Spatial and Temporal Constrained Ranked Retrieval over Videos

Yueting Chen
York University
ytchen@eecs.yorku.ca

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Xiaohui Yu*
York University
xhyu@yorku.ca

Ziqiang Yu
Yantai University
Key Laboratory of Urban Land Resources Monitoring and Simulation, MNR
zqyu@ytu.edu.cn

ABSTRACT

Recent advances in Computer Vision (CV) algorithms have improved accuracy and efficiency, making video annotations possible with high accuracy. In this paper, 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.

PVLDB Reference Format:
Yueting Chen, Nick Koudas, Xiaohui Yu, and Ziqiang Yu. Spatial and Temporal Constrained Ranked Retrieval over Videos. PVLDB, 15(11): 3226 - 3239, 2022.
doi:10.14778/3551793.3551865

1 INTRODUCTION

Large amounts of videos are being recorded and produced daily thanks to the popularity of modern computing devices and fast networks. Such abundance of video instigates pressing needs for subsequent analysis. With Deep Learning models, solid accuracy and efficiency have been achieved on many fundamental computer vision tasks such as image classification [18, 19, 39, 43], object detection [15, 17, 34, 35], and object tracking[4, 46, 47]. Such advances enable new opportunities for query processing over videos. For example, analyzing the results produced by object detection algorithms, objects that appeared per frame can be annotated automatically by type, enabling queries regarding the object types, the number of objects as well as the position of objects in the scene.

Recent works in the data management community utilize computer vision (CV) algorithms to extract data from raw frames to answer queries over videos with conditions such as object types[20, 25], object colors[23, 24], spatial constraints [27, 48], temporal constraints [8, 9] as well as interactions [6]. The current processing pipeline in the literature falls under two broad categories; one is to accelerate query answering via filtering based on cheap approximate filters utilizing DL models [20, 25, 48], while another approach applies CV algorithms as a pre-processing step and works on the extracted data only [8, 9]. The first approach usually requires extra training for cheap neural network models, while the second can adapt to new CV algorithms easily which can be further refined by humans to improve the annotation results if required.

In this paper, we take the second approach and 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 which contain 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 to 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 existing works [8, 9], 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, (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 1(a)-(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

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551865

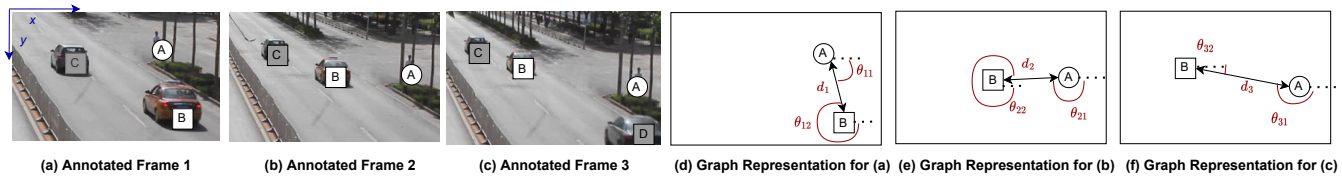


Figure 1: An Example

Figure 1(d)-(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 object (e.g., d_1 in Figure 1(d)), and (2) the angle values between two objects (e.g., θ_{11}, θ_{12} in Figure 1(d)).

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 predefined 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.

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.[1] 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 2.

In this paper, 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 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 Graph Index (GI) to precompute and materialize the graphs constructed 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.

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

The remainder of the paper is organized as follows. Section 2 formally defines the problem. Section 3 presents the overview of the framework, followed by the Graph Index (GI) in Section 4, and query processing in Section 5. Section 6 evaluates the proposed methods via experiments. Related works are discussed in Section 7, and Section 8 concludes the paper.

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 1. ***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, which 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., which 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 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 1(a). 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. We then define query graph matching as follows.

Definition 2. ***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 $h : V_P \rightarrow V_M$ 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 3. ***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 4. ***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 5. ***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 $h \in H$: ($|\mathcal{G}| = |\mathcal{P}| = l$)*

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

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 6. ***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 6 specifies STAR Retrieval on one video. Similarly, for a video repository, STAR Retrieval is conducted 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.

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 2.

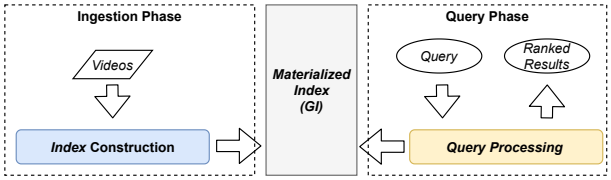


Figure 2: Framework Overview

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.

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 4, and then focus on query processing in Section 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 4.1); (2) introduce edge discretization to enable sharing between edges with the same attributes (Section 4.2).

4.1 Minimum Object Graph

As per Definition 1, 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 4.1. *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 4.1 leads to the following way of reducing the number of edges in the object graphs. We introduce *minimum object graphs.*

Definition 7 (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 4.1 iteratively, we can prove the following.

Theorem 4.2. *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 representation. As per Definition 7, 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 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 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. In the remaining sections, we use query graph and minimum query graph interchangeably, and use query graph sequence and minimum query graph sequence interchangeably.

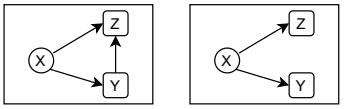


Figure 3: Minimum Object Graph

4.2 Edge Discretization

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 2.

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 5.2) as well as minimizing the storage overhead of the graph index (to be discussed in Section 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.

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 preprocess. 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 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

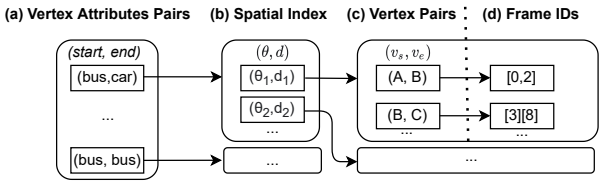


Figure 4: The Graph Index

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 4(a). For each edge, we obtain the attributes for both vertices. For example, the vertex attribute pair, *(bus, car)*, denotes the edge is 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 4(b) and Figure 4(c). 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 twofold: (1) grouping edges that share the same spatial relationships eliminates duplicate values and thus reduces the storage overhead; (2) as per Definition 5, all potential matches must be considered in computing the matching score. Separating the spatial attributes (d and θ) from the vertices (u and v) allow 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 4(d). 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. Both index levels are implemented as hash indexes to enable fast retrieval.

5 QUERY PROCESSING

5.1 Overview

A straightforward way to commence processing is to proceed in two steps, as shown in Figure 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 utilizing information provided in the query graph sequence is very crucial.

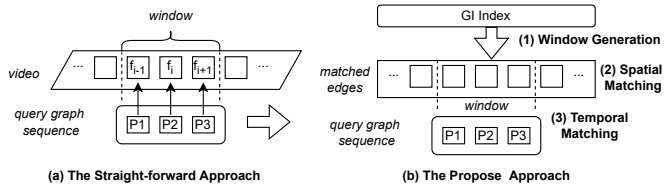


Figure 5: Comparison between Two Different Approaches

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(b): window generation, spatial matching, and temporal matching. Window generation (Section 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 5.3) produces matching data graphs from each frame, while temporal matching (Section 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.

5.2 Window Generation

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 for further processing.

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 $\mathcal{W} = \cup_{f_i \in F}\mathcal{W}_{f_i}$.

5.3 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(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 5.2). 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.

5.3.1 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.

5.3.2 Main Idea of Our Approach. Observe that for each Minimum Query Graph (Section 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 5.4) without generating all possible data graphs. The spatial matching step focuses only on

Algorithm 1: 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         I[idx + 1] ← I[idx + 1] ∪ {v_e};
13 r ← res.values();
14 foreach I ∈ r do
15     remove I if any element in I is empty;
16 return r;
```

producing intermediate data graphs by merging matched edges with the same anchor vertex (Section 5.3.3).

5.3.3 The SMA Algorithm. We propose the Spatial Matching Algorithm (SMA), depicted as Algorithm 1, 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.

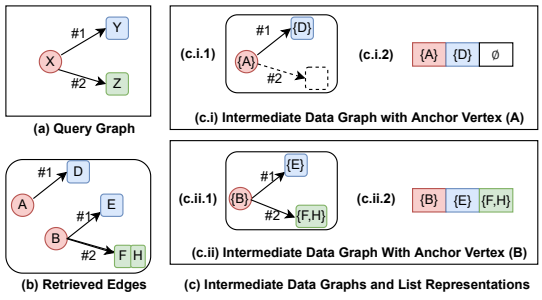


Figure 6: An Example for Spatial Matching

Example 1. *Figure 6 demonstrates the spatial matching between a query graph (Figure 1(a)) and a frame (Figure 1(b)). 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 6(b), 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 1.*

(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 5.2 (Line 5), shown in Figure 6(b). 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 6(b)).

(3) Intermediate data graph update (Lines 7-12). 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 12).

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 1(c.ii.2). The corresponding intermediate data graph is visualized in Figure 6(c.ii.1).

Note that the intermediate data graph with anchor vertex, A (Figure 6(c.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).

5.4 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 5.2). For a query graph sequence $\mathcal{P}$, the same vertex could appear in multiple query graphs. As per Definition 4, 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 5.4.1, and then discuss how windows can be prioritized in Section 5.4.3.

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 intermediate data graphs resulting from the previous steps. To store the contextual information

Algorithm 2: A Step in Temporal Matching (TM)

Input: the vertex to match v, Match Candidate Set S_M, the current window matching score $score$

1 $L_{\mathcal{P}} \leftarrow$ the ordered list of unique vertices in $V_{\mathcal{P}}$;
2 $M \leftarrow$ **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 $v \leftarrow$ **nextVertexToMatch**(M_V, $L_{\mathcal{P}}$);
5 $V' \leftarrow$ **getMatchingVerticesFor**(M_I, v);
6 **foreach** v' *in* V' **do**
7 **if** *not* M_V*.contains*(v') **then**
8 $M'_V \leftarrow$ add(M_V, v');
9 $M'_I \leftarrow$ **getMatchingIntermediateDataGraphs**(M_I,v');
10 **if** $|M'_V| == |V_{\mathcal{P}}|$ **then**
11 // means we have found one matched data graph sequence;
12 $M'_F \leftarrow$ **getRelevantFrames**(M'_I);
13 $score \leftarrow$ **max**($|M'_F|$, $score$);
14 **else**
15 $M' \leftarrow$ **MatchCandidate**(M'_V, M'_I);
16 $S_M \leftarrow S_M \cup \{M'\}$;

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 2 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 5.4.3). We use two sets to store the 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 10-13). In this case, we first obtain the set of relevant frames, M'_F, where at least one active intermediate data graphs is generated from each frame in M'_F (Line 12), and then update the matching score for the current window based on size of M'_F (Line 13). Otherwise, new Match Candidates are produced for further processing (Lines 15-16). 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 2. *Figure 7 presents an example on a window of three frames. The query graph sequence is shown in Figure 7(a), and the frames are shown in Figure 7(b). 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.*

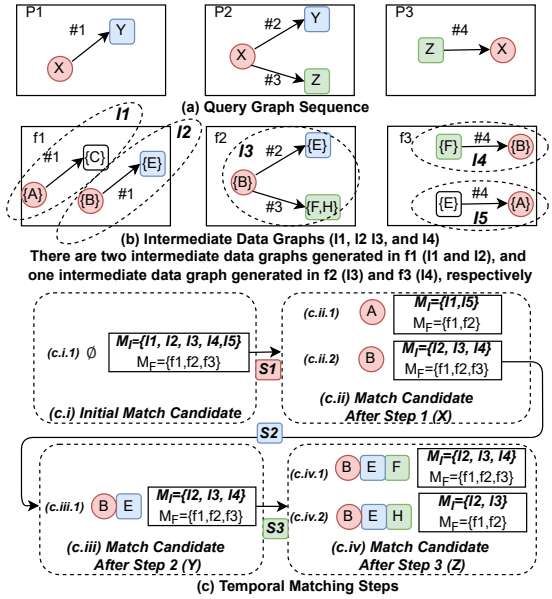


Figure 7: Temporal Matching Steps

Initially, the matching score of the current window is set to 0. We initialize one Match Candidate for the window, shown in Figure 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 7(c)), 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 2 , 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 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 7(c.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 the 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 7(c.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 7(c.iii, c.iv).

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 3. 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 2

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 3: 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
```

Example 3. *Based on the above example, Example 2, we can obtain the estimated score for each Match Candidate. For example, in Figure 7(c.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 7(c.iii), the Match Candidate with vertices (B,E) has the highest score of 3, so we continue the procedure. In step 3 (Figure 7(c.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 7(c.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.*

5.4.3 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 3, 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 3 and is omitted here for brevity.

6 EXPERIMENTS

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 except the object detection/tracking algorithms, for which we use open-source implementations [7, 10]. Specifically, we use Faster RCNN [35] as the object detection algorithm and Tracktor [4] as the tracking algorithm. Data are pre-loaded to memory before evaluation.

Datasets. Experiments are conducted on two real-world video datasets: De-trac [44] and Deep Drive [52]. 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).

Table 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

Data statistics are provided in Table 1, including the total number of frames (# frames), the average number of objects per frame (# avg obj/f), the total number of objects (# 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., lower average duration of each object).

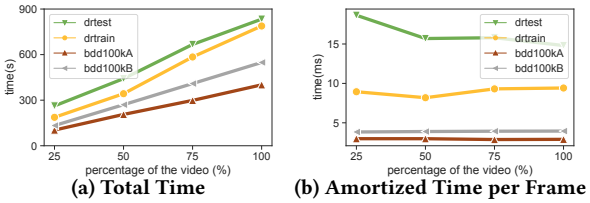
Query Processing Methods. We evaluate the query processing performance of three methods, namely, *base*, *prop* and *prop_s*. All three methods are based on the proposed index. The *base* method serves as the baseline method, which simply accepts the retrieved edges and performs graph matching utilizing Depth-First Search, as discussed in Section 5.3.1. 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 5.4.3).

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 [15] as the object detection algorithm and Tracktor [4] 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 [34], DeepSORT [46] etc.).

We report the query processing time based on a set of query graph sequences randomly generated from the raw videos 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. (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.

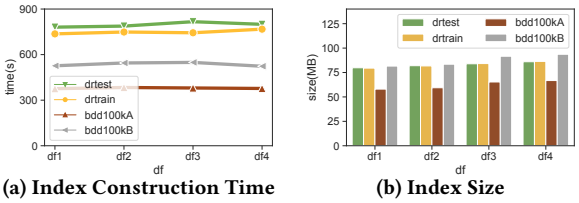
In our experiments, we set p_n to a fixed value, 20, which is sufficient to represent the distribution of query graph sequences. As can be validated by the experiment figures, query graph sequences generated in this way contain queries with different selectivities and their running time is representative 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



(a) Total Time (b) Amortized Time per Frame

Figure 8: Index Construction Time w.r.t. Number of Frames



(a) Index Construction Time (b) Index Size

Figure 9: Index Construction w.r.t. Discretization Granularity

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.

6.2 Index Construction

Varying datasets and the number of frames. We first report the index construction time for different number of frames in each video, shown in Figure 8. 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 8b, 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. As a result the index construction is highly scalable. We also observe an increasing or decreasing trend in time (drtest and drtrain) as the number of objects change in each frame, which demonstrates that the number of objects per frame dominates the index construction cost, since more edges could be generated and materialized with more objects.

Varying the discretization granularity. We vary the dicretization granularity and the results are depicted in Figure 9a. Different discretization granularities are applied 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 9b. 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.

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

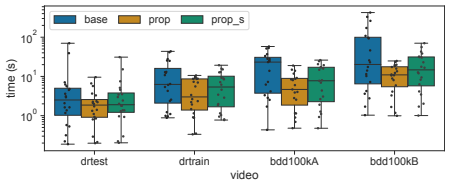
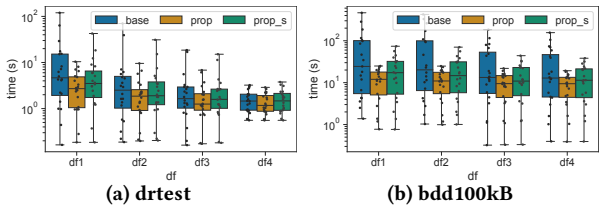


Figure 10: Query Time on Different Videos

queries from each video, issue these queries to the corresponding videos, and report the query processing time, as shown in Figure 10. Queries are generated following the rules presented in the Evaluation Methodology, in Section 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.

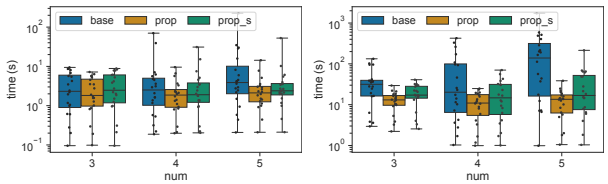
Both *prop* and *prop_s* are significantly faster than the *base* method, which can be observed from the median values of the three methods. The spread of the boxplot for *base* is also wider than the other two methods: for the queries in the bottom quarter of the boxplot, the processing times of these 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 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.



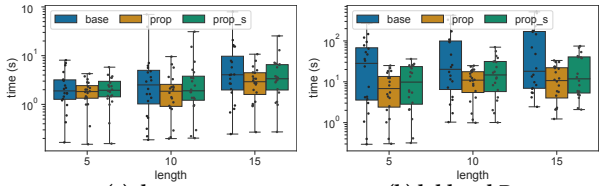
(a) drtest (b) bdd100kB

Figure 11: Varying *df*



(a) drtest (b) bdd100kB

Figure 12: Varying # of Unique Vertices

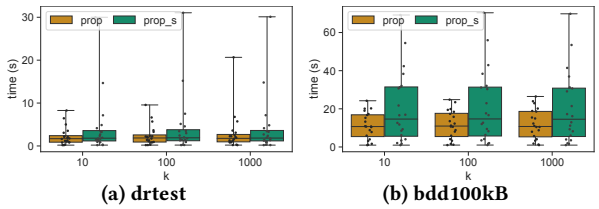


(a) drtest (b) bdd100kB

Figure 13: Varying the Length of Query Graph Sequences

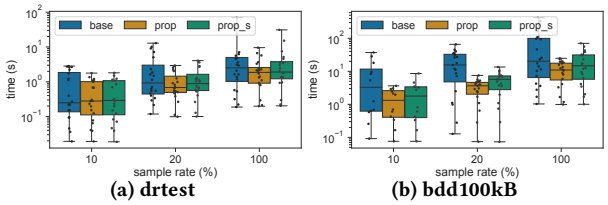
To evaluate the effect of discretization granularities, we build indexes with different granularities and measure the query processing time on the same set of queries. The results are shown in Figure 11. As we increase the number of buckets for edge attributes, the selectivity of the same query decreases, 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. 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 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.



(a) drtest (b) bdd100kB

Figure 14: Varying k



(a) drtest (b) bdd100kB

Figure 15: Varying Sample Rate

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 13. 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 13a) or a decreasing query processing time (Figure 13b, *base* method). In general, we can still see an increasing trend on all methods as we increase the query length. 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 14, 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 demonstrate 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 15. 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 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.

Summary. 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 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*).

7 RELATED WORK

Both the accuracy and efficiency have been improved in many computer vision tasks such as object detection [15, 17, 34, 35] and object tracking [4, 46, 47] utilizing Deep Learning models, enabling opportunities for declarative query processing over videos. Recent work has explored systematic ways of optimizing video queries leveraging cheaper DL models. For example, with specialized models and a cost-based optimizer, NoScope [25] achieves high efficiency on binary classification tasks. Similarly, BlazeIt [23, 24] further provides optimizations for aggregations and limit queries. To further improve the query efficiency, Focus [20] builds approximate indexes leveraging cheap NNs during the ingestion phase, while Kang et al. propose TASTI [26] to further accelerate proxy-based query processing algorithms (e.g., aggregations) via embeddings. Such works mainly focus on queries that can be answered either with single frames, or can be answered based on the aggregation of individual results evaluated on each frame in the window (i.e., object tracking algorithms are not involved). Queries with more complex conditions such as spatial constraints [27, 48], temporal constraints [8, 9], and interactions [6] have also been investigated. Such works can be generally summarized into two broad categories: one is to construct and train specialized DL models and utilize them as the basis for faster query evaluation [6, 27], in which case both the query accuracy and efficiency need to be considered; while another is to first annotate videos utilizing CV algorithms and then evaluate exact queries based on the annotations[8, 9], where the main focus is query efficiency. Our work falls into the second category, and we extend the semantics to further support spatial and temporal constraints on objects utilizing the graph data model.

Video indexing and retrieval have also been well studied in the multimedia literature [1, 21]. Hu et al. [21] outline a general framework consisting of six stages, among which two, namely *feature extraction*, and *query and retrieval* are most relevant to this paper. For feature extraction, most of the early work focuses on low-level features [5, 16, 51], 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 [28, 40]. Spatial-temporal information is considered either via trajectory-based methods [3, 50] or symbolic representation schemes [12]. In *query and retrieval*, existing work has considered queries issued using keywords, natural languages, or by examples (e.g., images, sketches). For queries using keywords and natural languages, the query results are based on the summarization or classification of videos [2], while for queries using examples, low-level features are usually considered in finding relevant videos [21]. Such works utilize similarity measures (e.g., feature matching, text matching, etc. [21]) to retrieve results based on the given query and do not support exact pattern matching as proposed in this paper. Recent advances also utilize end-to-end models to retrieve videos using a natural language interface [13, 30–32], which relies heavily on annotated datasets [49] and are tailored to specific query forms, such as natural languages, making it cumbersome to support new videos or new queries, especially to support queries with more precise constraints as adopted in our work.

Time-evolving graphs have been widely studied in application scenarios such as social networks [29, 42], where the main focus is either to identify sub-graphs that match the given pattern [14, 36, 37, 41], or perform data mining tasks such as interesting pattern mining [33, 42]. 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 the entire graph, which is infeasible in our setting.

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 [11, 38], 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 [38], or complex graphs that can be further decomposed into basic structures [22] or subgraphs [45]. In this work, 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.

8 CONCLUSIONS AND FUTURE WORK

In this paper, 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.

ACKNOWLEDGEMENT

This work was supported in part by NSERC Discovery Grants, NSFC Grant (No.62172351), and the Open Fund of Key Laboratory of Urban Land Resources Monitoring and Simulation, Ministry of Natural Resources. We would like to thank the anonymous reviewers for their valuable comments that have helped improve this paper.

REFERENCES

[1] Ahmad Sedky Adly, MS Abdelwahab, Islam Hegazy, and Taha Elarif. 2020. Issues and Challenges for Content-Based Video Search Engines A Survey. In *2020 21st International Arab Conference on Information Technology (ACIT)*. IEEE, 1–18.

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

[3] Faisal I Bashir, Ashfaq A Khokhar, and Dan Schonfeld. 2006. Real-time motion trajectory-based indexing and retrieval of video sequences. *IEEE Transactions on Multimedia* 9, 1 (2006), 58–65.

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

[5] Murray Campbell, Alexander Haubold, Shahram Ebadollahi, Dhiraj Joshi, Milind R Naphade, Apostol Natsev, Joachim Seidl, John R Smith, Katya Scheinberg, Jelena Tesic, et al. 2006. IBM Research TRECVID-2006 Video Retrieval System.. In *TRECVID*. 175–182.

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

[7] Kai Chen, Jiaqi Wang, Jiangmiao Pang, Yuhang Cao, Yu Xiong, Xiaoxiao Li, Shuyang Sun, Wansen Feng, Ziwei Liu, Jiarui Xu, Zheng Zhang, Dazhi Cheng, Chenchen Zhu, Tianheng Cheng, Qijie Zhao, Buyu Li, Xin Lu, Rui Zhu, Yue Wu, Jifeng Dai, Jingdong Wang, Jianping Shi, Wanli Ouyang, Chen Change Loy, and Dahua Lin. 2019. MMDetection: Open MMLab Detection Toolbox and Benchmark. *arXiv preprint arXiv:1906.07155* (2019).

[8] Yueting Chen, Xiaohui Yu, and Nick Koudas. 2020. TQVS: Temporal Queries over Video Streams in Action. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2737–2740.

[9] Yueting Chen, Xiaohui Yu, and Nick Koudas. 2021. Evaluating Temporal Queries Over Video Feeds. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 287–299.

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

[11] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.

[12] Alberto Del Bimbo, Enrico Vicario, and Daniele Zingoni. 1995. Symbolic description and visual querying of image sequences using spatio-temporal logic. *IEEE transactions on knowledge and data engineering* 7, 4 (1995), 609–622.

[13] Jianfeng Dong, Xirong Li, Chaoxi Xu, Shouling Ji, Yuan He, Gang Yang, and Xun Wang. 2019. Dual encoding for zero-example video retrieval. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 9346–9355.

[14] Maximilian Franzke, Tobias Emrich, Andreas Züfle, and Matthias Renz. 2018. Pattern search in temporal social networks. In *Proceedings of the 21st International Conference on Extending Database Technology*.

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

[16] Alex Hauptmann, Robert V Baron, Ming-yu Chen, Mike Christel, Pinar Duygulu, Chang Huang, Rong Jin, W-H Lin, T Ng, and Neema Moraveji. 2004. *Informedia at TRECVID 2003: Analyzing and searching broadcast news video*. Technical Report. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

[17] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 2961–2969.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[19] Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. 2019. Bag of tricks for image classification with convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 558–567.

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

[21] Weiming Hu, Nianhua Xie, Li Li, Xianglin Zeng, and Stephen Maybank. 2011. A survey on visual content-based video indexing and retrieval. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 41, 6 (2011), 797–819.

[22] Haoliang Jiang, Haixun Wang, S Yu Philip, and Shuigeng Zhou. 2007. Gstring: A novel approach for efficient search in graph databases. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 566–575.

[23] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. BlazeIt: optimizing declarative aggregation and limit queries for neural network-based video analytics. *Proceedings of the VLDB Endowment* 13, 4 (2019), 533–546.

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

[25] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1586–1597.

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

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

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

[29] Jure Leskovec, Lars Backstrom, Ravi Kumar, and Andrew Tomkins. 2008. Microscopic evolution of social networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 462–470.

[30] Xirong Li, Chaoxi Xu, Gang Yang, Zhineng Chen, and Jianfeng Dong. 2019. W2vv++ fully deep learning for ad-hoc video search. In *Proceedings of the 27th ACM International Conference on Multimedia*. 1786–1794.

[31] Jakub Lokoč, Werner Bailer, Klaus Schoeffmann, Bernd Münzer, and George Awad. 2018. On influential trends in interactive video retrieval: video browser showdown 2015–2017. *IEEE Transactions on Multimedia* 20, 12 (2018), 3361–3376.

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

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

[34] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[35] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems* 28 (2015), 91–99.

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

[37] Konstantinos Semertzidis and Evaggelia Pitoura. 2018. Top-k Durable Graph Pattern Queries on Temporal Graphs. *IEEE Transactions on Knowledge and Data Engineering* 31, 1 (2018), 181–194.

[38] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (2008), 364–375.

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

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

[41] Guohao Sun, Guanfeng Liu, Yan Wang, Mehmet A Orgun, and Xiaofang Zhou. 2018. Incremental graph pattern based node matching. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 281–292.

[42] Jimeng Sun, Christos Faloutsos, Spiros Papadimitriou, and Philip S Yu. 2007. Graphscope: parameter-free mining of large time-evolving graphs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 687–696.

[43] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[44] Longyin Wen, Dawei Du, Zhaowei Cai, Zhen Lei, Ming-Ching Chang, Honggang Qi, Jongwoo Lim, Ming-Hsuan Yang, and Siwei Lyu. 2020. UA-DETRAC: A New Benchmark and Protocol for Multi-Object Detection and Tracking. *Computer Vision and Image Understanding* (2020).

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

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

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

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

[49] Jun Xu, Tao Mei, Ting Yao, and Yong Rui. 2016. Msr-vtt: A large video description dataset for bridging video and language. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 5288–5296.

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

[51] Rong Yan and Alexander G Hauptmann. 2007. A review of text and image retrieval approaches for broadcast news video. *Information Retrieval* 10, 4-5 (2007), 445–484.

[52] Fisher Yu, Haofeng Chen, Xin Wang, Wenqi Xian, Yingying Chen, Fangchen Liu, Vashisht Madhavan, and Trevor Darrell. 2020. BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.