mathbb{L}$ is the class label. For example,

in frame #0, after applying object detection and tracking algorithms, we obtain two unique objects with ids A and B. We store such information as two rows in the Relation, VR: one for object A of class "person", and another for object B of class "car", both of which are generated at frame $fid = 0$.

With the above framework, we could easily obtain the annotations from videos. Moreover, the quality of the annotation could be further improved by better DNN models or by humans directly. In the following research problems, we adopt this framework and focus on answering queries efficiently based on the structured annotation from videos.

1.2.2 Evaluating Temporal Queries Over Video Feeds

In this problem, we initiate a study of temporal queries involving objects and their co-occurrences in video feeds. For example, queries that identify video segments during which the same two red cars and the same two humans appear jointly for five minutes are of interest to many applications ranging from law enforcement to security and safety. We take the first step and define such queries in a way that they incorporate certain physical aspects of video capture such as object occlusion. We present an architecture consisting of three layers, namely object detection/tracking, intermediate data generation, and query evaluation. We propose two techniques, Marked Frame Set (MFS) and Sparse State Graph (SSG), to organize all detected objects in the intermediate data generation layer, which effectively, given the queries, minimizes the number of objects and frames that have to be considered during query evaluation. We also introduce an algorithm called SSG-CM that processes incoming frames against the SSG and efficiently prunes objects and frames unrelated to query evaluation, while maintaining all states required for succinct query evaluation.

We present the results of a thorough experimental evaluation utilizing both real and synthetic data, establishing the trade-offs between MFS and SSG. We stress various parameters of interest in our evaluation and demonstrate that the proposed query evaluation methodology coupled with the proposed algorithms is capable to evaluate temporal queries over video feeds efficiently, achieving orders of magnitude performance benefits.

1.2.3 Ranked Window Query Retrieval over Video Repositories

In this problem, we focus the study of ranked window queries that aim to retrieve clips from large video repositories in which objects co-occur in a query-specified fashion. For example, ranked window queries allow retrieval of clips of a set duration (e.g., 10 seconds) with the highest score from a long video, where at least the same 3 cars (with matching conditions based on suitably defined metadata) appear jointly. To answer such queries, we propose a two-phased approach, which builds indexes for all desired objects of the given videos during an Ingestion Phase and evaluates query answers efficiently in the Query Phase. During the Ingestion Phase, the proposed Partition-Based Index Construction (PBIC) algorithm builds indexes on partitions obtained by splitting each given video. Leveraging such indexes, queries are answered in the Query Phase using the Partition-Based Query Processing (PBQP) algorithm, which efficiently produces the desired (query-specified) number of results with the highest scores. We present the outcome of a thorough performance study on real videos that evaluates the performance of the proposed algorithms by varying parameters of interest. Our results indicate that the proposed set of techniques are capable of processing queries efficiently at scale, demonstrating multiple orders of magnitude speedups over other applicable approaches.

1.2.4 Spatial and Temporal Constrained Ranked Retrieval over Videos

In this problem, we utilize the annotated data provided by such algorithms and construct graph representations to capture both object labels and spatial-temporal relationships of objects in videos. We define the problem of Spatial and Temporal Constrained Ranked Retrieval (STAR Retrieval) over videos. Based on the graph representation, we propose a two-phase approach, consisting of the ingestion phase, where we construct and materialize the Graph Index (GI), and the query phase, where we compute the top-ranked windows (video clips) according to the window-matching score efficiently. We propose two algorithms to perform Spatial Matching (SMA) and Temporal Matching (TM) separately with an early-stopping mechanism. Our experiments demonstrate the effectiveness of the proposed methods, achieving orders of magnitude speedups on queries with high selectivity.

1.3 Summary of Contributions

In this dissertation, we focus on three problems related to temporal query evaluation over videos. Our contributions are summarized as follows:

1. We present a thorough literature review related to our research topic from different aspects, including deep learning for computer vision tasks, which severs as the basic building blocks for obtaining video annotations; the current research status of video query processing systems, where we discuss and compare different approaches; video indexing and retrieval, where works form multimedia literature are discussed; and other works that are related to specific research problems.

2. We study the problem of evaluating temporal queries over video feeds, where we only consider object co-occurrence relationships between frames. Specifically, we assume queries are given in the Conjunctive Normal Form (CNF) and propose two algorithms to efficiently optimizes the input to the query evaluation. In this work, we also adopt a modified version of the CNF evaluation algorithm for faster query evaluation. We conduct a detailed experimental evaluation of the proposed algorithms utilizing both synthetic and real data, demonstrating the trade-offs between the two methods.

3. Based on a variant of the above problem, we further study the efficient evaluation of ranked queries. We propose to further reduce query evaluation time by constructing and materializing indexes. Specifically, we adopt a two-phased approach consisting of both the ingestion phase and the query phase. In the ingestion phase, we develop a Partition-Based Index Construction (PBIC) algorithm to build and materialize videos beforehand, while in the query phase, we propose the Partition-Based Query Processing (PBQP) algorithm to evaluate queries efficiently based on score estimations across multiple partitions. We evaluate our proposed approach with real-world datasets and demonstrate the effectiveness of the proposed methods.

4. We further consider both spatial and temporal information in videos, and propose to use graphs to capture such relationships. To this end, we introduce and formalize the Spatial and Temporal constrAined Ranked Retrieval (STAR Retrieval) problem. We

propose to build and materialize Graph Index (GI) and use two algorithms to perform both spatial and temporal matching efficiently. We conduct our experiments on real-world datasets and demonstrate the improvements in queries with high selectivities.

5. Besides the problems introduced and presented in this dissertation, we also discuss some promising directions toward building query processing systems for videos. We list some of the interesting problems or issues that are important and worth further exploring, which could also potentially enable a variety of systems and applications in different areas.

1.4 Published Works

The dissertation is based on the following published works:

1. Spatial and temporal constrained ranked retrieval over videos.

   Yueting Chen, Nick Koudas, Xiaohui Yu, and Ziqiang Yu. Proceedings of the VLDB Endowment, 15(11). 2022.

2. Ranked Window Query Retrieval over Video Repositories

   Yueting Chen, Xiaohui Yu, and Nick Koudas. 2022 IEEE 38th International Conference on Data Engineering (ICDE). IEEE, 2022.

3. Evaluating Temporal Queries Over Video Feeds.

   Yueting Chen, Xiaohui Yu, Nick Koudas, and Ziqiang Yu. Proceedings of the 2021 International Conference on Management of Data. 2021.

4. TQVS: Temporal Queries over Video Streams in Action.

   Yueting Chen, Xiaohui Yu, and Nick Koudas. Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020.

My other works during my Ph.D. study are listed below:

1. Track Merging for Effective Video Query Processing

Daren Chao, **Yueting Chen**, Nick Koudas, and Xiaohui Yu. 2023 IEEE 38th International Conference on Data Engineering (ICDE). IEEE, 2023.

2. Distributed Processing of k Shortest Path Queries over Dynamic Road Networks

   Ziqiang Yu, Xiaohui Yu, Nick Koudas, Yang Liu, Yifan Li, **Yueting Chen**, Dingyu Yang. Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2020.

2 Literature Review

2.1 Deep Learning for Computer Vision Tasks

Deep Learning [95, 50] has brought significant improvements to computer vision tasks such as image classification [92, 59], object detection [125, 60, 102], instance segmentation [60, 52], object tracking [153, 152, 18, 88, 144], etc. Some early works that have been studied decades ago in this area mainly built upon handcraft features due to the lack of effective image representation [35, 42, 178]. With Deep Learning techniques, convolutional neural networks (CNN) are served as the basic blocks to learn high-level features from images, where the effectiveness is first demonstrated on image classification tasks [92, 59].

Among CV tasks, image classification can be seen as the most fundamental and one of the early works that enable high accuracy on vision tasks. Inspired by the success of DNNs on image classification tasks, Object Detection is also being widely studied, which forms the basis for many other tasks [178] such as instance segmentation, object tracking, etc. In this section, we mainly discuss three main types of computer vision algorithms that are widely being applied to answer queries from videos, namely, image classification (Section 2.1.1), object detection (Section 2.1.2) and object tracking (Section 2.1.3). We then briefly discuss other tasks in Section 2.1.4.

2.1.1 Image Classification

Image classification, or object recognition, has been studied for decades, where the main objective is classifying images into different pre-defined classes. LeNet [97] is one of the earliest works that demonstrate the effectiveness of CNN on document recognition tasks. With advanced hardware, simple classification tasks on relatively small datasets (i.e., NORB

[96], CIFAR [91], etc.) can be solved quite well with simple architectures based on CNN, which can achieve the best error rate comparable to human performance [92, 31]. To further improve the classification accuracy on large datasets such as ImageNet [37], there are many works focusing on better architectures that are able to capture more complex information from images.

AlexNet [92] is one of the early works applying deep CNN networks on the image classification task, where the network contains eight layers: 5 convolutional layers and 3 fully-connected layers. AlexNet uses different convolutional kernel sizes, and large kernel sizes could lead to more parameters. To reduce the number of parameters in the convolutional layers and further increase the depth of the network, VGGNet [131] proposes to use a small kernel with fixed sizes (i.e., 3×3) as building blocks. By stacking a different number of convolutional layers (up to 19 weighted layers), one could construct VGG models with different depths, with deeper models tend to have higher classification accuracy. Instead of simply adding more layers to the network, Inception [138], which is the basic building block for GoogLeNet (which consists of 22 layers), chooses to also expand the width of each layer, where each layer consists of multiple kernels of different sizes to capture different levels of information. As the depth of the network increases, simply stacking more layers could lead to worse performance due to the vanishing gradient. To address the problem, ResNet [59] proposes residual blocks, which leverage identity shortcut connections to skip one or more layers, enabling networks with much deeper architectures (up to 152 layers) and better performance.

There are also many other network architectures proposed such as ZFNet [170], densenet [70], fractalnet [93], etc. As the number of parameters varies in networks with different architectures and depths, the training cost, inference time, and model accuracy also differ, which provides us the opportunities to select appropriate architecture and models according to the target task.

2.1.2 Object Detection

Object detection aims to detect all objects of interest from a given image, where the output result consists of a set of bounding boxes that are able to locate objects within the image.

Since the number of objects is unknown, localizing objects with a deep network is still a challenging task. Before CNNs are applied, Dalal et al. [35] propose to use grids of Histogram of Oriented Gradients (HOG) features along with SVM to create classifiers efficiently.

To leverage features produced by CNN, R-CNN [49] proposes an approach which consists of three main steps: region proposal, CNN feature computation, and region classification. First, selective search [141] is applied to extract around 2000 region proposals on the given image during region proposal stage. Then the proposed regions are wrapped and feed to CNN to obtain feature representations. Finally, SVM are utilized to train classification models based on the features obtained from each region. The final output for the image with localization result is produced based on the classification result on all region proposals with greedy non-maximum suppression [141]. To further speed up the training procedure, Fast R-CNN [48] first feed the input image to the CNN to generate a convolutional feature map, and then identify region proposals on feature maps directly, enabling sharing between regions during feature extraction. However, region proposal with selective search is still time-consuming and becomes the bottleneck. Faster R-CNN [125] uses a separate region proposal network to replace selective search and further improves the network performance.

The above methods can be seen as two-stage approaches, where algorithms consist of both region proposal and region classification. YOLO [122, 123] takes a different approach and predicts both bounding boxes and classification results with a single convolutional network. Specifically, YOLO divides the image into $S \times S$ grid and for each grid cell predicts B bounding boxes and C class probabilities. Since YOLO imposes strong spatial constraints on bounding box predictions (the number of bounding boxes and classes per grid cell are limited), it can struggle with small objects within the image. Similarly, SSD [103] also uses a single network to discretize the output space of bounding boxes into a set of default boxes, and then generates scores for each object category in each default box and produces location adjustments to the box. In general, two-stage approaches are more costly than one-stage approaches but could produce results with higher accuracy [178].

2.1.3 Object Tracking

While both image classification and object detection focus on producing results from one image, object tracking aims to associate objects between images, which is usually applied to obtain unique identifications for objects. According to how many objects are tracked, the tracking algorithms can be divided into two categories: Single Object Tracking (SOT) and Multiple Object Tracking (MOT).

Single Object Tracking (also known as visual tracking), aims to track objects based on a given target on the first image in the given sequence. Recent development of SOT algorithms can be grouped into two main categories: correlation filters-based algorithms and deep learning-based algorithms. Algorithms with correlation filters rely on manual features and are difficult to cope with changing environments [19, 63, 64, 174]. To further provide better feature extraction and fully utilize the recent development of Deep Learning, many different approaches are proposed based on various network architectures [174], including Autoencoder [146], CNNs [144, 62], recurrent neural networks [34], siamese network [139], etc.

Multiple Object Tracking relies on object detection algorithms to obtain bounding boxes and then associate objects across frames [160]. SORT [153] utilizes Faster R-CNN to obtain object detection results, and then performs estimation with Kalman filter [78] and data association with Hungarian algorithm, demonstrating that the object detection quality could have significant impact on tracking performance. To further improve the performance of SORT, an extension, DeepSORT [152], which utilizes features produced by CNN to track objects through longer periods of occlusions and aims reduce the number of identity switches in the tracking result, is also proposed. Tracktor [18] further demonstrates that tracking can be performed with only an object detection method (i.e., by training the network only with the detection task). Recently, many other approaches have also been proposed utilizing CNN with improved performance [160], including FairMOT [173], TransMOT [30], ByteTrack [172], etc.

2.1.4 Other CV Tasks

CNN has also been widely applied in many other computer vision tasks such as Image Segmentation [112], Object Re-identification [164], etc. With the same CNN backbones, multiple CV tasks could be solved within one neural network. For example, He [60] proposed a unified network to produce both object detection and instance segmentation results in one forward pass. Such works provides us the ability to extract various types of information from images directly, which could be further used to serve as the basis of video understanding. There are also other end-to-end networks addressing video tasks directly by taking a sequence of images as the input, such as action detector [176, 39], which is able to obtain complex annotations such as actions and events considering the spatial dimension. End-to-end networks have also been used to perform video retrieval directly, which will be discussed in details in Section 2.3.

2.2 Video Query Processing Systems

Many recent works focus on optimizing query processing over videos. According to their targeting problem and proposed approaches, we divide existing works into categories and discuss them in the following sections:

1. Proxy-based Approaches (Section 2.2.1). In this section, works focus on training specialized models (i.e., proxies) to minimize the evaluation time for given queries. Such proxies are either trained on-demand (with targeting reference model and queries) or pre-trained on specific datasets and available before queries, which will be discussed in Section 2.2.1.

2. Two-phased Approaches (Section 2.2.2). In this section, we discuss works that utilize two-phase approaches to amortize the ingestion cost among all queries to further speed up query answering, detailed in Section 2.2.2.

3. System-level optimizations (Section 2.2.3). In this section, we discuss works that tackle the query efficiency over videos from the systems perspective by applying system-level frameworks which usually consist of multiple steps or optimizations.

4. Other Video Systems (Section 2.2.4). Additionally, we discuss a wide range of video processing systems that do not fall into the above groups, such works focus on different aspects such as scheduling, operator placement, and storage optimizations.

2.2.1 Proxy-based Approaches

MCDNN. MCDNN [55] targets executing DNN models on resource-constrained devices that are intermittently connected to the cloud. Several model optimizations are introduced including specialization and model sharing. With model catalogs registered, the online scheduler uses a heuristic approach to maximize the average accuracy over all requests with constraints on resources used.

NoScope. NoScope [82] is one of the first systems that optimize for accelerating neural network analysis over videos via inference-optimized model search. The system accepts three inputs: the given video, the given object type to detect, and a reference neural network. The system constructs and trains a cascade of binary classification models to replace the reference neural network while preserving accuracy, which is referred to as model specialization. The key intuition is that specialized models can mimic the behavior of a reference model on a particular task while reducing the invocation cost. To leverage the facts that adjacent frames may share similar objects, NoScope also utilizes a difference detection module to efficiently determine whether the content of videos has already changed. Finally, with specialized models and difference detectors, NoScope adopts a cost-based model search strategy to construct, train and utilize models to achieve a target accuracy goal.

TAHOMA. TAHOMA [10] shares a similar idea with NoScope, where hierarchical models are utilized to reduce the runtime cost of queries of visual content, but with a wider design space. Specifically, TAHOMA considers different input shapes of specialized models for the same task, for example, images with different numbers of channels (i.e., RGB images vs grayscale images), with different resolutions, etc. During query plan generation, such transformations on input image representation have also be considered to generate better plans, thus could lead to higher speedups. The specialized models in TAHOMA are still restricted to binary classification models.

BlazeIt. BlazeIt [80, 81] further extends specialization and presents optimizations for

aggregation and limit queries. The system accepts queries in FrameQL format, which is a declarative extension of SQL with semantic support for video-specific queries. BlazeIt utilizes object detection as the reference model and performs object detection over a small sample of frames from videos to obtain annotations as metadata. For a given query, BlazeIt can infer proxy models and filters from query predicates and train them with trained data constructed based on the metadata. To optimize for aggregation queries, BlazeIt tries to train a specialized NN to estimate the statistics, if the training data is insufficient to do so, specialized NNs will be used to reduce the variance of approximate query processing via control variates. For limit queries, BlazeIt trains specialized NNs to predict counts instead of performing binary classifications to alleviate the class imbalance issue.

PP. Lu et al. propose to use Probabilistic Predicates [106], which are essentially cheaper binary classifiers, to accelerate machine learning inference queries with expensive User-Defined Functions (UDFs). PPs are trained beforehand based on binary labeled input data and are annotated with sufficient metadata including the predicate clause it corresponds to, the cost of execution and the predicted data reduction vs. accuracy curve, etc. A query optimizer is then responsible for generating a query plan based on the ordering of simple PPs with the query-level accuracy threshold being satisfied. PPs can be trained without any knowledge of the inference module and thus could be adapted to a broader class of queries and datasets compared with systems that are designed specifically for video queries (e.g., NoScope).

ABAE. ABAE [85] focuses on linear aggregation queries (SUM, COUNT, and AVG) utilizing stratified and pilot sampling with proxies. It assumes proxy models are available to obtain per-record proxy scores and leverages such proxy scores to split records into disjoint groups (strata). ABAE then aims to minimize the mean squared error between the approximate result and the actual result when the query is exhaustively executed by leveraging a two-stage sample-based approach by sampling between strata. Group by and complex predicates are also supported by extending the ABAE approach.

SUPG. SUPG [86] focuses on answering approximate selection queries with statistical accuracy guarantees. Queries can be answered approximately with proxies by specifying a precision target or a recall target. The key idea is to leverage importance sampling to

efficiently sample records from the dataset and label them using the oracle model, and estimate proxy threshold with probabilistic guarantees via confidence intervals.

SVQ. Video monitoring queries [87, 155] targets general queries and considers two types of filters: OD, which is based on object detection techniques, and IC, which utilizes classification models, to assess whether a frame should be processed by expensive DNN models. Moreover, it also provides optimizations for two special cases: aggregation queries involving objects with spatial constraints with statistical techniques, and detecting unexpected objects based on extreme value theory.

SVQ++. Similarly, querying for interactions [156, 24] focuses on identifying interactions from video streams efficiently by limiting the number of frames on which interaction DNN models are applied while maintaining high accuracy. This is realized through a filtering mechanism, where a sequence of filters is available to filter only objects that are promising to contain interactions based on the locations of objects. The optimal order of such filters can be updated dynamically when the selectivity of filters changes, which is realized by a dynamic Kolmogorov-Smirnov (KS) test.

Summary. The main idea of these works is to use cheap models (i.e., specialized models, proxies, filters) to accelerate query answering by replacing the reference model (e.g., NoScope) or by filtering unpromising frames early (e.g., SVQ, PP). Under this setting, one can introduce more knobs (e.g., including image transforms, etc.) to enlarge the design space of searching for the optimal query plan with a cost-based query optimizer. Based on the idea, aggregation queries can be further answered efficiently via statistical-based techniques such as stratified sampling (e.g., ABAE), and control variates (e.g., BlazeIt, SVQ). Such proxy-based approaches usually require extra training time to obtain specialized models and require training data labeled by reference models.

2.2.2 Two-phased Approaches

Focus. Focus aims to enable low-latency and low-cost querying over large historical video datasets by separating the ingestion time and query time. During ingestion time, Focus detects and indexes objects utilizing an approximate indexing scheme with cheap models. During query time, Focus reduces the query latency by clustering objects that are similar

(e.g., consecutive frames) and only applying the reference model on the cluster centroids. To provide trade-offs between ingestion cost and query latency, Focus adjusts parameters used during ingestion and query time according to the given recall and precision.

TASTI. TASTI [84] proposes a more general approach, where indexes are constructed leveraging semantic similarity across records via embedding. Instead of training per-query proxies, TASTI only requires a heuristic for "close" and "far" between a pair of records and trains the embedding with triplet loss. By clustering over the embeddings, TASTI stores the distances of all embeddings to each cluster representative, and uses the k nearest cluster representatives for query processing. Downstream queries could also be answered based on clusters with query-specific proxy scores propagated from the cluster representatives to the unannotated records, which allows for more general and accurate proxy scores across different queries.

Summary. Two-phased approaches are feasible only if: (a) the video is available before queries (i.e., historical videos), and (b) the scope of queries is known (i.e., models or proxies used to answer queries are given beforehand). The main idea of the above works is to build indexes via clustering based on the output or embeddings produced by DNN models, and leverage such indexes to answer queries efficiently.

2.2.3 System-level Optimizations

EVA. EVA [161] based on the idea of materializing and reusing existing results automatically to facilitate faster video analysis via a symbolic approach to analyze reuse opportunities between queries. The system consists of three steps: First, identify overlapping between two UDF invocations via the symbolic approach, which considers three derived predicates: intersection, difference, and union. Then, compute the cost of UDF invocation given the availability of materialized results. Finally, by rewriting the original queries, materialized results could be used or produced by queries. The paper shows via experiments that simple use of materialized data could achieve significant speedups over video queries.

VIVA. VIVA [126] proposes the use of relational hints to consider two main relationships between models: when a model can replace another, and when a model can be used to filter frames for another. VIVA automatically rewrites queries and selects and validates the hints

applicable to the query to find the best performance plan that meets the desired accuracy, which is done via three steps: enumerating all valid plans, estimating plan accuracies using canary inputs and ground truths, and finally selecting plans that satisfy users' requirements.

FiGO. FiGO [22] aims to provide fine-grained query optimization by splitting the video into a sequence of chunks and picking an appropriate model for each chunk. The optimizer starts by treating the video as one chunk, and then iteratively splits it into smaller chunks based on accuracy constraints to achieve the trade-off between query optimization time and query execution time. Models are selected online based on a variant of Thompson sampling.

MIRIS. MIRIS [17] focuses on multi-object tracking queries, where the main idea is to adjust the framerate according to video characteristics. The system starts with tracking objects at a minimum sampling framerate utilizing a graph neural network, followed by the filtering phase to reduce the number of video frames for later processing. The remaining tracks are then determined by uncertainty resolution and refinement, additional video frames will be processed as needed. Recurrent neural networks are trained and applied in the filtering and refinement phases.

OTIF. OTIF [16] focuses on the video pre-processing step, where the objective is to extract all object tracks efficiently from videos. To apply OTIF on a new video dataset, the training set is required for training proxy models, while the validation set is used by the OTIF parameter tuner to produce a speed-accuracy curve, from which a point representing one parameter configuration on the curve will be selected as the parameter for processing the entire video dataset. Besides joint parameter tunning, two main optimizations are introduced in the paper: segmentation proxy model, which identifies regions of frames with no objects and skips costly model inference on those regions, and recurrent reduced-rate tracking, which aims to reduce the cost of identifying all tracks.

Summary. Targeting different aspects in the video analytical systems, the above works either focus on plan optimization based on ideas such as materialized results [161] and relational hints provided by users [126], or focus on fine-grained optimization utilizing chunks [22], or target queries with track results [17, 16]. As more videos and queries are being studied, such works together could lead to more complex and complete video analytical systems that could be applied and deployed to a wide range of application areas.

2.2.4 Other Video Systems

Besides proxy-based and two-phased query evaluations, there are also other works focusing on optimizing the efficiency of query evaluation from other aspects (such as scheduling, operator placement, etc.), which will be discussed in this section.

Smol. Unlike other proxy-based approaches, where the main focus is to reduce DNN invocation cost, Smol [83] targets the cost introduced by preprocessing steps (decoding, transforming, and transferring image data to target devices). Two optimizations are introduced: using natively available low-resolution visual data, and placing preprocessing operators on different devices (i.e., CPU or accelerator) to balance the throughput of preprocessing and DNN execution. Smol also adopts a cost model with preprocessing costs being accounted.

Panorama. Panorama [175] is a data system for unbounded vocabulary querying over videos, where a cascaded model is applied in object recognition. The system generates different feature embeddings for the same input with varying qualities. The performance-accuracy tradeoff is realized via the short-circuiting mechanism based on the accuracy requirement from users and the model accuracy on a validation set.

Optasia. Optasia [105] focuses on executing user-defined queries over surveillance videos. It provides several vision modules as basic building blocks for vision tasks, where each module produces a relational table, which is materialized on demand. Users then issue queries from these tables and the query optimizer optimizes for a collection of queries to find a logical plan with the lowest cost. Optasia also parallelizes the query plans automatically based on factors such as input video size, number of cameras, etc.

VideoStorm. VideoStorm [171] automatically adjusts parameters such as resolution, frame rate, and internal algorithmic parameters to maximize performance on quality and latency for video processing pipelines. The system consists of a centralized manager, which accepts queries, and a set of workers, which execute queries. Queries are described with DAGs of transforms, where each transform processes a sequence of frames and passes its output downstream. The system utilizes transforms as standard components and the core aspect of VideoStorm is the scheduler it provides, which considers the resource-quality profiles of queries and tolerance to latency in processing.

VideoEdge. VideoEdge [72] considers the setting of hierarchical clusters, where the main objective is to select the query plan and place it across the cluster, with more factors considered (such as network, compute, etc.) compared with VideoStorm. VideoEdge also considers merging common components across queries and different implementations of vision components. By treating each cluster as an aggregate bin of resources, the problem of placing query components across the clusters can be mapped to the multiple-choice multi-dimensional knapsack problem, and a heuristic approach with polynomial complexity is proposed.

Chamelon. Chamelon [75] focuses on the configuration selection problem to minimize resource usage while achieving the desired accuracy. The main challenge in this setting is to efficiently perform profiling for the online adaptation of configurations. The key idea in Chamelon is to leverage the underlying characteristics that could persist for each camera over time (temporal correlations) and the similarities across multiple cameras (spatial correlations). With the fact that individual knobs in the configurations have an independent impact on accuracy, the profiling procedure can be done incrementally with grouping optimizations, which can significantly reduce the cost of profiling and searching for the optimal solution.

Scanner. Scanner [119] addresses two main efficient issues in video analysis: efficient pixel data access and parallel processing of pixel-level operations. Scanner organizes video collections as tables in a data store, with optimizations for compressed videos to address the first issue, while the second issue is addressed by treating pixel-analysis tasks as dataflow graphs and optimizing graph scheduling and parallelization on parallel processing resources.

VStore. VStore [159] provides a data store for supporting efficient analytics over large archival videos. For given video streams, VStore saves multiple video versions in different storage formats, which will then be retrieved and converted to consumption formats that are suitable for the executed operators. With a given set of consumers, VStore derives the optimal configuration backward starting from the consumption formats to the storage formats to trade off between consumption speed and storage cost.

Video-zilla. Video-zilla [68] is an indexing layer between video stores and analytical systems, which uses the data unit abstraction called semantic video stream (SVS) to reduce

the redundancy of videos. A hierarchical index is built considering the semantic similarity within and across camera feeds to quickly cluster video feeds.

Smokescreen. Smockscreen [61] based on the video degradation-accuracy profiling model to achieve the tradeoff between aggregation query accuracy and the mounts of degradation. The paper addresses four aggregation functions with reduced frame sampling and proposes a profile repair policy for further improvement.

Summary. As can be noted from the above works, video query evaluation can be further optimized in different settings and from different aspects. For example, Smol considers preprocessing operators in the query plan; with complex network structures, one can further optimize based on a set of queries (i.e., MCDNN), or optimize operator placement in the cluster (i.e., VideoStorm and VideoEdge); as videos captured by a cluster of cameras change over time, one can optimize the configuration selected for the video processing pipelines (i.e., Chamelon); moreover, the cost of storing and retrieving videos for analysis can also be optimized (i.e., Scanner, VStore). As video queries are being studied and applied in more complex scenarios with rich semantics, there are still many promising topics worth further investigation, as pointed out by Hayes et al. [58].

2.3 Video Indexing and Retrieval

Video indexing and retrieval have also been well studied in the multimedia literature [69, 5, 134]. Hu et al. [69] outline a general framework consisting of six stages, among which two, namely *feature extraction*, and *query and retrieval* are most relevant to our research.

For feature extraction, most of the early work focuses on low-level features [21, 163, 56], such as color-based, texture-based, and shape-based features. Although there are some works considering high-level object features, the object types are usually limited to very specific types, such as faces [132, 94]. Spatial-temporal information is considered either via trajectory-based methods [15, 162] or symbolic representation schemes [36]. In *query and retrieval*, existing work has considered queries issued using keywords, natural languages, or by examples (e.g., images, sketches). Early works in this area mainly focus on indexing videos using tags (keywords) based on captions [7] or features extracted from videos directly

[69]. For queries using keywords and natural languages, the query results are based on the summarization or classification of videos [13], while for queries using examples, low-level features are usually considered in finding relevant videos [69]. Such works utilize similarity measures (e.g., feature matching, text matching, etc. [69]) to retrieve results based on the given query and do not support exact pattern matching (i.e., exact matching between object types, etc.).

Recent advances also utilize Deep Learning models to train joint networks [38, 168] aiming to support video retrieval using natural language queries. Moment retrieval problems (i.e., retrieving video clips according to a given query within a video repository) have also been studied (e.g., [113, 101]). The main approach is to map both videos and queries into a common latent space [100, 71] via multimodal machine learning techniques [14]. These methods require models to be trained end-to-end for specific tasks with annotated training data [90, 158] limiting the expressibility of the query types that can be answered to those present on training data. Moreover, they return subsets of all possible answers, typically evaluated via precision and recall (as opposed to exact answers). Sufficiently large and error-free training data are still not available for effective learning of models for complex and variable video data [104], making it cumbersome to support new videos or new queries, especially to support queries with more precise constraints.

2.4 Other Related Works

We further discuss some existing works related to specific research problems. Specifically, we first discuss related works to the first research problem in data mining in Section 2.4.1, and then discuss top-k query processing related to research problems 2 and 3 in Section 2.4.2, finally, graph-related problems that are relevant to research problem 3 are discussed in Section 2.4.3.

2.4.1 Data Mining

If we treat objects as items and consider the video as a sequence of itemsets, the problem is also related to data mining problems. In the data mining community, frequent pattern

mining is a well-studied topic [26, 53], where the goal is to mine the hidden patterns of the frequent itemsets. Many classic algorithms and their variants are proposed and widely applied to many applications, including Apriori [8], FP-Growth [54], Tree-Projection [6], etc. In a similar context, sequence pattern mining [9, 43] is another related topic that has been widely explored, where the sequential order of data is further being considered. Numerous methods, including Generalized Sequential Patterns (GSP) [135], SPADE [169], PrefixSpan [118], SPAM [12], BIDE [143], etc., have been proposed to discover interesting subsequences from a set of sequences. Most of the early works are based on the entire database and thus cannot be applied to temporal data directly.

Recently, there are also works focused on frequent itemset mining [47], association rule mining [98, 145], temporal pattern mining [27], frequent arrangements discovery [117] in the context of temporal data. Some work also has been conducted in the context of mining frequent itemsets [29, 76, 116] and sequential patterns [120, 65, 23] over sliding windows. The main focus of these works is to produce itemsets given a support/confidence threshold and do not support query constraints. Moreover, such works mainly consider the general cases where no assumption can be made on the temporal data. In the context of video processing, objects produced by adjacent frames are likely to be stable (i.e., objects identified from adjacent frames are likely to overlap), which could be taken into consideration.

2.4.2 Top-k Query Processing

Both problems 2 and 3 target ranked queries. In relational databases, top-k Query Processing is a well-studied problem [73], where many classic algorithms are proposed.

Many works focus on producing exact top-k items from the ranked results. Such works assume multiple lists that rank the same set of objects based on different scoring predicates, and a score aggregation function is used to aggregate partial scores to obtain final top-k results [73]. In [73], according to the assumption on access methods supported by data sources, top-k processing techniques can be divided into three groups: (1) require both sorted and random access, e.g., TA algorithm [41]; (2) require no random access, e.g., NRA algorithm [40], Stream-Combine algorithm [51]; (3) require sorted access with controlled random probes, e.g., Upper and Pick algorithms [20, 108]. Among them, the threshold

algorithm [41], also known as TA, is one of the best known algorithms that targets for general purposes, which assumes that the costs of different access methods are the same and there is no restrictions on the number of random access performed. Many variants are also being proposed based on TA [115, 25]. However, such approaches assumes objects and their scores are already computed and can be easily retrieved, which is not the case for queries on video annotations since results for complex queries need to be computed dynamically. In our work, we take similar ideas from existing works and mainly focus on how to produce score estimation efficiently based on the data available.

Instead of answering exact queries, approximate top-k query processing has also been studied. For example, the approximation version of J* algorithm [115] further improves the performance by producing an ϵ-approximation to the top-k answers with correctness being sacrificed. Theobald et al. [140] further provide accuracy guarantees based on approximate variants of the TA algorithm. On another related thread, top-k query processing over uncertain data has also been investigated. In probabilistic databases, an additional ranking dimension, probabilities, along with scores need to be factored in the interpretation of top-k queries [73, 133, 46, 77, 121] Recent works also focus on top-k queries in specific settings, including queries over spatial data [147], efficient parallel processing [129], queries on encrypted databases [110], etc. In our work, we focus on specific top-k scenarios in videos and existing approaches cannot be applied directly.

2.4.3 Graph-related Works

In problem 3, we utilize graphs to model the spatial relationship between objects. Subsequently, videos can be modeled as either a time-evolving graph or a sequence of graphs.

Time-evolving graphs have been widely studied in application scenarios such as social networks [137, 99], where the main focus is either to identify sub-graphs that match the given pattern [44, 127, 128, 136], or perform data mining tasks such as interesting pattern mining [114, 137]. However, existing graph algorithms proposed in the context of social networks are not directly applicable here; the typical assumption in the social network setting is that the patterns are to be evaluated directly on the entire graph, which is infeasible in our setting.

If we consider videos as sequences of graphs, executing queries on graphs can be modeled

as subgraph isomorphism problems. On another related thread, the subgraph isomorphism problem has been well-studied, where the main objective is to find graphs containing a given query graph from a large set of graphs. Such works mainly target large graphs [130, 33], where the number of nodes is relatively large and the degree of each node varies, which can be used as the filter condition or part of the query constraints [130], or complex graphs that can be further decomposed into basic structures [74] or subgraphs [151]. In problem 3, we leverage the characteristics of spatial relationships between video objects and simplify the query graph to reduce the complexity of graph matching, such that graph isomorphism testing can be avoided.

3 Evaluating Temporal Queries Over Video Feeds

3.1 Introduction

3.1.1 Background and Motivation

In this work, we are concerned with applications that involve video collected from static cameras as it is prevalent in surveillance and security applications. Our focus however is on the efficient execution of *temporal queries* over video feeds. A video feed is an ordered sequence of frames, presented at a specific *frame rate*, typically 30 frames per second (*fps*) to align with human visual perception. Different types of objects appear in frames and can be easily detected by state-of-the-art object detection algorithms. Similarly, DL models are highly effective in tracking the same object (e.g., car) from one frame to the next, as objects persist across frames. Given this basic ability to detect and track objects across the temporal evolution of a video feed, several queries of interest arise. For example, we may be interested to discover video segments during which only the exact same two cars (and no other object) appear continuously within a 10-second interval; similarly, we may wish to detect video segments during which the same three trucks and the same human object appear for a duration of one minute jointly (possibly along with many other cars/trucks and humans among them). These queries become highly sophisticated when the query objects can be associated with arbitrary Conjunctive Normal Form (CNF) expressions. Moreover, when query-specified objects can be connected to external identities (e.g., license plates) such queries become highly powerful.

Such queries have obvious applications in video surveillance analysis. As an additional application domain, consider the large repositories of driving footage utilized in training autonomous vehicles [142]. Our techniques assist in "debugging" training data, namely

identifying whether sufficient clips in the training data exist for specific query conditions. During a recent accident with autonomous vehicles [142] it was determined that training data did not contain sufficient footage of multiple white trucks jointly in the screen. Techniques such as those proposed herein assist to effectively "debug" training data and pinpoint whether the issues are with the training data or the supporting algorithms. Efficient execution of such queries can automate many of these tasks.

3.1.2 Our Proposal

Evaluating queries on video feeds involving multiple objects and temporal constraints presents numerous challenges, however. First, finding video frames where a query (e.g., two white trucks appear jointly for a period of time) is satisfied can be very expensive since both the actual objects and frames that satisfy the query are uncertain, which means that a brute-force approach to consider all possible object combinations and frames, thus leading to a large search space. Sets of objects that are not promising to be part of any query answer should be pruned immediately from further evaluation. Second, *occlusions* occur naturally among objects in videos, where some objects disappear from the visible screen in some frames (because of being blocked by other objects) and then reappear. For example, in traffic camera video footage, smaller cars may disappear behind a tractor-trailer or bus for a few seconds or longer and reappear. Occlusions can be captured by state-of-the-art object tracking algorithms [89, 153], and thus they need to be considered in the query semantics. Similarly, although state-of-the-art object tracking algorithms perform well when tracking an object across frames, they are still error-prone. As a result, some objects may disappear or be assigned new identifiers in some frames. Temporal query semantics has to account for pragmatic challenges such as occlusion, and subsequent query evaluation needs to take such semantics into account.

In this chapter, we place the problem of evaluating temporal queries over video feeds into perspective, and in particular, make the following contributions:

- We define semantics for temporal queries on video feeds accounting for aspects intrinsic to video objects such as object occlusion.

- We decouple object detection/classification from query evaluation, introducing an intermediate layer (MCOS Generation) that optimizes the input to the query evaluation module.

- We fully develop the MCOS Generation module, detailing a way to organize objects detected through state maintenance.

- We propose *Marked Frame Set* (MFS) to prune unnecessary states early. Additionally, to provide even better pruning efficiency in certain scenarios, we also propose to maintain states in a graph structure called *Sparse State Graph* (SSG).

- We propose an algorithm called *SSG-CM* that processes incoming frames against the SSG, dynamically adjusting its structure and incrementally and efficiently maintaining all primitives necessary for succinct query evaluation.

- We propose several modifications to a CNF query execution algorithm, and demonstrate how to efficiently couple CNF query execution with algorithm SSG-CM, yielding improved performance in our application scenario.

- We present a detailed experimental evaluation of the proposed algorithms utilizing both synthetic and real data, demonstrating the trade-offs between MFS and SSG. Both approaches for state maintenance yield significant performance improvements over baselines on real datasets, with benefits increasing as query parameters vary. We also demonstrate that adapting our state maintenance during query evaluation yields an optimized approach that exhibits significant speedup on query evaluation, up to more than 100 times in our experiments.

3.2 Problem Definition

We consider a video feed V as a bounded sequence of frames, i.e., $V = \langle f_0, \ldots, f_i, \ldots, f_{N-1} \rangle$, where N is the total length of the video. We use $\mathbb{ID}$ to denote the set of possible object identifiers (id) that can appear in the video feed (unique identifiers that are persistent for each detected object across video frames). Typically such identifiers are extracted by the

application of object tracking models [89]. We use $\mathbb{L}$ to denote the set of class labels for all objects. Class labels (e.g., *person*, *car*) are extracted by the application of object classification/detection models on each frame. Object detection algorithms produce each detection result with an associated score. In our approach, in accordance to prior work [177], we specify a cut-off threshold, which is applied to the detection results directly. Each result with a score above the threshold is considered a positive detection, and only such results are processed further. Such an approach has clear semantics and yields results that are easy to interpret. We wish to highlight, however, that such an approach has inherent limitations. Ultimately the results we present and their accuracy depend on the accuracy of the underlying object detection models. By imposing a cutoff threshold and treating results above the threshold as positive, we may miss detected results that are positive but have a lower score. Moreover, false positives may be introduced in the query processing pipeline. Although state-of-the-art detection algorithms showcase impressive accuracy [125] this is still a possible concern. An alternative approach would be to associate with each detection a confidence level and embed the detections into a probabilistic framework; such confidence levels can be computed in various ways (e.g., they can be derived from the detection scores, or calculated based on an ensemble of detection models). This is an orthogonal approach that is of interest to explore further as part of future work.

Figure 3.1 presents an example of a relation obtained from some video feed consisting of five frames. Specifically, utilizing object detection and tracking algorithms, we can extract a set of objects O_i from each frame f_i, and for each $o_i \in O_i$, its id $id(o_i)$ and its class label $label(o_i)$. Thus, we obtain a structured relation VR from the video feed V, with schema $(fid, id, class)$, where fid is the frame id i ($i \in [0, N)$), $id \in \mathbb{ID}$ is the object id, and $class \in \mathbb{L}$ is the class label. For example, in frame #0, after applying object detection and tracking algorithms, we obtain two unique objects with ids A and B. We store such information as two rows in the Relation, VR: one for object A of class "person", and another for object B of class "car", both of which are generated at frame $fid = 0$. Object tracking algorithms ensure that an object has the same id for all frames in which it appears, even under the presence of occlusions [109, 153, 154].

Processing relation VR one can specify numerous queries to reveal certain important

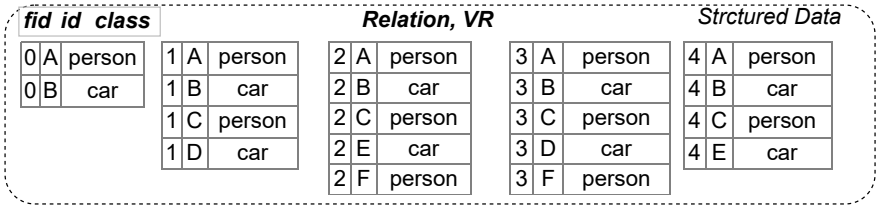


Figure 3.1: A Relation Representing a Video Feed

properties regarding the objects in the underlying video stream. One important class of queries concerns conducting query processing on sets of objects that are co-occurring in a collection of frames (co-occurrence object sets). Without occlusions, such queries can be evaluated by comparing the object sets from each frame with its previous frame. However, it is typical that objects would be occluded from the visible scene (e.g. a pedestrian may be hidden behind a vehicle for a while).

We wish to support arbitrary *Conjunctive Normal Form* (CNF) queries over video feeds. To express queries involving co-occurrence object sets involving occlusions, we specify a user-defined function (UDF) named COEVAL based on standard SQL, shown in Figure 3.2. VR represents the relation extracted from the given video feed. Each query is expressed along with a temporal window context, which is defined through WINDOW_SPEC. Rows falling into the same window are evaluated by COEVAL, which accepts two parameters: CNF_EXP, which expresses the query condition in CNF (being fully aware of the schema of VR), and the occlusion parameter, d, which sets a minimum threshold for the number of frames that objects co-occur within a window. Each condition c in the CNF expression is of the form $obj(x, threshold)\ \theta\ n$, where obj is a user-defined function, which determines the number of objects that are detected as label x with score $\geq threshold$ for a given object set, n is an integer value, and $\theta \in \{\leq, =, \geq\}$. Different from any other aggregation functions, the CNF expression is evaluated on co-occurring objects, where the exact same objects have to appear in at least d frames from the same window. Based on the syntax, occlusions can be easily considered by setting d to some value that is smaller than the window size.

For example, a query requesting that at least one person and one car appear jointly can be expressed as Figure 3.3. In this example, we assume the query is evaluated based on a sliding window consisting of 4 frames (specified by "RANGE BETWEEN ... ROW"), and

```
SELECT *,
    COEVAL(
        CNF_EXP,
        d
    ) OVER
        WINDOW_SPEC
    AS eval
FROM
    VR
WHERE eval = TRUE
```

Figure 3.2: SQL Reference

```
SELECT DISTINCT FID,
    COEVAL(
        obj(person,0.5) ≥ 1 ∧ obj(car,0.5) ≥1,
        3
    ) OVER (
        ORDER BY FID
        RANGE BETWEEN INTERVAL '4' FRAMES
        PRECEDING AND CURRENT ROW
    ) AS eval
FROM
    VR
WHERE eval = TRUE
```

Figure 3.3: Example Query

we require the number of frames the objects appear jointly to be at least 3 frames (that is, we allow any of the objects to be occluded for at most 1 frame). The associated CNF expression is $obj(person, 0.5) \geq 1 \wedge obj(car, 0.5) \geq 1$, which means we consider object sets containing at least 1 person and 1 car with detection score ≥ 0.5. In this example, we the last frame id from VR that satisfy the query in each window. In some cases, users may only focus on the result frames or the objects only, for which the select clause can be altered accordingly. Such queries can also be used as building blocks to support higher-level queries, such as querying the maximal video segment consisting of a series of sliding windows on which the query evaluates to TRUE.

To answer such queries, we have to enumerate all possible co-occurrence object sets in each window during evaluation. The user-defined function $obj(x, threshold)$ is treated as a filtering operation based on the score of each object during object detection on each frame. Thus, all objects and tracked results are produced after applying such parameters. For this reason, the CNF expression can be simplified to $person \geq 1 \wedge car \geq 1$. Suppose we evaluate the above query on the relation extracted in Figure 3.1.

In the window of frames #1 to #4, we can compute one of the co-occurrence object sets as {A,B,C}, consisting of objects appearing jointly in the frame set {1,2,3,4}. This object set consists of 2 persons and 1 car; thus it satisfies the query conditions and should be produced as a result. By the same token object sets {A,B} as well as {B,C} evaluate to TRUE on the same query (both consist of 1 person and 1 car), but arguably are subsumed by the object set {A,B,C} since they all appear in the same frame set.

From this simple example, it is evident that the query answer semantics merit careful

consideration, as we do not wish to overwhelm users with multiple query results from each window. As the CNF expressions vary along with queries, each window can generate a very large number of possible query answers. To address this issue, we introduce stricter semantics, necessitating that queries are evaluated over maximal co-occurring object sets within a window. That way, each query will be evaluated on a single object set within each window. We refer to such object sets as Maximum Co-occurrence Object Sets, defined as follows:

DEFINITION 1 (Maximum Co-occurrence Object Set (MCOS)). *Given a set of frames* $\mathbb{F}'$, $\mathbb{ID}'$ *is the maximum co-occurrence object set iff:*

- $\forall f \in \mathbb{F}'$, *all objects in* $\mathbb{ID}'$ *appear in* f.
- $\forall id \in \mathbb{ID}$, $\exists f \in \mathbb{F}'$, *where* id *does not appear in* f.

For a given frame set $\mathbb{F}'$, an object set $\mathbb{ID}'$ is a maximum co-occurrence object set if and only if none of its supersets is a co-occurrence object set of $\mathbb{F}'$. In this case, we say $\mathbb{ID}'$ is the MCOS of $\mathbb{F}'$, or simply, $\mathbb{ID}'$ is an MCOS. Each unique frame set can produce only one MCOS. In the example above, for the frame set {1,2,3,4}, the only MCOS is {A,B,C}, which contains two persons and one car. Notice that if the query requests the co-occurrence of exactly one car and one person in that window, the query will evaluate to false in the window {1,2,3,4}, as it will be executed on the MCOS of the frame set only[1]. That way we can regulate the query results per window, as we are operating on the MCOS of each window only and adjusting the window size and occlusion parameter d per query. Nevertheless, our framework can easily support alternate desired semantics within a window once these are specified. Trivially, we can support the execution of the query over all co-occurrence object sets within a window if that is required.

In this problem, we focus on sliding window query semantics[2]; each query is expressed with a temporal window context w. The window advances every time a new frame is encountered and the query is evaluated over the most recently encountered w frames.

[1]It will, however, evaluate to TRUE and report on {A,B} in the previous window {0,1,2,3}.

[2]Other options are possible, such as tumbling window, and our solution will work equally well with such semantics.

3.3 Overview

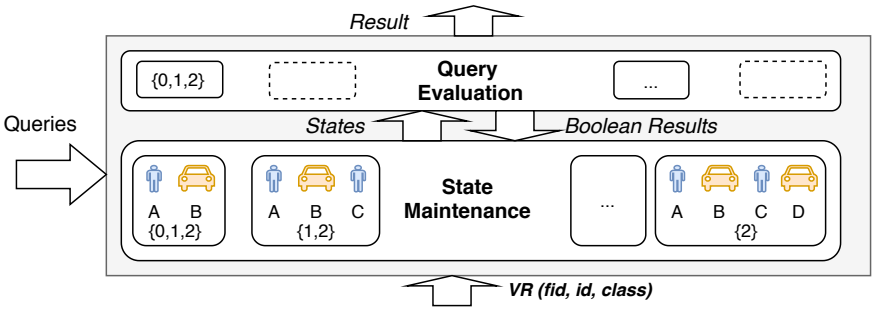


Figure 3.4: Architecture Overview

The overall architecture of our solution is shown in Figure 3.4. Our solution consists of two modules, namely State Maintenance, and Query Evaluation. Object detection and tracking algorithms are applied to obtain the structured relation VR, as described in Section 3.2. This design allows any state-of-the-art algorithm from the computer vision community to be adopted as it becomes available.

Instead of evaluating queries based on the relation directly, we compute all possible object sets and their frame sets via the State Maintenance module. This module is responsible for generating and materializing all possible candidates for query evaluation. We use **State** as the base unit to store candidates. States in this module can be maintained incrementally by reusing intermediate states from the previous window. Each state consists of two types of information: an object set and the frames in which the objects appeared[3]. Note that the object set may not be an MCOS in some states. A state s is a *valid state* iff $\mathbb{ID}_s$ is an MCOS of frame set $\mathbb{F}_s$. Otherwise, it is an *invalid state*. State Maintenance should only produce valid states to Query Evaluation since, in our treatment, queries are evaluated on MCOS's only.

We use $\mathbb{S}$ to represent the set of all maintained states in the current window. For each valid state, $s \in \mathbb{S}$, processing on the MCOS can be divided into three steps:

1. Check against the occlusion parameter d. The query could be evaluated to TRUE only if $|\mathbb{F}_s| \geq d$.

[3]This definition of states can easily support any semantics for query evaluation, not just MCOS.

2. For each condition in CNF expressions evaluate its result.

3. Once all conditions of the expression have been processed, we can move on to evaluate CNF expressions. We adapt CNF evaluation algorithms [150] for this step.

In the State Maintenance module, simple optimizations can reduce the number of states evaluated on CNF. For example, testing the occlusion parameter d can be conducted when the MCOS's are generated (a form of push-down optimization). Namely, a state s containing an MCOS can be pruned (not relayed to the Query Evaluation module) if $|\mathbb{F}_s| < d$ (*unsatisfied states*). States with $|\mathbb{F}_s| \geq d$ will not be pruned (*satisfied states*). Subsequently, states containing an MCOS (i.e., *valid* states) will be relayed to the Query Evaluation module. Since we are evaluating queries over a window, some states that are not satisfied at frame i may be satisfied in the future as the window slides. We may still have to maintain them. Consequently, state maintenance requires succinct book-keeping to ensure both efficient execution and correctness.

In summary, queries are analyzed first, and objects that are not required for the given query are discarded immediately. The State Maintenance module maintains states for each window and updates them incrementally. States can be generated, satisfied, and invalid as new frames arrive. Only satisfied and valid states ($\mathbb{F}_s \geq d$, containing an MCOS) are relayed to the Query Evaluation module, where the final result is produced based on CNF expressions.

We propose the Marked Frame Set approach in the next section to reduce the number of states maintained. We further explore the relationships between states and propose an improved approach, Sparse State Graph, to reduce the number of states processed with minimum overhead (in Section 3.5). Finally, we discuss the Query Evaluation module in Section 3.6.

3.4 Marked Frame Set

3.4.1 A First Attempt

According to its definition, an MCOS can be simply generated by intersecting object sets from all of the frames in a given window. However, there might exist numerous possible combinations of frames inside a given window, which could lead to large computational costs.

A simple way to optimize this is to maintain only one state per unique object set, and reuse states from the previous window to derive new states. When the window slides, frames that are removed from the window are denoted as *expired*. For each existing state, we remove the expired frame ids from its frame set. A state can be removed immediately once its frame set is empty. New states can then be generated based on object set intersections between existing states and the new arriving frames.

The procedure can be described formally as follows. Let the set of all states in the current window be $\mathbb{S}$, and assume that the last frame id in the current window is i. When a new frame with id $i' = i + 1$ arrives with associated object set $\mathbb{ID}_{o_{i'}}$, the following steps are conducted: (1) Append the new frame id: $\forall s \in \mathbb{S}$, if $\exists s' \in \mathbb{S}$, $\mathbb{ID}_s \cap \mathbb{ID}_{o_{i'}} = \mathbb{ID}_{s'}$, the new frame id can be simply appended to the existing state s'. (2) Create new states: (a) For each state $s \in \mathbb{S}$, let $\mathbb{ID}' = \mathbb{ID}_s \cap \mathbb{ID}_{o_{i'}}$. If $\forall s' \in \mathbb{S}$, $\mathbb{ID}' \neq \mathbb{ID}_{s'}$, then a new state ns is created with object set $\mathbb{ID}'$. In this case, $\mathbb{F}_s \subset \mathbb{F}_{ns}$. (b) If $\forall s \in \mathbb{S}$, $\mathbb{ID}_s \cap \mathbb{ID}_{o_{i'}} \neq \mathbb{ID}_{o_{i'}}$. This is the case where no state has the same object set as the arriving frame. A new state ns is created with the new frame id as the only element in $\mathbb{F}_{ns}$.

Example 1. *Consider the state maintenance of the example query (Figure 3.3) on the relation shown in Figure 3.1. To better illustrate the procedure, we first show all possible object sets that can be generated at different frames in Figure 3.5. In Figure 3.6, we highlight all valid states (states having MCOS) with shade areas and use a bold font to denote a satisfied state (number of frames* ≥ 3*). We ignore the asterisk(*) marked in each state in this example. As can be observed, all states are generated with an MCOS. However, as window slides, although the state with O*1 *(*$\{A,B\}$*) is still satisfied, it may become invalid at frame #4, where the only MCOS for frame set* $\{1,2,3,4\}$ *is object set O*4*.*

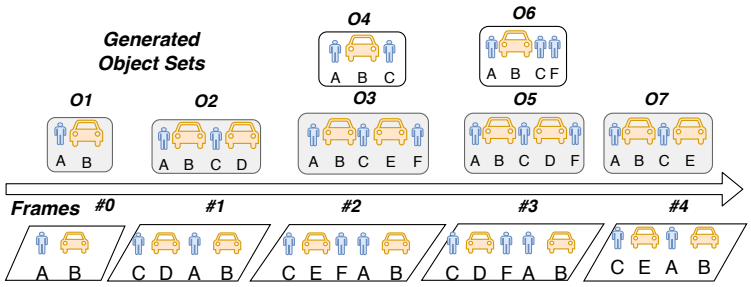


Figure 3.5: Object Sets Generated for the Example Query

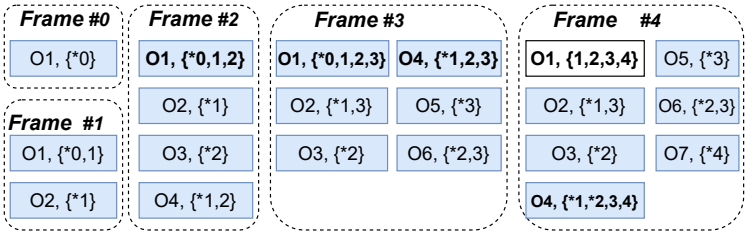


Figure 3.6: States Generated from Each Frame for the Example Query

In the above example, the state with object set $O1$ at frame #4 is invalid and should not be processed further and/or evaluated. Maintaining invalid states could impose unnecessary overhead as they will be processed against new arriving frames. A new mechanism has to be introduced to address this issue.

3.4.2 Principal States

From the example shown in Section 3.4.1, we can observe that certain states have the same object sets with some frames in the video, while for other states their object sets are generated from more than one frame. To distinguish these two types of states, we introduce the notion of *Principal State* to denote states that have the same object sets with some frames in the current window, defined as follows.

Definition 2 (Principal State). *For a window size w, let $V_{i,w} =< f_{max(0,i-w)}, ..., f_i >$ be the corresponding video frames. A state s in the current window is a principal state iff $\exists f_j \in V_{i,w}$, such that $\mathbb{ID}_{f_j} = \mathbb{ID}_s$.*

A state may become a principal state when a new frame, k, arrives: a) if $\exists s \in \mathbb{S}$, such that s has the same object set as the new frame k, namely $\mathbb{ID}_{f_k} = \mathbb{ID}_s$, then s will become a principal state; 2) if none of the states in $\mathbb{S}$ has the same object set as the new frame, a

new state s will be created with object set of frame k and become a principal state. We say that a principal state s is created from frame k, and that frame k created principal state s. A principal state will cease being one and become invalid, once all the frames that have created it expire from the current window.

Principal states carry an important property. Consider a window on the relation; starting with one frame, the only existing state (that was created from the frame) is a principal state. If we add one more frame to the window, a new state may be created based on the existing principal state and the new principal state. By keeping adding more frames, new states can always be created based on existing states and the new principal state. Thus, every state in the current window is generated by principal states either *directly or indirectly*.

3.4.3 The MFS Approach

We propose to mark, under certain conditions, frames in the frame set, and denote the marked set as **Marked Frame Set**. Suppose a new frame i arrives, with the principal state ns. For each state $s \in \mathbb{S}$, frames are marked according to the **Frame Marking Rules**, as follows:

1. For the state ns that is created by frame i (Step 2.b in Section 3.4.1), frame i will always be marked in $\mathbb{F}_{ns}$.

2. If $\exists s' \in \mathbb{S}$, $\mathbb{ID}_{s'} \cap \mathbb{ID}_{ns} = \mathbb{ID}_s$, $\forall f \neq i \in \mathbb{F}_{s'}$, if f is marked in $\mathbb{F}_{s'}$, then f will also be marked in $\mathbb{F}_s$.

When these rules are applied, we have the following theorem.

THEOREM 1. *With the Frame Marking Rules, a state is valid iff at least one frame in its frame set is marked.*

PROOF 1. *Recall that all states are generated based on object set intersections between some existing state and the new principal state. Let s be a state generated from two other states s_i and s_j, where s_j is the new principal state. (1) If s_i is a principal state created from frames i, where $i < j$, then only frame i is marked in s. Assume that the object set of s cannot be generated from other states. After frame i expires, s will become invalid as it shares the*

same frame set as s_j. (2) If s_i is not a principal state, then we can find two other states s'_i and s'_j to generate s_i, one of which is also a principal state. By keeping doing this, we can find a set of principal states S' that generate state s indirectly. Let s' be the earliest principal state that is created at some frame i'. Then only frame i' is marked in state s. After frame i' expires, s will also become invalid, while no frame is marked in the frame set.

According to Theorem 1, a state can be pruned safely once none of the frames in its frame set is marked. Following the Frame Marking Rules, we propose the MFS approach, which utilizes a set of states to maintain all possible object sets along with their Marked Frame Set. In particular, in MFS, we maintain a set of existing states from the previous window; when a new frame arrives, we compute the object set intersection between each existing state and the new frame and mark frames according to the Frame Marking Rules. We show in the following example that the Marked Frame Set can reduce the maintenance cost significantly while ensuring the correct result is produced.

EXAMPLE 2. *Consider the example query (Figure 3.3) over the video feed (Figure 3.1) again. We add an asterisk (*) before a frame id to denote a marked frame (Figure 3.6). The state with object set $O1$ is generated at frame #0 with frame 0 marked. From frame #1 to #3, the new frame id is appended to the existing state. When frame #4 arrives, frame #0 is removed from the set along with the mark. As all marked frames have been removed from the state, the state becomes invalid and gets removed. Thus, at frame #4, only the state with $O4$ will be selected as the satisfied state.*

3.5 Sparse State Graph

3.5.1 State Graph

We have established that by marking certain frames, the number of states maintained can be minimized to provide efficiency. To further reduce the number of states to be explored in each window, investigating the relationship between states is a natural extension. To capture the relationship of how the states are generated, we use **State Graph** to connect those states, defined as follows:

Definition 3 (State Graph (SG)). *A state graph is a directed graph represented as* $G = (\mathbb{S}, \mathbb{E})$*, where* $\mathbb{S}$ *is the set of all states (nodes) and* $\mathbb{E}$ *is the set of all edges. Each state* $s \in \mathbb{S}$ *contains a unique object set along with its Marked Frame Set. Each edge* $(s_i, s_j) \in \mathbb{E}$ *(where* $s_i, s_j \in \mathbb{S}$*) represents that state* s_j *could be generated from state* s_i*.*

As discussed in Section 3.4.2, each state is generated by an existing state, and the principal state is created from the new arriving frame. An SG can be constructed by simply connecting states naturally in the way they are generated. In the case of a new state s being generated by some existing state s' and a new principal state ns, we can simply add two edges (s', s) and (ns, s), since the new state s could be generated from both states. To update the states in SG, we can use the set of principal states as starting points, as all other states are generated by principal states either directly or indirectly (as introduced in Section 3.4.2) and are thus reachable from principal states by a Depth-First Search (DFS) on SG.

Note that we do not have to traverse the whole graph for state maintenance (e.g., to process the arrival of a new frame), as the structure of SG allows us to stop early when certain conditions are met. More specifically, for an edge (s_i, s_j) in SG, according to Definition 3 the following must hold: $\mathbb{ID}_{s_j} \subset \mathbb{ID}_{s_i}$. Therefore, during the insertion of a new principal state s', if we find that the object set intersection between s_i and s' is empty, then there is no need to visit the adjacent states of s_i anymore, as the object set intersection between any of those states and s' is guaranteed to be empty. However, even with this optimization, there will still be duplicate visits to the states, as shown by the example below.

Example 3. *Suppose we are building a state graph starting from frame #1 for the video sequence shown in Figure 3.5. To ease the illustration, we omit the frame set and use the object set to annotate each state. The state graph generated at frame #2 is shown in Figure 3.7a, where principal states are shaded. When frame #3 ({A,B,C,D,F}) arrives, we visit the graph starting with principal state* s_1 *using depth-first search. The object set intersections computed based on states* s_1 *and* s_3 *remain the same as the existing states, and no new state is generated. A new state* s_5 *is generated by computing the object set intersection between the principal state* s_2 *and the new frame as well as connecting the states based on how they are generated, as shown in Figure 3.7b.*

To understand the problem with this way of constructing the graph, consider the scenario

where another new frame with object set $\{E,G\}$ arrives afterward. Its intersections with states s_1 and s_4 (which are empty) render visiting state s_3 unnecessary according to Property 1. Unfortunately, due to the existence of edge (s_2, s_3), state s_3 will still be visited nonetheless, since the object set intersection between s_2 and the new frame is non-empty.

(a) Before Processing Frame #3

(b) After Processing Frame #3

Figure 3.7: The State Graph for the Example Shown in Figure 3.5

The above example illustrates that simply connecting states in the way they are generated does not help to prune due to redundant edges. The graph structure should be restructured to provide better pruning power.

3.5.2 Sparse State Graph

The main challenges are therefore (1) how to restructure SG to obtain better pruning power, and (2) how to achieve that while minimizing the maintenance cost.

We first focus on the pruning issue. In the above example, edge (s_2, s_3) is considered redundant as it prevents us from pruning state s_3, and removing it will improve the pruning power of the graph. However, this edge was not redundant at creation; it only becomes redundant after frame #3 arrives. Evidently, dynamic modifications on the SG are required after a new frame arrives.

One straightforward solution is to check each existing edge and restructure the graph if necessary every time a new frame arrives. However, if many edges have to change as new frames arrive, the benefit of removing redundant edges may be outweighed by the extra maintenance cost.

To reduce the maintenance cost, we seek to further reduce the number of edges in the graph. Since we are visiting SG starting from *all* principal states, all edges between principal states can be removed. Moreover, if a state s is relevant to the new frame (i.e., the object

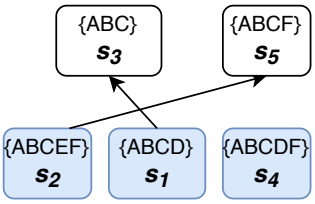


Figure 3.8: Reducing the Number of Edges in Figure 3.7b

set intersection between s and the new frame is non-empty), s is guaranteed to be visited as long as there is a path from some principal state to s. This allows us to limit each state to have at most one incoming edge and still obtain the correct result. Thus, in what follows we will focus on a specialized type of state graph called Sparse State Graphs (SSG).

Definition 4 (Sparse State Graph (SSG)). *A Sparse State Graph is a state graph* $G = (\mathbb{S}, \mathbb{E})$ *that satisfies the following properties:*

- $\nexists s_i, s_j$ *s.t.* $\exists e = (s_i, s_j) \in \mathbb{E}$*, where* s_i *and* s_j *are principal states.*
- $\forall e_1, e_2 \in \mathbb{E}$*, where* $e_1 = (s_i, s_j), e_2 = (s_k, s_l)$*, we always have* $s_j \neq s_l$*; in another word, there does not exist a state with more than one incoming edge.*

An example of SSG based on the SG in Figure 3.7b is shown in Figure 3.8. Note that compared with Figure 3.7b, edge (s_4, s_1) is removed as both s_1 and s_4 are principal states, as per property (1). Edges (s_2, s_3), (s_4, s_3), and (s_4, s_5) are removed due to property (2). In the remainder of this section, we present our method to build and maintain such SSGs.

3.5.3 Edge Maintenance

As per Definition 4, we can only keep at most one incoming edge for each state. To determine the strategy of edge maintenance on SSG, two questions have to be answered: When should we perform edge maintenance, and how to perform it?

Before answering the above questions, we first consider the life cycle of a state. A state can be created or removed when the window slides. Since each state in SSG contains a unique object set, a state must also be updated when it is possible to generate its object set by computing object set intersections between some existing states and the new arriving frame.

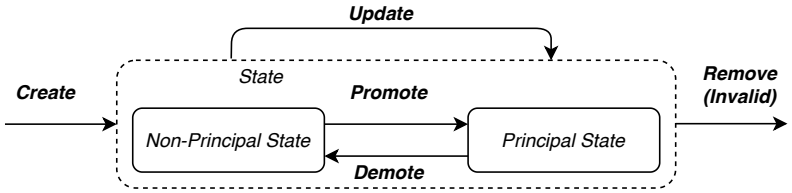


Figure 3.9: Life Cycle of a State

Thus, the complete life cycle of a state consists of three actions, *create*, *update* (optional), and *remove*. In the meantime, a non-principal state can also later become a principal state (called *promote*, if the state shares the same object set as the new frame), and vice versa (called *demote*, caused by frame expiry). The complete life cycle is illustrated in Figure 3.9.

When to Perform State Maintenance: As discussed in the previous section, minimizing maintenance cost is crucial to performance improvement; frequent edge creation and removal should be avoided. Therefore, we apply edge modification only when necessary, namely,

1. When a state is created or removed: edge maintenance is essential to guarantee that each state is reachable from principal states;

2. When a state s is promoted: the incoming edge of s should be removed; or

3. When a state s is demoted: a new edge should be added to guarantee that s is still reachable from a principal state.

In short, edge maintenance is required if any action, except for an update, is taken in a life cycle of a state.

How to Perform State Maintenance: When a state s is promoted, edge maintenance can be done by simply removing the existing incoming edge (if any) of s, since we can now reach s directly as a principal state.

When a state s is demoted, a new edge is required to guarantee the demoted state is still reachable from some principal state. To avoid extra computation, we simply retrieve the first marked frame, i', of the demoted state s and add a new edge (ps, s), where ps is the principal state created by frame i'.

Recall that a state s is created only if the object set intersection between some existing state s' and the new frame is non-empty. We use ns to denote the principal state that shares the same object set as the new frame. To provide better pruning power, edge (s', s) should be created instead of (ns, s). The intuition behind this is that adding the edge (ns, s) would result in a "shallower" graph, where all states can be directly reached from principal states; but preference should be given to "deeper" graphs as they would afford us more opportunities to prune away unnecessary states thanks to the hierarchical structure.

When a state s is removed, its incoming edge (if any) should also be removed. Consequently, the adjacent states of s are no longer reachable. This can be fixed by reconnecting the adjacent states of s to some existing states. Assume that s', s'' are two other states with edges $(s'', s), (s, s')$. Before removing both edges, we check if state s'' is still valid. If so, we simply create a new edge (s'', s'). In this case, state s' is still reachable from some existing state s'' on the graph. Otherwise, we can try to apply this rule recursively, until we find such a state to connect to or end up with no such state found. In the latter case, we still need to add state s' to the graph, which is the same as the demotion case as above.

In summary, we apply the following rules to each state s triggered by actions in the life cycle:

1. If state s is created by object set intersections between some existing state s' and a new frame, we add a new edge(s', s).

2. If state s is updated, we only update its Marked Frame Set, with no edge modification performed.

3. If s is removed, we remove its associated edges, and add new edges between some existing states and s's adjacent states.

4. If s is promoted, we simply remove its incoming edge.

5. If s is demoted, we find an existing principal state and add an edge between that principal state and s.

3.5.4 SSG-CM Algorithm

3.5.4.1 Algorithm Overview

Based on the above rules, we present our algorithm, called the SSG-CM Algorithm (for SSG Construction and Maintenance), to build and maintain SSG as shown in Algorithm 1. Similar to the MFS Approach (Section 3.4.3), to guarantee that each state contains a unique object set, we use a hash table OSSHash to store the mapping between each unique object set and the state that contains it. Once a new object set has arrived or been created, we first perform a lookup in OSSHash to determine whether there is already a state with the same object set (Line 6). Since all states are generated based on principal states, we maintain a list of principal states in the SSG in the order of the smallest frame number in its Marked Frame Set and use them as the starting points of the graph visit. For each state s we keep its adjacent states in a list (denoted as *s.next*), and the previous state in the graph (denoted as *s.prev*).

Algorithm 1: SSG-CM Algorithm

```
   Input: a new frame i, with object set objSet
 1 if PSList.size() = w then
 2 |   os ← PSList.remove(0);
 3 |   updateEdges(os, null);
 4 end
 5 // look up the state in the hash table and create one if not exists.;
 6 ns ← lookupOrCreate(objSet);
 7 add and mark frame i in ns;
 8 for ps : PSList do
 9 |   visitStates(ps, i, ns, true);
10 end
11 if ns.prev ≠ null then
12 |   ns.prev.next.remove(ns);
13 |   ns.prev ← null;
14 end
15 PSList.add(ns);
```

When a new frame arrives, we first check if there is any expired frame, and remove all invalid states (Lines 1-4). Then, starting from each principal state in order, we visit states in the graph and update or generate states accordingly (Lines 8-10). If a state is promoted

to a principal state, the existing incoming edge is removed (Lines 11-14).

To provide all satisfied states to the Query Evaluation module, we test whether a state is satisfied once a state is created or updated. Since it is trivial to add the pseudo-code of such testing, it is not presented in Algorithm 1 for simplicity. In the algorithm, we utilize two procedures, *visitStates* and *updateEdges*, which are presented in detail in Sections 3.5.4.2 and 3.5.4.3.

3.5.4.2 Visiting States

The *visitStates* procedure visits all states in the graph, but only computes object set intersections for states that are relevant to the new arriving frame. States could be created, updated, or removed during this process.

Algorithm 2: visitStates

```
   Input: state s being visited, current frame i, new principal state ps, whether to
          compute object set intersections toCompute
 1 if s.flag = i then return;
 2 s.flag ← i;
 3 pruneState(s);
 4 if s is invalid then
 5 |   remove s and modify related edges;
 6 else
 7 |   inter ← ID_s ∩ ID_ps;
 8 |   if |inter| > 0 then
 9 |   |   if |inter| = |ID_s| then
10 |   |   |   appendFrameToAllNodes(s, i); return;
11 |   |   else
12 |   |   |   s' ← lookupOrCreate(inter);
13 |   |   |   mergeFrames(s', s); add frame i in s';
14 |   |   |   toCompute ← true;
15 |   |   |   if s' is newly created then s.next.add(s') ;
16 |   |   end
17 |   else
18 |   |   compute ← false;
19 |   end
20 end
21 for next : s.next do visitStates(next, i, ps, toCompute) ;
```

Starting from each principal state, we perform a Depth-First Search on all states in the

graph by applying Algorithm 2 recursively. We use a Boolean value *toCompute* to indicate whether we need to compute the object set intersection for the current state. Note that since multiple frames may share the same principal state, we use a flag to avoid visiting the same state more than once (Lines 1-2). For each state s to visit, we first remove expired frames from its Marked Frame Set, utilizing the *pruneState* function. If the current state s is invalid, we remove it and reconnect related edges (Line 5). Otherwise, if the object set intersection result is non-empty, we look up the state with this intersection in OSSHash, or create a new state if needed (Line 12). We use s' to denote the retrieved or new state. The Marked Frame Set of s' is updated according to state s (Line 13) (same as Step 2 in Frame Marking Rules, Section 3.4.3). We create an edge between s and s' only if state s' is newly created (Line 15). The adjacent states of s will be further visited until all states are processed (Line 18).

We also provide a minor optimization of this procedure. If the object set of the current state s (ID_s) is the same as the intersection result (*inter*), for any state s' that is adjacent to s, the result of object set intersection will always be the same as the object set in s' ($\mathbb{ID}_{s'} \cap \mathbb{ID}_{ps} = \mathbb{ID}_{s'}$). As such, we can simply append the new frame to the MFS of all adjacent states of s without computing the object set intersections (Line 10). This applies recursively.

3.5.4.3 Updating Edges

Before updating existing edges, we have to check whether there is an expiring frame. If so, its associated principal state should be demoted or removed. This is done through Algorithm 3, which also adopts the DFS algorithm.

The key task of this algorithm is to handle edge modification after some states expire, starting with the given principal state. If the current state s is invalid, we clean up related edges and visit its adjacent states recursively (Line 7). Otherwise, we find a principal state according to the first marked frame in s (Lines 9-10) and update related edges. The removal of other invalid states is handled by the *visitStates* procedure.

Algorithm 3: updateEdges

```
   Input: current state s
 1 pruneState(s);
 2 if no frame is marked in s then
 3 |   remove s from hash table;
 4 |   if s.prev ≠ null then
 5 |   |   s.prev.next.remove(s); s.prev ← null;
 6 |   end
 7 |   for next : s.next do updateEdges(next) ;
 8 else
 9 |   mark ← the next marked frame in as;
10 |   as ← the principal state created at frame mark;
11 |   if as ≠ s then
12 |   |   remove incoming edges on state s if there is any;
13 |   |   if s is not a principal state then
14 |   |   |   as.next.add(s); s.prev ← as;
15 |   |   end
16 |   end
17 end
```

3.5.4.4 Complexity Analysis

Suppose the number of frames that share the same object set is λ on average. So for a given window size w, approximately $\frac{w}{\lambda}$ unique object sets are obtained by processing the raw frames, which result in $\frac{w}{\lambda}$ unique principal states, denoted as $x = \frac{w}{\lambda}$. In the worst case, new states are generated by each existing state and the object set of a newly arriving frame. We use a_x to denote the total number of states corresponding to the x principal states. We have $a_x = 2 * a_{x-1} + 1$, starting with $a_1 = 1$. Thus, the total number of states is $a_x = 2^x - 1$. Since each state has at most one incoming edge, the maximum number of edges in an SSG would be the same as the number of states. Moreover, principal states do not have incoming edges. Therefore, the total number of edges is $2^x - 1 - x$. The time complexity of the algorithm is thus $O(V + E)$=$O(2^x - 1 + 2^x - 1 - x) = O(2^x)$, where $x = \frac{w}{\lambda}$.

3.6 Query Evaluation

3.6.1 The CNFEval Algorithm

For our query evaluation module, any CNF evaluation algorithm can be employed. We choose to utilize the evaluation algorithm proposed in [150], which will be referred to as *CNFEval*.

The CNFEval algorithm evaluates a set of CNF queries that include $\in$, $\notin$ (set membership) predicates. The algorithm first builds an inverted index for the starting set of queries. The inverted index is dynamically maintained as queries are inserted and/or deleted.

For example, query $q_1 = age \in \{2, 3\} \wedge (state \in \{CA\} \vee gender \in \{F\})$ specifies a CNF on name-value pairs. Specifically, this query can be interpreted as follows: the value for "age" is 2 or 3; the value for "state" is "CA"; and the value for "gender" is "F".

An inverted index for q_1 containing posting lists is generated. A posting list is a list of triples that are generated according to query conditions. A triplet is represented as $(qid, p, disjId)$, where qid is the query id, p is the predicate ($\in$ or $\notin$), and $disjId$ is the disjunction id (starting from 0) of the condition. For example, condition $gender \in \{F\}$ is in the second disjunction, and thus a triplet (1, $\in$, 1) is generated.

For a given set of name-value pairs, the algorithm retrieves posting lists from the inverted index first. Queries can be answered by ordering and scanning all retrieved posting lists. For example, given an input $\{(age, 3), (gender, F)\}$, two posting lists $(1, \in, 0)$ and $(1, \in, 1)$ are retrieved. From the two triplets in the lists, we determine that both disjunctions 0 and 1 are satisfied. Thus, query q_1 is evaluated as TRUE. The details of the algorithm are available elsewhere [150].

3.6.2 CNF Evaluation with Inequality Predicates

The CNFEval algorithm only handles queries containing set predicates. As a result inequality conditions are not supported naturally. Thus, we propose to build three separate inverted indexes for the query conditions of the form $label\ \theta\ n$ ($label$ is used as the key, while n is used as the value), with an extra column, value, associated to each of the three cases, $\leq, \geq, =$.

Table 3.1: $\geq$ Index

Key	Value	Posting List
Car	2	(2, 0)
	3	(2, 1)
Person	2	(2, 1)

Table 3.2: $\leq$ Index

Key	Value	Posting List
Car	5	(2, 2)
Person	3	(2, 0)

Consider a query $q_2 = (car \geq 2 \vee person \leq 3) \wedge (car \geq 3 \vee person \geq 2) \wedge (car \leq 5)$ in our framework. Tables 3.1 and 3.2 depict the inverted indexes for this query. Since this query contains both $\geq$ and $\leq$ conditions, two inverted indexes are built. Each key in the inverted list is associated with an ordered list; the ordering takes place by value in descending (if θ is $\leq$) or ascending (if θ is $\geq$) order. Given an input with a set of name-value pairs, posting lists are retrieved as follows. For each name-value pair, (k, v):

1. Retrieve the ordered list from both tables where key=k.
2. Retrieve posting lists in order where value $\leq v$ (if θ is $\geq$) or value $\geq v$ (if θ is $\leq$).

We refer to this version of the algorithm, which encompasses our enhancements, as *CNFEvalE*. It is utilized to evaluate queries on the states produced by the State Maintenance module. The procedure to evaluate queries on video feeds is described as follows:

1. For the given queries, we first build inverted lists.
2. For each state, s, generated by the State Maintenance module:
   (a) A set of aggregate values, $\mathbb{A}_s$, is computed based on the MCOS of state s. Each aggregate value is represented as (l, v), where $l \in \mathbb{L}$ is the class label, while v is the number of objects of class l.
   (b) The aggregate value set, $\mathbb{A}_s$, is utilized as input to CNFEvalE to produce the results.
   (c) If some query q evaluates to TRUE, the frame set of the state, $\mathbb{F}_s$, is produced as the result.

3.6.3 Pruning States by Evaluation Results

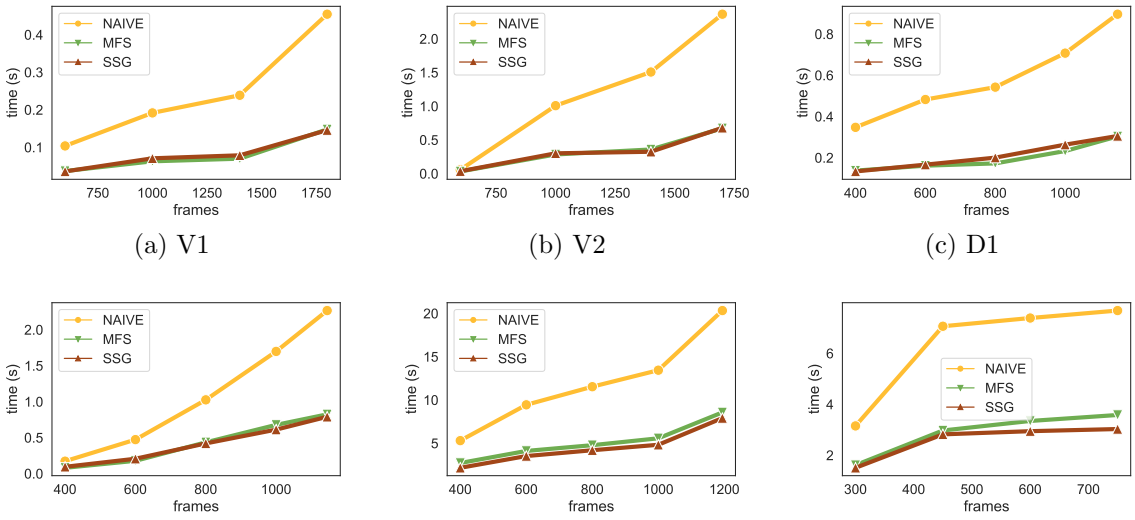
Query evaluation proceeds on the MCOS of a satisfied and valid state produced by the State Maintenance module. We make the following observation: if queries evaluate to FALSE on the MCOS generated by some state s, they will always evaluate to FALSE on every possible MCOS generated by any state s' generated from s. This observation is captured by the following proposition.

PROPOSITION 1. *Let s be some state in an SSG, and $\mathbb{ID}_s$ be the MCOS generated by s. For each condition c in query q, if c evaluates to FALSE on $\mathbb{ID}_s$, then for any state s' with $\mathbb{ID}_{s'}$, where $\mathbb{ID}_{s'} \subset \mathbb{ID}_s$, the condition c will always evaluate to FALSE on $\mathbb{ID}_{s'}$.*

If Proposition 1 holds, such states can be safely removed from the graph while guaranteeing correctness. Proposition 1 can be satisfied only if θ is $\geq$. To adopt such a pruning strategy, for a given set of queries, we test whether they contain $\geq$ predicates only. If so, when a new state is created (Figure 3.9), we evaluate the MCOS of that state using the Query Evaluation module. In this case, we do not wait for the entire Result State Set to be generated. If all queries evaluate as FALSE, the state will be marked as "terminated". Terminated states will no longer be processed, and thus we can reduce the number of states maintained in the State Maintenance module. This provides additional pruning flexibility to the State Maintenance module in certain cases.

3.7 Experiments

In this section, we present a thorough experimental evaluation of the two proposed approaches (Marked Frame Set and Sparse State Graph) on both synthetic and real datasets and study the tradeoffs between them. We have also evaluated an algorithm based on the original State Graph presented in Section 3.5.1, but it performs significantly (at least 50%) worse than MFS and SSG in all cases, and thus is omitted in this section. We first evaluate the State Maintenance methods proposed in this chapter in Section 3.7.2, and then study the performance of Query Evaluation in Section 3.7.3.



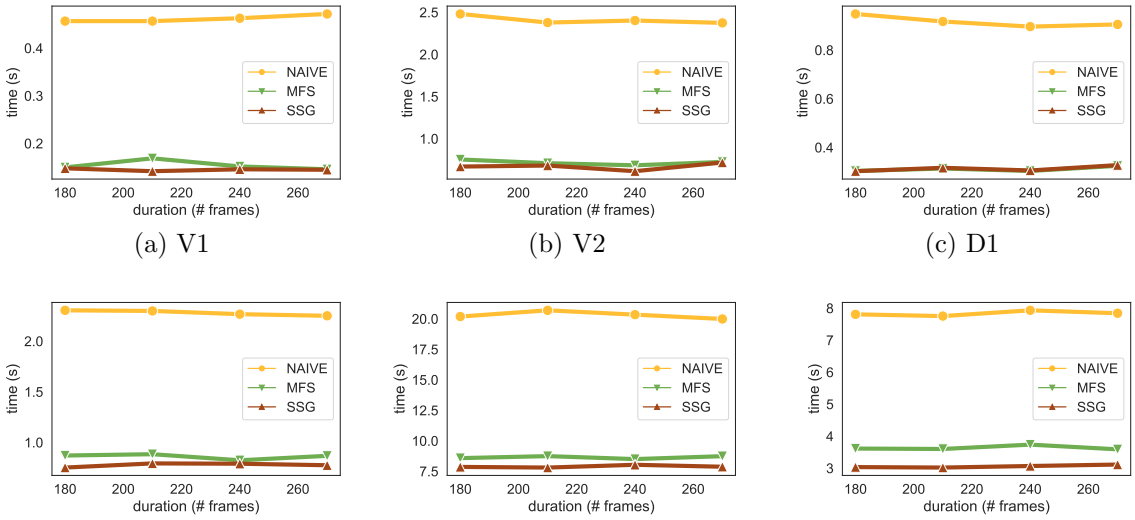
(a) V1 (b) V2 (c) D1

(d) D2 (e) M1 (f) M2

Figure 3.10: Varying the Total Number of Frames

Table 3.3: Dataset Statistics

Dataset	V1	V2	D1	D2	M1	M2
Frames	1800	1700	1145	1150	1194	750
Objects	84	131	155	117	175	181
Obj/F	4.62	7.54	7.36	9.92	11.83	13.81
Occ/Obj	1.07	1.16	0.60	0.89	2.6	1.07
F/Obj	99.08	97.87	72.43	73.35	80.8	57.47



(a) V1 (b) V2 (c) D1

(d) D2 (e) M1 (f) M2

Figure 3.11: Varying Duration d

3.7.1 Settings

Environment. Our experiments are conducted on a machine with one Intel i5-9600K CPU, one GeForce GTX 1070 GPU and 16 GB RAM. All algorithms are implemented in Java [4] except the object detection and tracking algorithms for which we utilize standard implementations.

Object Detection and Tracking. We adopt YOLO [122] as the object detection algorithm to identify all objects in videos and Deep SORT [153] as the object tracking algorithm. For both algorithms, we utilize standard implementation from open-source projects [1, 2]. Note that the object detection and tracking algorithms can be replaced with any other available algorithms. The main difference between different algorithms is the number of objects produced in the annotation result. For simplicity, in our experiment, we use the same object detection algorithm, while the effect of variable objects in each frame is mainly studied by varying different videos. As a comparison, we also adopt another implementation [3] which uses MobileNet [66] as the object detection algorithm.

Synthetic Datasets. We use synthetic videos generated by the VisualRoad benchmark [57], which is proposed to evaluate the performance of video database management systems (VDBMSs). We select two representative videos provided on the project homepage[4]: rain with light traffic (V1), and postpluvial with heavy traffic (V2). We also generate videos with different configurations to study the relationship between the performance and the number of objects per frame.

Real Datasets. We also evaluate our approach based on four videos from Detrac [107] and MOT16 datasets [111]. From Detrac, we use MVI_40171 (D1) and MVI_40751 (D2). From MOT16, we use MOT16-06 (M1) and MOT16-13 (M2). We also evaluated our approach utilizing large publicly available data sets from Youtube as outlined in Section 3.7.3.

Dataset Statistics. In the rest of this section, we use V1 and V2 to denote two synthetic videos, while using D1, D2, and M1, M2 to denote four real videos. D1 and D2 are captured by static cameras, *while M1 and M2 are captured by moving cameras.* In our experiments, we only identify objects belonging to one of the following classes: person, car, truck, and

[4]Code available at `https://github.com/yestinchen/cnf_eval`

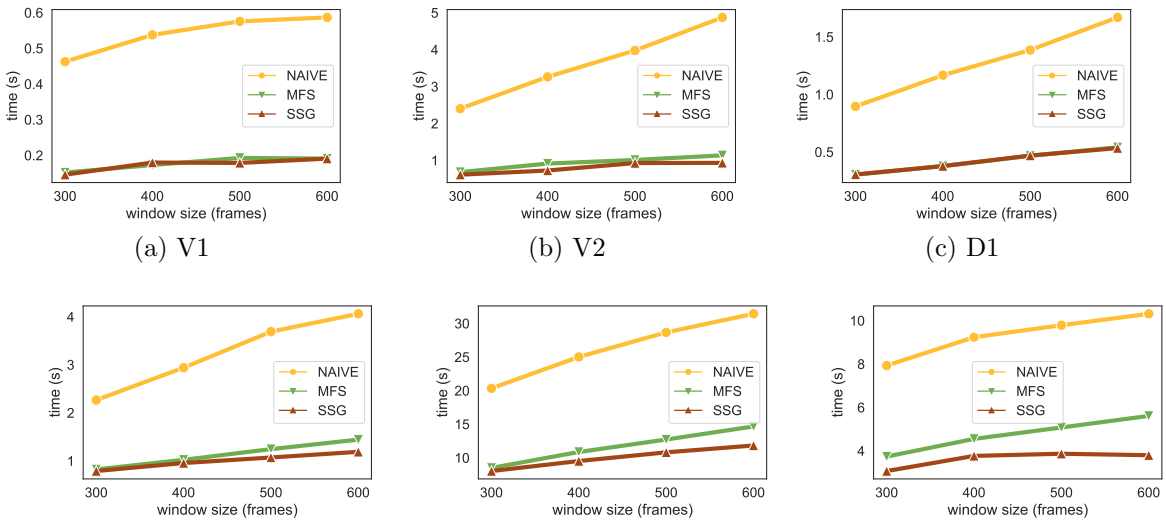
bus. The statistical information on these videos after applying object detection and tracking algorithms is summarized in Table 3.3 with default settings. In the table, Frames and Objects represent the total number of frames and unique objects, respectively; Obj/F represents the average number of objects per frame; Occ/Obj represents the average number of occlusions per object; while F/Obj denotes the number of frames that each object appears on average.

3.7.2 State Maintenance

We implement a baseline approach that simply stores the set of frames that each object set that appears. We first collect all the object sets that satisfy the given occlusion threshold and then check whether they share the same frame set. If that is the case, according to the definition of MCOS, we only keep the object set with the maximum size. We refer to this approach as *NAIVE*. We also implemented the MFS approach as described in Section 3.4.3, as well as SSG as presented in Section 3.5.4. All three methods are memory-based, where a hash table is used to map the object sets to frame sets (NAIVE), or Marked Frame Sets (MFS), or internal states (SSG). To study the performance of the three methods, we design experiments to measure only the State Maintenance time.

We vary the window size, w (default value is set to 300 frames), and the occlusion parameter, d (default value is set to 240 frames). At 30 fps, the default setting aims to identify objects that appear for at least 8 seconds in a window of 10 seconds. The number of occlusions per object also varied during experiments. By default we do not vary occlusion; each video has a number of occlusions per object occurring naturally in each dataset (Table 3.3).

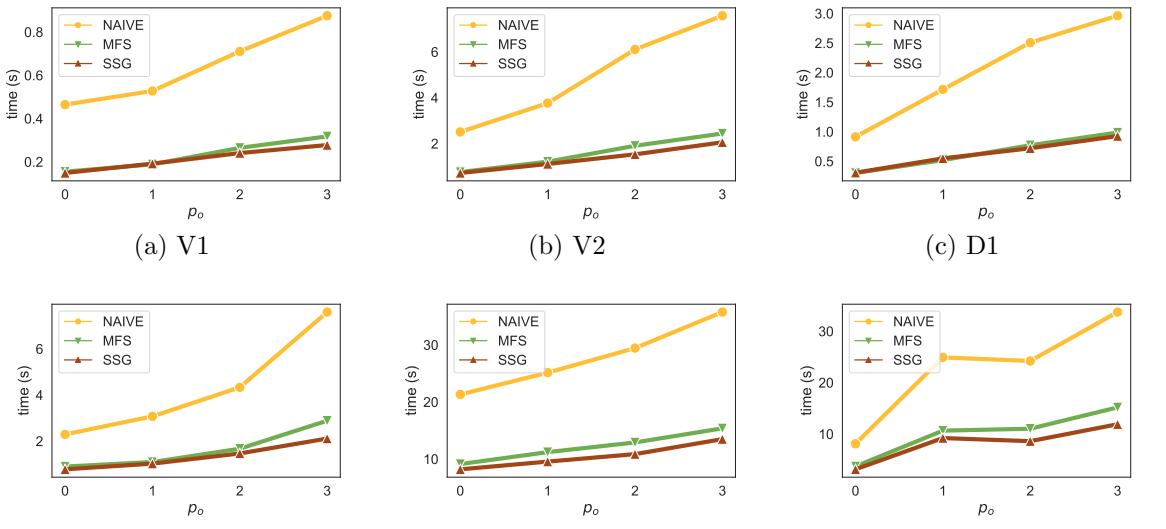
We then vary the number of total frames evaluated on datasets, shown as Figure 3.10. The performance of all methods demonstrates a similar general trend: when the number of frames increases, the total time also increases. In all three methods, MCOS's are generated based on the object sets; thus the number of distinct object sets can have a profound impact on the overall performance. As such, even under the same window size and occlusion threshold, their performance on different videos can vary significantly. For example, in Figure 3.10f, processing the first 450 frames is very expensive, but the total cost does not increase significantly as the number of frames increases. In contrast, processing the *last* 194



(a) V1 (b) V2 (c) D1

(d) D2 (e) M1 (f) M2

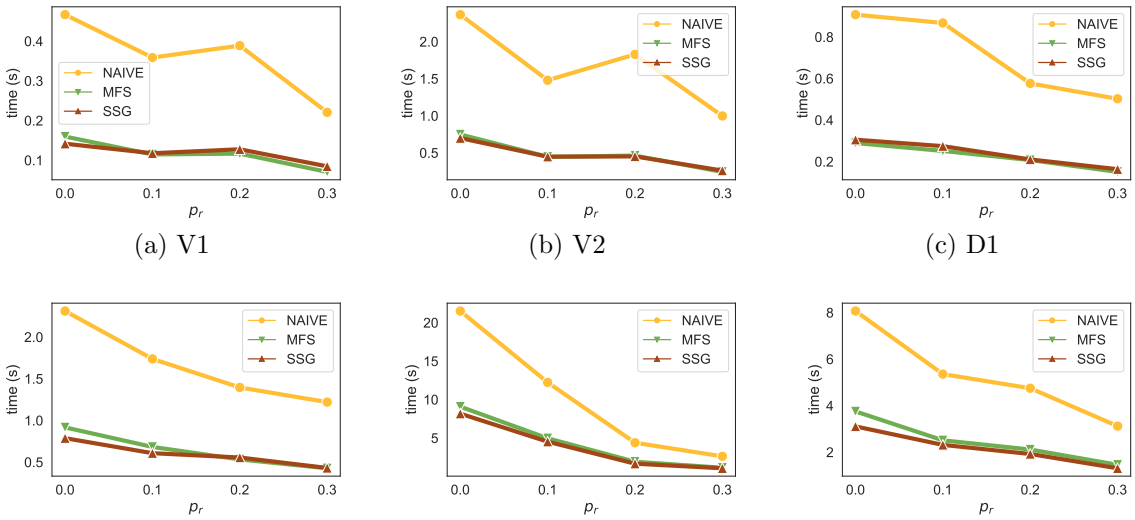
Figure 3.12: Varying Window Size w



(a) V1 (b) V2 (c) D1

(d) D2 (e) M1 (f) M2

Figure 3.13: Varying # Occlusions with p_o



(a) V1 (b) V2 (c) D1

(d) D2 (e) M1 (f) M2

Figure 3.14: Varying # Objects with p_r

frames is much more costly on video M1 (Figure 3.10e). This is because these frames include more objects per frame than the average objects per frame in the clip. As a result, these frames contribute significantly to the number of states created overall and subsequently to the overall cost.

Both MFS and SSG perform similarly on all videos generated by the VisualRoad benchmark. These videos have a smaller number of objects per frame, while the duration for each object in the visible screen (F/Obj in Table 3.3) is relatively longer than other videos. In such scenarios, the number of unique states is relatively small, while most states can be generated directly from principal states. Thus, SSG does not provide significantly better pruning power than MFS since its graph structure in such cases does not provide a clear advantage. In contrast, on other datasets (D1 to M2), the performance of SSG is better than MFS due to the pruning effectiveness attributed to the graph structure. For example, in Figure 3.10f SSG is around 18% faster than MFS and 12% faster in Figure 3.10d. In these datasets, the number of objects per frame is much larger (Obj/F in Table 3.3) or the duration of objects in the visible screen is lower, and thus more states are maintained.

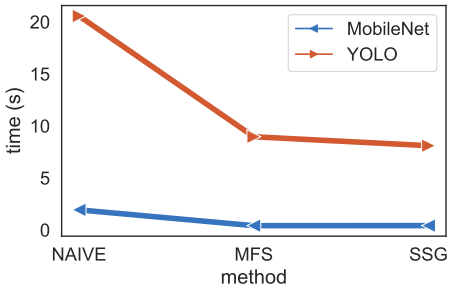
With a window size $w = 300$ frames, we vary the duration parameter d from 180 to 270 frames, and the results are shown in Figure 3.11. The performance of all methods is relatively stable since the duration parameter does not affect the number of states maintained. The

performance gap between MFS and SSG is very narrow on datasets V1 and V2, with a speedup of 3.1 and 3.3 times respectively over NAIVE. It is a different story on M2 however: MFS has only a 2.1 times speedup over NAIVE, but SSG can still achieve a speedup of more than 2.5 times thanks to its pruning power. This is consistent with the trends shown in previous figures.

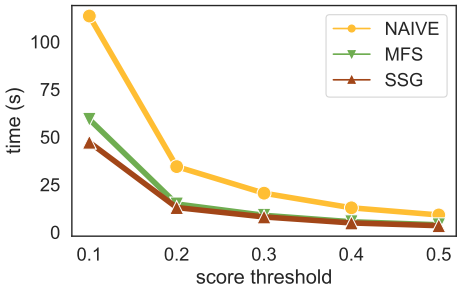
We next vary the window size w and fix the duration parameter $d = 240$ showing the results as Figure 3.12. All methods require more time to compute as the window size increases since more states have to be maintained and computed. The NAIVE and MFS methods are penalized more, since they both have to compute object set intersection between each existing state and the new arriving frame. As we increase the window size, the trends remain overall the same. It is interesting to observe how data set characteristics affect performance. For example, Dataset M1, M2 are captured by moving cameras; this means the duration of each object in the visible screen (Obj/F in Table 3.3) is smaller and new objects are introduced frequently to the visible screen, leading to more unique states. By increasing the window size, SSG benefits most due to its pruning power, which can be observed from Figure 3.12f, where SSG is almost 50% faster than MFS. Such trends can also be observed on other datasets (Figures 3.12c, 3.12d, 3.12e).

So far for both real and synthetic datasets we have utilized the intrinsic number of object occlusions present in them. To introduce more occlusions (and be able to vary them) we reuse the same object id after an object disappears from the video. Thus, to simulate more occlusions, we introduce an occlusion parameter, p_o. Each object id will be reused at most p_o times. Figure 3.13 depicts the results, where p_o varies from 0 to 3. In general, more occlusions increase the chances that the intersection of object sets between states is non-empty. This penalizes the overall performance as a larger number of states have to be maintained. Such trends can be observed for both synthetic and real data sets. Increasing p_o leads to higher speedups for both MFS and SSG on all datasets. For example, the speedup of MFS increases from 2.1 to 2.2 times compare to NAIVE, while the speedup of SSG increases from 2.6 to 2.7 times as p_o increases from 0 to 3 in Figure 3.13f.

Similarly, we introduce another parameter p_r to artificially control and vary the total number of unique objects in each video. For example, with $p_r = 0.1$, we randomly remove



(a) Varying Models (Log-scale)



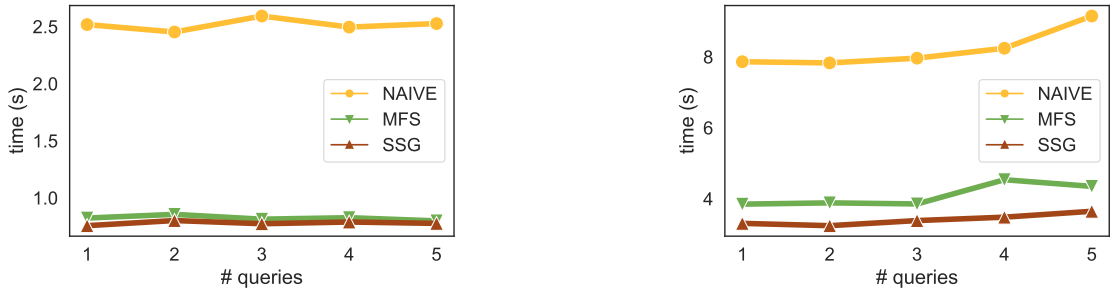
(b) Varying Score Thresholds With YOLO

Figure 3.15: More Evaluations on M1

(by excluding the associated object id from further processing) 10% objects from the original video to understand the performance of our methods. The result is shown in Figure 3.14. As we decrease the number of unique objects on each video, the performance of MFS and SSG becomes close. On videos where MFS and SSG perform similarly with $p_r = 0$, SSG does not perform much worse than MFS as p_r increases, implying that the edge maintenance cost of SSG is negligible.

We then vary the score threshold and the object detection model, shown as Figure 3.15. MobileNet has lower accuracy but a higher speed compared to YOLO. As a result, objects have a higher chance of being misclassified, thus leading to less detected objects after applying the object tracking algorithm. Similarly, increasing the score threshold for object detection algorithm also decreases the number of detected objects in total. Consequently, the overall performance of state maintenance improves. In all cases, both MFS and SSG still perform better than NAIVE. Other videos show a similar trend.

In summary, it is evident that both MFS and SSG offer large performance improvements compared to the NAIVE method. On videos with simple scenes (e.g. less objects, less occlusions) the performance of MFS is very similar to that of SSG. When the number of objects per frame is relatively larger and/or the duration of objects in the visible screen is relatively smaller, SSG has the best performance, up to more than 3 times faster than the NAIVE method and can be up to 50% faster than MFS. In general, SSG provides better pruning power than MFS while minimizing maintenance cost. On datasets with more objects per frame and/or data sets captured by moving cameras (yielding shorter duration of objects



(a) Synthetic (V1)

(b) Real data (M2)

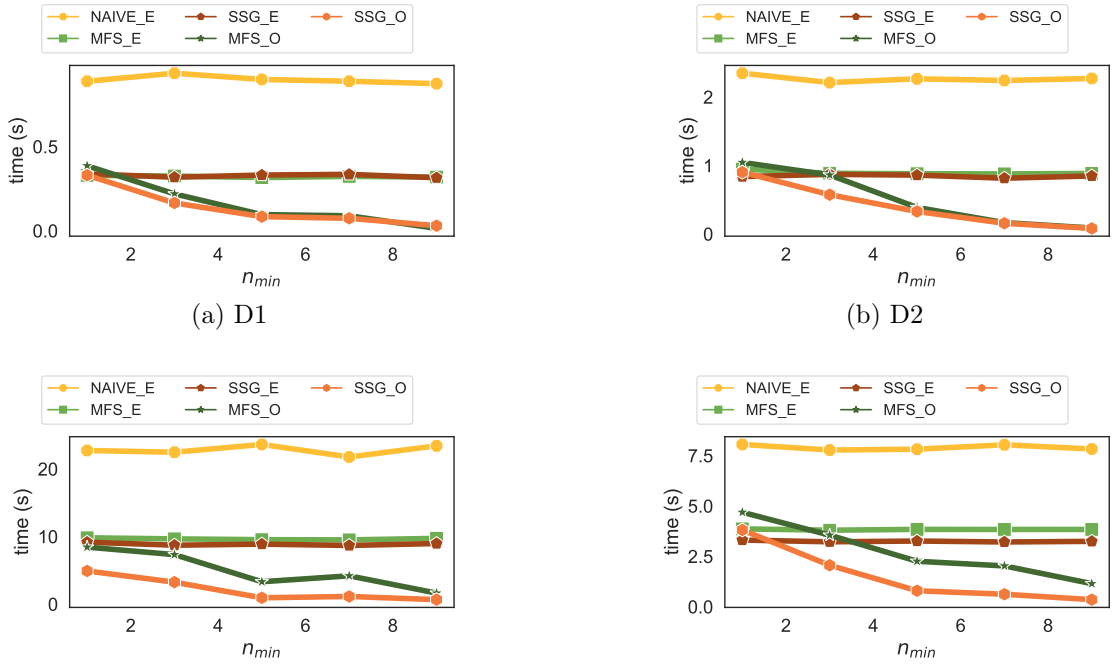
Figure 3.16: Varying # Queries

in the visible screen and increased number of new objects entering the visible screen, thus more unique states), SSG achieves higher speedups, as is observed from experimental results.

3.7.3 Query Evaluation

Based on State Maintenance methods, we benchmark our Query Evaluation module (Section 3.6.2). We vary the number of queries from 10 to 50, as shown in Figure 3.16. The y-axis depicts the overall time including both state maintenance and query evaluation. The performance of each method remains almost the same even when the number of queries increases. Both MFS and SSG are more than 3.1 times faster than NAIVE in Figure 3.16a, while MFS performs worse than SSG in Figure 3.16b, with an overall speedup of 2.1. Thus, it is evident that the overhead of query evaluation on MCOS is negligible compared to that of state maintenance.

To evaluate the pruning strategy introduced in Section 3.6.3, we implemented five different methods: MFS_O and SSG_O, which implement the pruning strategy based on MFS and SSG methods; NAIVE_E, MFS_E and SSG_E, which only follow the CNFEvalE method proposed in Section 3.6.2. We generate 100 queries containing $\geq$ conditions only. Let C be the set containing all conditions from the given queries; we use $n_{min} = \min_{\forall c \in C} n_c$ to denote the minimum value in all conditions, where n_c is the threshold value in condition c (conditions are of the form $x \geq n_c$, where x is the object class). We vary n_{min} from 1 to 9, and the evaluation result on real datasets is shown in Figure 3.17. As n_{min} increases, methods adopting SSG perform better than those adopting MFS in general. For example, in Figure



(a) D1 (b) D2

(c) M1 (d) M2

Figure 3.17: Varying n_{min} in $\geq$ Queries

3.17c, when $n_{min} = 3$, SSG_O is 6 times faster than NAIVE, while MFS_O is only 3 times faster. Similarly, in Figure 3.17d, when $n_{min} = 5$, SSG_O is also around 6 times faster than MFS_O compared to NAIVE, where the speedup is more than 9 times. When n_{min} is large enough, we observe a significant performance improvement on all methods that adopt the pruning strategy (MFS_O and SSG_O). Among them, the SSG_O has the best performance in general. When $n_{min} = 1$, methods adopting the pruning strategy are slightly more expensive on some datasets (Figure 3.17a, 3.17b) since more states in the graph are evaluated on the given queries. We observe that methods based on SSG are always the best compared to other approaches, while the pruning strategy offers significant improvements on both MFS and SSG methods. When the pruning strategy is enabled, the optimized approach *provides more than 100 times speedup on real datasets*, as is evident in Figure 3.17, when n_{min} is set to 9.

We have also evaluated our approach by utilizing numerous videos consisting of a large number of frames. For example, on a longer video clip consisting of more than 50k frames

	MFS	SSG
State Maintenance	2.60	3.19
With Query Evaluation	127	380

Figure 3.18: Speedup on a Long Video

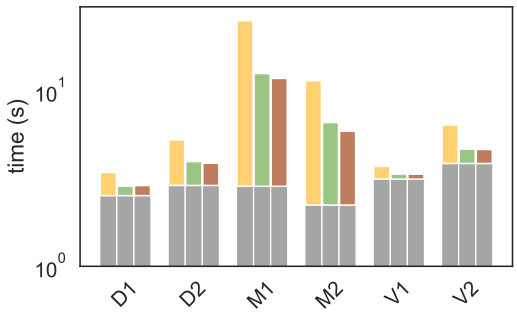


Figure 3.19: End-to-end Evaluation (Log-scale)

[5], the speedup is presented in Figure 3.18. Considering the overhead of state maintenance only, SSG provides a speedup of 3.2 times compared to the baseline. For the entire query evaluation, we generate 100 queries containing $\geq$ conditions only (same as methodology above), and the optimization offered by SSG provides the highest speedup of over 380 times, with $n_{min} = 15$. Note that the query optimization method based on SSG (SSG_O) performs much faster than that based on MFS (MFS_O) thanks to the pruning power of SSG. The speedup of both methods increases as we increase the value of n_{min}. These speedups are consistent with our observations on other large publicly available video feeds [6].

We finally present the performance of the three methods in an end-to-end manner, as shown in Figure 3.19. We measure the average processing time per query (over 50 queries) for each dataset (the lower the better), including the object detection and tracking time (denoted as D&T in the figure). As can be observed, both MFS and SSG have superior performance, and SSG performs the best overall.

To study the impact of upstream models on the result accuracy, we manually annotate videos and use that as the ground truth. For comparison, we evaluate the same queries on both models (YOLO and MobileNet) and compute the precision and recall according to the windows produced in the result. In general, the implementation with MobileNet is more than twice faster compared with that with YOLO. However, when both models have a similar recall, YOLO offers 3 times better precision than MobileNet. We stress that any

[5] `https://www.youtube.com/watch?v=wqctLW0Hb_0`

[6] `https://www.youtube.com/watch?v=e_WBuBqS9h8`
`https://www.youtube.com/watch?v=Inbd-YOdPUE`

model can be adapted to our framework, while more accurate models produce results with higher accuracy in general.

3.8 Summary

In this chapter, we have considered the problem of evaluating CNF temporal queries over video feeds. Utilizing DL approaches, it is possible to extract valuable context from frames and enable advanced query processing. We have introduced the Marked Frame Set and Sparse State Graph approaches to maintain context that aims to reduce the overhead of processing new frames and facilitate CNF query evaluation. We demonstrate via experiments that our approaches yield vast performance improvements during query execution over alternate approaches. In this chapter, we handle uncertainty by applying a cut-off in the object detection result score. Future work may explore using object detection score as an uncertainty measure and utilize that during State Maintenance.

4 Ranked Window Query Retrieval over Video Repositories

4.1 Introduction

4.1.1 Background and Motivation

In this chapter, we focus on structured query processing involving quantitative constraints over video repositories. For example, consider a query that aims to identify clips of a set duration (e.g., 10 seconds) in which at least the same say three cars (possibly each with query-specified labels such as colors, brand logos, and commercial advertisements) appear jointly. The query result consists of video clips from the repository, each of which contains at least one group of objects that satisfy the given query. Since there could be many such video clips in the result, they have to be ranked according to some scoring function that reflects how well a video clip matches the query, to reduce the number of video clips returned in the result.

For example, in news broadcasting, the editorial team is interested to identify historical footage showing a car with a Coca-Cola sign and a car with a Red-Bull sign appearing jointly for at least five seconds on a clip. Such queries can be especially useful for retrieving specified video clips from large video repositories that are commonplace in numerous settings such as broadcast TV but also surveillance and law enforcement as well as autonomous driving, where the retrieved clips can be used for content creation and DL model training.

To answer such queries, we assume that a detected object (e.g., a car) can be tracked across frames, establishing a unique identity (a key in a relational sense) utilizing well-studied object tracking algorithms [88], which are robust and capable of tracking the same object

even in the presence of occlusions from other objects [153, 152]. Utilizing such algorithms, we can pre-process large collections of video data and make them amenable to the interactive processing of such structured queries. As the query parameters are provided at run-time (e.g., window size, type, and the number of objects), such queries are challenging to answer with known indexing methodologies.

4.1.2 Our Proposal

In this chapter, we develop special indexing techniques to efficiently answer such queries over large video repositories. It is evident that such queries may return a large number of query results. As such, we develop algorithms to rank query results and efficiently return only a number of desired results with the highest ranking (scores) as specified by the query. Our framework supports any monotonic ranking scheme.

We propose a two-phased approach consisting of an *ingestion* (or pre-processing) phase during which videos are processed and suitably indexed, and a *query* phase during which ranked window queries are executed. Our main contributions are summarized as follows:

- We propose an overall methodology that partitions a video repository and accelerates query processing for given queries with a two-phase approach.
- We develop a *Partition-Based Index Construction (PBIC)* algorithm, which partitions a given video repository into *fixed-size clips* and builds indexes called *Score Ordered Prefix Forests (SOPF)*, which provides efficient object set (for objects identified in the query) computation.
- We design the *Partition-Based Query Processing (PBQP)* algorithm to evaluate queries efficiently with an early-stopping mechanism, prioritizing windows based on score estimation.
- We perform thorough experiments using real-world datasets, which demonstrate that for real videos, our method can achieve speedups up to more than three orders of magnitude compared to baseline methods.

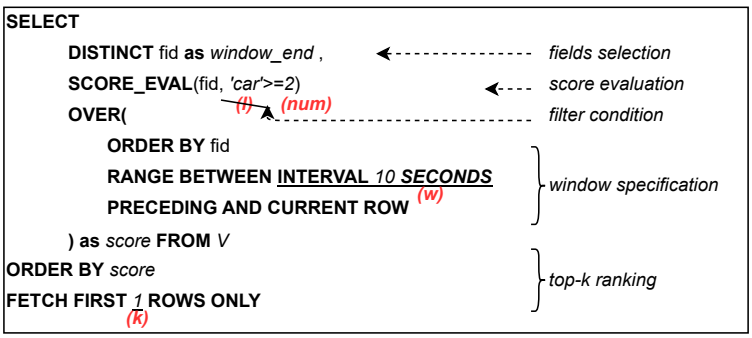


Figure 4.1: A Query Example

4.2 Problem Definition

We consider the video repository, R, as a set of videos. Each video in the repository, $V \in R$, is a bounded frame sequence, $V = \langle f_0, \ldots, f_i, \ldots, f_{N-1} \rangle$, where N is the number of frames in video V. With object detection and tracking algorithms, we can obtain a set of objects from each frame. We use O_{f_i} to denote the set of objects that appear in frame f_i, and use $\mathcal{O}$ to denote the set of all objects that appear in the video, V, i.e. $\mathcal{O} = \bigcup_{f_i \in V} O_{f_i}$.

Each object $o \in \mathcal{O}$ is associated with a set of attributes, which are derived utilizing object detection algorithms and other applicable DL models (texture, color detection, etc.). We use labels to denote such attributes and assume each object is associated with one or more labels, and these labels do not change throughout the whole video. Moreover, an object is assigned a *tracking identifier* when it is first detected and this identifier is persistent for the duration of the appearance of the object in consecutive frames (possibly accounting for object occlusions as well if supported by the object detection/tracking algorithms [152, 153]). Using such an identifier, an object can be tracked across different frames of the video.

Figure 4.1 illustrates the general query form. V is the video source, while *fid* is a frame identifier in the video. The SQL statement consists of three main clauses: (1) the select clause, which selects both the start frame of the window (the *window_end* field) and the score computed on each window (the *score* field); (2) the window clause, which specifies a window type (e.g., sliding, tumbling) and size, w (In this example, a sliding window of length 10 seconds is specified, which can be easily transformed to the number of frames according to the video's frame rate, e.g., 30 frames/second); (3) the rank clause, which sorts all windows

according to their scores in descending order, and selects a query-specified number (k) of results (k is 1 in this case) with the highest scores only.

In the select clause, **SCORE_EVAL** is a user-defined scoring function, which computes a score value for each window identified by a frame id and the size of the window. The function accepts two parameters: the starting frame of the current window, f, and a *filter condition*, which is written in Conjunctive Normal Form (e.g., *count({'car', 'red'}) $\geq 2\wedge$ count({'person'}) ≥ 1* in Figure 4.1, which specifies at least two red cars and one person appear jointly). In this chapter, we present our algorithms and techniques for the evaluation of conditions with greater than or equal to ($\geq$) operator to simplify the presentation; other operators (such as $=$ and $\leq$) can be supported similarly. Since we target long video sequences with our techniques, without loss of generality and to ease exposition we develop our framework for executing queries on a single long video sequence. Notice that multiple videos in a repository can be processed in the same way[7].

At query time, three parameters need to be specified, namely k, the number of results to produce; w, the size of each window; and *cond*, the filter condition in Conjunctive Normal Form (highlighted in Figure 4.1). We use $\mathcal{O}_W$ to denote the set of unique objects in the current window W (i.e. $\mathcal{O}_W = \{o|\exists f_i \in W, o \in O_{f_i}\}$). Note that an object can appear across multiple frames and it is assigned the same *tracking identifier*. Thus, the set of unique objects in the window is computed utilizing the tracking identifiers of the objects present in the window. The object set O is going to be considered for evaluating the score of the window W, only if the object set O satisfies the filter condition (i.e. $eval(O, cond)$ is *TRUE*, where *eval* is the evaluation function). In our example, the filter condition evaluates to *TRUE* for all object sets that contain at least two objects with both labels *red* and *car* and one object with label *person*. Only those windows with at least two red cars and one person appearing will be considered during ranking. The score of a window W is computed as the maximum score that could be produced by any object set that satisfies the given filter condition, specifically, the score of window W is subsequently computed as:

[7]E.g., by encoding all videos in a repository as a long sequence and annotating each frame with the video it belongs to, or by processing each video separately utilizing our techniques.

$$S(W) = \max_{\{O|O \subseteq \mathcal{O}_W \wedge eval(O,cond)\}} S(O, W) \tag{4.1}$$

In the equation above, $S(O, W)$ is the score of object set, O, in the current window, W, defined as:

$$S(O, W) = \sum_{f_i \in W} s(O, O_{f_i}), \tag{4.2}$$

where $s(O, O_{f_i})$ is a base scoring function that produces the score of object set O in a given frame f_i. Any monotonic function that satisfies the following condition can be used:

$$\forall O' \subset O, s(O', O_{f_i}) \geq s(O, O_{f_i}) \tag{4.3}$$

In general, the scoring function should reflect how important the object is in the window (e.g., the duration of the object in the current window). In this paper, we adopt a simple scoring function that computes the score of an object set, O, based on whether all objects in O are present in the given frame f_i, defined as:

$$s(O, O_{f_i}) = \begin{cases} 1 & O \subseteq O_{f_i} \\ 0 & O \not\subseteq O_{f_i} \end{cases} \tag{4.4}$$

A ranked window query is formally defined as follows.

Definition 5 (Ranked Window Query). *Given a video V, and a query with parameters k, w, cond, we use $\mathcal{W}$ to represent all the possible windows with size w generated on video V. The result of the query is a ranked list of k windows, Result $= (W_1, \ldots, W_i, \ldots, W_k)$, where $\forall W_i \in$ Result, $S(W_i) \geq S(W_{i+1})$ and $\forall W_j \in \mathcal{W} \setminus$ Result, $S(W_j) \leq S(W_k)$, with $S(W_i)$ defined as per Equation (4.1).*

In this paper, we focus on sliding window semantics where a new window is generated for each frame. Other window semantics can be supported trivially. In the highest-ranked results, ties are broken arbitrarily. For queries that are issued on multiple videos $R' \subseteq R$ in the video repository, we can treat multiple videos as a long video, $V' = \cup_{V \in R'} V$, and evaluate the query over the long video V' only.

4.3 Overview of the Approach

We propose a two-phased approach, consisting of an Ingestion Phase and a Query Phase, shown in Figure 4.2. The general idea is to materialize intermediate data representations (including indexes and associated metadata) during the ingestion phase and utilize such data representations subsequently for fast query answering.

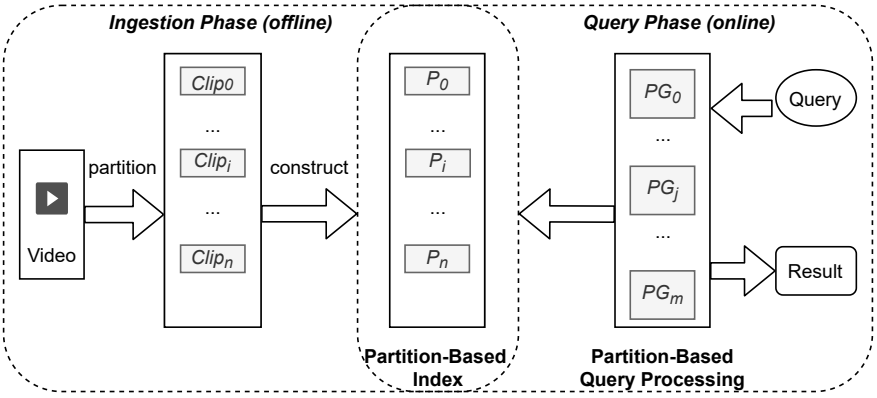


Figure 4.2: Overview of Top-k Window Query Processing

The ingestion phase takes place offline, during which videos are pre-processed into non-overlapping partitions (frame sequences or clips) of a given size. For each partition, we build a Partition-Based Index (PBI), which is then materialized along with the associated metadata. The main idea is to pre-compute the occurrence of all possible object sets in each partition and use them to accelerate query answering. Since the PBI index is a localized structure built for each partition, its incremental updates can be performed easily. For example, new videos can be added in two steps: (1) generating partitions based on the new video, and (2) building a PBI index for each generated partition.

Query processing takes place during the query phase, which is handled by the Partition-Based Query Processing (PBQP) algorithm. In a nutshell, the PBQP algorithm first retrieves relevant object sets from each PBI that satisfy the given query condition and then constructs candidate windows by merging partitions on the fly via a mechanism called Partition Groups (PG), which are then processed to produce the final result. Such processing is prioritized by the estimated scores of the object sets, and we introduce an early-stopping mechanism to terminate the algorithm once the top-k windows with the highest scores are guaranteed to

be found.

In the next section, we present our index structure (PBI) in detail. Based on such indexes, in Section 4.5 we present the PBQP algorithm to answer queries efficiently.

4.4 Partition-Based Index

The main objective of the ingestion phase is to pre-compute and materialize sufficient data for query answering. This is challenging since: (1) both the window size w and the filter conditions are only available at query time; such information cannot be utilized during ingestion phase. It is infeasible to enumerate all possible window sizes and filter conditions; (2) the index structure should support incremental updates to the video repository such as adding or removing videos, or restricted query scope of videos (i.e., querying on a subset of videos in the repository). We propose to split videos into smaller partitions, where each partition has a fixed size of p frames. The index structure is built for each partition, and the final results for a given query can be computed based on the materialized index across multiple partitions during query processing.

During ingestion, once a partition is created, all objects and their labels are obtained utilizing object detection/tracking algorithms. To capture the co-occurrence relationships of objects, we use object sets to store objects together and use a set of frames in the partition to denote the frames where all objects in the set appear jointly. We use $\mathcal{O}_P$ to denote the set of unique objects obtained from the frames in the partition P. The general idea is to pre-compute object sets $O \subseteq \mathcal{O}_P$ along with their scores in each partition, and materialize them for query processing. By materializing sufficient information for each object set, answers to given queries can be constructed based on object sets from one or more partitions.

A straightforward approach to compute all possible object sets and their scores is to intersect all object sets across frames. For example, Figure 4.3a presents a video clip with three frames, where each set corresponds to a frame, and each letter represents a unique object, recognized using the same identifier provided by the object tracking algorithm. By intersecting the first and second frames, an object set $\{C\}$ is produced, and its score can be computed based on Equation 4.2. All unique object sets with different scores are shown

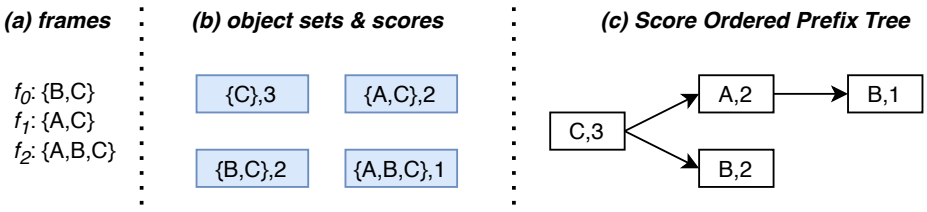


Figure 4.3: Score Ordered Prefix Tree (SOPT): an Example

in Figure 4.3b. However, this could be very expensive, especially when the partition size is large.

We propose to build a Score Ordered Prefix Forest (SOPF) in each partition. Specifically, we build tree indexes, where each node in the tree represents one object set appearing in the partition. To provide sufficient data for fast window score computation across multiple partitions, each node also includes an interval list (containing all frames in the partition), and a bitmap representation. We also build hash indexes based on labels to allow fast node retrieval based on given filter conditions. In Section 4.4.1, we discuss first the main structure of the index. We then present the PBIC (Partition-Based Index Construction) algorithm to construct the index in Section 4.4.2, while introducing the bitmap and hash index in Section 4.4.3.

4.4.1 Score Ordered Prefix Forest

The main objective of Score Ordered Prefix Forest is to capture all possible object set combinations along with their scores inside each partition. We propose the use of prefix trees (tries) [45] to reduce both computational and memory costs during index construction. Tries have been successful in encoding correlations across strings or lists with tree paths. Since we require nodes in the prefix trees to be sorted, namely object sets with higher scores should be placed closer to the top of the prefix trees, we construct a Score Ordered Prefix Tree (SOPT) as shown in Figure 4.3c (see Section 4.4.2 for the precise algorithm).

Definition 6 (Score Ordered Prefix Tree (SOPT)). *A Score Ordered Prefix Tree,* $\mathbb{T} = (\mathbb{E}, \mathbb{V})$*, is a prefix tree, where* $\mathbb{E}$ *is the set of edges, while* $\mathbb{V}$ *is the set of nodes. We use* $v_r \in \mathbb{V}$ *to denote the root of the tree. For any node* $v_i \in \mathbb{V}$*, we use* P_{v_r,v_i} *to denote the path from node* v_r *to* v_i*, and use* O_{v_r,v_i} *to denote the object set represented by* P_{v_r,v_i}*. We refer to*

the object set O_{v_r,v_i} as the object set of node v_i. Each v_i contains at least two fields: key_{v_i}, which represents the object in the current node, and $score_{v_i}$, which represents the score of the object set O_{v_r,v_i}. The following properties hold:

1. *For any edge $(v_i, v_j) \in \mathbb{E}$, we have $O_{v_r,v_j} = O_{v_r,v_i} \cup \{key_{v_j}\}$, and $score_{v_i} \geq score_{v_j}$.*
2. *For any two nodes $v_i, v_j \in \mathbb{V}$, $O_{v_r,v_i} \neq O_{v_r,v_j}$.*

In Definition 6, property (1) guarantees that all nodes in the tree are sorted according to scores, while property (2) aims to eliminate redundant nodes such that the total number of nodes can be minimized. In Figure 4.3 and the rest of the chapter, we use the object tracking identifier associated with the object of the node as the key for each node.

For a specific partition, the object with the maximum score should be selected for the root node. In many cases, it is infeasible to cover all object sets in a partition using one tree. For example, If we add another frame, $f_3 : \{D, E\}$, to the frames in Figure 4.3a, it is not possible to build a single tree that includes all objects. As such, we propose to build a Score Ordered Prefix Forest for each partition, defined as follows.

Definition 7 (Score Ordered Prefix Forest (SOPF)). *A Score Ordered Prefix Forest $\mathbb{G}$ is a set of independent trees, where each tree $\mathbb{T}_i \in \mathbb{G}$ is a Score Ordered Prefix Tree.*

4.4.2 The PBIC Algorithm

Let $\mathcal{O}_P$ be the object set constructed from the current partition, P. For any two object sets $O_i, O_j \subseteq \mathcal{O}_P$, where $O_j \subset O_i$, according to Equations(4.2) and (4.3) in Section 4.2, we have $S(O_i, P) \geq S(O_j, P)$, where the partition P is treated as a window of size p. As a result, if we add new objects to an object set O_i, the new object set O_j will have the same or lower score. Based on this observation, we can always start with one object which has the maximum score in the partition as the root of a SOPT and build the tree recursively by adding another object that has the maximum score among all the remaining objects. We propose Algorithm 4 to construct SOPF.

We first select the root. We use *objSet* to store the object set of the current node being built, which is initialized as empty (Line 1). *fList* is the current frame list, which is

Algorithm 4: PBIC

```
1  objSet ← ∅;
2  fList ← the raw frame list in this partition;
3  countMap ← compute the map containing all (object, frame count) entries based on
     fList;
4  allCountMaps ← {countMap} // a set of maps ;
5  rootList ← [];
6  while countMap.size() > 0 do
7  |  e ← remove the entry with the max count in countMap;
8  |  objSet ← objSet ∪ {e.obj};
9  |  score ← compute the score for objSet;
10 |  node ← (e.obj, score);
11 |  fList′ ← filter fList according to e.obj;
12 |  buildNext(node, allCountMaps, fList′,countMap, objSet);
13 |  objSet ← objSet \ {e.obj};
14 |  rootList.add(node);
15 end
```

initialized as the list of all frames in this partition (Line 2). For each unique object appearing in the partition, we compute the number of times it appears in $fList$. We use $countMap$ to store the association between the objects and their numbers of occurrences in $fList$ (Line 3). The main idea of the algorithm is to iteratively remove the most frequent object from $countMap$ and use it to initialize the root (Lines 7-10). Then we compute the list of all frames in which the object at the root appears ($fList'$, Line 11) and call function $buildNext$ (detailed in Algorithm 5) to build the remainder of the tree utilizing $fList'$ (Line 12).

Function $buildNext$ works in a similar fashion (shown in Algorithm 5). It utilizes $fList'$ and recursively identifies the most frequent object associated with the node being built when $buildNext$ is invoked (Lines 4-9), while refining $fList'$ (Line 10). It utilizes the frequent objects at each iteration to build children of the root, operating recursively (Line 12).

With these algorithms, we can prove the following theorem.

THEOREM 2. *For each tree in the SOPF constructed by Algorithms 4 and 5, both properties of SOPT hold.*

PROOF 2. *Property (1) follows immediately from Equations (4.2) and (4.3). We now prove that property (2) holds by contradiction. Assume that the procedure violates property (2), for some two nodes* $v_i, v_j \in \mathbb{T}$*, we have* $O_{v_r,v_i} = O_{v_r,v_j}$*. Then both nodes are in the same level of the tree, and*

Algorithm 5: buildNext($pNode$, $allCountMaps$, $fList$, $countMap$, $objSet$)

Input: parent node, $pNode$; a list containing all count maps from previous levels, $allCountMaps$; the object-count map from the previous level, $countMap$; the filtered frame list, $fList$; current object set, $objSet$;

```
1  newCountMap ← all objects in countMap and their frame count based on fList;
2  allCountMaps ← allCountMaps ∪ {newCountMap};
3  while newCountMap.size() > 0 do
4  |  e ← remove the entry with the max count in newCountMap;
5  |  for map in allCountMaps do
6  |  |  if map.get(e.obj) = e.count then map.remove(e.obj) ;
7  |  end
8  |  objSet ← objSet ∪ {e.obj};
9  |  score ← compute score for objSet;
10 |  node ← (e.obj, score);
11 |  fList' ← filter fList according to e.obj;
12 |  pNode.addNext(node);
13 |  buildNext(node, allCountMaps, fList', newCountMap,objSet);
14 |  objSet ← objSet \ {e.obj};
15 end
16 allCountMaps ← allCountMaps \ {newCountMap};
```

$score_{v_i} = score_{v_j}$. We use X and Y to denote two objects that are generated in different order in O_{v_r,v_i} and O_{v_r,v_j} respectively. We use $node_{v_{xi}}$ and $node_{v_{yi}}$ to denote two nodes on path, P_{v_r,v_i}, where $key_{v_{xi}} = X$, $key_{v_{yi}} = Y$. Similarly, we use $node_{v_{xj}}$ and $node_{v_{yj}}$ to denote two nodes on path, P_{v_r,v_j}, where $key_{v_{xj}} = X$ and $key_{v_{yj}} = Y$. Then we have $node_{x_{xi}} \prec node_{v_{yi}}$, and $node_{v_{yj}} \prec node_{v_{xj}}$.

Assume that $node_{v_{xi}}$ is generated first, then $score_{v_{xi}} \geq score_{v_{yi}}$. Once $node_{v_{xi}}$ is generated, according to the algorithm, object X will be removed from count maps; thus it's impossible to generate $node_{v_{xj}}$ which is a contradiction.

We demonstrate the procedure via the following example.

EXAMPLE 4. *Let us build the SOPF for a partition containing three frames: {A,B,C}, {B,C}, and {A,C}. Figure 4.4 depicts this process. We highlight selected objects and newly created nodes in shade. We first compute the object-count map for level 1, shown in Figure a. The object with the max count is C; thus we choose C as the key for the root node and remove (C,3) from the level 1 count map.*

Based on (C,3), we compute the level 2 count map, shown in Figure b. Since all frames contain object C, the level 2 count map is the same as level 1. Both objects A and B have

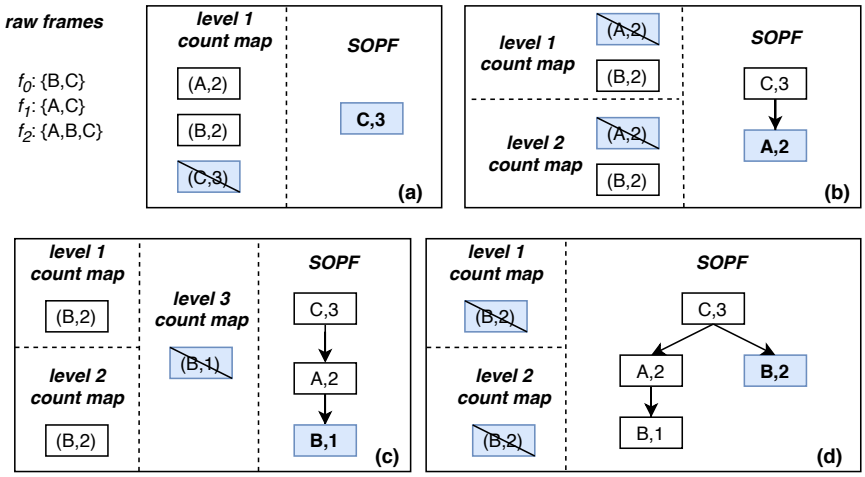


Figure 4.4: An Example of Building SOPF

the same count value; so we use lexicographic order to break the tie, and object A with count 2 is chosen as the next node. The corresponding object-count pair is then removed from both level 1 and level 2 count maps.

Similarly, we select the node (B,1) in Figure c. Since the level 3 count map is empty, we move back to the previous node and select (B,2) as the next node after (C,3) in Figure (d). The algorithm stops when all count maps are empty.

In the above example, one SOPT is sufficient to represent all object sets in the partition. In other cases where one SOPT cannot cover all objects, the level 1 count map will not be empty; thus such building procedures can be applied recursively until all remaining object-count pairs are processed.

4.4.3 Bitmap Encoding and Hash Index

Although SOPF stores all object sets and scores in a compact manner, random access to any object set is still inefficient; we have to start the search process from the root node every time. Instead of using the SOPF directly, we augment the data representation with bitmap and hash indexes in SOPF to support random access.

We utilize bitmaps to encode the object set corresponding to each node. We first identify all unique objects (based on the tracking identifiers) with the label l (where l is the label of the object the current index of the partition is built for) within the partition and impose a

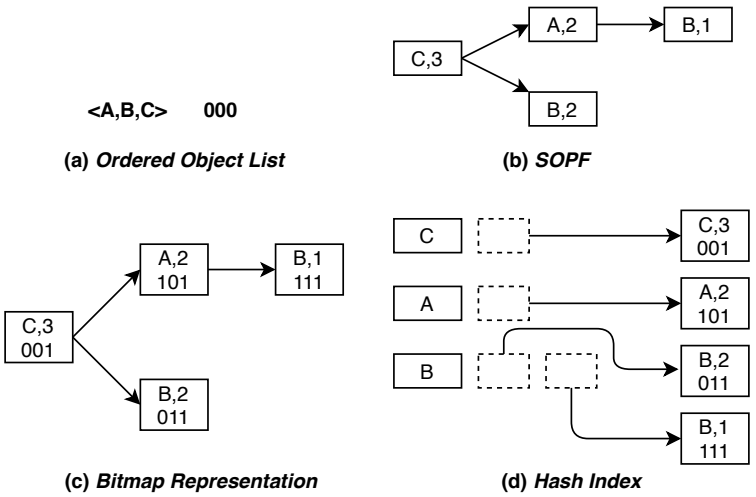


Figure 4.5: An Example Using Hash Index and Bitmap

total ordering by sorting them based on their tracking identifier. The bitmap for each node is initialized by representing the object at the node with a bit in the total object ordering (Figure 4.5a). Starting from the root node, we set the bit associated with the object (tracking identifier) in that node. All the bitmaps of the remaining nodes are computed by bit-wise "or" between the bitmap of the previous node (the parent in the tree) and the bitmap generated by the object (tracking identifier) in the current node (e.g., Figure 4.5c, which is generated from Figure 4.5b).

We adopt a hash index to facilitate efficient node lookups given an object. We use an individual object tracking identifier for each node in SOPF as the hash key. Based on this idea, we build the hash index between each object and all nodes containing the object, shown as Figure 4.5d. Both the bitmap representation and the hash index can be built based on a single pass of the SOPF. The SOPF is discarded once the bitmap and hash index is constructed.

After generating all bitmaps and the hash index, all nodes are sorted according to the bitmap cardinality (i.e., number of bits set to 1) in each node in descending order, breaking ties using the score. For each partition P_i, we materialize the nodes in sorted order and we refer to the materialized node list of the partition as its *node list*, $\mathbb{V}_{P_i}$. We also materialize the hash index for each partition, H_{P_i}. To provide sufficient information while enumerating windows during query processing (see Section 4.5.3), we store for each node an *interval list*,

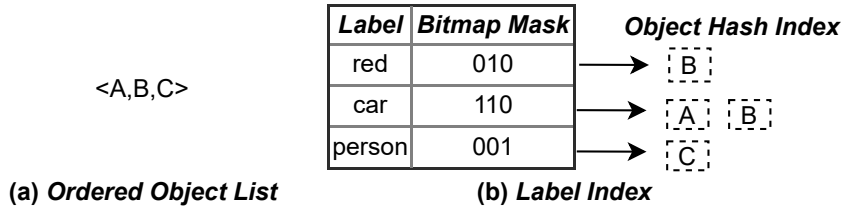


Figure 4.6: Metadata For the Example in Figure 4.5

which contains the identifiers of all the frames in which the object set appears when building the index.

We also materialize additional information (Figure 4.6), for each partition P_i, including (a) the sorted (by tracking identifier) object list, $L^O_{P_i}$, which will be used to reconstruct the object set from bitmaps (see Section 4.5.3); and (b) the hash index built on labels, which will be used to early-prune unrelated partitions (see Section 4.5.2). In the label index, we store a bitmap mask for each label (where each bit is set to 1 if the corresponding object has that label , and the object list, with the hash index built in Figure 4.5d.

4.4.4 Complexity Analysis

To understand the time required to construct the SOPF we provide an analysis of the number of nodes that can be generated in the forest. Let the maximum number of objects per frame be m, and the number of unique frames (frames with unique object sets) be p'. Assume that we already have a SOPF for $p' - 1$ frames, which consists of $a_{p'-1}$ nodes. Two cases arise when the p'th frame is added to the SOPF (assume that we obtain the new SOPF based on the existing SOPF consisting of $p' - 1$ frames):

1. The p'th frame shares no object with the existing SOPF. In this case, the new SOPF is simply the old one plus the new nodes, one for each object in the p'th frame; we have $a_{p'} = a_{p'-1} + m$.

2. The p'th frame shares some objects with the existing SOPF. In this case, two types of nodes are generated: (a) Nodes representing the current frame, which generates at most m new nodes; (b) For each object that appears in both the p'th frame and any of the previous $p' - 1$ frames, at most one new node will be created. This is because

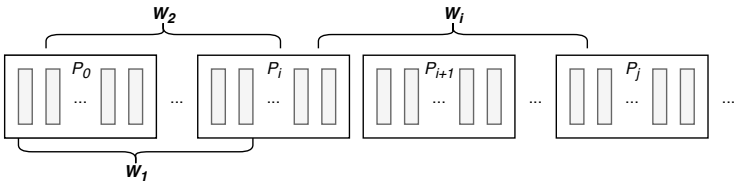


Figure 4.7: Different Windows with Fixed Size w

we generate nodes based on the object set scores, and with a new frame added, in the worst case we cannot update any node on the previous forest directly (i.e., the object sets of nodes from the previous forest are not subsets of the object set from the p'th frame); as a result, a new node with a new score will be created. In this case, m new nodes will be generated. Thus, we have $a_{p'} = a_{p'-1} + 2m$.

To summarize, the number of nodes in the SOPF that is built from p' frames is $2mp'$ in the worst case. In practice, we expect the number of unique frames to be much less than the number of frames in each partition, i.e. $p' \ll p$. In practice, we expect the number of unique frames to be much less than the number of frames in each partition, i.e. $p' \ll p$.

4.5 Query Processing

4.5.1 Overview

A brute-force approach is to first compute and then rank the score of each window that satisfies the filter condition. For a given query defined over a window size w, the score of each window can be computed based on the PBI indexes of the partitions the query encloses. We use λ to denote the number of partitions involved in each window; then the lower bound λ_l and upper bound λ_u of λ can be computed as follows:

$$\lambda_l = \lceil \frac{w}{p} \rceil, \lambda_u = \lceil \frac{w-1}{p} \rceil + 1 \tag{4.5}$$

where p is the size of each partition. Thus, the score computation for an arbitrary window requires at most λ_u partitions. For example, in Figure 4.7, W_1,W_2 are two windows with $\lambda = \lambda_l$, while W_i depicts the case where $\lambda = \lambda_u$.

For long videos, computing scores for each window can be extremely costly since the

same partitions could be involved in multiple windows. We, therefore, seek to develop an algorithm capable of computing the result without actually computing the scores of all windows involved. We introduce the notion of *Partition Groups* based on equal-sized grouping of the partitions to facilitate query processing. Each Partition Group contains pg number of partitions, where $pg \geq \lambda_u$, to guarantee that all windows of interest to a query are covered by at least one Partition Group. Since a partition is the smallest unit of indexing for PBI (i.e., no further details are available beyond the partition-level information) and windows sharing the same Partition Group (i.e., enclosing the same partitions) share the same estimated score, we provide score estimation for Partition Groups instead of individual windows. We further divide the query processing into two main steps: (1) we produce an estimated score for each Partition Group, and prioritize Partition Groups utilizing the estimated score; (2) for a given Partition Group, we compute all the window scores enclosed in it and introduce an early-stopping mechanism by prioritizing nodes with the highest score retrieved from the PBI indexes.

In the next section, we first discuss how to estimate the score for each Partition Group, and then introduce the window score computation in each Partition Group, along with an example in Section 4.5.3.

4.5.2 The PBQP Algorithm

We present the Partition-Based Query Processing (PBQP) algorithm to estimate scores for all Partition Groups and iteratively process the one with the next highest score. We discuss score estimation in Section 4.5.2.1, and present the algorithm in Section 4.5.2.2.

4.5.2.1 Score Estimation

We use G_i to denote a Partition Group starting from partition P_i, (i.e., $G_i = \{P_i, \ldots, P_{i+pg-1}\}$), and use C_{P_i} to denote the underlying video clip (sequence of frame identifiers) that constitutes partition P_i. Since each window only involves at most λ_u partitions, we use C_i^P to denote the set of *adjacent* λ_u partitions starting from P_i (i.e., $C_i^P = \{P_i, \ldots, P_{i+\lambda_u-1}\}$). Roughly speaking, a Partition Group consists of a larger or equal number of partitions than

a window. That is, in each G_i, there are at most $pg - \lambda_u + 1$ different subsequences (adjacent partitions) with size λ_u. When $pg = \lambda_u$, we have $C_i^P = G_i$.

We use $\mathcal{W}_{C_i^P}$ to denote the set of windows that enclose the partitions C_i^P (i.e., $\forall W \in \mathcal{W}_{C_i^P}$, all frame ids in W are in $\cup_{P \in C_i^P} P$). It is easy to show that the maximum score of any window in $\mathcal{W}_{C_i^P}$ is upper-bounded by the sum of the maximum score we could derive from each partition this window encloses, i.e.,

$$\max_{W \in \mathcal{W}_{C_i^P}} S(W) \leq \sum_{0 \leq j < \lambda_u} maxScore(P_{i+j}, cond) \tag{4.6}$$

where *maxScore(P, cond)* represents the maximum node score that could be produced from partition P. It is computed as the highest score among all nodes in $\mathbb{V}_P$ with object sets satisfying *cond* in partition P, formally (where O_v and $score_v$ denote the object set and the score of node v respectively, as per Definition 6):

$$maxScore(P, num) = \max_{\{v | v \in \mathbb{V}_P \wedge eval(O_v, cond)\}} score_v \tag{4.7}$$

Based on Equation (4.6), we choose to use the sum of all maximum scores produced by Equation (4.7) from each partition as a proxy to estimate the score upper bound of windows in $\mathcal{W}_{C_i^P}$. That is, $\tilde{S}(C_i^P)$, the estimated score upper bound for windows $\mathcal{W}_{C_i^P}$, can be computed by

$$\tilde{S}(C_i^P) = \sum_{0 \leq j < \lambda_u} maxScore(P_{i+j}, cond) \tag{4.8}$$

The score upper bound $\tilde{S}(G_i)$ of all windows enclosed in Partition Group G_i (which we call the estimated score of G_i for simplicity) can be estimated as follows:

$$\tilde{S}(G_i) = \max_{0 \leq j < pg - \lambda_u + 1} \tilde{S}(C_{i+j}^P) \tag{4.9}$$

4.5.2.2 The Algorithm

The PBQP algorithm is shown in Algorithm 6. For the given list of partitions, the first step is to tag those partitions that are guaranteed not to be included in the final query result as

inactive, so that they will be simply ignored when we process each Partition Group, leading to cost savings. This can be achieved utilizing the label index (Figure 4.6 (b)) in each partition. Specifically, we retrieve the bitmap masks for labels in the filter condition and use them to evaluate the condition result (by counting whether the number of bits set satisfies the given value in the filter condition). As such, partitions for which the label masks do not satisfy the filter conditions are tagged *inactive*; all other partitions are tagged *active* (Line 1). Partition Groups are then created on all partitions in *partitions* using a preset pg value (Line 2) in a sliding-window fashion (e.g., every partition serves as the starting point of one Partition Group when $pg = \lambda_u$).

Algorithm 6: PBQP

Input: list of partitions *partitions*; partition size p; window size w; number of results k; filter condition *cond*; number of partitions in each Partition Groups pg;

Output: the list of windows with the highest score;

1 tag *partitions* (*active/inactive*) using metadata and *num*;
2 $\mathcal{G} \leftarrow$ create Partition Groups based on *partitions* and pg;
3 $res \leftarrow []$ // *the list of windows with the highest scores*;
4 estimate the score $\tilde{S}(G)$ for each $G \in \mathcal{G}$;
5 *priorityQG* $\leftarrow$ build a priority queue for $\mathcal{G}$;
6 **while** *maxScore(priorityQG)* $\leq$ *res.min* $\wedge$ *maxScore(priorityQG)* $>$ *0* $\wedge$ *res.min* $< w$ **do**
7 $G \leftarrow$ dequeue the Partition Group with the highest score from *priorityQG*;
8 *pauseScore* $\leftarrow$ the current highest score in *priorityQG*;
9 processPG(PG, *pauseScore*, res) // $\tilde{S}(G)$ *and* res *will be updated*;
10 enqueue G to *priorityQG*;
11 **end**
12 **return** res;

Next, we build a priority queue of Partition Groups with scores estimated as per Equation (4.9) for subsequent iterative processing (Lines 4-5). The score estimation of each partition is initialized as follows: (1) we retrieve nodes indexed by at least one label in the filter condition, and denote them as candidate nodes; (2) sort the candidate nodes according to their score and use the max score as the estimated score.

Utilizing the priority queue, we iteratively process the Partition Groups. At each step, we dequeue the Partition Group with the highest estimated score at the head of the priority

queue for processing. This process terminates when the upper bound on the score of the remaining Partition Groups in the queue are less than or equal to the minimum score in the current result (Line 6). Result candidates are computed and updated when processing each Partition Group, which is conducted by *processPG* (detailed in Section 4.5.3).

The estimated score for G is updated after it is processed (Line 9). Whenever there is such an update, we will check if this new score is still greater than or equal to that for the Partition Group currently at the head of the queue. If this is not true (i.e, $\tilde{S}(G) < pauseScore$, Lines 8-9), we pause the processing of G, store any intermediate results, and enqueue G back to the priority queue and continue with the current head of the queue. The processing of G may resume from where it left off if it becomes the head of the queue again. Thus, we always work on the most promising Partition Group, shifting to another whenever necessary.

4.5.3 The *processPG* Algorithm

4.5.3.1 The Main Idea

The *processPG* algorithm computes the actual score of each window enclosed in the given Partition Group and updates the result set along the way. The general idea is to always work on the node with the highest score in the partition.

When processing node v from partition P, we compute the score of its object set along with that for all its possible subsets, which requires both node retrieval and object set intersections between v and nodes from other partitions. For each object $o \in O_v$, we use o as the key to retrieve nodes containing this object from other partitions in G utilizing their hash indexes $H_{P'}$, where $P' \in G, P' \neq P$. For each retrieved node, v', by computing the intersection between O_v and $O_{v'}$, we can obtain all possible object sets along with their associated interval lists (maintained in each node), based on which the window scores can be computed. Leveraging the bitmap in each node, object set intersections can be computed efficiently. We also introduce an early-stopping mechanism to terminate the algorithm early once the rank results are finalized.

4.5.3.2 The Algorithm

We now present the complete algorithm for computing all the window scores in a given Partition Group, shown in Algorithm 7. To always work on the most promising Partition Group, the *processPG* algorithm could stop on a Partition Group once its estimated score is lower than that of some other Partition Group. We introduce a parameter *pauseScore* to stop processing once the estimated score of the current Partition Group is lower than the *pauseScore*. The algorithm accepts three parameters: the current Partition Group, referred to as G; the threshold score to pause the processing of this Partition Group, *pauseScore*; and the list of windows with highest scores, *res*.

We use a priority queue to drive the processing of partitions in each Partition Group (Line 1). During processing, we use *es* to record the estimated score (based on Equation (4.9)) for the current Partition Group, and update its value if necessary after a node is processed. When *es* becomes less than the threshold *pauseScore* or less than the smallest score in the current result *res*, this algorithm *processPG* stops (Line 2). Otherwise, we choose the partition with the highest score to process (Line 3) and start with the next unprocessed node with the highest score in that partition, denoted as *node* (Line 4). We retrieve nodes utilizing the label index (Figure 4.6b) to reduce the number of nodes processed and visit them according to their scores in descending order to enable early stopping.

We first test whether the bitmap produced by the retrieved node satisfies the given filter condition (by computing bitwise 'and' with corresponding label masks within its partition), and skip the node if the condition is not met (Line 5). Nodes from other partitions are retrieved utilizing hash indexes (Lines 7-10). Bitmap intersections are carried out between different nodes to identify the intersection of their object sets (Line 16). Since the bitmap for a node encodes only the objects at that node, we transform such local bitmaps to a global bitmap with each bit representing a distinct object in the Partition Group, before computing the intersections (Lines 11 and 15).

We use *GOSI* to denote the generated object sets from the above steps plus the interval list produced by each node (Line 18) (the list of frames in which the object set appears). For each unique object set O in *GOSI*, the complete list containing all frames in which O

Algorithm 7: processPG(PG, pauseScore, res)

Input: the Partition Group to be processed, G; the score threshold, *pauseScore*; the current list of windows with highest scores, *res*; the processed set of bitmaps, *pBitMaps*;

```
1  PQ ← the priority queue containing all partitions in G;
2  while es ≥ pauseScore ∧ es ≥ res.min do
3      partition ← dequeue the partition from PQ;
4      node ← the next node with the max score in partition;
5      if eval(node.bitmap, cond) then
6          objSet ← retrieve the object set of node;
7          retrievedNodes ← ∅;
8          for obj : objSet do
9              retrieve nodes indexed by obj from other partitions;
10             add to retrievedNodes;
11         end
12         bitmap ← compute the global bitmap from node;
13         GOSI ← []; // Generated Object Sets and Intervals
14         GOSI.add(bitmap, node.intervals);
15         for rNode : retrievedNodes do
16             bitmap′ ← compute the global bitmap from rNode;
17             bitmapR ← bitmap′&bitmap;
18             if bitmapR ∉ pBitMaps ∧eval(bitmapR, cond) then
19                 GOSI.add(bitmapR, rNode.intervals);
20         end
21         cResults ← compute the complete interval list for each unique object set in
             GOSI;
22         update res according to cResults;
23     end
24     update the score of partition to the next max score among unprocessed nodes;
25     update processedBitmaps, es; enqueue partition to PQ;
26 end
```

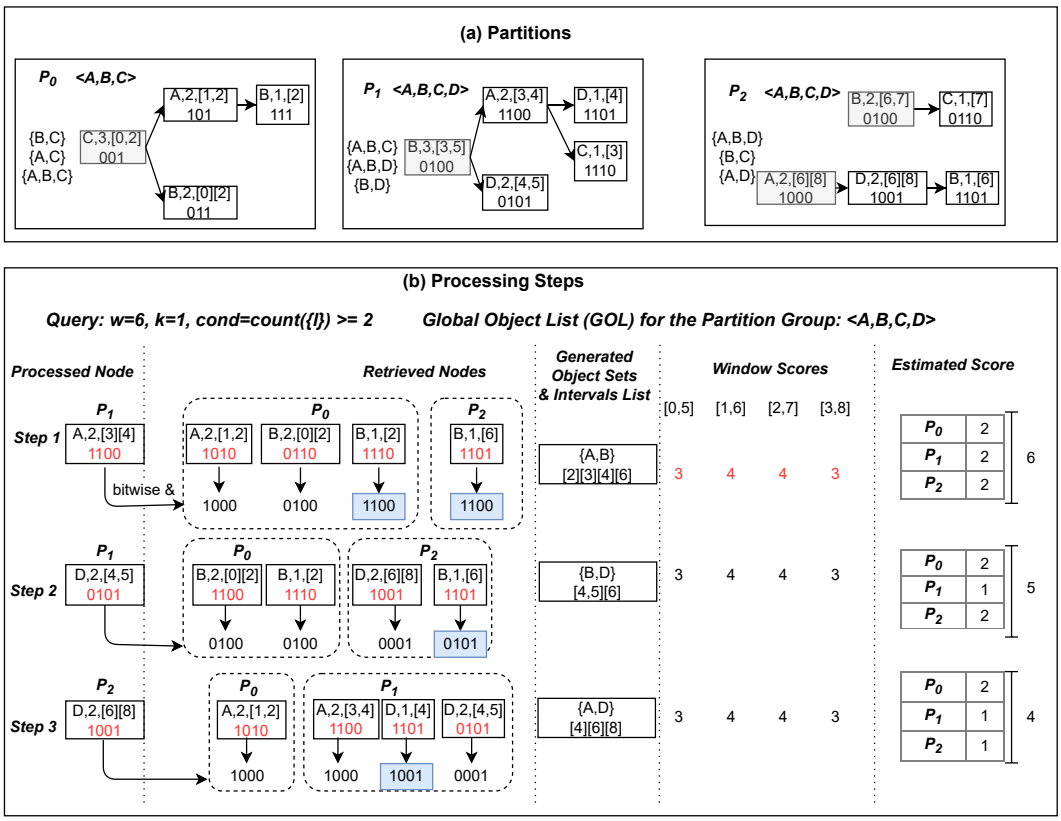


Figure 4.8: An Example of Window Score Computation in a Partition Group

appears in the current Partition Group can be obtained by scanning the *GOSI* list (Line 19). Window scores are computed trivially and *res* is updated (Line 20). Both the score of the current partition (Line 21) and the estimated score of this Partition Group are updated (Line 22).

4.5.3.3 An Example

We illustrate how *processPG* works using an example. To make our discussion more focused, we assume that we finish processing the Partition Group in one shot without pausing caused by conditions in Line 2 in Algorithm 7.

EXAMPLE 5. *Figure 4.8a shows a Partition Group consisting of 3 partitions. To provide better readability, we show the SOPF structure for each partition and omit the hash index. Each node contains four fields: the object (key), the score, the bitmap, and an interval list of frame identifiers. The frames are depicted on the left side of each partition. For simplicity,*

we assume the index is built on a single label l, with the filter condition queries the same label: count($\{l\}$) $\geq num$, where l is the label and $num = 2$, and we set $k = 1$ and $w = 6$. We maintain for this Partition Group a Global Object List (GOL) with each object corresponding to a bit in the global bitmap. In the example, the GOL is $\langle A, B, C, D \rangle$.

As the filter condition $num = 2$, all nodes with bitmap cardinality < 2 are skipped since they do not meet the filter condition (shaded nodes in Figure 4.8a). We select the max score over all unprocessed nodes in each partition as the estimated score for that partition. Initially, all partitions (P_0, P_1, and P_2) happen to have the same score, 2. Ties are broken arbitrarily; assume that we start with P_1 and choose to process node (A,2) (as (D,2) has the same score). The following actions are performed (Figure 4.8b, Step 1):

1. *Retrieve the object set represented by node (A,2): {A,B}. Use each object as the key to retrieve all nodes containing this object using the hash index from other partitions in this Partition Group. The retrieved nodes are shown in the "Retrieved Nodes" column.*

2. *Transform local bitmaps to the global bitmap for each node (highlighted in each node in the figure).*

3. *Compute the bit-wise "and" between the bitmap of node (A,2) (from P_1) and that for each of the other retrieved nodes.*

4. *Based on bit-wise computation results with cardinality (i.e., the number of bits with value 1) $\geq num$, generate object sets and their interval lists correspondingly (shown in the "Generated Object Sets & Interval List" column).*

5. *Based on the generated object sets, compute scores for all windows with window size $w = 6$ according to Equation 4.1 (the "Window Scores" column).*

6. *Update the estimated score for partition P_1 if necessary. In this case, the estimated score remains 2 (provided by node (D,2), shown in the "Estimated Score" column).*

At the end of Step 1, the estimated score for this Partition Group is 6, according to the estimation method introduced in Section 4.5.2. This is greater than the current highest score

of all windows (the "Window Scores" column), which is 4. We continue with the next node (D,2).

After Step 2 in Figure 4.8b, the max score of unprocessed nodes in P_1 is 1, which reduces the estimated score of this Partition Group to 5, as per Equation (4.9). Since the current highest score is 4, more nodes have to be further processed.

Without loss of generality, we choose node (D,2) from P_2 next. After processing, the max unprocessed node score for P_2 reduces to 1, which leads to the overall score estimation of 4. This means that the score of any unprocessed object sets will not exceed 4. Since we already have windows with a score of 4, the query processing for this Partition Group stops and produces the window with a score of 4 as the final query result.

Our discussion above assumes that we can finish processing G in one shot. When there are multiple partitions, this may not always be the case. We may have to pause its processing when its estimated score falls below that of the current head of the priority queue (*pauseScore*) and switch processing to that (more promising) Partition Group instead (see Algorithm 6). Filter conditions with multiple labels (e.g., at least two red cars) can also be processed similarly, with two minor modifications: (1) only a subset of nodes are retrieved utilizing the label index in each partition. (2) all conjuncts in the filter condition have to be satisfied at evaluation.

4.5.4 Complexity Analysis

We now analyze the query processing cost. Table 4.1 summarizes the notations used in this section.

4.5.4.1 Overall Query Cost

The overall cost $cost_{total} = cost_{spg} + cost_{sp} + cost_{pn} + cost_{and}$, where $cost_{spg}$ is the cost of sorting all possible Partition Groups, $cost_{sp}$ is the cost of sorting partitions inside Partition Groups, $cost_{pn}$ is the cost of sorting candidate nodes inside each partition, while $cost_{and}$ is the cost of computing bitmap "and" between nodes.

Assume the total length of the video is N, and the partition size is p. In the worst

Table 4.1: Table of Notations

N	the total length of a video
w	window size
p	partition size
pg	the number of partitions per Partition Group
N_{pg}	the total number of Partition Groups
N_{pn}	the maximum number of candidate nodes in a partition
$sc(O)$	the score of object set O
$ts(O)$	the time span of object set O
λ	the number of partitions involved for any window
λ'	the number of partitions involved for any object set

case, the total number of Partition Groups is $N_{pg} = \frac{N}{pg/2} - 1 = \frac{2N}{pg} - 1$. During processing, we maintain all Partition Groups in a priority queue and select the one with the highest estimation score, and repeat the process until top-k results are finalized. Let C_{pg} be the total number of Partition Groups picked up in this procedure. The time complexity of $cost_{spg} = O(N_{pg} + C_{pg} * log_{N_{pg}})$.

In each Partition Group, we also maintain a priority queue of partitions and choose which to process next. Similarly, let C_p be the total number of partitions picked up in this procedure, we have $cost_{sp} = O(N_{pg} * pg + C_p * log_{pg})$.

Let N_{pn} be the maximum number of candidate nodes retrieved from one partition. Then the cost of sorting N_{pn} nodes is $O(N_{pn} * log_{N_{pn}})$. Since there are at most $\lceil \frac{N}{p} \rceil$ partitions, we have $cost_{pn} = O(\lceil \frac{N}{p} \rceil * N_{pn} * log_{N_{pn}})$.

In each partition, multiple nodes could be retrieved and processed. Let C_{node} be the total number of nodes processed in the procedure. For each node, assume at most x number of nodes from its adjacent partitions are retrieved and bitmap "and" is computed. We have $cost_{and} = O(x * C_{node})$.

Normally, we have $C_{node} \gg C_p > C_{pw}$. Thus $cost_{and}$ dominates the overall performance. We further analyze the value of C_{node} and x in the next section.

4.5.4.2 Analysis of Bitmap Processing Cost

For any object, o, we use F_o to represent the frameset in which o appears in the given video. We use $f_{of} = min(F_o)$ and $f_{ol} = max(F_o)$ denote the first and last frame the object o

appears, separately.

We use $ts(o) = f_{ol} - f_{of} + 1$ to denote the time span of the object o, which computes the span between the first frame and the last frame o appears. We use $sc(o) = count(f_o)$ to represent the score of the object o, which computes the total number of frames the object o appears. For object o, its frame count does not exceed its time span, i.e. $ts(o) \geq sc(o)$.

Similarly, for an object set O with frame set F_O, we have $F_O = \bigcap_{o \in O} F_o$, the time span of object set O, $ts(O) = F_{Ol} - F_{Of} + 1$, and the score of object set O, $sc(O) = count(F_O)$. For all object $o \in O$, we have $ts(o) \geq ts(O)$ and $sc(o) \geq sc(O)$.

The value of C_{node}. According to Equation 4.5 (in Section 4.5), we estimate the possible remaining max score based on every adjacent λ_u partitions, where $\lambda_u = \lceil \frac{w-1}{p} \rceil + 1$. Similarly, for any object set O, we have the number of partitions involved $\lambda' \in [\lceil \frac{ts(O)}{p} \rceil, \lceil \frac{ts(O)-1}{p} \rceil + 1]$.

Assume the minimum score of the top-k result is sc_k, according to the algorithm, for every λ' partition, the algorithm stops when $\sum_{0 \leq i < \lambda'} sc_i \leq sc_k$.

For simplicity, we assume scores are equally distributed in all partitions and consider one partition only. In this case, for each partition, the stop condition is $sc_{stop} = \frac{sc_k}{\lambda_u} = \frac{sc_k}{\lceil \frac{w-1}{p} \rceil + 1}$. All nodes with $sc(O', i) \in [sc_{stop}, p]$ will be processed. To provide better performance, the value of sc_{stop} should be maximized to reduce the number of nodes visited. Thus, $\frac{w-1}{p}$ should be minimized, which means $p > w$.

The value of x. For object set O with time span $ts(O)$, the maximum number of partitions O involved is $\lambda'_u = \lceil \frac{ts(O)-1}{p} \rceil + 1$. For any $ts(O) > 1$, we have $\lambda'_u \in [2, +\infty)$. From Equation 4.5, we have the number of partitions involved in each window, w, $\lambda_u \in [2, +\infty)$.

For an object set O from partition i, the number of nodes retrieved and computed from other partitions, $x = \sum_{0 \leq j < \lambda, j \neq i} count_{node}(O, j)$, where $count_{node}(O, j)$ represents the number of nodes that are relevant to (i.e., can be retrieved using the hash index) object set, O, in partition j.

Since multiple partitions may share nodes with the same object set, reducing the number of partitions involved can also reduce the value of x.

Trend analysis. Based on the above discussions, the performance trend regarding different w and p is as follows:

If w is fixed: (1) If $p \geq w$, as p increases, the performance overhead of enumerating

window scores based on the interval list also increases, which leads to higher cost. (2) If $p \leq w$, as p increases, $\frac{w}{p}$ decreases, thus sc_{stop} increases, which lead to better performance.

If p is fixed, two cases arise: (1) w increases and sc_k increases, this happens when $w \leq ts_{max}$. The performance depends on which increases faster. (2) w increases and sc_k remains, in this case, $\frac{w-1}{p}$ also increases, leading to sc_{stop} decrease, thus the overall performance decreases.

To summarize, the major overhead of this method is the cost of computing bitmap "and". If the size of w is given, the partition size should be greater than the window size, $p > w$. When this is not feasible, we could set partition size $p \geq ts_{max}$, where ts_{max} is the maximum value of the object time span in the video to provide better performance.

4.6 Experimental Results

4.6.1 Settings

Environment. Our experiments are conducted on a machine with an Intel i5-9600K CPU, a GeForce GTX 1070 GPU, and 16GB RAM. All algorithms are implemented in Python [8] except the object detection/tracking algorithms for which we use open-source implementations [1].Similar to the experiments in Chapter 3, object detection and tracking algorithms can be replaced with any other available algorithms. In our experiment, we only focus on the efficiency aspect of the query evaluation, which is orthogonal to the accuracy of upstream algorithms. Moreover, different object detection and tracking algorithms may produce different numbers of objects in the annotation result, which is equivalent to varying the number of objects in the video annotations. For simplicity, we use the same object detection algorithm in the experiment, while the number of objects is considered as the characteristic of the videos and can be studied by using different videos. Data are pre-loaded to memory before evaluation.

Dataset. Experiments are conducted on real-world videos. We select three surveillance videos (S1-3, up to half an hour in length each), three news videos (N1-3, up to 45 minutes

[8]Code available at `https://github.com/yestinchen/ranked_wr`

Table 4.2: Video Statistics

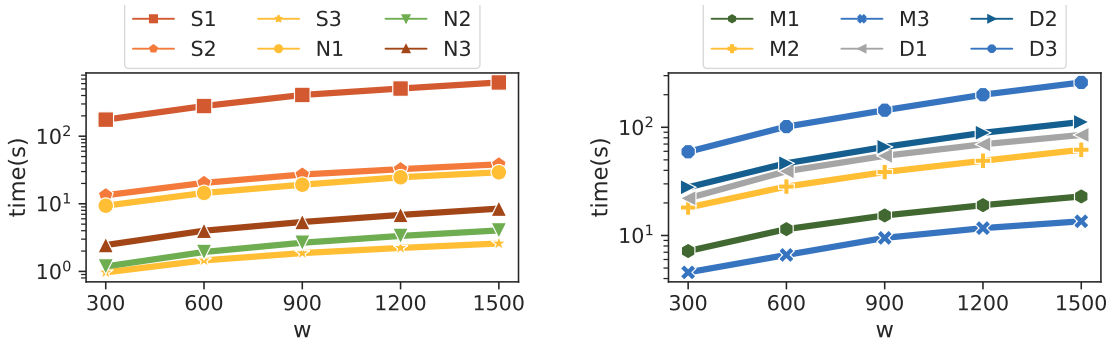
Video	**S1**	**S2**	**S3**	**N1**	**N2**	**N3**
# frames	51k	53k	36k	82k	38k	38k
# objects	7.95k	3.83k	1.14k	2.52k	1.11k	1.44k
# objs/f	13.66	7.97	4.75	2.61	2.17	2.54
max # objs/f	31	16	12	43	21	33
Video	**M1**	**M2**	**M3**	**D1**	**D2**	**D3**
# frames	195k	195k	202k	79k	79k	79k
# objects	6.34k	7.58k	4.30k	22.23k	22.44k	25.99k
# objs/f	2.04	2.51	1.64	7.20	7.35	9.13
max # objs/f	28	41	23	27	29	35

(a) N1 (b) D1

Figure 4.9: The Number of Objects with Different Time Spans

each), three movies (M1-3, up to two hours each), and three videos from an autonomous driving dataset (D1-3, up to one hour each). The surveillance and news videos are obtained from Youtube [9]. We arbitrarily choose three movies in their entirety: *Fast and Furious*, *Midway*, and *Inception*. We utilize YOLO as the object detection algorithm [122] and DeepSORT as the object tracking algorithm [153, 152] on S1-3, N1-3, and M1-3 (with similar trends observed when using other detection/tracking algorithms). We also use the MOT-labeled data from the DeepDrive dataset [167]. The labeled data from the DeepDrive dataset are split into small video clips and tracked independently; each clip consists of around 200 frames. We concatenate 400 clips together to form a long video and generate three independent videos (D1-3). Each video contains one primary object type (the type of objects that appear most in the video). For the experiments with single labels, we evaluate our method utilizing type *person* as query label (l) for videos N1-3, M1-3, and type *car* as query label for videos S1-3

[9]Videos are from the following URLs: `https://www.youtube.com/watch?v=wqctLW0Hb_0`; `https://www.youtube.com/watch?v=e_WBuBqS9h8`; `https://www.youtube.com/watch?v=Inbd-YOdPUE`; `https://www.youtube.com/watch?v=mLm0iZMIxWA`; `https://www.youtube.com/watch?v=GsQ4BPHiFRM`; `https://www.youtube.com/watch?v=ORlCakn1-hw`



(a) S & N (b) M & D

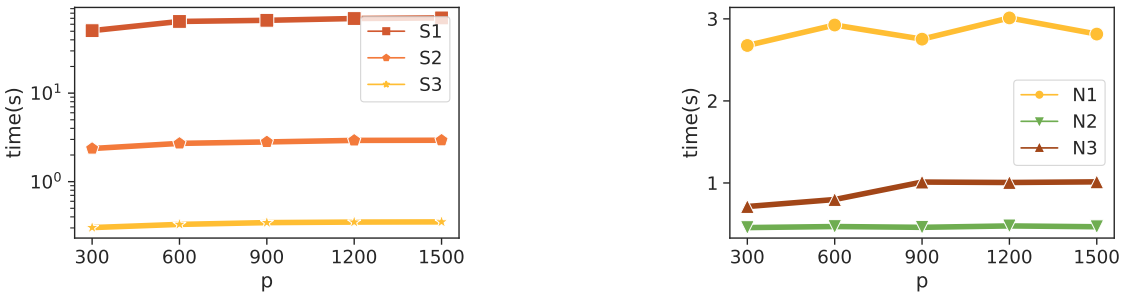
Figure 4.10: Query Processing Time without Index (Log-Scale)

and D1-3. Figure 4.9 presents the number of detected and tracked objects that have a given *time span* (number of contiguous frames in which those objects appear) in N1 and D1. As can be observed, most objects have a time span of less than 200 frames. Other videos share a similar distribution.

Baseline. We maintain a hash map between each unique object set (computed by objects intersection between frames) and the list of frames in which it appears. The hash map is shared across windows and is updated once the window slides. We also discard object sets that do not satisfy the filter condition immediately as an optimization. Even accounting for these optimizations, query execution is still very slow (e.g., up to more than 100 seconds for S1), as shown in Figure 4.10, where results are presented in two plots for better readability.

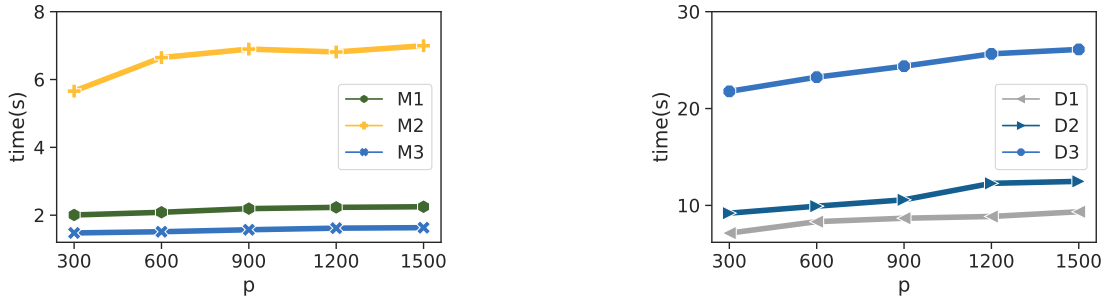
4.6.2 Index Construction

We vary the partition size, p, and evaluate both the index construction time and index size. Figure 4.11 shows the index construction time. As we increase p: (1) the number of nodes (unique object sets) increases in each partition; (2) for each object set, the frames in which it appears may also increase, leading to a higher cost of score computation. Thus, the overall cost of index construction increases, which is consistent across all videos. Similar trends can be observed in Figure 4.12: the index size increases as p increases. Note that the index size could vary significantly for different videos. For videos with more objects per frame (e.g., D1-D3, S1), the index size is much larger than others.



(a) S1-3 (log-scale)

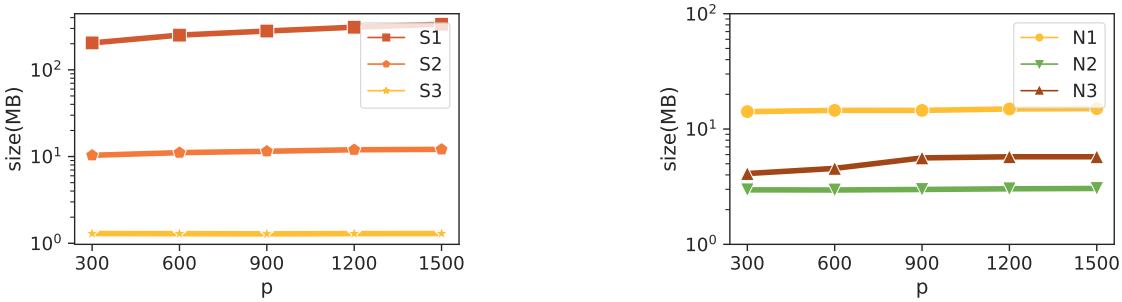
(b) N1-3



(c) M1-3

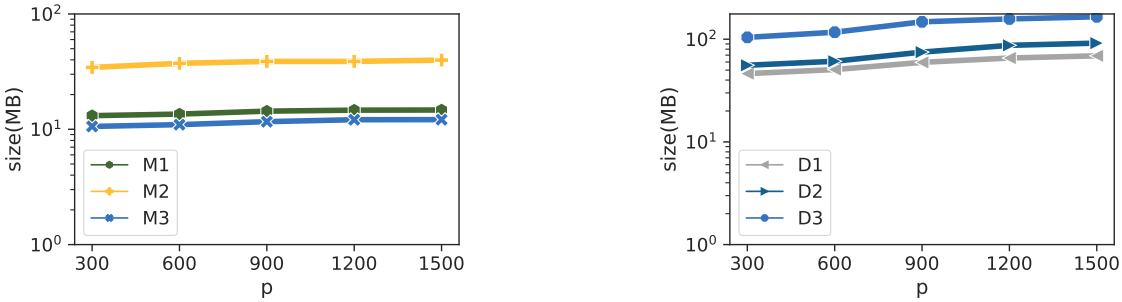
(d) D1-3

Figure 4.11: Index Construction Time



(a) S1-3 (log-scale)

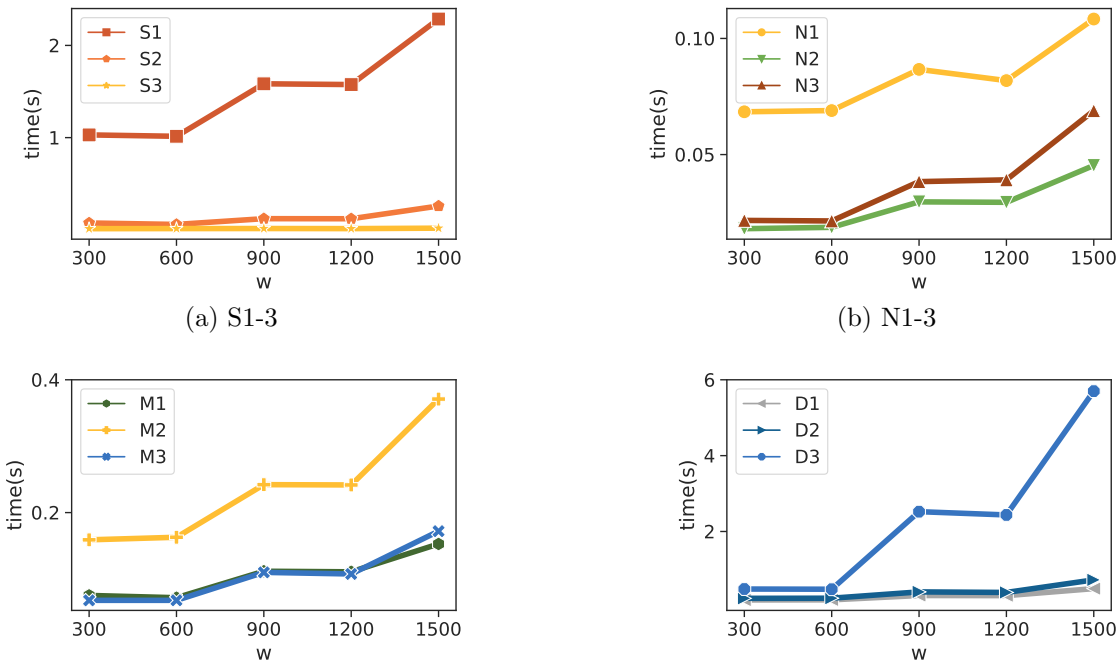
(b) N1-3 (log-scale)



(c) M1-3 (log-scale)

(d) D1-3 (log-scale)

Figure 4.12: Index Size



(a) S1-3 (b) N1-3

(c) M1-3 (d) D1-3

Figure 4.13: Query Processing Time with Varying w

4.6.3 Query Processing

Default Settings. We start with queries with single labels. Similar to the example demonstrated in Section 4.5.3.3, the condition is set to $count(\{l\}) \geq num$, where l is the query label and num is the filter value. By default, we set both w and p at 600 frames (20 seconds with 30fps), $num = 6$, $k = 1$, and $pg = \lambda_u$. We vary each parameter and study its impact on query processing time.

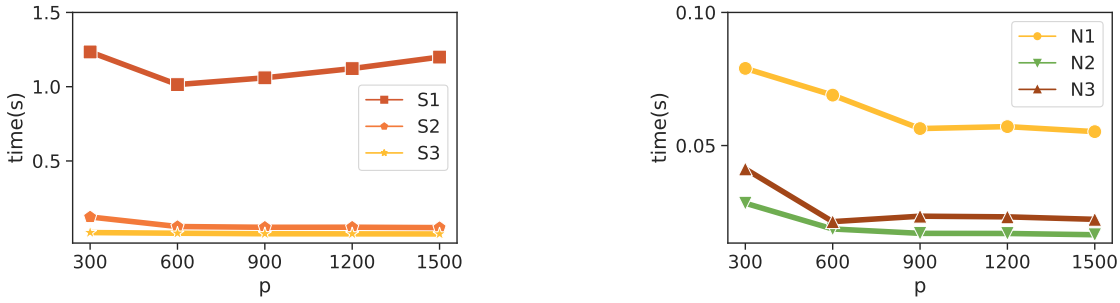
Varying w. We set $p = 600$, and vary w, and the result is shown in Figure 4.13. In general, more objects in videos lead to higher query processing time (e.g., S1 and D3). As we increase w, the overall cost increases due to the following reasons. (1) With partition size p fixed, larger w leads to more partitions in each Partition Group, resulting in increased cost of computing score for windows in each Partition Group; (2) One partition is involved in multiple Partition Groups, and therefore the same node could be processed multiple times within different Partition Groups; (3) According to Equation (4.8), the value of the estimated score for each Partition Group also increases due to larger λ_u. If the minimum

score in the result does not increase as w increases, then more nodes have to be processed in each partition, thus leading to a higher cost. For videos with more objects (e.g., S1, D3), this trend is even more pronounced since different object sets in each partition are more likely to share similar scores, and these high-scored nodes in each partition may not contribute to the final results.

Varying p. We next set $w = 600$ and vary p from 100 to 1500, as shown in Figure 4.14. When $p < w$, we observe decreasing query time as p increases. When $p = 100$, each Partition Group consists of 7 partitions (i.e. large λ_u), with the worst performance among all settings. This is because there are still many objects with a time span > 100 (shown in Figure 4.9), and smaller p means smaller scores for nodes in each partition. In each partition, nodes with a higher score may not contain the final results; thus more nodes should be processed. When $p > w$, we have $\lambda_u = 2$. Two factors could impact the query time: (1) object sets that split in two partitions with smaller p could now fall into the same partition, thus leading to a lower cost during computation (less computation between different partitions); (2) if the same object set still splits into more than one partition, the cost of processing each partition increases due to larger p (more nodes in each partition to process), thus the overall cost could increase. Moreover, since more windows are enclosed in each Partition Group, more window scores are computed for each generated object set, leading to a higher overall cost. These indicate that on different videos, we can observe either an increasing (S1) trend, decreasing (N1) trend, or even a peak point at a certain p value (M3, $p = 1200$).

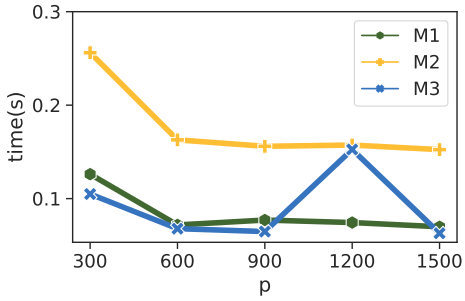
Varying *num*. As shown in Figure 4.15, as *num* increases from 2 to 10, the query time shows an increasing trend in general. This is because as we increase *num*, the number of object sets that satisfy the query in each video decreases, leading to a lower score value in the final results. Meanwhile, all nodes are sorted with scores and we always select nodes with the highest score first. Although a lower number of results may be returned when the threshold *num* increases, we have to explore more nodes in the index (as more objects are involved in satisfying the given condition), leading to a higher cost. This also demonstrates that even when *num* is small, the early-stopping mechanism is able to reduce the overall query time.

Varying k. We then vary k from 1 to 2000, controlling how many results are selected.

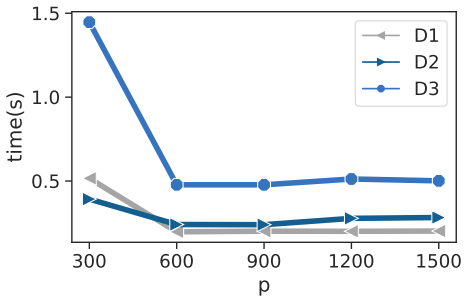


(a) S1-3

(b) N1-3

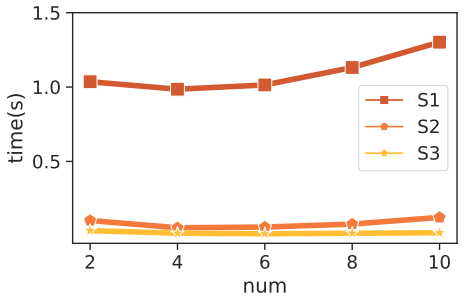


(c) M1-3

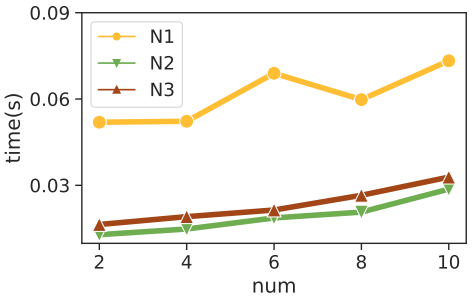


(d) D1-3

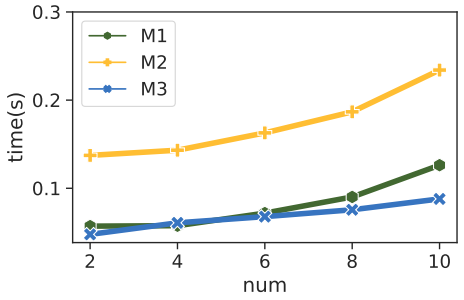
Figure 4.14: Query Processing Time with Varying p



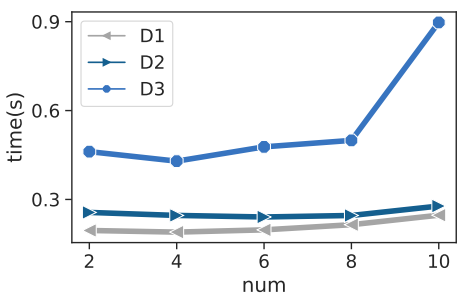
(a) S1-3



(b) N1-3

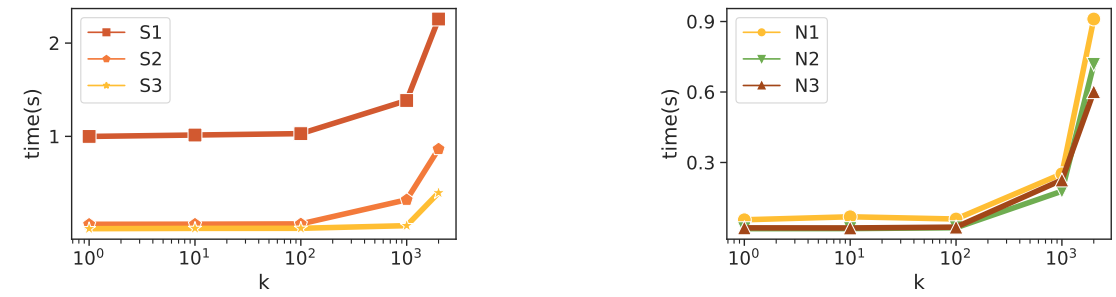


(c) M1-3

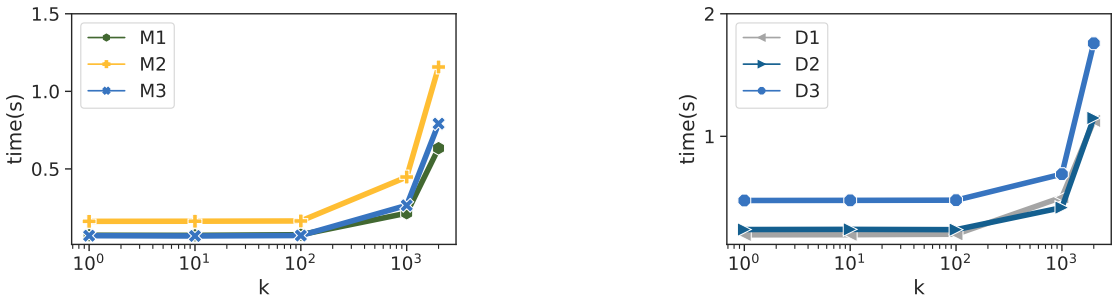


(d) D1-3

Figure 4.15: Query Processing Time with Varying *num*



(a) S1-3 (b) N1-3

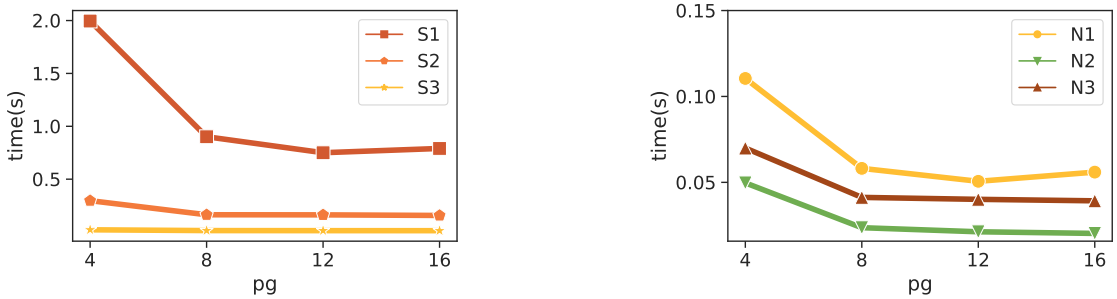


(c) M1-3 (d) D1-3

Figure 4.16: Query Processing Time with Varying k

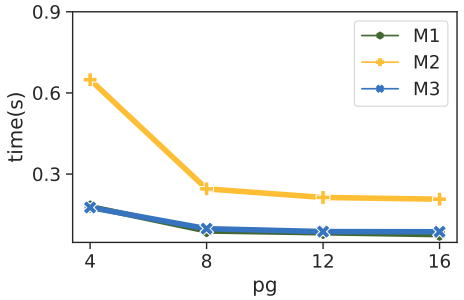
As expected, larger k leads to higher costs, since more nodes are retrieved and processed in each partition. Evaluating videos with more objects has an even higher cost when k is large. However, most queries terminate within 1 second, which is still much faster than the baseline. When $k = 1$, for example, our method is more than 100 times faster than the baseline on videos M2 and S1. Large savings persist even as k increases. For example, on M2 our method is more than 20 times faster, while on S1 it is still more than 100 times faster when $k = 2000$. These results point to the significant advantages of our approach thanks to the early-stopping mechanism from a performance perspective, making our technique highly attractive for the specific class of queries we focus on.

Varying pg. Finally, we set $p = 300$, $w = 900$, and vary pg from 4 ($= \lambda_u$) to 16. As pg increases, each Partition Group consists of more partitions. This leads to two possible scenarios: (1) each Partition Group could generate more windows; (2) each partition participates in fewer Partition Groups, and thus the amortized processing time for each partition is reduced. We can observe from Figure 4.17 that the query time decreases when pg varies from 4 to 8, indicating that the cost is reduced due to scenario (2). We can also observe a

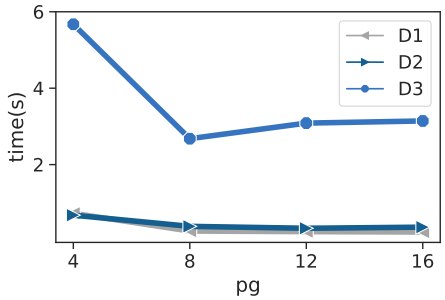


(a) S1-3

(b) N1-3



(c) M1-3

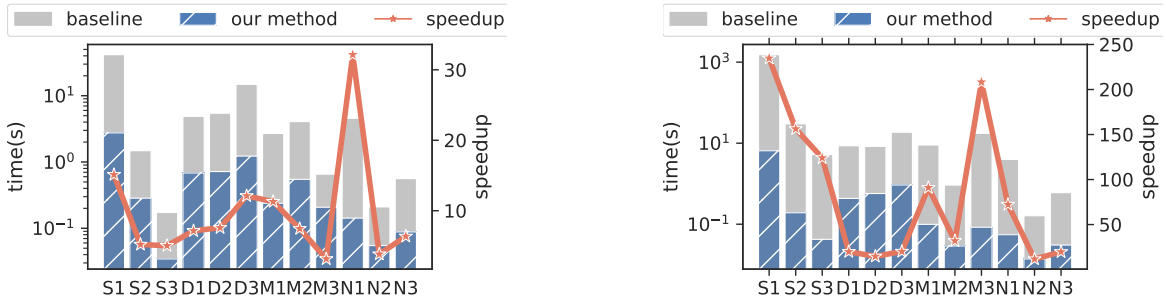


(d) D1-3

Figure 4.17: Query Processing Time with Varying pg

slightly increasing trend when pg varies from 8 to 16 due to scenario (1). In general, setting $pg = 2\lambda_u$ seems to be an appropriate value in most cases, where each partition only appears in at most 2 Partition Groups. For $pg > 2\lambda_u$, although some partitions may appear in one Partition Group only, the cost of computing window scores inside each Partition Group may increase, thus leading to the non-decreasing overall cost.

Summary. From the above experiments, we observe that query time can be minimized if $p = w$. However, it is not always feasible to assume exact knowledge of w prior to queries. Both Figures 4.13 and 4.14 show that larger values of w have more impact on query performance compared to larger p. From Figure 4.14, we can also observe that the query performance is the worst when p is too small to cover the time span (time that objects appear in the video) of most objects. Thus, from a query evaluation perspective, we should prefer a larger p (with a value that at least covers the time span of most objects in the video). Meanwhile, a larger p could lead to higher index construction costs. Object time spans and precise statistics can be obtained with a single pass of the video and be utilized during ingestion time to construct partitions. These statistics can guide the choice of p in the



(a) Q0 (Log-Scale)

(b) Q1 (Log-Scale)

Figure 4.18: Queries with multiple labels in the Filter Condition

absence of query workload information. If such a workload is available, further optimization on the choice of p is possible.

Finally, we experiment with the execution of two conjunctive queries: Q0 and Q1. Q0 inquires for at least 2 red objects ($count(\{l, red\}) \geq 2$, where l is *car* or *person*, depending on the type of objects present in each video), while Q1 focuses on at least 2 different types of objects with count ≥ 2 (either $car \geq 2 \wedge person \geq 2$ or $car \geq 2 \wedge truck \geq 2$, depending on the video type). Since the color label is not included in the detection results, we generate 5 different colors and assign them randomly to each object in the video. We build partitions on both labels and evaluate the query. The results are consistent with those presented for the case of single conditions, shown in Figure 4.18. Our proposed technique exhibits vast performance benefits of up to 100 times speedup on certain videos (e.g., S1 and S2). We can also observe that the speedups on Q0 are generally lower than those on Q1. This is because the filter condition in Q0 has a lower selectivity: objects have to satisfy both conditions *red* and *car*, while only approximately 20% objects are associated with *red* label; and in Q1, the filter condition is much looser. Lower selectivity leads to lower scores in the final results, meaning more nodes have to be explored to guarantee that the estimated score for each Partition Group is less than the score obtained in the top-ranked results. These results attest to the benefits of our overall approach. It would be also interesting to study the potential optimizations on multiple-label queries (e.g., building a joint index on multiple labels to further speed up the query evaluation); precise exploration of such optimizations is a promising area of future work.

4.7 Summary and Future Work

The advances in Deep Learning have enabled the accurate identification of objects along with other useful information such as object attributes, opening new avenues for processing expressive queries on videos. We have presented a study of ranked window queries over video repositories. We take a partition-based approach that builds and materializes indexes along with metadata for each partition (the PBI indexes). We have presented two algorithms that construct indexes and evaluate queries efficiently. Via experiments, we have demonstrated that our proposal achieves orders of magnitude improvement over baseline methods. As future works, we could further explore index optimizations for more complex queries.

5 Spatial and Temporal Constrained Ranked Retrieval over Videos

5.1 Introduction

5.1.1 Background and Motivation

In this chapter, we focus on a new type of video retrieval (called *STAR Retrieval*) that seeks to identify video segments of a set duration, while satisfying user-provided constraints on the spatial relationships among objects of interest, as well as other conditions on the object labels (such as object type, object color, etc.). Such queries are helpful in scenarios where one is interested in identifying video clips from a large video repository that contains user-specified objects with user-specified spatial and temporal constraints among them. Such operations are prevalent in video editing, video content creation, news production, marketing, and advertising of social media content. For example identify historical footage in which Barack Obama is on the right of a fighter Jet and left of Michelle Obama during the preparation of a news story, using archive videos, identify archive clips of a coastguard vessel next to a burning tanker to easily assemble footage from related past events, or creating popular social media compilations based on social media challenges. Such queries usually only focus on particular moving patterns of selected objects, based on the relative spatial relationships between objects being considered.

Different from works discussed in the previous chapters, where only temporal relationships are supported, we consider both the spatial relationships between objects in the same frame and the temporal moving pattern across frames. We use graphs as the data model to encode both object attributes and the spatial-temporal relationships between objects. Specifically,

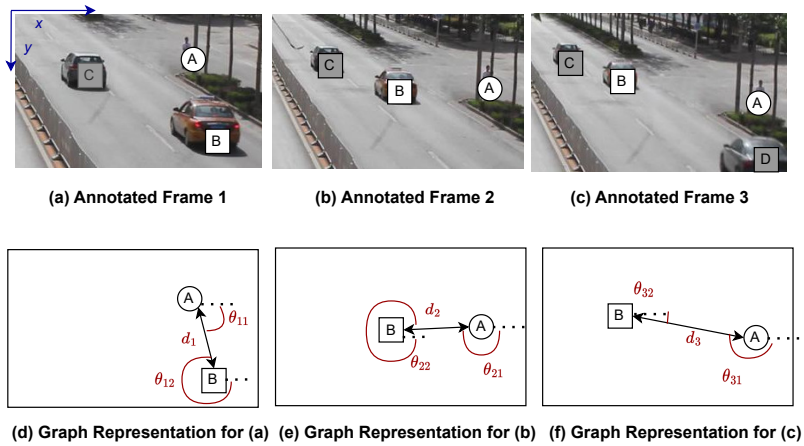


Figure 5.1: An Example

1. Each vertex on the graph represents an object, with object labels as vertex attributes, which can be obtained via CV algorithms.

2. Each edge represents the spatial relationship between two objects. We consider relative positions between any two objects, since in most cases objects are moving, and the absolute position of objects in the frame is less important. Moreover, queries that require conditions on the absolute position of objects can be easily supported by incorporating the position as an attribute on the vertex.

3. Based on the above, a sequence of graphs is used to capture the temporal relationships of objects between frames. We employ the track ID, produced by object tracking algorithms, as the unique identifier for each vertex, where the same track ID is used across the graph sequence for vertices that refer to the same object.

Figure 5.1a-c depicts an example scenario where a car (annotated by B) is passing the pedestrian (annotated by A) from the left. For each frame, we can construct a corresponding graph, shown in Figure 5.1d-f. Utilizing the unique track ID on vertices, the temporal information between frames is easily captured by the sequence of graphs. For each graph, the object labels are represented with different shapes (circle and square) in the vertices, while the spatial information is represented in two groups of values: (1) the distance between two objects (e.g., d_1 in Figure 5.1d), and (2) the angle values between two objects (e.g., θ_{11}, θ_{12} in Figure 5.1d).

In the above example, we only consider object type for brevity. Other attributes of interest such as object colors, vehicle brand, etc., can be easily added and supported. Moreover, since both the angle and the distance between two objects can be continuous values, retrieving videos while requiring exact matches in these values would be too strict to apply in any practical scenario. Thus, we introduce edge discretization to restrict edge attributes to a pre-defined number of buckets. As such we reduce the value range for edge attributes while guaranteeing that attribute values in close proximity share the same discretized values.

Graph representations capture both the object labels on nodes and relationships between objects on edges explicitly, making it easier to support arbitrary query conditions on both objects and relationships between objects. Utilizing graph modeling, any video of arbitrary length can be represented with a sequence of graphs, where the spatial information of objects is captured implicitly by track IDs throughout the sequence. Queries are provided as a short sequence of graphs (one graph per frame of the query), representing a certain movement pattern amongst objects, with the spatial relationships between objects in each frame being considered. We use query graph sequences to denote such queries. The query graph sequence can be obtained either by examples or by sketches. Specifically, with a query-by-example approach, users can provide a short video clip as the initial step. With object detection and tracking algorithms applied, users then specify the set of objects that are of interest. Based on the annotation results, a graph sequence can be generated. Similarly, with a query-by-sketch approach, users construct a sequence of graphs directly with specified attributes on nodes (objects) and edges (spatial relationships between objects, i.e., d and θ values from the above example). Such approaches can be easily realized and are useful to retrieve specific moving patterns from videos in scenarios such as scene retrieval and pattern search. As such, we can composite only selected objects of interest into the query graph, making the query more specific and precise, which cannot be done utilizing Machine Learning-based approaches.

5.1.2 Our Proposal

The main objective is to retrieve ranked video clips from a video repository such that: (1) the duration of video clips (number of frames) is the same as the number of frames in the

given query, and (2) each video clip is associated with a matching score, where video clips with higher matching scores are ranked higher in the query results. The candidates of such video clips can be obtained by imposing a sliding window over whole videos to obtain clips, where the window size is equal to the length of the given query.[10] In a similar manner, we also construct graph representations for each video clip. By aligning two graph sequences obtained from the query and the video clip, we derive a one-to-one mapping between graphs in the video clip and graphs in the given query. We propose a matching framework to produce the matching score of each video clip, where the score reflects the closeness of the video clip and the given query, which will be formally defined in Section 5.2.

We refer to this problem as Spatial and Temporal constrAined Ranked Retrieval (STAR Retrieval) over videos. For ease of presentation, we use the term *windows* to refer to the video clips extracted from videos in the repository and use the term *query graph sequence* to denote the graph representation of the given query, which will be formally defined in Section 5.2.

Answering such queries by enumerating all possible windows and computing their matching scores as described above could be very inefficient and costly. To accelerate query answering, We simplify the query graph and propose a two-phase framework, consisting of both an ingestion phase, where a Graph Index (GI) is constructed and materialized, and a query phase, where queries are evaluated efficiently utilizing the proposed algorithms. The GI index provides efficient edge retrieval according to the given vertex attributes and edge attributes while minimizing the storage overhead. During the query phase, a matching procedure is performed utilizing spatial and temporal information stored in the GI index to prune unrelated edges or data graphs early. By adopting an early-stopping mechanism based on score estimation, the matching procedure can terminate as soon as the ranked result is finalized.

Our contributions are summarized as follows:

- We introduce and formalize the STAR Retrieval problem using graphs to represent spatial and temporal relationships of interest in both videos and queries.
- We propose to use Graph Index (GI) to precompute and materialize the graphs con-

[10]We use sliding window semantics as an example; other window semantics can also be easily supported.

structed from annotations with edge discretization, which is then used to accelerate query answering.

- We propose two algorithms, namely, Spatial Matching to perform efficient graph matching for each query graph, and Temporal Matching to perform temporal matching across query graphs with an early-stopping mechanism.

- We evaluate our approach via real-world videos, demonstrating orders of magnitude performance improvement over the baseline on queries with high selectivity.

5.2 Problem Definition

A video, $\mathcal{V}$, is a sequence of frames, $\mathcal{V} =< f_1, \ldots, f_n >$. For each frame, we can obtain a set of objects, where each object is represented by a unique identifier (i.e., object ID), along with other labels such as object type, object color, etc. Let $\mathbb{L}$ be the set of all labels, and $\mathbb{ID}$ be the set of all object identifiers. We use a function from $\mathbb{ID}$ to $\mathbb{L}$ to describe the labels of each object represented by its object identifier. We utilize a graph abstraction to represent the information extracted from each frame. In this representation, each vertex is an object identifier (i.e., object ID), and each edge represents the spatial relationships between two vertices. We use Object Graphs to denote such graphs, defined as follows:

Definition 8 (Object Graph). *We represent a given frame f by a graph $G_f = (V_{G_f}, E_{G_f}, L_{G_f})$, where $V_{G_f} \subseteq \mathbb{ID}$ is the set of object IDs, E_{G_f} is the set of edges, and $L_{G_f} : V_{G_f} \rightarrow \mathbb{L}^*$ is a labeling function that maps an object ID to a set of labels. Each edge $e_i \in E_{G_f}$ takes the form of $e_i = (u, v, \theta_i, d_i)$ that denotes the spatial relation between two objects with IDs, u and v, where θ_i represents the angle from the object with ID u to the object with ID v, while d_i represents the distance between the two objects.*

For each frame, $f_i \in \mathcal{V}$, the object graph can be extracted with the following information captured on the graph:

1. *Vertices (Object IDs).* The unique identifier (i.e., object ID) of each object in the frame, can be obtained utilizing object tracking algorithms.

2. *Vertex Attributes (Object Labels).* The labels (metadata) regarding any object in the frame, including attributes, such as color, object type, etc., can be obtained directly from object detection results. We assume that such object labels are static attributes and do not change for each unique object recognized by the same object identifier.

3. *Edge Attributes (Spatial Relationship between Vertices).* θ and d values are associated with edges, representing the spatial relationships between objects in the frame, which can be derived from the relative positions of bounding boxes produced by object detection algorithms.

The edge attributes can be obtained as follows: we represent each frame in a two-dimensional space with the origin at the top-left corner of the frame, as shown in Figure 5.1a. For each object with identifier, o, we can obtain its position (x_o, y_o, h_o, w_o) in pixels from the object detection results, where x_o and y_o denote the horizontal and vertical positions of the object center, h_o is the height of the object, and w_o is the width of the object, assuming that each object is recognized as a rectangle. For any given pair of objects with ID u and v in the frame, the edge attributes can be computed based on the center points as follows: $d = \sqrt{(\Delta x)^2 + (\Delta y)^2}$, $\theta = \arctan \frac{\Delta y}{\Delta x}$, where $\Delta x = x_v - x_u$ and $\Delta y = y_v - y_u$.

We use *Query Graph* to denote the Object Graph corresponding to a single frame of the given query and *Data Graph* to denote the Object Graph constructed based on a frame from the video repository. Then the matching between Query Graphs and Data Graphs is defined as follows.

Definition 9 (Query Graph Matching). *Given a query graph $P = (V_P, E_P)$ and a data graph $G = (V_G, E_G)$, the Query Graph Matching returns all subgraphs $M = (V_M, E_M)$ of G if there exists a bijective function $h : V_P \to V_M$ such that for each $v \in V_P, L_P(v) \in L_M(h(v))$, and for each edge $(u, v, \theta, d) \in E_P$, $(h(u), h(v), \theta, d) \in E_M$. We use $P \underset{h}{\sim} M$ to denote that graph M is a match of P with function h, and $P \underset{h}{\sim} G$ to denote that there is at least one subgraph of G that matches P with h.*

We reuse the same vertices (Object IDs) across frames utilizing the unique tracking identifiers provided by object tracking algorithms. The object graph sequence is defined as follows.

DEFINITION 10 (Object Graph Sequence). *An object graph sequence,* $\mathcal{G}_F = \langle G_{f_1}, \ldots, G_{f_l} \rangle$ *is a sequence of graphs constructed based on a sequence of frames* $F = \langle f_1, \ldots, f_l \rangle$*. The length of an object graph sequence is the total number of frames in* F*, i.e.,* $|\mathcal{G}_F| = |F|$.

We use $V_{\mathcal{G}_F}$ to denote the set of unique vertices in the graph sequence. We obtain a *Query Graph Sequence* from the given query and a *Data Graph Sequence* based on any video clip from the video repository. We can now introduce the notion of matching between a query graph sequence and a data graph sequence of the same length based on the pairwise matching of individual query graphs and data graphs.

DEFINITION 11 (Query Graph Sequence Matching). *Given a query graph sequence* $\mathcal{P}$ *(which we consider a* pattern*) and a data graph sequence* $\mathcal{G}$*, with* $|\mathcal{P}| = |\mathcal{G}| = l$*, if there exists an injective function* $h : V_{\mathcal{P}} \rightarrow V_{\mathcal{G}}$ *and* $\exists i \in [1, l]$*, such that* $P_i \underset{h}{\sim} G_i$*, where* $P_i \in \mathcal{P}$ *and* $G_i \in \mathcal{G}$*, then we say that graph sequence* $\mathcal{G}$ matches *pattern* $\mathcal{P}$ *with* h*. The matching score is defined as:*

$$score(\mathcal{P} \underset{h}{\sim} \mathcal{G}) = \sum_{P_i \in \mathcal{P}, G_i \in \mathcal{G}} score(P_i \underset{h}{\sim} G_i)$$

where

$$score(P_i \underset{h}{\sim} G_i) = \begin{cases} 1, & P_i \underset{h}{\sim} G_i \\ 0, & otherwise \end{cases}$$

For any arbitrary function h, we have $0 \leq score(\mathcal{P} \underset{h}{\sim} \mathcal{G}) \leq l$. When $score(\mathcal{P} \underset{h}{\sim} \mathcal{G}) = l$, then $\mathcal{G}$ *fully* matches $\mathcal{P}$ with h (i.e., $\mathcal{G}$ is a *complete match* of $\mathcal{P}$ with h). When $0 < score(\mathcal{P} \underset{h}{\sim} \mathcal{G}) < l$, then $\mathcal{G}$ *partially* matches $\mathcal{P}$ with h (i.e., $\mathcal{G}$ is a *partial match* of $\mathcal{P}$ with h). Otherwise, $\mathcal{G}$ is not a match of pattern $\mathcal{P}$ with h.

DEFINITION 12 (Query Graph Sequence Matching Score). *We use* H *to represent the set of all possible functions between domain* $V_{\mathcal{P}}$ *and range* $V_{\mathcal{G}}$*. The matching score of the data graph sequence* $\mathcal{G}$ *with respect to the query graph sequence* $\mathcal{P}$ *is defined as the maximum score produced by function* $h \in H$*: (*$|\mathcal{G}| = |\mathcal{P}| = l$*)*

$$score(\mathcal{P} \sim \mathcal{G}) = \max_{\forall h \in H} score(\mathcal{P} \underset{h}{\sim} \mathcal{G})$$

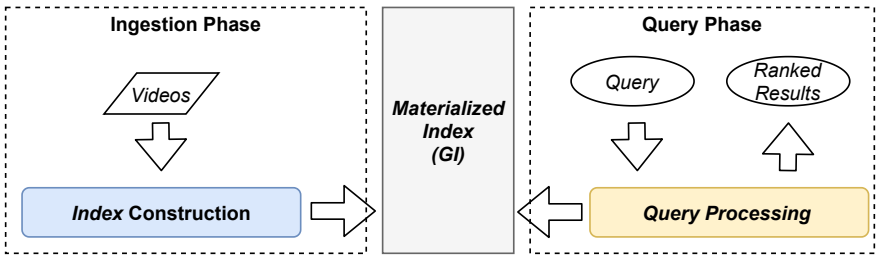


Figure 5.2: Framework Overview

Given the query graph sequence $\mathcal{P}$, and a (long) video $\mathcal{V}$, we wish to determine the video clips (i.e., windows of frames) with the highest matching scores, defined as follows.

Definition 13 (Spatial and Temporal Constrained Ranked Retrieval (STAR Retrieval)). *Given a video $\mathcal{V}$, a query graph sequence $\mathcal{P}$ and a query parameter k, we use $\mathcal{W}$ to represent all windows of frames with size $|\mathcal{P}|$ generated on video $\mathcal{V}$, and use $\mathcal{G}_W$ to represent the data graph sequence constructed from video clip $W \in \mathcal{W}$. The result of STAR Retrieval is a ranked list of k video clips, $Result = (W_1, \ldots, W_k)$, where $\forall W_i \in Result$, $score(\mathcal{P} \sim \mathcal{G}_{W_i}) \geq score(\mathcal{P} \sim \mathcal{G}_{W_{i+1}})$ and $\forall W_j \in \mathcal{W} \setminus Result$, $score(\mathcal{P} \sim \mathcal{G}_{W_j}) \leq score(\mathcal{P} \sim \mathcal{G}_{W_k})$.*

Definition 13 specifies STAR Retrieval on one video. Similarly, for a video repository, STAR Retrieval is conducted by either processing all videos iteratively or abstracting the video repository as a long video. By applying windows on a video, the video clip is formed with only frames enclosed in one window. For simplicity, in the remaining sections, we use *matching score* to denote the query graph sequence matching score for each window.

5.3 Framework Overview

Performing STAR Retrieval relies on the graph representation of videos. However, it is infeasible to build graph sequences using all videos in the repository, from scratch, each time a query is posed. We adopt a two-phase approach to amortize graph construction cost over all queries and accelerate query answering on the pre-materialized index, which consists of a video ingestion phase and a query phase, as shown in Figure 5.2.

The ingestion phase is conducted as a pre-processing step, where each frame is routed to applicable object detection and object tracking algorithms which provide all suitable annotations for each frame. Our objective is to compute all spatial relationships between

vertices in each frame and materialize it as edge attributes to facilitate query processing. Based on the vertex attributes and edge attributes, we build and materialize the Graph Index (GI), which is meant to serve all possible incoming queries; thus the cost of video ingestion is amortized over all queries. Since this phase is independent of any query, it is conducted as a pre-processing step.

During the query phase, ranked results for a given STAR Retrieval query are produced utilizing the materialized index, which involves two main steps: window generation and matching score computation. We deploy pruning strategies in both steps to reduce the computational cost. Specifically, utilizing edge and vertex attributes in the query graph sequence, we only retrieve relevant vertices and edges from the materialized index, reducing both the size of the data graph for each frame and the number of windows generated; during matching score computation, we introduce score estimation to prioritize windows that are likely to produce higher matching scores and apply an early-stopping mechanism to avoid further computation once the ranked result is finalized.

We first discuss how to build the GI index during the ingestion phase in Section 5.4, and then focus on query processing in Section 5.5.

5.4 Graph Index (GI)

The Graph Index is built on the video repository during the ingestion phase. The main objective is to pre-compute the data graph per frame in the video repository and materialize sufficient data such that for any given query, the data graph sequence for each window can be constructed efficiently. We achieve this based on two main ideas: (1) simplify the query graph sequence, such that the number of edges retrieved from the index can be minimized (Section 5.4.1); (2) introduce edge discretization to enable sharing between edges with the same attributes (Section 5.4.2).

5.4.1 Minimum Object Graph

As per Definition 8, given a frame, for any vertex (i.e. Object ID) in the frame, $v \in V_{G_p}$, there must be an edge, $(v, v', \theta, d) \in E_{G_f}$, where v' is any other vertex in V_{G_f} ($v' \neq v$). In

other words, the graph G_f for frame f is a complete graph. This implies that the number of edges could increase dramatically when the number of objects increases in each frame. We, therefore, focus on developing a more compact data structure to store the information contained in the object graph.

To begin with, instead of keeping and matching two directed edges between any two vertices, we could use one directed edge without losing any information. Specifically, consider the spatial relationship between vertices, u and v, in the object graph. There exist two edges $e_1, e_2 \in E_G$, where $e_1 = (u, v, \theta_1, d_1)$ and $e_2 = (v, u, \theta_2, d_2)$. Given the values of distance d_1 and angle θ_1, we can easily derive those of d_2 (which is equal to d_1) and θ_2.

We can further reduce the number of edges maintained by exploiting their geometric properties. We first show that for any trio of vertices with an edge between each vertex pair, we can derive the attributes of any edge from those of the other two, as stated in the following lemma.

LEMMA 3. *Let X, Y, Z be three distinct vertices in the graph G_f for frame f. The edge attributes of any two pairs of vertices from X, Y, Z sufficiently determine the edge attribute of the remaining pair.*

It is easy to show that the attributes for the third edge can be derived utilizing trigonometric functions based on the triangle formed by X, Y and Z in the 2D space for the frame. The formal proof is omitted for brevity.

Lemma 3 leads to the following way of reducing the number of edges in the object graphs. We introduce *minimum object graphs*.

DEFINITION 14 (Minimum Object Graph). *A minimum object graph M_f is a graph derived from G_f by retaining all the vertices in G_f but only the minimum number of edges in G_f to ensure that it remains a connected graph, i.e. $|E_{M_f}| = |V_{M_f}| - 1$.*

By applying Lemma 3 iteratively, we can prove the following theorem.

THEOREM 4. *The edge attributes for any pair of vertices in a minimum object graph can be derived using edges solely from this graph.*

To reduce the complexity of retrieving data graph sequences and computing query graph matching, we use minimum object graphs instead of object graphs for the query representa-

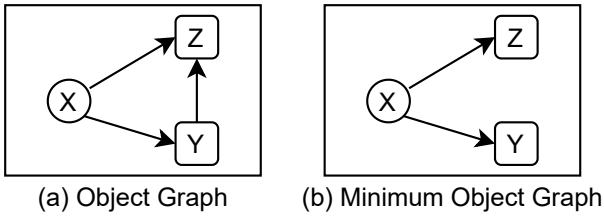


Figure 5.3: Minimum Object Graph

tion. As per Definition 14, to construct a minimum object graph, we can select any vertex as the start vertex (which we call the *anchor vertex*), and keep only those edges from the start vertex to all other vertices. In this case, the minimum object graph can be viewed as a tree, where the height of the tree is 1 and the anchor vertex is the root of the tree. Without loss of generality, in the following discussion, we select the top-left most vertex, o, as the anchor vertex (i.e., the object with the lowest (x_o,y_o) value, where x_o, y_o denote the horizontal and vertical positions of the object center in the frame) and only keep edges starting from this vertex in the minimum object graph. Figure 5.3 depicts an example of converting an object graph to a minimum object graph. We use circles and squares to denote different types of objects (i.e., with different labels), and use X, Y, Z to denote three unique vertices.

We use *Minimum Query Graph Sequence* to denote the sequence of minimum object graphs generated from a given query. Since we only represent minimum query graphs and minimum query graph sequences, in the remaining sections, we use query graph and minimum query graph interchangeably and use query graph sequence and minimum query graph sequence interchangeably.

5.4.2 Edge Discretization

We now discuss how to compute the distances and angles in the edge attributes and how to compress such information in the graph index.

To ensure consistency between different video resolutions, the distance between vertices in a frame is normalized. For two vertices u and v, their distance $d(u, v)$ is computed as $d(u, v) = \frac{d_p(u,v)}{d_l}$, where $d_p(u, v)$ is the Euclidean distance between the two vertices, and d_l is the diagonal length of the frame. Both $d(u, v)$ and $d_p(u, v)$ have the same units (say, pixels) and $d \in [0, 1]$. Angles can be computed as per their definition in Section 5.2.

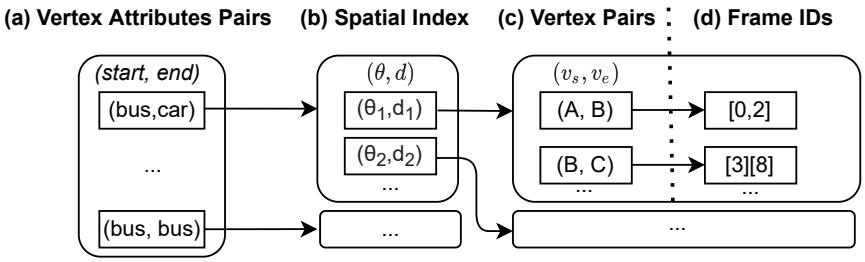


Figure 5.4: The Graph Index

Typically, for querying purposes, the exact distances and angles between the vertices are not critical, thus we reduce the distances and angles to a lower resolution via discretization. This has the benefit of lowering the cost of edge retrieval during query processing (to be detailed in Section 5.5.2) as well as minimizing the storage overhead of the graph index (to be discussed in Section 5.4.3). For discretization, we determine the number of buckets allocated to storing θ and d values, and simply keep their respective bucket index as the discretized value. We stress however that our approach would work unchanged without any discretization if that is required.

5.4.3 Index Construction

To guarantee that the relationships between any two vertices are captured and materialized, during the ingestion phase we generate and materialize complete graphs for each frame in the videos we pre-process. Since the object graphs generated from adjacent frames in a video are likely to share the same vertices and same edge attributes, we could reuse such information and minimize the storage overhead. Based on the above along with edge discretization, we propose the Graph Index (GI), illustrated in Figure 5.4.

For each video, $\mathcal{V} =< f_1, \ldots, f_n >$, in the repository, we first obtain the data graph sequence, denoted by $\mathcal{G}_V$. We use $E_{\mathcal{G}_V}$ to denote the set containing all edges in $\mathcal{G}_V$, and use $V_{\mathcal{G}_V}$ to denote the set containing all vertices in $\mathcal{G}_V$. To materialize the entire data graph sequence, three types of information need to be captured: vertex attributes attached to each vertex $v \in V_{\mathcal{G}_V}$, edge attributes stored in edges $e \in E_{\mathcal{G}_V}$, and the temporal information (i.e., the temporal relationships of vertices between frames), captured by the sequence of frames. We start with indexing the vertex attributes from each data graph, as shown in Figure 5.4a.

For each edge, we obtain the attributes for both vertices. For example, the vertex attribute pair, *(bus, car)*, denotes the edge starting from a vertex with an attribute of bus, to another vertex with an attribute of car. The unique vertex attribute pairs are used as the first-level index.

For each vertex attribute pair, we then build a spatial index, which maps a (θ, d) pair to a set of vertex pairs i.e., the object ID pairs, as illustrated in Figure 5.4b and Figure 5.4c. For example, the mapping between (θ_1, d_1) and vertex pair, *(A,B)*, denotes that there is one edge starting from vertex (i.e., object ID) A to vertex B, with edge attribute values (θ_1, d_1). The benefits of this spatial index are two-fold: (1) grouping edges that share the same spatial relationships eliminates duplicate values and thus reduces the storage overhead; (2) as per Definition 12, all potential matches must be considered in computing the matching score. Separating the spatial attributes (d and θ) from the vertices (u and v) allows us to retrieve all vertex pairs that match a given edge with the specified spatial attributes.

Finally, for each vertex pair, the temporal information (i.e., the temporal relationships of vertices between frames) is captured by a set of frames. The set of frames can be stored in intervals to further reduce storage overhead, as illustrated by Figure 5.4d. By grouping frames that share the same vertex attributes and edge attributes, we can obtain a set of frames, where the elements in the set are frame identifiers denoting frames in the video $\mathcal{V}$. Since the same vertices and their spatial relationships often span multiple frames, this index provides a compact way of storing such information.

To summarize, the Graph Index has two index levels. The first-level index provides efficient lookup on vertex attributes while reducing duplicate values on edges. The second-level index allows the retrieval of all vertex pairs that satisfy given edge attributes (i.e., the spatial relationships between two vertices) , where each vertex pair is associated with a list of frames, indicating where the corresponding edge was generated from the original video. Both index levels are implemented as hash indexes to enable fast retrieval.

5.4.4 Complexity Analysis

We use m to denote the maximum number of objects per frame. For each frame, we need to store one edge for each unique object pair. Thus, the total number of edges in the graph is

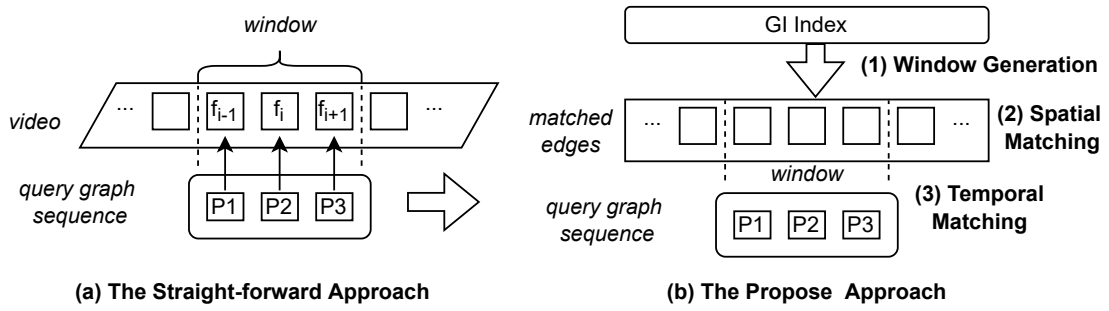


Figure 5.5: Comparison between Two Different Approaches

$m * (m-1)/2$, with the complexity of $O(m^2)$. For a video with p frames, the total number of edges is $O(pm^2)$. With the proposed index, we store all edges in a more compact manner while the total number of edges remains the same.

5.5 Query Processing

5.5.1 Overview

A straightforward way to commence processing is to proceed in two steps, as shown in Figure 5.5(a): we first generate sliding windows from the video $\mathcal{V}$, and then align each window with the query graph sequence to compute the window's matching score (i.e., the query graph sequence matching score). After all windows are processed, the scores are ranked and the final results are produced. However, this approach would be very inefficient because: (1) the number of windows could be very large (query processing is linear to the size of the repository) and most windows may not contribute to the final top k results; (2) even within one window, there could be many partial matches for the given query graph sequence. Thus, identifying efficient pruning techniques to eliminate unrelated results utilizing information provided in the query graph sequence is very crucial.

We can optimize the query processing from three aspects to further speed up the evaluation: (1) reducing the number of windows to be examined, (2) pruning partial matches that do not contribute to the final matching score early in each window, and (3) stopping early once the highest matching score in a given window is found. Based on the above ideas, we divide the pipeline into three steps, shown in Figure 5.5b: window generation,

spatial matching, and temporal matching. Window generation (Section 5.5.2) retrieves only matched edges for edges in the given query graph sequence and generates a set of windows based on the retrieved edges for further processing. Spatial matching (Section 5.5.3) produces matching data graphs from each frame, while temporal matching (Section 5.5.4) evaluates the matching score in each window iteratively according to the spatial matching results. Both steps utilize pruning strategies to reduce the evaluation cost.

5.5.2 Window Generation

5.5.2.1 Edge Retrieval

For a given edge $e = (u, v, \theta, d)$ from a query graph, $P \in \mathcal{P}$, where $\mathcal{P}$ is a query graph sequence. Let $L_{\mathcal{P}}$ be its labeling function. We call an edge $e' = (u', v', \theta', d')$ generated from a frame in the video a *matched edge*, if their vertex attributes and edge attributes both match, i.e., $L_{\mathcal{P}}(u) = L_{\mathcal{P}}(u')$, $L_{\mathcal{P}}(v) = L_{\mathcal{P}}(v')$, $\theta = \theta'$ and $d = d'$. The objective of the edge retrieval step is to produce the matched edges for all edges in $\mathcal{P}$ from the video $\mathcal{V}$.

Utilizing the GI, edge retrieval is conducted for each query graph P in the query graph sequence $\mathcal{P}$. Specifically, for each edge $e = (u, v, \theta, d), e \in P$, we fist obtain the vertex attributes pair, ($L_{\mathcal{P}}(u)$ and $L_{\mathcal{P}}(v)$). Utilizing the vertex attribute pairs and the spatial index on (θ, d), we obtain a set of tuples from GI, denoted by $\mathcal{R}_e$, where each element $r \in \mathcal{R}_e$ is a tuple, $r = (e_r, F_r)$, such that e_r is a matched edge, and F_r is the set of frames where the matched edge can be found. We further group together the matched edges for each frame, denoted by S_{f_e}, where f is the frame, and e is the given edge from the query graph. If a frame f cannot produce any valid data graph, namely there is no matched edge (i.e., $S_{f_e} = \emptyset$) for any edge $e \in E_P$, it is pruned from further processing.

Once we have retrieved the edges for all query graphs $P \in \mathcal{P}$, we group the matched edges by frames, which will then be used to generate windows (Section 5.5.2.2) for further processing.

5.5.2.2 Window Generation

Let F be the set of all frames where at least one matched edge is retrieved, where $F \subseteq \mathcal{V}$. Let l be the length of the query graph sequence $\mathcal{P}$, i.e., $l = |\mathcal{P}|$. For any frame $f_i \in F$, we use $\mathcal{W}_{f_i}$ to denote the set of windows of size l it could be enclosed in. Then we have $\mathcal{W}_{f_i} = \cup_{j \in [i-l+1,i]}\{W_j\}$, where W_i represents the window starting from frame f_i. We use $\mathcal{W}$ to denote the set of all windows that are generated from the retrieved frames, then $\mathcal{W} = \cup_{f_i \in F}\mathcal{W}_{f_i}$.

5.5.3 Spatial Matching

For any given frame, f, in the window, we can identify the corresponding query graph P it aligns with, after aligning each window with the query graph sequence (illustrated in Figure 5.5(b)). Spatial matching operates on aligned pairs of P and f. We use S_f to denote the set of matched edges for all edges in the query graph P, i.e., $S_f = \cup_{\forall e \in E_P} S_{f_e}$, where S_{f_e} is the set of matched edges for edge e in frame f (retrieved using the procedure described in Section 5.5.2). The objective of the spatial matching step is to produce a set of data graphs S_G, given S_f, where each $G \in S_G$ matches P.

5.5.3.1 A First Approach

A possible solution is to perform query graph matching utilizing graph traversal algorithms. This would involve two main steps. (1) Enumerate all data graphs that could be generated from S_f, denoted as *candidate data graphs*. Specifically, we form a data graph G by selecting one matched edge e' from the matched edges S_{f_e} for each edge $e \in P$. (2) For each candidate data graph G, we run Depth-First Search or Breadth-First Search on G and the query graph P in parallel, starting from the vertex with the same attributes on both graphs. G is considered as a match of P if we can traverse both graphs with attributes on all vertices and edges matched. However, this approach could lead to high computational cost, since the number of data graphs generated in step (1) could be very large, especially when there are multiple matched edges for each edge in the query graph.

5.5.3.2 Main Idea of Our Approach

Observe that for each Minimum Query Graph (Section 5.4.1), there is only one anchor vertex. This indicates that a candidate data graph G is a match of the query graph P, only if (1) all edges in G share the same anchor vertex, and (2) all the vertices are unique. Thus, generating the matching data graphs can be done in two steps: (1) *grouping*: group the matched edges according to the anchor vertex; (2) *enumeration*: enumerate all possible data graphs in each group. However, when the number of matched edges is large, the *enumeration* step could be costly. We therefore defer the enumeration of precise data graphs and only generate *Intermediate Data Graphs* to group matched edges with the same anchor vertex together.

An *Intermediate Data Graph*, I, has the following characteristics: (1) Each vertex $S_v \in V_I$ in I is a set, where each element, $v \in S_v$, in the set is a vertex in a data graph. (2) A data graph can be constructed by taking one element from every vertex of the intermediate data graph.

We could operate on such intermediate data graphs directly to facilitate temporal matching (Section 5.5.4) without generating all possible data graphs. The spatial matching step focuses only on producing intermediate data graphs by merging matched edges with the same anchor vertex (Section 5.5.3.3).

5.5.3.3 The SMA Algorithm

We propose the Spatial Matching Algorithm (SMA), depicted as Algorithm 8, to generate Intermediate Data Graphs. We use lists to represent intermediate data graphs by imposing a particular order of the graphs to further reduce the memory footprint. Specifically, we store only the set of vertices as an element in the list; the edge attributes (θ and d values) are omitted since they are identical to the edge attributes in the query graph. The vertex mapping between the query graph and the intermediate data graphs is captured implicitly by visiting edges from the query graph in order. To better illustrate the whole procedure, we explain the algorithm with the following example.

Example 6. *Figure 5.6 demonstrates the spatial matching between a query graph (Figure*

Algorithm 8: Spatial Matching Algorithm (SMA)

Input: the query graph P, the current frame f;
Output: a set of Intermediate Data Graphs

```
1  Q_E ← the ordered edge list constructed from P;
2  res ← dict();
3  foreach idx, e in Q_E do
4      // idx is the current position of the ordered edge list, and e is the edge;
5      S_fe ← the set of matched edges retrieved for the edge e;
6      foreach s in S_fe do
7          let v_s, v_e be the vertices in s, and v_s is the anchor vertex;
8          I ← res.get(v_s);
9          if I == null then
10             I ← new IntermediateDataGraph(v_s);
11             res.put(v_s, I);
12         end
13         I[idx + 1] ← I[idx + 1] ∪ {v_e};
14     end
15 end
16 r ← res.values();
17 foreach I ∈ r do
18     remove I if any element in I is empty;
19 end
20 return r;
```

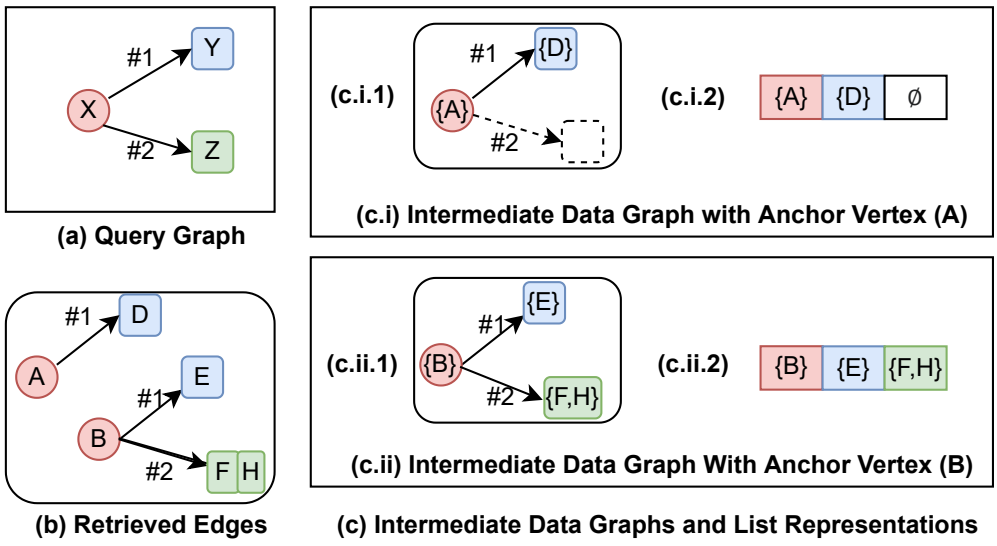


Figure 5.6: An Example for Spatial Matching

8a) and a frame (Figure 8b). We use X, Y, Z to denote vertices in the query graph and A-H to denote vertices in the frame from videos. Different shapes (circles and squares) are used to represent different object types (e.g., cars and buses). For simplicity, we show only matched edges in Figure 5.6b, and use different edge IDs to refer to edges with different attributes. We demonstrate the procedure in three steps: Initialization, Edge Matching, and Intermediate Data Graph Update, with line numbers referring to those in Algorithm 8.

(1) Initialization (Lines 1-2). The first step is to extract an edge list for each query graph (Line 1). Edge matching is then performed for each edge from the edge list in order. Assume that we store the two edges #1 and #2 in that order. We also create a hash map, which maps an anchor vertex to an intermediate data graph (represented by a list of vertex sets), to store the result (Line 2).

(2) Edge matching (Lines 3-6). For each edge in Q_E (Line 3), we retrieve the matched edges, as per Section 5.5.2 (Line 5), shown in Figure 5.6b. For simplicity, we use a vertex pair (v_s, v_e) to represent each edge, where v_s is the anchor vertex and v_e is the other vertex. We start with edge #1(X, Y). There are two matched edges, (A,D) and (B,E), that can be retrieved from the GI index (Figure 5.6b).

(3) Intermediate data graph update (Lines 7-13). Assume we start with the matched edge (A,D). We first extract the two vertices (Line 7): v_s=A and v_e=D, where v_s is the anchor vertex. Since there are no intermediate data graphs yet, we initialize a new one. The new intermediate data graph is stored in a list, where the first element is a set, containing the anchor vertex A (Lines 9-11). We place vertex D as an element in the set at position 1 in the list (Line 13).

Similarly, we also create another list representing the intermediate data graph with anchor vertex, B. Subsequently, we continue with matching edges for edge #2(X,Z). In this case, there are two matched edges, (B,F) and (B,H), that share the same anchor vertex. We place both vertices, F and H, in one set, and store them in the 2nd position of the list, as shown in Figure 8c.ii.2. The corresponding intermediate data graph is visualized in Figure 5.6c.ii.1.

Note that the intermediate data graph with anchor vertex, A (Figure 5.6c.i.1), does not contain any valid data graph due to missing matched edges for edge #2(X,Z) in the query graph. Thus it can be pruned immediately (Lines 14-15).

5.5.4 Temporal Matching

The main objective of temporal matching is to produce the matching score for a given window W based on the set of intermediate data graphs obtained for each frame f (Section 5.5.2). For a query graph sequence $\mathcal{P}$, the same vertex could appear in multiple query graphs. As per Definition 11, to identify a data graph sequence that is a complete match of $\mathcal{P}$, one must ensure that the same vertex v across different data graphs is matched to the same vertex v' across the corresponding query graphs. A data graph sequence $\mathcal{G}_V$ that matches the query graph sequence $\mathcal{P}$ is identified when we can establish a one-to-one mapping between the vertex sets of $\mathcal{G}_V$ and $\mathcal{P}$, i.e., (1) a matching vertex v' is determined for each vertex $v \in V_{\mathcal{P}}$; (2) all matching vertices are unique (i.e., for any $u, v \in V_{\mathcal{P}}$, $u' \neq v'$, where u', v' are the matching vertex for u and v, respectively). As such, temporal matching can be implemented as an iterative procedure, and each iteration focuses on matching one particular vertex $v \in V_{\mathcal{P}}$. We use *step* to denote such an iteration and present the algorithm to perform such temporal matching steps iteratively for each window. We first introduce the algorithm for the temporal matching step based on the above idea in Section 5.5.4.1, and then discuss how windows can be prioritized in Section 5.5.4.3.

5.5.4.1 Temporal Matching Steps

For a given window, initially, no matching vertices are identified for any vertex in $V_{\mathcal{P}}$. We start by finding matching vertices for a vertex $v \in V_{\mathcal{P}}$ utilizing all intermediate data graphs (such a vertex can be chosen randomly or by imposing a topological order of the vertices in $V_{\mathcal{P}}$). Recall that each intermediate data graph contains a list of vertex sets. The set of all matching vertices for v, denoted by V', can thus be easily identified from such lists. For a matching vertex $v' \in V'$, we mark the intermediate data graphs in which v' has been identified as *active*. After the first step, we have identified a set of all matching vertices, V', for v, and for each $v' \in V'$, a set of active intermediate data graphs are determined, which can be used in subsequent processing.

Without loss of generality, in each following step, we assume that a set of matching vertices have been identified, denoted by M_V. We use M_I to denote the set of active inter-

mediate data graphs resulting from the previous steps. To store the contextual information between steps, we introduce a structure called *Match Candidate*, M, consisting of M_I and M_V. As an initial step, we create one Match Candidate per window, W, where $M_V = \emptyset$, and M_I contains all intermediate data graphs, as discussed above.

Algorithm 9: A Step in Temporal Matching (TM)

```
   Input: the vertex to match v, Match Candidate Set S_M, the current window
          matching score score
 1 L_P ← the ordered list of unique vertices in V_P;
 2 M ← getOneMatchCandidate(S_M);
 3 let M_V be the set of matched vertices from previous steps in M; let M_I be the set of
   intermediate data graphs in M;
 4 v ← nextVertexToMatch(M_V, L_P);
 5 V' ← getMatchingVerticesFor(M_I, v);
 6 foreach v' in V' do
 7 |   if not M_V.contains(v') then
 8 |   |   M'_V ← add(M_V, v');
 9 |   |   M'_I ← getMatchingIntermediateDataGraphs(M_I,v');
10 |   |   if |M'_V| == |V_P| then
11 |   |   |   // means we have found one matched data graph sequence;
12 |   |   |   M'_F ← getRelevantFrames(M'_I);
13 |   |   |   score ← max(|M'_F|, score);
14 |   |   end
15 |   |   else
16 |   |   |   M' ← MatchCandidate(M'_V, M'_I);
17 |   |   |   S_M ← S_M ∪ {M'};
18 |   |   end
19 |   end
20 end
```

Algorithm 9 depicts one step in the iteration. We use S_M to denote the set of Match Candidates in the current window. Initially, S_M contains only one Match Candidate initialized from the current window W, as discussed in the initial step. Let $L_\mathcal{P}$ be the list of vertices constructed by imposing a total order on $V_\mathcal{P}$ (Line 1). The order of $L_\mathcal{P}$ can be obtained by sorting the vertices in $V_\mathcal{P}$ according to the number of occurrences in the query graph sequence. At each step, we pick an arbitrary Match Candidate (Line 2) (an optimized way of choosing the next Match Candidate will be discussed in Section 5.5.4.3). We use two sets to store sufficient information, where M_V consists of vertices, and M_I contains the references

to the active intermediate data graphs from the last step (Line 3). We first derive the next vertex $v \in V_{\mathcal{P}}$ to match based on the size of M_V and $L_{\mathcal{P}}$ by selecting the next vertex in the L_P based on the current number of matched vertices in M_V (Line 4). Then we retrieve matching vertices from the intermediate data graphs, M_I (Line 5); each matching vertices, v', is processed only if it is not matched to some vertex in $V_{\mathcal{P}}$ already (Lines 6-7). A data graph sequence that matches to the query graph sequence is identified if a matching vertex for each vertex $v \in V_{\mathcal{P}}$ is found (Line 10-14). In this case, we first obtain the set of relevant frames, M'_F, where at least one active intermediate data graph is generated from each frame in M'_F (Line 12), and then update the matching score for the current window based on the size of M'_F (Line 13). Otherwise, new Match Candidates are produced for further processing (Lines 16-17). If no matching vertex for v can be found (i.e., $V' = \emptyset$, in Line 5), then the processing of the current Matching Candidate terminates directly. We explain this algorithm in detail with an example.

EXAMPLE 7. *Figure 5.7 presents an example on a window of three frames. The query graph sequence is shown in Figure 5.7a, and the frames are shown in Figure 5.7b. There are two intermediate data graphs generated in $f1$ ($I1$, $I2$), and one in $f2$ ($I3$) and two in $f3$ ($I4$, $I5$). Take the intermediate data graph, $I3$, as an example, each vertex in $I3$ is a set of matching vertices for a vertex in $P2$. E.g., the vertex set $\{B\}$ in $I3$ stores all matching vertices of vertex X in $P2$, while the vertex set $\{F, H\}$ stores all matching vertices of vertex Z.*

Initially, the matching score of the current window is set to 0. We initialize one Match Candidate for the window, shown in Figure 5.7(c.i.1) (Line 1). The set of matched vertices is empty, denoted by $\emptyset$. All intermediate data graphs are considered as active, i.e., $M_I = \{I1, I2, I3, I4, I5\}$. For better readability, we also provide the relevant set of frames in the figure, i.e., $M_F = \{f1, f2, f3\}$.

At Step 1 (the arrow annotated with S1 in Figure 5.7c), we try to identify matching vertices for the vertex X in the query graph sequence. All matching vertices from the intermediate data graphs are highlighted in red. In Algorithm 9 , we retrieve the matching vertices from the intermediate data graphs (Line 5). Since we use lists to store the intermediate data graphs, this can be easily done by selecting the vertex set at the corresponding

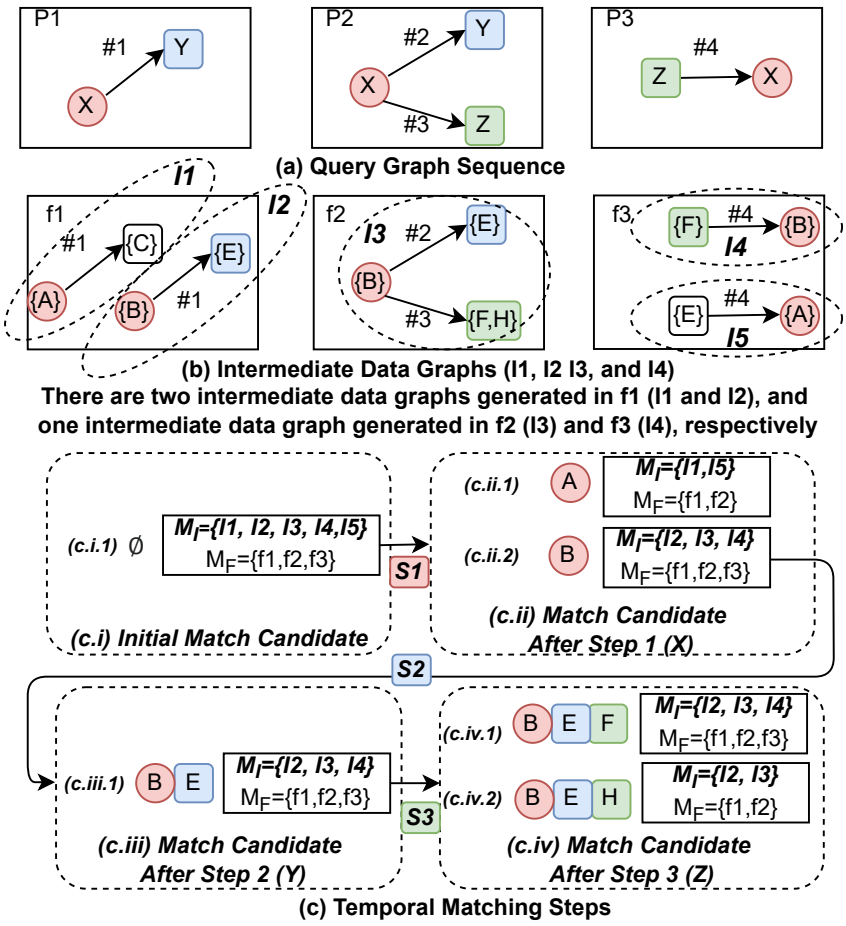


Figure 5.7: Temporal Matching Steps

position in the list. There are two matching vertices, A and B, in this case. We take vertex A as an example, shown in Figure 5.7c.ii.1. We first add vertex A to the matched vertices, M_V (Lines 7-8). Vertex A appeared in $I1$ from frame $f1$, and in $I5$ from frame $f2$; we update M'_I=$\{I1, I5\}$ (Line 9). Since there is only one matched vertex in M'_V (Line 11), we produce a new Match Candidate (Line 15) and add it to the Match Candidate Set for further processing (Line 16). The matched vertex B is also processed in a similar manner (Line 6); the result is shown in Figure 5.7c.ii.2. Similarly, we can continue the procedure by selecting the next vertex to match for Steps 2 and 3, with results shown in Figure 5.7(c.iii, c.iv).

5.5.4.2 Prioritizing Match Candidates

There could be many Match Candidates in each window. At each step, we wish to always select the most promising Match Candidate to explore. Since we maintain a set of frames for each Match Candidate, intuitively, we can use such a set to estimate the maximum matching score that could be produced by each Match Candidate. We introduce the estimated score of each Match Candidate, M, computed as $|M_F|$, where M_F is the set of relevant frames (i.e., frames for which at least one intermediate data graph is marked as active) of M. At each step, we pick the Match Candidate with the highest estimated score.

The procedure is outlined in Algorithm 10. We use Q_M to represent the priority queue of Match Candidates in the current window (Line 1). We always select the Match Candidate with the highest estimated score at each step of the iteration (Line 3). Algorithm 9 also updates the matching score of the current window W, as well as adds new Match Candidates to the priority queue. The procedure stops if there is no Match Candidate that has an estimated score higher than the score of the current window (Line 2).

Algorithm 10: Prioritizing Match Candidates

```
1 Q_M ← the priority queue containing all Match Candidates in W;
2 while W.score ¡ Q_M.peek().estimate_score do
3 |   OneStepTemporalMatching(Q_M);// this also updates Q_M if new Match
  |     Candidates are produced
4 end
```

Example 8. *Based on the above example, Example 7, we can obtain the estimated score for*

each Match Candidate. For example, in Figure 5.7c.ii.1, the estimated score for the Match Candidate with vertex A is 2, while the estimated score for the one with vertex B is 3. If we prioritize Match Candidates with higher scores, then we select the Match Candidate with vertex B to explore first. Similarly, after step 2 in Figure 5.7c.iii, the Match Candidate with vertices (B,E) has the highest score of 3, so we continue the procedure. In step 3 (Figure 5.7c.iv), we have obtained the highest score of 3, which is produced by the Match Candidate with vertices (B,E,F). Meanwhile, we have another unexplored Match Candidate with vertex A (produced in step 1 in Figure 5.7c.ii.1), whose estimated score is 2. Since 2 < 3, it is guaranteed that no other Match Candidates could produce a matching score higher than 3, and thus the algorithm stops.

5.5.4.3 Prioritizing Windows

There could be many window candidates for a video; we adopt the same strategy and prioritize windows based on the estimated scores. Let S_M be the set of Match Candidates in a given window W, and the estimated score of window W is computed as $\max_{M \in S_M} |M_F|$. Similar to Algorithm 10, we maintain a priority queue to store all generated windows. Windows are processed iteratively according to their estimated scores. During the procedure, we also keep track of the final ranked results and their scores. The procedure stops once the highest estimated score produced by the windows remaining in the priority queue is less than or equal to the current minimum matching score in the ranked results. The algorithm follows the same structure as Algorithm 10 and is omitted here for brevity.

5.5.5 Complexity Analysis

In the worst case, the edge retrieval procedure retrieves all edges from the index, and all candidate windows are examined during spatial matching and temporal matching. For simplicity, we assume all objects occur in each graph in the query graph sequence. We use n to denote the number of unique objects in the query and use m to represent the maximum number of objects in each window. To match the query graph, there are maximum $m * (m-1) * \cdots * (m-n+1)$ possible combinations. Let w be the length of the query

pattern sequence (i.e., the window size), and p be the total number of frames in the video. In the worst case, the computational complexity is $O(m^n wp)$. In practice, the number of examined windows can be reduced thanks to the early stopping mechanism introduced in Section 5.5.4.3.

5.6 Experiments

5.6.1 Settings

Environment. Our experiments are conducted on a machine with an Intel i5-9600K CPU, a GeForce GTX 1070 GPU and 48GB RAM. All algorithms are implemented in Python [11] except the object detection/tracking algorithms, for which we use open-source implementations [28, 32]. Specifically, we use Faster RCNN [125] as the object detection algorithm and Tracktor [18] as the tracking algorithm. Similar to the experiments in Chapter 3, object detection and tracking algorithms can be replaced with any other available algorithms. Different object detection and tracking algorithms may produce different numbers of objects in the annotation result, which is equivalent to varying the number of objects in the video annotations. For simplicity, we use the same object detection algorithm in the experiment, while the number of objects is considered as the characteristic of the videos and can be studied by using different videos. Data are pre-loaded to memory before evaluation.

Datasets. Experiments are conducted on two real-world video datasets: De-trac [148] and Deep Drive [166]. Both datasets consist of short videos (image sequences). For the De-trac dataset, we take videos from both the training set and the test set, and concatenate them into two long videos, *drtrain* and *drtest*, respectively. For the Deep Drive dataset, we take videos from the test set and split them into two groups to form two separate long videos (*bdd100kA* and *bdd100kB*). We apply object detection and tracking algorithms to obtain structured data on the De-trac dataset (drtest and drtrain), while using annotated data (ground-truth) for the Deep Drive dataset (bdd100kA and bdd100kB).

Data statistics are provided in Table 5.1, including the total number of frames (# frames), the average number of objects per frame (# avg obj/f), the total number of objects (#

[11] Code available at `https://github.com/yestinchen/star_retrieval`

Table 5.1: Database Statistics

video	drtest	drtrain	bdd100kA	bdd100kB
# frames	56.30k	83.73k	138.25k	138.78k
# avg obj/f	24.64	17.32	9.51	11.33
# objects	37.48k	32.86k	53.21k	59.61k
avg duration	38.41	45.71	25.48	27.85

objects), and the average duration of each object (avg duration). In general, there are more objects per frame in De-trac, and objects are changing more frequently in Deep Drive (i.e., the lower average duration of each object).

Query Processing Methods. We evaluate the query processing performance of three methods, namely, *base*, *prop* and *prop_s*. All three methods are based on the proposed index. The *base* method serves as the baseline method, which simply accepts the retrieved edges and performs graph matching utilizing Depth-First Search, as discussed in Section 5.5.3.1. The temporal matching step of the *base* method then simply enumerates all possible mappings to the vertices in the query graph pattern and subsequently computes the window scores. The *prop_s* method is a sequential version, where all windows are processed sequentially without prioritization across windows, while the *prop* method is the proposed method with prioritizing windows enabled (Section 5.5.4.3).

Evaluation Methodology. Object detection/tracking algorithms are applied as part of preprocessing. As such algorithms are not applied during query execution and we do not account for the time required by such operations in our evaluation. The preprocessing time could vary depending on the object detection and tracking algorithms used. For example, in our setting, with high accuracy algorithms, i.e., Faster R-CNN [48] as the object detection algorithm and Tracktor [18] as the object tracking algorithm, we could achieve a speed of 4 frames per second during the preprocessing step on average. The speed could be further improved if faster models are applied (e.g., YOLO [124], DeepSORT [152] etc.).

We report the query processing time based on a set of query graph sequences randomly generated from the raw videos. Each query graph sequence is generated as follows:

1. Determine the length p_d of the query graph sequence and the number of objects p_o in the query graph sequence.

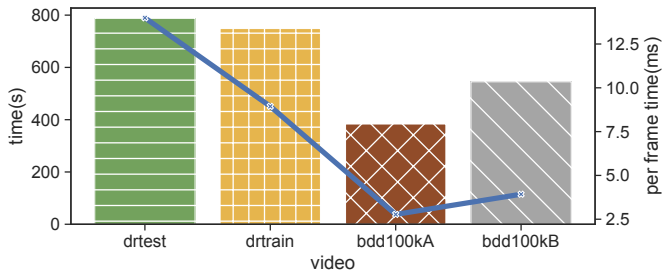


Figure 5.8: Index Construction Time on Different Videos

2. Select p_n video clips randomly from the original dataset.

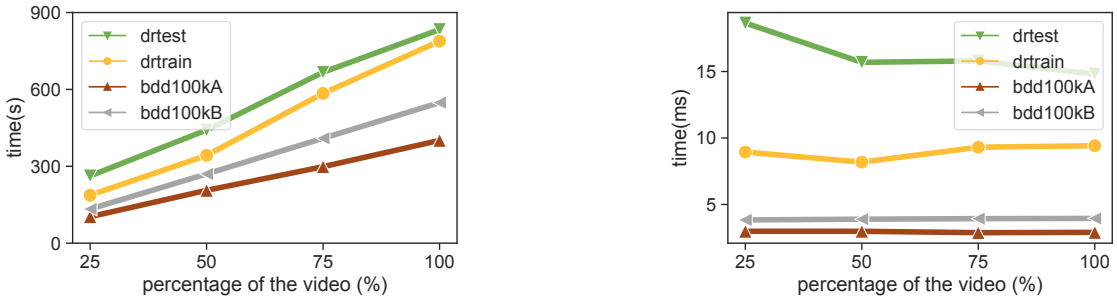
3. From each short video clip, we randomly select p_o number of objects from p_d consecutive frames to generate one query graph sequence.

In our experiments, we set p_n to a fixed value, 20, which is sufficient to represent the distribution of query graph sequences. As can be validated by the experiment figures, query graph sequences generated in this way contain queries with different selectivities and their running time is representative of the general trend. Increasing p_n would show similar trends in the experiment results. We vary p_d and p_o in the following experiments and study the performance of different methods. Increasing either p_d or p_o also increases the complexity of queries either on the temporal dimension (p_d) or on the spatial dimension (p_o).

Discretization granularity. To study the impact of discretization granularity, we vary it and present both the index construction time and query time. We use a tuple (s_θ, s_d) to represent the number of buckets assigned for θ and d values. In the experiments, we use 4 different discretization granularities: $df1$=(4, 10), $df2$=(8, 10), $df3$=(8, 15), $df4$=(12, 15). Unless otherwise specified, $df2$ is the default discretization granularity used in the experiments.

5.6.2 Index Construction

Varying datasets. Figure 5.8 presents the time utilized for index construction on different datasets. The primary y-axis is the time consumed to build the index on the entire video, which is shown as the bar chart, while the secondary y-axis is the amortized time on each frame in the video, shown as the line chart. In general, the index construction time is highly



(a) Total Time

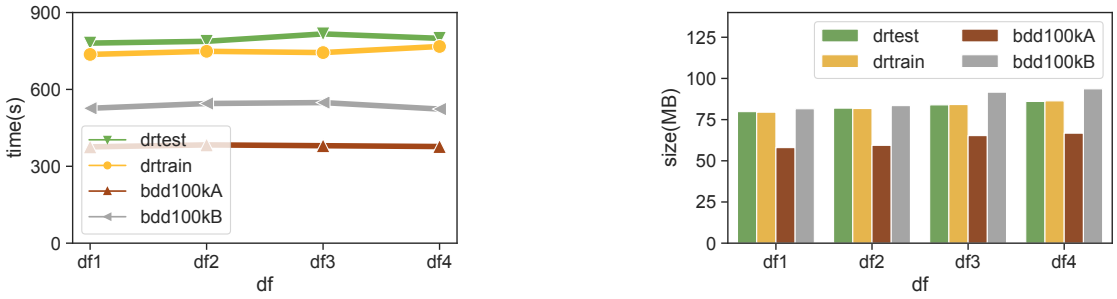
(b) Amortized Time per Frame

Figure 5.9: Index Construction Time w.r.t. Number of Frames

related to the number of objects per frame. As more objects appear in each frame, more edges will be generated and materialized, thus leading to a higher construction time. For example, even though drtest and drtrain have fewer frames compared with bdd100kA and bdd100kB, both the total index build time and the amortized time per frame are still higher than the other two videos. Furthermore, drtrain has the most number of objects per frame and it also has the highest build time among all videos. As a contrast, bdd100kA has the least number of objects per frame and the lowest build time. This confirms that more objects per frame leads to higher index construction cost.

Varying the number of frames. We further report the index construction time for different numbers of frames in each video, shown in Figure 5.9. We select 4 checkpoints starting from 25% frames to 100% frames starting from the beginning of each video. The total build time grows linearly as more frames are processed. As shown in Figure 5.9b, the amortized time per frame remains stable for both bdd100kA and bdd100kB. The reason is we process one frame at a time during index construction, and there are no inter-frame computations. Thus the overall index construction time is the sum of the construction time consumed by each frame. As a result index construction is highly scalable. We also observe an increasing or decreasing trend in time (drtest and drtrain) as the number of objects changes per frame, which also confirms that the number of objects per frame dominates the index construction cost.

Varying the discretization granularity. We vary the discretization granularity and the results are depicted in Figure 5.10a. Different discretization granularities are applied



(a) Index Construction Time

(b) Index Size

Figure 5.10: Index Construction w.r.t. Discretization Granularity

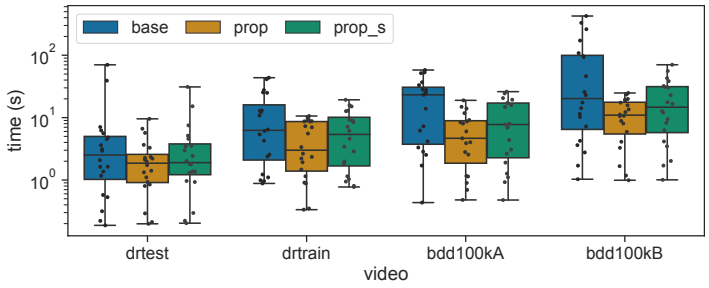


Figure 5.11: Query Time on Different Videos

when materializing edge attributes. From $df1$ to $df4$, the attributes on different edges are more likely to have diverse values. Theoretically, changing the discretization granularity will not impact the computational cost since neither the number of edges nor the number of frames changes. The result confirms that varying the discretization granularity does not have a significant impact on the index construction time. We have also reported the index size with different discretization granularities, shown in Figure 5.10b. Having more buckets for edge attributes is likely to increase the number of edges with unique values, thus leading to higher space consumption. The trend is more obvious on videos with more objects, such as bdd100kB. The figure also shows that even with more frames, a smaller number of objects per frame could still have a significant impact on the storage saving, as can be observed from bdd100kA, where it has the lowest space consumption among all videos.

5.6.3 Query Processing

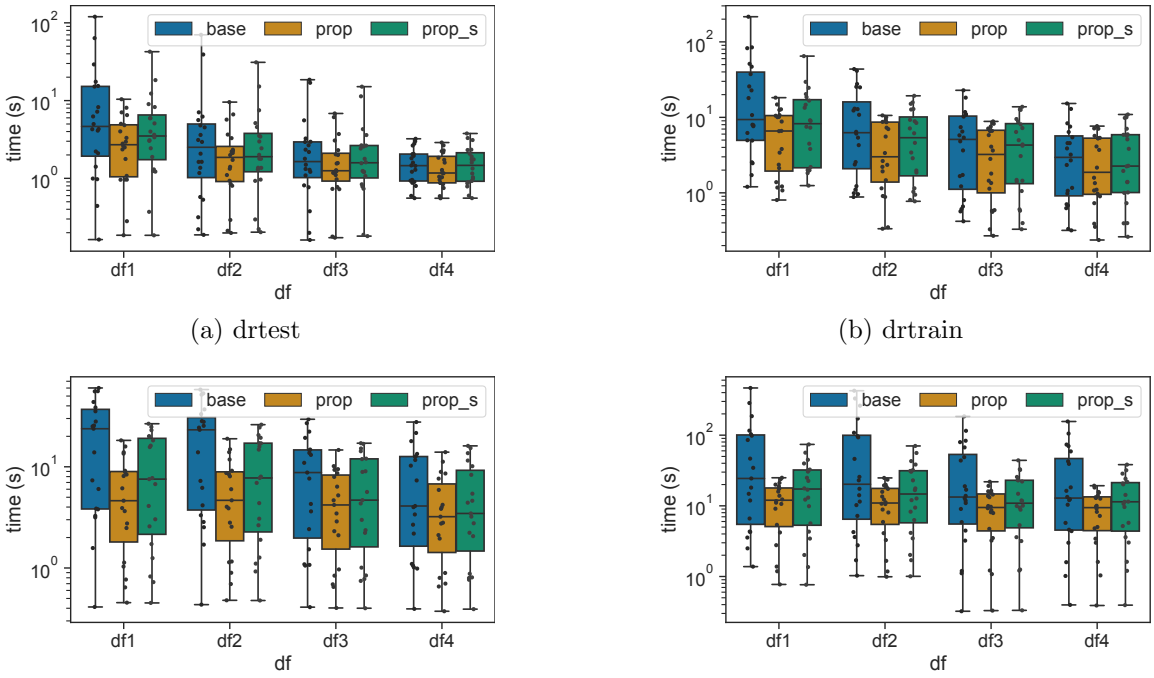
Query processing time under default settings. As the default setting, we set the query length (i.e., the length of the query graph sequences), $p_d = 10$, the number of objects

in each query, $p_n = 4$, and use $df2$ as the discretization granularity. By default, we only retrieve $k = 100$ results with the highest scores. We generate 20 queries from each video, issue these queries to the corresponding videos, and report the query processing time, as shown in Figure 5.11. Queries are generated following the rules presented in the Evaluation Methodology, in Section 5.6.1. The boxplot depicts the processing time distribution of 20 different queries, while the dots on the boxplot represent the actual data points, with each point representing the processing time obtained on one query instance.

Both *prop* and *prop_s* are significantly faster than the *base* method, which can be observed from the median values of the three methods. The spread of the boxplot for *base* is also wider than the other two methods: for the queries in the bottom quarter of the boxplot, the processing times of these three methods are very close, except for drtrain, where *base* is much slower; for queries in the top quarter, *prop* has the best performance among all these methods, while *base* is the worst. The result aligns with our intuition: queries having a lower processing time with the *base* method are likely to have fewer data graph sequences generated, i.e., there are less Match Candidates to prune for both *prop* and *prop_s*. Moreover, the benefit of introducing the early-stopping mechanism may not be sufficient to cover the additional maintenance overhead. For queries with a longer execution time, there are more Match Candidates to prune, and thus the benefit of the early-stopping mechanism is much more pronounced. The results indicate that our proposed method is effective, especially on queries with high selectivities. Even without prioritizing windows (i.e., *prop_s*), there is still a huge improvement over the *base* method. For the remainder of this section, we present the result based on two representative videos due to space limitation, but the trend is similar on other videos.

Varying the discretization granularity. We first vary the discretization granularity, as introduced in Section 5.6.1. From $df1$ to $df4$, as the number of buckets for either θ or d increases, the selectivity of a given edge decreases due to the increase in the value ranges for edge attributes. For example, the θ values of two edges that could fall into the same bucket with the discretization granularity applied may belong to two different buckets if the number of buckets increases, leading to a lower selectivity.

To evaluate the effect of discretization granularities, we build indexes with different gran-



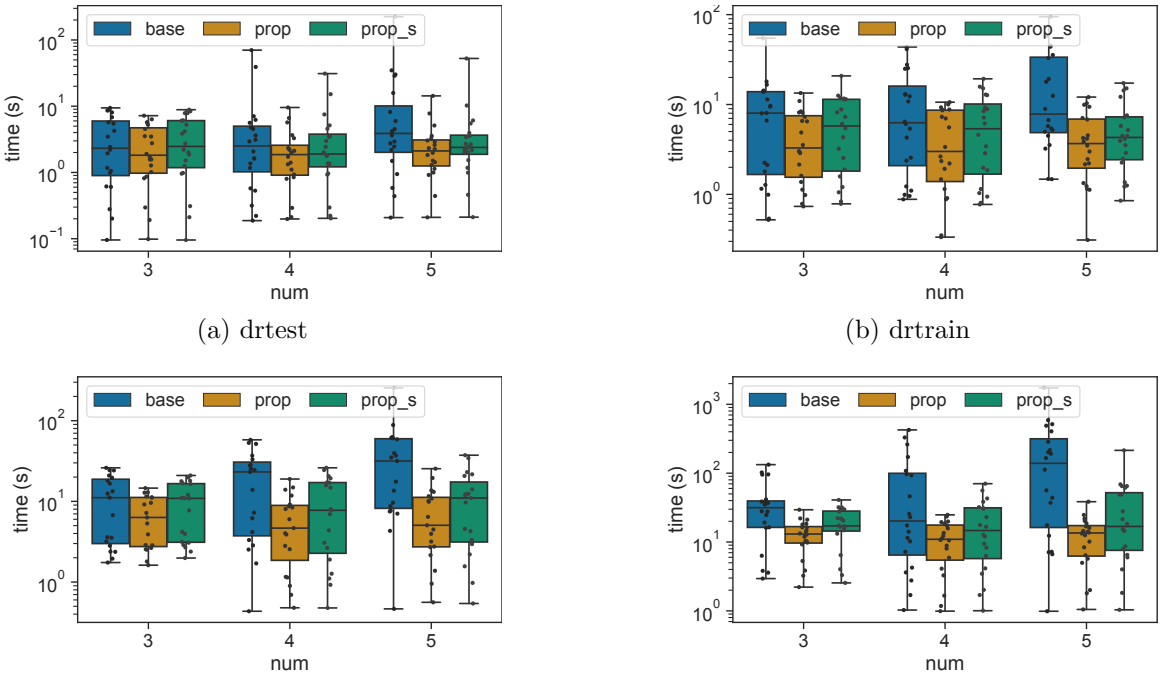
(a) drtest (b) drtrain

(c) bdd100kA (d) bdd100kB

Figure 5.12: Varying *df*

ularities and measure the query processing time on the same set of queries. The results are shown in Figure 5.12. As we increase the number of buckets for edge attributes, the selectivity of the same query decreases, thus leading to less processing time thanks to the index structure. This is because in the first step of all three methods, we retrieve only relevant edges from the index; therefore, more discretized values mean fewer vertex pairs are retrieved for each edge, and thus less time is needed. With the retrieved vertex pairs, the *prop* method still outperforms the other two methods, while *base* has the longest processing time overall. In general, as we increase the number of buckets for either θ or d, the query evaluation time for all methods decreases; the *prop* method still outperforms the other two methods, while *base* has the longest processing time overall.

Varying the number of unique vertices in the query. We vary the number of unique vertices in the query from 3 to 5. Specifically, we generate the query graph sequence based on 5 vertices (i.e., 5 different object IDs), and then reduce the number of vertices included in each query from 5 to 3. As we increase the number of unique vertices in the query, more edges could be retrieved in each frame, leading to higher computational cost



(a) drtest

(b) drtrain

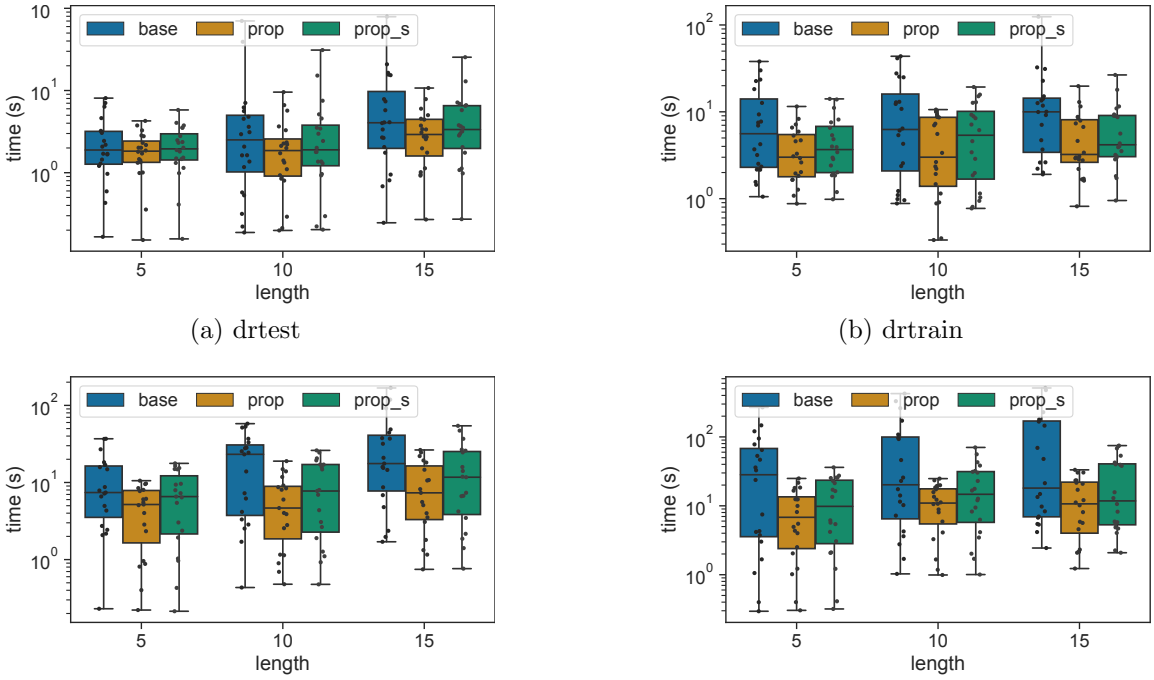
(c) bdd100kA

(d) bdd100kB

Figure 5.13: Varying # of Unique Vertices

for query processing. Consequently, both methods *prop* and *prop_s* have much lower query processing time compared with *base*. Moreover, for those expensive queries (queries with long processing time, shown in the top quarter of the boxplot), the query processing time of *base* grows exponentially while the time for the other two methods grows at a much lower rate thanks to the early-pruning mechanism.

Varying the length of query graph sequences. Similarly, we vary the length of query graph sequences from 5 to 15. We first generate queries based on the longest length (i.e., 15), and then reduce the query length from 15 to 5. The results are shown in Figure 5.14. As we increase the query length, the number of query graphs that each frame needs to match increases; thus more data graphs could be generated for each window, leading to higher computational cost. Meanwhile, increasing the query length may also reduce the number of Match Candidates since the query could be less selective due to the information from the extra frames. Consequently, we could either see an increasing mean query processing time (Figure 5.14a) or a decreasing query processing time (Figure 5.14d, *base* method). In general, we can still see an increasing trend on all methods as we increase the query length.



(a) drtest (b) drtrain

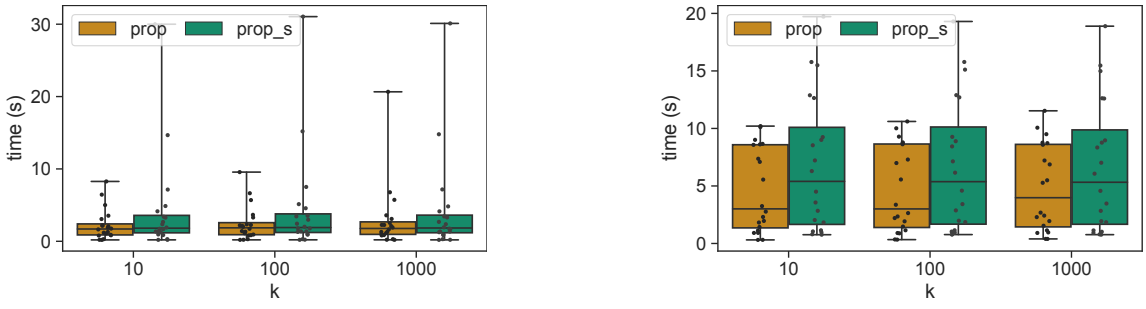
(c) bdd100kA (d) bdd100kB

Figure 5.14: Varying the Length of Query Graph Sequences

Among all three methods, *prop* performs the best, which is in agreement with our other experiments, demonstrating the effectiveness of the proposed methods.

Varying k. We further vary the number of results retrieved for each query, k, from 1 to 1000. The result is shown in Figure 5.15, where *base* is omitted since its query time remains stable for all k values. As can be observed, increasing k leads to increasing query time for the *prop* method, thanks to the early stopping mechanism brought by prioritizing windows, while the query time for *prop_s* remains stable. This further demonstrates the effectiveness of the early stopping mechanism in the proposed method.

Varying the sample rate. Due to the characteristics of videos, sampling is commonly used to reduce the computation by reducing the number of frames. To study the impact of different sample rates, we vary the value from 10% to 100%, and generate different queries at different sample rates with the same query parameters (i.e., same p_d and p_o). For example, a sample rate of 10% means that we sample only 10% of the frames. The results are shown in Figure 5.16. As the sample rate decreases (from 100% to 10%), the index is constructed on fewer frames, and the spatial relationships between objects are likely to change more



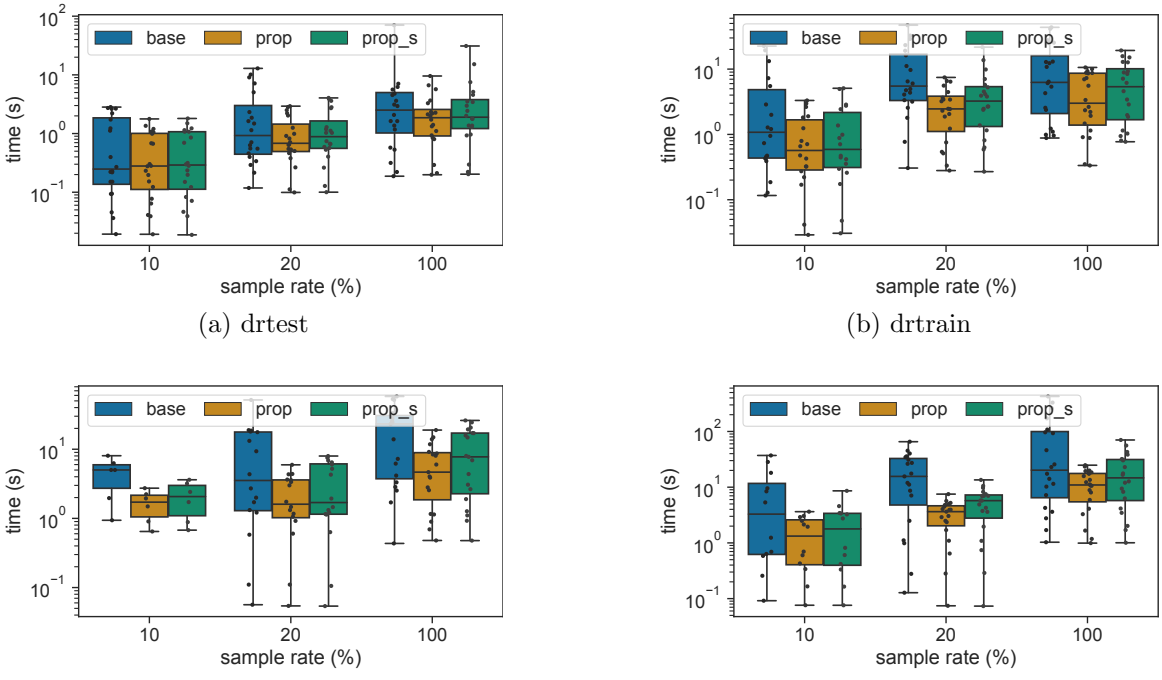
(a) drtest

(b) drtrain

(c) bdd100kA

(d) bdd100kB

Figure 5.15: Varying k



(a) drtest

(b) drtrain

(c) bdd100kA

(d) bdd100kB

Figure 5.16: Varying Sample Rate

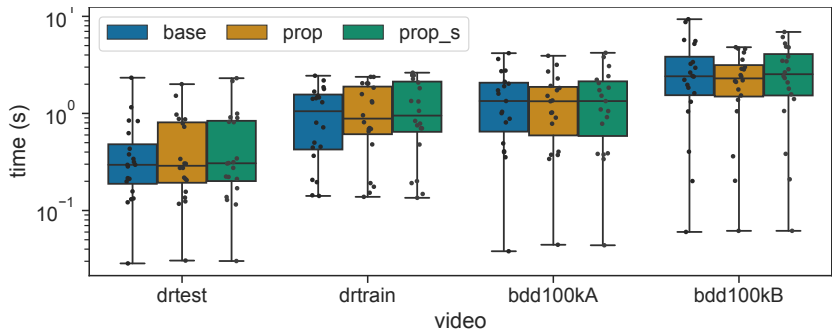


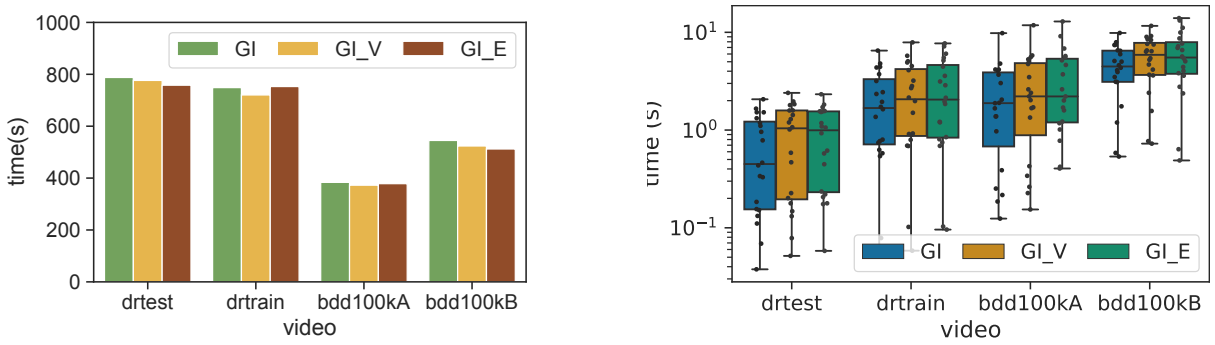
Figure 5.17: Query Time on Two Labels

frequently. As a result, adjacent frames are likely to have different values for the same edge. Consequently, the number of vertex pairs retrieved from the index could be reduced, and the number of Match Candidates during query processing is also reduced, leading to faster query processing. Among the three methods, *prop* is still the fastest, demonstrating the effectiveness of our proposal when sampling is enabled.

Query with multiple types of labels. Finally, we synthetically generate another type of label representing the color of objects. We randomly assign one out of three colors to each unique object, and then build indexes on two labels and evaluate the performance with the same set of queries as in the previous experiments. The results are shown in Figure 5.17. Compared with the results based on a single type of label (Figure 5.11), the query processing time for all methods decreases due to the lower selectivity of queries. Both *prop* and *prop_s* are still faster than *base* in general.

5.6.4 Ablation Study

Different index structures. We perform an ablation study to further understand the pruning power brought by the GI index. We use *GI* to denote our proposed method utilizing the full GI index, based on which we implement two variants: *GI_E*, which builds indexes based on the edge attributes only, and *GI_V*, which builds indexes based on the vertex attributes only. We measure the index construction time for all three methods, and the result is shown in Figure 5.18a. The overall construction time of either *GI_V* or *GI_E* is slightly less than that of *GI*, due to the simplified index structure. Building the index for *GI_V* can be faster than for *GI_E* on some videos (e.g., drtrain and dbb100kA), and vice



(a) Index Construction Time

(b) Edge Retrieval Time

Figure 5.18: Varying Index Types

versa on other videos (e.g., drtest and bdd100kB), depending on video characteristics.

For these three methods, we further evaluate the performance of query processing. Specifically, we only include the Edge Retrieval time, where the objective is to produce only related edges to the given query, and omit the processing time for the remaining steps (i.e., spatial matching in Section 5.5.3, etc.) as they are identical in all methods. The result is presented in Figure 5.18b. The *GI* method has the lowest edge retrieval time, while the other two methods are slower than the *GI* method. This demonstrates that having indexes on both edge and vertex attributes leads to more efficient query processing than using either vertex or edge attributes alone.

The effect of intermediate data graphs. To further study the impact of having or not having intermediate data graphs during query processing, we further implement another method *noimd*, which enumerates all possible matching graphs directly and uses them during temporal matching. The matching graphs are processed in a similar manner, where the matching candidates are prioritized and processed iteratively. The result is depicted in Figure 5.19. Since intermediate data graphs are mainly optimized for those cases where multiple vertices match the same vertex in the query graph, the benefit of intermediate data graphs only becomes significant on queries with high selectivities (e.g., drtrain, bdd100kB).

Summary. Results from the above experiments demonstrate the effectiveness of our proposed methods. There are two main steps in query execution: edge retrieval and Match Candidate pruning. The edge retrieval reduces the number of Match Candidates significantly if the number of vertex pairs retrieved from the index is small (i.e., with fewer objects in the

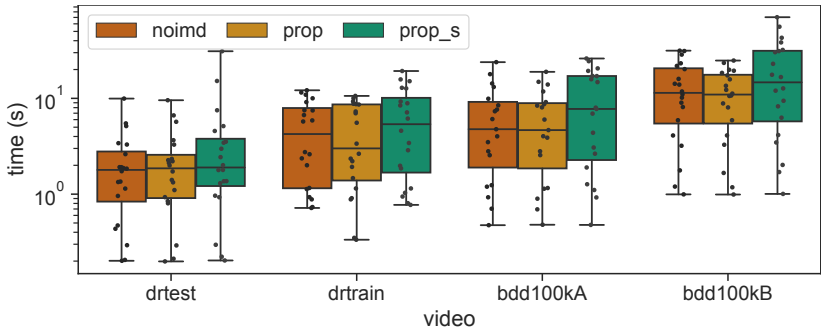


Figure 5.19: Query Time w./w.o. Intermediate Data Graphs

query, or fewer frames in the video). Even with many vertex pairs retrieved, our proposed methods can still answer queries efficiently by pruning Match Candidates early. The main saving comes from the spatial matching process (*prop_s*), while the algorithm to prioritize windows further provides a better early-stopping mechanism and achieves faster execution (*prop*).

5.7 Summary and Future Work

In this chapter, we define the problem of STAR Retrieval. Utilizing CV algorithms, we obtain graph representations for videos and propose to build and fully specify the GI index to accelerate query answering. We process queries based on the GI index, along with two proposed algorithms, Spatial Matching (SMA) and Temporal Matching (TM). Experiments based on real-world videos confirm the effectiveness and utility of the proposed algorithms. In this work, we have mainly considered the spatial relationships between objects, along with the labels on objects and temporal constraints. Other complex attributes such as events involving objects may be of interest for more complex pattern queries. Other directions, such as supporting variable window sizes for different playback speeds, and further improvements to the index structure with subgraphs, are also worth exploring in future work. In addition to fixed window size, more complex matching considering variable window sizes or different playback rates could also be considered.

6 Conclusion and Future Work

6.1 Summary of the Dissertation

In this dissertation, we studied several problems of evaluating temporal queries over videos, specifically considering temporal information. We start with a general framework, which utilizes state-of-the-art algorithms to obtain object detection and tracking results as video annotations. Queries can solely be evaluated on the annotations, which reflect the underlying video content. With such a framework, we investigated three types of queries. First, we focus on temporal queries involving objects and their co-occurrences in video feeds, where we proposed two algorithms to prune states early to accelerate downstream query evaluation. Then, for the same query, we aim to produce only the results with the highest ranks with a two-phased approach, which consists of an ingestion phase, to build and materialize indexes for query acceleration, and a query phase, which evaluates the given queries efficiently with the materialized index. Specifically, we take a partition-based approach to build indexes along with metadata for each partition and evaluate queries across multiple partitions. Finally, we further enrich the queries by taking both the spatial and temporal constraints in videos. We model the annotations in videos as graphs and propose to simplify the graph for faster query evaluation. In this problem, both videos and queries are modeled as a sequence of graphs. With the pre-materialized graph index, our framework consists of window generation, which is used to generate only windows that are promising; spatial matching, which is used to perform intra-frame level matching based on spatial information; and temporal matching, which is used to perform inter-frame level matching based on temporal matching.

6.2 Discussions and Future Work

Through the works presented in this dissertation, we have already made our first steps to answering queries over videos. With the general framework we adopted, we can always obtain unique object identifiers and associated properties of the objects from videos utilizing CV algorithms. We refer to such an approach as query evaluation on video annotations.

The main benefit of such an approach is that we can only focus on query efficiency based on the annotations directly with exact evaluation based on query semantics. Such queries can be useful for retrieving certain patterns from a small set of videos where the cost of obtaining the annotation is acceptable or the annotations are already present. On a high level, this property also makes the query problems related to some existing research topics in data mining (Chapters 3 and 4) or graph matching (Chapter 5). For example, if we simply ignore object labels and treat each unique object as a unique item, the problem of finding co-occurrence objects can be treated as a special case for finding item sets from item sequences. In our proposed methods, we rely on some characteristics of videos such as the locality of videos (i.e., objects that appeared in the scene are relatively stable in adjacent frames) to speed up the query execution. In the future, we can further explore whether the proposed methods can be extended to different domains other than video data.

Such an approach also has some limitations. For example, obtaining annotations for videos is not always feasible, especially for long videos or large video repositories, due to the significant amount of computational resources required by the annotation procedure. Moreover, for some queries, we may also want to consider the accuracy produced by CV algorithms with some mathematical guarantees. Ultimately, we aim to build query processing and analytic systems for videos. Along this line, there are still a lot of problems worth further exploring.

In this section, we discuss some of the promising directions from the following three aspects: query evaluation on video annotations, query evaluation on raw videos, and query evaluation on videos with multiple data sources.

6.2.1 Query Evaluation on Video Annotations

In this direction, we can still use the general framework discussed in this dissertation to obtain video annotations utilizing state-of-the-art algorithms. The underlying assumptions are: 1) the video is of high value and with high information density. 2) the query focus on the pattern of some specific objects included in the video. 3) the annotation results are relatively accurate, or inaccurate results produced by algorithms could be calibrated manually with acceptable costs. With such a framework, we could answer more complex queries on videos, some possible problems are listed as follows.

More complex queries with the graph modeling. In Chapter 5, we focused on the exact matching of graphs with equal window sizes. However, this matching mechanism could be too strict for queries in certain scenarios. For example, considering videos with different frame rates, both the video and the given pattern could share similar movements with different frame rates. Moreover, we may also wish to find certain cases where graphs are not exactly matched but are somewhat matched. For example, we may allow at most one object to be missing in the frame. Such cases require a different approach since the relaxation of the matching criteria would lead to potentially a high number of candidates during execution, making such problems even more challenging.

Queries with semantics-based representations. Another problem is modeling the semantics of objects in the frame. For example, in a video, we may wish to describe queries such as "the sun is rising from the sea". Although such queries can be answered with end-to-end CV models [11, 157], such models require extra training data and time. One way to answer such queries is to consider the semantics of the objects in the modeling and develop optimized algorithms. In such cases, the queries would be more focused on the semantics rather than the exact matching of temporal or spatial information and thus require special treatment to eliminate unpromising results early for faster execution.

Query optimizations considering the relationships between predicates. For complex queries with multiple predicates, selecting which one to apply first could also bring significant savings on the evaluation cost. This could be further explored since on videos with different characteristics, the pruning power of different predicates could vary. The precise

optimization in such cases is also worth exploring.

6.2.2 Query Evaluation on Raw Videos

Another line of work executes queries on raw videos directly. There have been many pieces of works focusing on specific aspects as discussed in Chapter 2, such as the use of cheap proxy models [24, 80, 79, 82], etc. Such techniques are suitable for scenarios where: 1) it is infeasible to apply detection or tracking algorithms on all frames in videos. 2) the system needs to trade off between query execution time and results accuracy. Despite some existing works, there are still some promising problems, listed as follows.

Efficient annotations for video query processing. One critical issue is the efficient annotations for video query processing. Previous work mainly focuses on annotating object detection results [82, 79]. However, the quality of tracking results could also have a significant impact on the query result. How to annotate videos efficiently, considering various annotation types such as color, object detection, object tracking, and even semantics segmentation, etc., is also an important topic. Moreover, considering the neural network structures that most CV algorithms utilize, the pipeline of applying such neural network models could be further optimized by enabling sharing between different models.

Semantics-aware query optimizations on raw videos. Another important piece of information carried with videos is video semantics. For example, by looking at only several frames from a video, we could infer whether the video is news broadcasting or surveillance footage. Depending on different videos, we have different assumptions on the characteristics of adjacent frames in the video, which could be further utilized for pruning when executing queries. This problem requires us to model the video semantics and apply such semantics for query answering, which could potentially improve the efficiency of query answering.

Cost models for video query processing. On a higher level, there are some queries that only focus on the final results, regardless of what algorithms are applied during execution. In such cases, queries are expected to return results that satisfy given metrics (i.e., accuracy, precision, etc.), while there could be multiple models or algorithms for performing the same query while satisfying the given criteria. A system could further utilize query rewriting techniques, which have been successfully applied in relational database systems,

to optimize the generated query plan. Cost models that are specifically designed for video processing systems, which consider both the inference time and the metrics of models, could be utilized here to provide better query rewriting results, leading to faster execution time or higher throughput. Additionally, providing mathematical guarantees could also be very useful for query answering with different target metrics.

6.2.3 Query Evaluation on Videos with Multiple Data Sources

In the above two sections, we assume that the queries are executed either on raw videos or on annotations. However, to design an end-to-end system for video processing and analysis, we may need to consider executing queries on a mixed format of data, including videos, annotations, or even external data sources such as text or audio. We further discuss the potential problems from the following aspects.

Query processing with materialized views. For queries that are executed directly on raw videos, one way to accelerate queries is to materialize data that have already been annotated and reuse the result for later queries, as demonstrated in [161], where a simple materialized strategy is discussed. However, if we consider different models that target the same task in a system, how to materialize and reuse such annotations across various models becomes a very challenging task. Precise exploration of such optimizations considering the trade-off between accuracy and efficiency is also worth studying.

Query processing with external information. In the real world, videos are normally produced along with other types of information, such as video metadata, audio, or even text. Such information could be further utilized to provide higher pruning power, in cases where the relationship between videos and other data sources can be easily established. In this scenario, developing a systematic approach for query evaluation, possible along with some cost model, across multiple data sources could also be very useful.

Aside from the problems discussed above, how to develop a systematic approach for video query processing, as well as benchmarking video systems, are also important. We believe that videos, as a media format that contains a large volume of unstructured data and stores explicit or implicit information of various kinds, could bring significant change to people's daily lives. Currently, the ways that people understand and retrieve videos are still relatively

primitive yet continuously involving. We believe that building video processing and analytic systems could bring significant change to the way people obtain information from a large set of videos. We will continue to work towards this ultimate goal and we hope that the studies presented in this dissertation would also be helpful to the community, as well as encourage more researchers to join us in this promising topic.

Bibliography

[1] https://github.com/pythonlessons/TensorFlow-2.x-YOLOv3.

[2] https://github.com/nwojke/deep_sort.

[3] https://github.com/adipandas/multi-object-tracker.

[4] https://db.cs.washington.edu/projects/visualroad/.

[5] Ahmad Sedky Adly et al. "Issues and Challenges for Content-Based Video Search Engines A Survey". In: *2020 21st International Arab Conference on Information Technology (ACIT)*. IEEE. 2020, pp. 1–18.

[6] Ramesh C Agarwal, Charu C Aggarwal, and VVV Prasad. "A tree projection algorithm for generation of frequent item sets". In: *Journal of parallel and Distributed Computing* 61.3 (2001), pp. 350–371.

[7] Arnav Agharwal et al. "Tag-based video retrieval by embedding semantic content in a continuous word space". In: *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE. 2016, pp. 1–8.

[8] Rakesh Agrawal, Ramakrishnan Srikant, et al. "Fast algorithms for mining association rules". In: *Proc. 20th int. conf. very large data bases, VLDB*. Vol. 1215. Citeseer. 1994, pp. 487–499.

[9] Rakesh Agrawal and Ramakrishnan Srikant. "Mining sequential patterns". In: *Proceedings of the eleventh international conference on data engineering*. IEEE. 1995, pp. 3–14.

[10] Michael R Anderson et al. "Physical representation-based predicate optimization for a visual analytics database". In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE. 2019, pp. 1466–1477.

[11] Lisa Anne Hendricks et al. "Localizing moments in video with natural language". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 5803–5812.

[12] Jay Ayres et al. "Sequential pattern mining using a bitmap representation". In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2002, pp. 429–435.

[13] Yusuf Aytar, Mubarak Shah, and Jiebo Luo. "Utilizing semantic word similarity measures for video retrieval". In: *2008 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2008, pp. 1–8.

[14] Tadas Baltrušaitis, Chaitanya Ahuja, and Louis-Philippe Morency. "Multimodal machine learning: A survey and taxonomy". In: *IEEE transactions on pattern analysis and machine intelligence* 41.2 (2018), pp. 423–443.

[15] Faisal I Bashir, Ashfaq A Khokhar, and Dan Schonfeld. "Real-time motion trajectory-based indexing and retrieval of video sequences". In: *IEEE Transactions on Multimedia* 9.1 (2006), pp. 58–65.

[16] Favyen Bastani and Samuel Madden. "OTIF: Efficient tracker pre-processing over large video datasets". In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 2091–2104.

[17] Favyen Bastani et al. "MIRIS: Fast Object Track Queries in Video". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1907–1921.

[18] Philipp Bergmann, Tim Meinhardt, and Laura Leal-Taixe. "Tracking without bells and whistles". In: *Proceedings of the IEEE international conference on computer vision*. 2019, pp. 941–951.

[19] David S Bolme et al. "Visual object tracking using adaptive correlation filters". In: *2010 IEEE computer society conference on computer vision and pattern recognition*. IEEE. 2010, pp. 2544–2550.

[20] Nicolas Bruno, Luis Gravano, and Amélie Marian. "Evaluating top-k queries over web-accessible databases". In: *Proceedings 18th International Conference on Data Engineering*. IEEE. 2002, pp. 369–380.

[21] Murray Campbell et al. "IBM Research TRECVID-2006 Video Retrieval System." In: *TRECVID*. 2006, pp. 175–182.

[22] Jiashen Cao et al. "FiGO: Fine-Grained Query Optimization in Video Analytics". In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 559–572.

[23] Lei Chang et al. "Seqstream: Mining closed sequential patterns over stream sliding windows". In: *2008 Eighth IEEE international conference on data mining*. IEEE. 2008, pp. 83–92.

[24] Daren Chao, Nick Koudas, and Ioannis Xarchakos. "SVQ++: Querying for Object Interactions in Video Streams". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2769–2772.

[25] Surajit Chaudhuri, Luis Gravano, and Amélie Marian. "Optimizing top-k selection queries over multimedia repositories". In: *IEEE Transactions on Knowledge and Data Engineering* 16.8 (2004), pp. 992–1009.

[26] Chin-Hoong Chee et al. "Algorithms for frequent itemset mining: a literature review". In: *Artificial Intelligence Review* 52.4 (2019), pp. 2603–2621.

[27] Yi-Cheng Chen, Wen-Chih Peng, and Suh-Yin Lee. "Mining temporal patterns in time interval-based data". In: *IEEE Transactions on Knowledge and Data Engineering* 27.12 (2015), pp. 3318–3331.

[28] Kai Chen et al. "MMDetection: Open MMLab Detection Toolbox and Benchmark". In: *arXiv preprint arXiv:1906.07155* (2019).

[29] Yun Chi et al. "Moment: Maintaining closed frequent itemsets over a stream sliding window". In: *Fourth IEEE International Conference on Data Mining (ICDM'04)*. IEEE. 2004, pp. 59–66.

[30] Peng Chu et al. “Transmot: Spatial-temporal graph transformer for multiple object tracking”. In: *arXiv preprint arXiv:2104.00194* (2021).

[31] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. “Multi-column deep neural networks for image classification”. In: *2012 IEEE conference on computer vision and pattern recognition*. IEEE. 2012, pp. 3642–3649.

[32] MMTracking Contributors. *MMTracking: OpenMMLab video perception toolbox and benchmark*. `https://github.com/open-mmlab/mmtracking`. 2020.

[33] Luigi P Cordella et al. “A (sub) graph isomorphism algorithm for matching large graphs”. In: *IEEE transactions on pattern analysis and machine intelligence* 26.10 (2004), pp. 1367–1372.

[34] Zhen Cui et al. “Recurrently target-attending tracking”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 1449–1458.

[35] Navneet Dalal and Bill Triggs. “Histograms of oriented gradients for human detection”. In: *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*. Vol. 1. Ieee. 2005, pp. 886–893.

[36] Alberto Del Bimbo, Enrico Vicario, and Daniele Zingoni. “Symbolic description and visual querying of image sequences using spatio-temporal logic”. In: *IEEE transactions on knowledge and data engineering* 7.4 (1995), pp. 609–622.

[37] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[38] Jianfeng Dong et al. “Dual encoding for zero-example video retrieval”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 9346–9355.

[39] Kevin Duarte, Yogesh Rawat, and Mubarak Shah. “Videocapsulenet: A simplified network for action detection”. In: *Advances in neural information processing systems* 31 (2018).

[40] Ronald Fagin, Ravi Kumar, and Dakshinamurthi Sivakumar. “Comparing top k lists”. In: *SIAM Journal on discrete mathematics* 17.1 (2003), pp. 134–160.

[41] Ronald Fagin, Amnon Lotem, and Moni Naor. "Optimal aggregation algorithms for middleware". In: *Journal of computer and system sciences* 66.4 (2003), pp. 614–656.

[42] Pedro Felzenszwalb, David McAllester, and Deva Ramanan. "A discriminatively trained, multiscale, deformable part model". In: *2008 IEEE conference on computer vision and pattern recognition*. Ieee. 2008, pp. 1–8.

[43] Philippe Fournier-Viger et al. "A survey of sequential pattern mining". In: *Data Science and Pattern Recognition* 1.1 (2017), pp. 54–77.

[44] Maximilian Franzke et al. "Pattern search in temporal social networks". In: *Proceedings of the 21st International Conference on Extending Database Technology*. 2018.

[45] Edward Fredkin. "Trie memory". In: *Communications of the ACM* 3.9 (1960), pp. 490–499.

[46] Tingjian Ge, Stan Zdonik, and Samuel Madden. "Top-k queries on uncertain data: on score distribution and typical answers". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 2009, pp. 375–388.

[47] Mazaher Ghorbani and Masoud Abessi. "A new methodology for mining frequent itemsets on temporal data". In: *IEEE Transactions on Engineering Management* 64.4 (2017), pp. 566–573.

[48] Ross Girshick. "Fast r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1440–1448.

[49] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587.

[50] Ian Goodfellow et al. *Deep learning*. Vol. 1. MIT press Cambridge, 2016.

[51] J Guntzer, W-T Balke, and Werner Kießling. "Towards efficient multi-feature queries in heterogeneous environments". In: *Proceedings International Conference on Information Technology: Coding and Computing*. IEEE. 2001, pp. 622–628.

[52] Abdul Mueed Hafiz and Ghulam Mohiuddin Bhat. "A survey on instance segmentation: state of the art". In: *International journal of multimedia information retrieval* 9.3 (2020), pp. 171–189.

[53] Jiawei Han, Jian Pei, and Hanghang Tong. *Data mining: concepts and techniques*. Morgan kaufmann, 2022.

[54] Jiawei Han, Jian Pei, and Yiwen Yin. "Mining frequent patterns without candidate generation". In: *ACM sigmod record* 29.2 (2000), pp. 1–12.

[55] Seungyeop Han et al. "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints". In: *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. 2016, pp. 123–136.

[56] Alex Hauptmann et al. *Informedia at TRECVID 2003: Analyzing and searching broadcast news video*. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 2004.

[57] Brandon Haynes et al. "Visual Road: A Video Data Management Benchmark". In: *Proceedings of the 2019 International Conference on Management of Data*. ACM. 2019, pp. 972–987.

[58] Brandon Haynes et al. "VisualWorldDB: A DBMS for the Visual World". In: *CIDR*. 2020.

[59] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[60] Kaiming He et al. "Mask r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969.

[61] Wenjia He and Michael Cafarella. "Controlled Intentional Degradation in Analytical Video Systems". In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 2105–2119.

[62] David Held, Sebastian Thrun, and Silvio Savarese. "Learning to track at 100 fps with deep regression networks". In: *European conference on computer vision*. Springer. 2016, pp. 749–765.

[63] Joao F Henriques et al. "Exploiting the circulant structure of tracking-by-detection with kernels". In: *European conference on computer vision*. Springer. 2012, pp. 702–715.

[64] João F Henriques et al. "High-speed tracking with kernelized correlation filters". In: *IEEE transactions on pattern analysis and machine intelligence* 37.3 (2014), pp. 583–596.

[65] Chin-Chuan Ho et al. "Incremental mining of sequential patterns over a stream sliding window". In: *Sixth IEEE International Conference on Data Mining-Workshops (ICDMW'06)*. IEEE. 2006, pp. 677–681.

[66] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[67] Kevin Hsieh et al. "Focus: Querying large video datasets with low latency and low cost". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 269–286.

[68] Bo Hu, Peizhen Guo, and Wenjun Hu. "Video-zilla: An Indexing Layer for Large-Scale Video Analytics". In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 1905–1919.

[69] Weiming Hu et al. "A survey on visual content-based video indexing and retrieval". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 41.6 (2011), pp. 797–819.

[70] Gao Huang et al. "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[71] Po-Yao Huang et al. "Informedia@ TRECVID 2018: Ad-hoc video search with discrete and continuous representations". In: *TRECVID Proceedings*. Vol. 70. 2018.

[72] Chien-Chun Hung et al. "Videoedge: Processing camera streams using hierarchical clusters". In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2018, pp. 115–131.

[73] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. "A survey of top-k query processing techniques in relational database systems". In: *ACM Computing Surveys (CSUR)* 40.4 (2008), pp. 1–58.

[74] Haoliang Jiang et al. "Gstring: A novel approach for efficient search in graph databases". In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2007, pp. 566–575.

[75] Junchen Jiang et al. "Chameleon: scalable adaptation of video analytics". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 253–266.

[76] Nan Jiang and Le Gruenwald. "CFI-Stream: mining closed frequent itemsets in data streams". In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2006, pp. 592–597.

[77] Cheqing Jin et al. "Sliding-window top-k queries on uncertain streams". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 301–312.

[78] Rudolph Emil Kalman. "A new approach to linear filtering and prediction problems". In: (1960).

[79] D. Kang, P. Bailis, and M. Zaharia. "BlazeIT: Fast Exploratory Video Queries Using Neural Networks". In: *PVLDB*. 2019.

[80] Daniel Kang, Peter Bailis, and Matei Zaharia. "BlazeIt: optimizing declarative aggregation and limit queries for neural network-based video analytics". In: *Proceedings of the VLDB Endowment* 13.4 (2019), pp. 533–546.

[81] Daniel Kang, Peter Bailis, and Matei Zaharia. "Challenges and Opportunities in DNN-Based Video Analytics: A Demonstration of the BlazeIt Video Query Engine." In: *CIDR*. 2019.

[82] Daniel Kang et al. “NoScope: optimizing neural network queries over video at scale”. In: *Proceedings of the VLDB Endowment* 10.11 (2017), pp. 1586–1597.

[83] Daniel Kang et al. “Jointly optimizing preprocessing and inference for DNN-based visual analytics”. In: *Proceedings of the VLDB Endowment* 14.2 (2020), pp. 87–100.

[84] Daniel Kang et al. “Task-agnostic Indexes for Deep Learning-based Queries over Unstructured Data”. In: *arXiv preprint arXiv:2009.04540* (2020).

[85] Daniel Kang et al. “Accelerating approximate aggregation queries with expensive predicates”. In: *Proceedings of the VLDB Endowment* 14.11 (2021), pp. 2341–2354.

[86] Daniel Kang et al. “Approximate Selection with Guarantees using Proxies”. In: *Proceedings of the VLDB Endowment* 13.11 ().

[87] Nick Koudas, Raymond Li, and Ioannis Xarchakos. “Video Monitoring Queries”. In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE. 2020, pp. 1285–1296.

[88] Sebastian Krebs, Bharanidhar Duraisamy, and Fabian Flohr. “A survey on leveraging deep neural networks for object tracking”. In: *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2017, pp. 411–418.

[89] Sebastian Krebs, Bharanidhar Duraisamy, and Fabian Flohr. “A survey on leveraging deep neural networks for object tracking”. In: *20th IEEE International Conference on Intelligent Transportation Systems, ITSC 2017, Yokohama, Japan, October 16-19, 2017*. 2017, pp. 411–418.

[90] Ranjay Krishna et al. “Dense-captioning events in videos”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 706–715.

[91] Alex Krizhevsky, Geoffrey Hinton, et al. “Learning multiple layers of features from tiny images”. In: (2009).

[92] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[93] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. "Fractalnet: Ultra-deep neural networks without residuals". In: *arXiv preprint arXiv:1605.07648* (2016).

[94] Duy-Dinh Le, Shin'ichi Satoh, and Michael E Houle. "Face retrieval in broadcasting news video by fusing temporal and intensity information". In: *International Conference on Image and Video Retrieval*. Springer. 2006, pp. 391–400.

[95] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[96] Yann LeCun, Fu Jie Huang, and Leon Bottou. "Learning methods for generic object recognition with invariance to pose and lighting". In: *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.* Vol. 2. IEEE. 2004, pp. II–104.

[97] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[98] Yong Joon Lee et al. "Mining temporal interval relational rules from temporal data". In: *Journal of Systems and Software* 82.1 (2009), pp. 155–167.

[99] Jure Leskovec et al. "Microscopic evolution of social networks". In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2008, pp. 462–470.

[100] Xirong Li et al. "W2vv++ fully deep learning for ad-hoc video search". In: *Proceedings of the 27th ACM International Conference on Multimedia*. 2019, pp. 1786–1794.

[101] Zhijie Lin et al. "Moment retrieval via cross-modal interaction networks with query reconstruction". In: *IEEE Transactions on Image Processing* 29 (2020), pp. 3750–3762.

[102] Li Liu et al. "Deep learning for generic object detection: A survey". In: *International Journal of Computer Vision* 128.2 (2020), pp. 261–318.

[103] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[104] Jakub Lokoč et al. “On influential trends in interactive video retrieval: video browser showdown 2015–2017”. In: *IEEE Transactions on Multimedia* 20.12 (2018), pp. 3361–3376.

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

[106] Yao Lu et al. “Accelerating machine learning inference with probabilistic predicates”. In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 1493–1508.

[107] Siwei Lyu et al. “UA-DETRAC 2017: Report of AVSS2017 & IWT4S Challenge on Advanced Traffic Monitoring”. In: *Advanced Video and Signal Based Surveillance (AVSS), 2017 14th IEEE International Conference on*. IEEE. 2017, pp. 1–7.

[108] Amélie Marian, Nicolas Bruno, and Luis Gravano. “Evaluating top-k queries over web-accessible databases”. In: *ACM Transactions on Database Systems (TODS)* 29.2 (2004), pp. 319–362.

[109] Xue Mei et al. “Minimum error bounded efficient l 1 tracker with occlusion detection”. In: *CVPR 2011*. IEEE. 2011, pp. 1257–1264.

[110] Xianrui Meng, Haohan Zhu, and George Kollios. “Top-k query processing on encrypted databases with strong security guarantees”. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 353–364.

[111] Anton Milan et al. “MOT16: A benchmark for multi-object tracking”. In: *arXiv preprint arXiv:1603.00831* (2016).

[112] Shervin Minaee et al. “Image segmentation using deep learning: A survey”. In: *IEEE transactions on pattern analysis and machine intelligence* (2021).

[113] Niluthpol Chowdhury Mithun, Sujoy Paul, and Amit K Roy-Chowdhury. “Weakly supervised video moment retrieval from text queries”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 11592–11601.

[114] Yuuki Miyoshi, Tomonobu Ozaki, and Takenao Ohkawa. "Mining interesting patterns and rules in a time-evolving graph". In: *Proc. Int. MultiConf. Engineers and Computer Scientists 2011* 1 (2011), pp. 448–453.

[115] Apostol Natsev et al. "Supporting incremental join queries on ranked inputs". In: *VLDB*. Vol. 1. 2001, pp. 281–290.

[116] Fatemeh Nori, Mahmood Deypir, and Mohamad Hadi Sadreddini. "A sliding window based algorithm for frequent closed itemset mining over data streams". In: *Journal of Systems and Software* 86.3 (2013), pp. 615–623.

[117] Panagiotis Papapetrou et al. "Discovering frequent arrangements of temporal intervals". In: *Fifth IEEE International Conference on Data Mining (ICDM'05)*. IEEE. 2005, 8–pp.

[118] Jian Pei et al. "Mining sequential patterns by pattern-growth: The prefixspan approach". In: *IEEE Transactions on knowledge and data engineering* 16.11 (2004), pp. 1424–1440.

[119] Alex Poms et al. "Scanner: Efficient video analysis at scale". In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), pp. 1–13.

[120] Chedi Raıssi, Pascal Poncelet, and Maguelonne Teisseire. "Need for speed: Mining sequential patterns in data streams". In: *BDA05: Actes des 21iemes Journees Bases de Donnees Avancees* (2005).

[121] Christopher Re, Nilesh Dalvi, and Dan Suciu. "Efficient top-k query evaluation on probabilistic data". In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2007, pp. 886–895.

[122] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *CoRR* abs/1804.02767 (2018).

[123] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 779–788.

[124] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[125] Shaoqing Ren et al. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems* 28 (2015), pp. 91–99.

[126] Francisco Romero et al. "Optimizing Video Analytics with Declarative Model Relationships". In: *Proceedings of the VLDB Endowment* 16.3 (2022), pp. 447–460.

[127] Konstantinos Semertzidis and Evaggelia Pitoura. "Durable graph pattern queries on historical graphs". In: *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE. 2016, pp. 541–552.

[128] Konstantinos Semertzidis and Evaggelia Pitoura. "Top-k Durable Graph Pattern Queries on Temporal Graphs". In: *IEEE Transactions on Knowledge and Data Engineering* 31.1 (2018), pp. 181–194.

[129] Anil Shanbhag, Holger Pirk, and Samuel Madden. "Efficient top-k query processing on massively parallel hardware". In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 1557–1570.

[130] Haichuan Shang et al. "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 364–375.

[131] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[132] Josef Sivic, Mark Everingham, and Andrew Zisserman. "Person spotting: video shot retrieval for face sets". In: *International conference on image and video retrieval*. Springer. 2005, pp. 226–236.

[133] Mohamed A Soliman, Ihab F Ilyas, and Kevin Chen-Chuan Chang. "Top-k query processing in uncertain databases". In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2006, pp. 896–905.

[134] Newton Spolaôr et al. “A systematic review on content-based video retrieval”. In: *Engineering Applications of Artificial Intelligence* 90 (2020), p. 103557.

[135] Ramakrishnan Srikant and Rakesh Agrawal. “Mining sequential patterns: Generalizations and performance improvements”. In: *International conference on extending database technology*. Springer. 1996, pp. 1–17.

[136] Guohao Sun et al. “Incremental graph pattern based node matching”. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 281–292.

[137] Jimeng Sun et al. “Graphscope: parameter-free mining of large time-evolving graphs”. In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2007, pp. 687–696.

[138] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[139] Ran Tao, Efstratios Gavves, and Arnold WM Smeulders. “Siamese instance search for tracking”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 1420–1429.

[140] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. “Top-k query evaluation with probabilistic guarantees”. In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 2004, pp. 648–659.

[141] Jasper RR Uijlings et al. “Selective search for object recognition”. In: *International journal of computer vision* 104.2 (2013), pp. 154–171.

[142] R. Urtasun. “Self Driving Vehicle Technology”. In: *CVPR 2020, Tutorial* (2020).

[143] Jianyong Wang, Jiawei Han, and Chun Li. “Frequent closed sequence mining without candidate maintenance”. In: *IEEE Transactions on Knowledge and Data Engineering* 19.8 (2007), pp. 1042–1056.

[144] Lijun Wang et al. “Visual tracking with fully convolutional networks”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 3119–3127.

[145] Ling Wang et al. “Mining temporal association rules with frequent itemsets tree”. In: *Applied Soft Computing* 62 (2018), pp. 817–829.

[146] Naiyan Wang and Dit-Yan Yeung. “Learning a deep compact image representation for visual tracking”. In: *Advances in neural information processing systems* 26 (2013).

[147] Xiang Wang et al. “Top-k spatial-keyword publish/subscribe over sliding window”. In: *The VLDB Journal* 26.3 (2017), pp. 301–326.

[148] Longyin Wen et al. “UA-DETRAC: A New Benchmark and Protocol for Multi-Object Detection and Tracking”. In: *Computer Vision and Image Understanding* (2020).

[149] Simon Wenkel et al. “Confidence score: The forgotten dimension of object detection performance evaluation”. In: *Sensors* 21.13 (2021), p. 4350.

[150] Steven Euijong Whang et al. “Indexing boolean expressions”. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 37–48.

[151] David W Williams, Jun Huan, and Wei Wang. “Graph database indexing using structured graph decomposition”. In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2007, pp. 976–985.

[152] Nicolai Wojke and Alex Bewley. “Deep Cosine Metric Learning for Person Re-identification”. In: *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE. 2018, pp. 748–756.

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

[154] Yi Wu, Jongwoo Lim, and Ming-Hsuan Yang. “Online object tracking: A benchmark”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2013, pp. 2411–2418.

[155] Ioannis Xarchakos and Nick Koudas. “Svq: Streaming video queries”. In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 2013–2016.

[156] Ioannis Xarchakos and Nick Koudas. “Querying for Interactions”. In: *IEEE Transactions on Knowledge and Data Engineering* (2021).

[157] Shaoning Xiao et al. “Boundary proposal network for two-stage natural language video localization”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 4. 2021, pp. 2986–2994.

[158] Jun Xu et al. “Msr-vtt: A large video description dataset for bridging video and language”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 5288–5296.

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

[160] Yingkun Xu et al. “Deep learning for multiple object tracking: a survey”. In: *IET Computer Vision* 13.4 (2019), pp. 355–368.

[161] Zhuangdi Xu et al. “EVA: A symbolic approach to accelerating exploratory video analytics with materialized views”. In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 602–616.

[162] Chikashi Yajima, Yoshihiro Nakanishi, and Katsumi Tanaka. “Querying video data by spatio-temporal relationships of moving object traces”. In: *Working Conference on Visual Database Systems*. Springer. 2002, pp. 357–371.

[163] Rong Yan and Alexander G Hauptmann. “A review of text and image retrieval approaches for broadcast news video”. In: *Information Retrieval* 10.4-5 (2007), pp. 445–484.

[164] Mang Ye et al. “Deep learning for person re-identification: A survey and outlook”. In: *IEEE transactions on pattern analysis and machine intelligence* 44.6 (2021), pp. 2872–2893.

[165] Serena Yeung et al. “End-to-end learning of action detection from frame glimpses in videos”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2678–2687.

[166] Fisher Yu et al. "BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning". In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2020.

[167] Fisher Yu et al. "BDD100K: A diverse driving dataset for heterogeneous multitask learning". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 2636–2645.

[168] Youngjae Yu, Jongseok Kim, and Gunhee Kim. "A joint sequence fusion model for video question answering and retrieval". In: *Proceedings of the European Conference on Computer Vision*. 2018, pp. 471–487.

[169] Mohammed J Zaki. "SPADE: An efficient algorithm for mining frequent sequences". In: *Machine learning* 42 (2001), pp. 31–60.

[170] Matthew D Zeiler and Rob Fergus. "Visualizing and understanding convolutional networks". In: *European conference on computer vision*. Springer. 2014, pp. 818–833.

[171] Haoyu Zhang et al. "Live Video Analytics at Scale with Approximation and Delay-Tolerance". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 377–392.

[172] Yifu Zhang et al. "Bytetrack: Multi-object tracking by associating every detection box". In: *arXiv preprint arXiv:2110.06864* (2021).

[173] Yifu Zhang et al. "Fairmot: On the fairness of detection and re-identification in multiple object tracking". In: *International Journal of Computer Vision* 129.11 (2021), pp. 3069–3087.

[174] Yucheng Zhang et al. "Recent advances of single-object tracking methods: A brief survey". In: *Neurocomputing* 455 (2021), pp. 1–11.

[175] Yuhao Zhang and Arun Kumar. "Panorama: a data system for unbounded vocabulary querying over video". In: *Proceedings of the VLDB Endowment* 13.4 (2019), pp. 477–491.

[176] Yue Zhao et al. “Temporal action detection with structured segment networks”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 2914–2923.

[177] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. “Objects as points”. In: *arXiv preprint arXiv:1904.07850* (2019).

[178] Zhengxia Zou et al. “Object detection in 20 years: A survey”. In: *arXiv preprint arXiv:1905.05055* (2019).