VQPy: An Object-Oriented Approach to Modern Video Analytics

Shan Yu [1] **Zhenting Zhu** [1] **Yu Chen** [1] **Hanchen Xu** [1] **Pengzhan Zhao** [1] **Yang Wang** [2] **Arthi Padmanabhan** [3] **Hugo Latapie** [4] **Harry Xu** [1]

Abstract

Video analytics is widely used in contemporary systems and services. At the forefront of video analytics are *video queries* that users develop to find objects of particular interest. Building upon the insight that video objects (*e.g.*, human, animals, cars, *etc.*), the center of video analytics, are similar in spirit to objects modeled by traditional object-oriented languages, we propose to develop an object-oriented approach to video analytics. This approach, named VQPy, consists of a front-end—a Python variant with constructs that make it easy for users to express video objects and their interactions—as well as an extensible backend that can automatically construct and optimize pipelines based on video objects. We have implemented and open-sourced VQPy, which has been productized in Cisco as part of its DeepVision framework.

1 Introduction

The widespread deployment of surveillance cameras and the expansion of online video platforms have resulted in a tremendous surge in video data. Harnessing this extensive video data for intelligent video analytics is vital for practical applications such as enhancing safety in smart cities, optimizing traffic management, and enabling autonomous driving, among others.

At the core of video analytics lies the concept of video queries, which serve as a crucial link between users and video data. Video queries enable users to specify and extract video objects or events that align with their particular interests. For instance, a traffic planner may wish to analyze patterns leading to traffic accidents, such as jaywalking or vehicles speeding past pedestrians. Likewise, a police officer might be interested in identifying suspicious activities, such as someone lingering in a restricted area or attempting to tamper with security equipment. Supporting video queries is a complex task because real-world video queries typically involve a combination of various traditional computer vision (CV) tasks, including image classification, object detection, object tracking, activity recognition, and more. Moreover, tailoring this combination of CV tasks to suit the unique query requirements of different users or applications adds an additional layer of complexity.

State-of-the-art. The state-of-the-art approaches for addressing video queries can be categorized into three primary methods: (1) constructing a pipeline by hand; (2) employing a SQL-like language; and (3) using a multimodal large language model (MLLM), which provides a versatile zero-shot solution capable of handling a wide range of video queries. Our main observation behind this work is that *video queries are concerned about video objects (such as humans, animals, vehicles, etc.) and their spatial and temporal interactions*. A key limitation in these existing techniques is the lack of an *object-based abstraction*, making it hard for them to both describe and optimize complex queries that center around the existence of and/or the relationship between a variety of video objects. We will elaborate on these approaches below; a detailed description of related work can be found in §A.

Handcrafting pipelines. The most widely adopted approach in industry for implementing a specific task is the manual crafting of pipelines that connect pretrained vision models. However, it can be a labor-intensive and error-prone process, demanding deep expertise in computer vision and significant engineering resources. In the construction of such a pipeline, CV experts are required to identify objects and analyze their relationships manually by selecting the appropriate models from model repositories (such as HuggingFace (Wolf et al., 2020), MMDetection (Chen et al., 2019), or various GitHub contributions), writing inference code to query these models, and creating programs to link these tasks together. This is a formidable undertaking and it needs to be completely redone for each new application.

SQL-based frameworks. To alleviate the laborious manual efforts involved in constructing and configuring pipelines, recent video database management systems (VDBMS), exemplified by (Lu et al., 2016; Kang et al., 2020; 2022; Xu et al., 2022), have introduced a high-level interface that permits expressive querying using a SQL-like language. This

[1]University of California, Los Angeles, California, USA [2]Intel, Santa Clara, California, USA [3]Harvey Mudd College, Claremont, California, USA [4]Cisco Research, San Jose, California, USA. Correspondence to: Shan Yu <shanyu1@g.ucla.edu>, Harry Xu <harryxu@cs.ucla.edu>.

Proceedings of the 7^{th} MLSys Conference, Santa Clara, CA, USA, 2024.

interface allows for the automatic construction of pipelines through SQL queries. However, SQL-based frameworks were originally designed for processing structured tabular data and are, therefore, not ideally suited for handling object-based video queries. This discrepancy introduces challenges in effectively expressing and executing video queries. To express complex queries that involve temporal and spatial relationships between objects (*e.g.*, scenarios like a car approaching a cyclist), a SQL-based approach requires developers to "think like a table"—they typically treat camera frames as if they were relational tables, with queries often involving complex nesting, joins and group-bys on these frame tables. Implementing such queries necessitates the use of many UDFs that must be coded in imperative languages like Python.

Moreover, the disparity between video objects in video queries and the structured data model used in SQL-based frameworks can result in suboptimal query optimization. The latter poses challenges in carrying out optimizations specifically focused on individual objects, such as memoization. For example, static properties (*e.g.*, color) of a video object remain unchanged once computed. As demonstrated in our evaluation, remembering the values of such properties for new frames (as opposed to recomputing them) can lead to a ten-fold performance increase (§5.2), while doing so in SQL is a formidable challenge.

Multimodal LLMs. The latest development in multimodal large language models (MLLMs) empowers users to interact with and comprehend videos using natural language queries. This approach offers a solution that appears to eliminate the necessity for constructing pipelines. While MLLMs show great promise in video understanding, they excel most in the realm of exploratory video analytics, where human users engage with the video query system to iteratively explore and gain a deeper understanding of the video content. They are unable to answer questions regarding video objects on a specific frame. Moreover, MLLMs come with a high computational cost, which can result in substantial delays. This, combined with the iterative process of generating responses, often renders the latency too long to be suitable for time-sensitive applications like real-time surveillance.

Insight. Our key insight is video objects are fundamentally similar to the objects modeled in traditional object-oriented programming languages like Java or Python. Consequently, the creation of a video-object-oriented query framework allows for the straightforward development of complex queries. Moreover, the video-object-oriented design enables optimizations *at the object level*, which can greatly enhance query performance.

VQPy. The original Python language lacks built-in support for representing object interactions, spatial and temporal relationships, and constraints on video frames. As a solution, we have created VQPy, which is a Python variant featuring constructs specifically tailored for expressing and modeling video objects and their relationships. VQPy places video objects at the core of all queries. These video objects exhibit inheritance relationships and possess properties and operations, much like those in a conventional object-oriented (OO) language. Basing VQPy off Python is intentional to tap into the extensive machine learning ecosystem, making it seamless for developers to construct end-to-end pipelines, ranging from queries to subsequent tasks.

VQPy makes the following contributions:

- **A video-object-oriented frontend:** VQPy employs an object-oriented approach to represent video objects and their interactions, enabling developers to create intricate queries with ease, without the need for additional code to connect various CV tasks or express object interactions through a relational data model for SQL. Additionally, our design embraces object-oriented concepts like inheritance and polymorphism, promoting code reusability and facilitating query composition.
- **An efficient backend with an object-centric data model:** Our backend is constructed based on a data model that revolves around video objects rather than relational tables. This approach streamlines the efficient execution of video-object-oriented queries, enabling the integration of numerous object-level computation reuses that were not feasible in a SQL-based framework primarily geared toward relational data.
- **An extensible optimization framework:** VQPy's optimization engine has been crafted as a flexible optimization framework, making it effortless to integrate diverse query optimizations like frame filtering and specialized neural networks (NNs) as plug-and-play operators, requiring minimal adjustments to the code.

Results. We have written 14 queries with VQPy and evaluated on 5 datasets from real-word surveillance video streams. On average, VQPy achieved more than 10× query speedup over state-of-the-art systems without sacrificing accuracy. We also implemented three optimizations commonly used in previous works showcasing VQPy's ability to integrate customized optimizations. VQPy has been open-sourced at *https://github.com/vqpy/vqpy* and integrated into DeepVision, which is Cisco's comprehensive video analytics framework, for commercial purposes.

2 VQPy Overview

As shown in Figure 1, VQPy's architecture contains three core components: frontend, backend, and library.

Frontend. The frontend of VQPy augments Python's capacity to articulate video queries through the introduction of

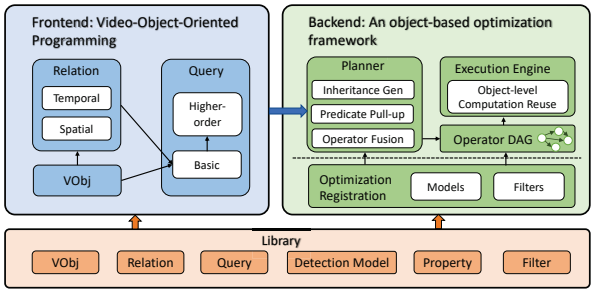


Figure 1. VQPy architecture.

three key constructs: `VObj`, `Relation`, and `Query`. `VObj` serves as the central abstraction within VQPy, defining the primary objects of interest within video data. `Relation` builds upon `VObj`, specifying spatial or temporal relationships among these objects, while `Query` further extends the concepts of `VObj` and `Relation` to define a comprehensive video query.

Backend. VQPy's backend framework uses an object-based data model, and comprises three fundamental components: operators, planner, and execution engine. When presented with a video query, our planner constructs a sequence of operators, such as object detection and object tracking, using a graph-based data model centered around `VObj`. Subsequently, the execution engine carries out the execution of this pipeline. Moreover, VQPy's backend simplifies the process of optimization registration, allowing users to effortlessly incorporate their filters and specialized neural networks for video objects into our backend through straightforward Python annotations. The planner, equipped with metadata from the property library and profiling data from the model zoo, then reorganizes the operators within the directed acyclic graph (DAG) to generate an optimized query plan that aligns with the users' specified accuracy targets.

Library. VQPy provides a library that encompasses a model zoo, which integrates state-of-the-art models both for common CV tasks including object detection, action recognition, and object tracking, and for specific property functions like license plate recognition or color detection. The models in the library can be selected to construct `VObj` and `Relation`. VQPy's library also provides commonly used `VObjs`, `Relations` and `Queries` that serve as building blocks for constructing other queries. Besides, VQPy's library also includes backend optimizations including specialized NNs and filters corresponding to the built-in `VObjs` and `Queries`.

3 Frontend: Video-Object-Oriented Programming

This section discusses VQPy's frontend, with a focus on how users can express queries in an object-oriented manner. As discussed in §2, to empower Python with video query abilities, VQPy extends Python's syntax with three major constructs: `VObj`, `Relation`, and `Query`. These constructs are similar to Python classes but carry special properties and constraints to ease the development of video analytics.

VObj. Much like an object in vanilla Python, `VObj` defines the video object type users want to query on (*e.g.*, vehicle, person, *etc.*). `VObj` supports the definition of *properties*, *e.g.*, the color of the car, which can be used in `Query` for detecting objects with such properties, such as a red car.

Figure 2 demonstrates how to construct a vehicle `VObj` with the properties "center", "direction", and "color". In `VObj`, properties can be built with other properties defined in the same `VObj` (*e.g.*, the "direction" property takes as input the "center" property of the vehicle `VObj`), or with pre-defined properties in `vqpy.VObj` (*e.g.*, `bbox`, `frame_rate`, `vobj_image`, *etc.*), or with properties in its super-`VObjs`.

```
vobj Vehicle(vqpy.VObj):

    def __init__(self):
        self.model = "yolox"
        self.class_names = ["car", "truck", "bus"]

    @stateless(input="bbox")
    def center(self, bbox):
        pass

    @stateful(input="center", history_len=5)
    def direction(self, hist_centers):
      # self-customized
      if hist_centers[0][0] - hist_centers[4][0] > 0:
          return "up"
      else:
          return "other directions"

    @stateless(model="color_detect")
    def color(self, images):
      # built-in color_detect model
      pass
```

Figure 2. VQPy Vehicle `VObj`.

In `VObj`, each property can be either *stateless* or *stateful*, indicating whether the property requires cross-frame information (*e.g.*, `direction`) or not (*e.g.*, `color`). If a property is stateless, that is, the property is static in the `VObj` and only depends on the current frame (such as "color" and "license"), users can modify the property with a `@stateless` annotation, which allows the developer to specify other properties within the same frame as dependencies. Similarly, to compute a stateful property, users can annotate it with a `@stateful` annotation, which takes as input the length of the history of its dependent property. In the Vehicle `VObj` example shown in Figure 2, the `direction` property of a car is a stateful property, and computing it requires five consecutive frames of the `center` property.

To build a `VObj` in VQPy, users can directly utilize the vision models from VQPy's library by referring to the model name. For example, the `Vehicle VObj` uses the built-in "yolox" as its model to detect vehicle objects, and the "color_detect" model to compute the color property. Besides, developers can also write customized code to define properties, such as the `direction` property in Figure 2.

A special VObj we provide is the scene VObj, which represents the scene of each frame. This can be used to define background properties, such as day or night, rainy or sunny, whether at an intersection, *etc.*

Relation. To ease the query development for object interactions, we introduce the Relation construct. Taking VObjs as input, Relation models spatial or temporal relations between the input VObjs. Similarly to properties on VObj, properties on Relation can be either stateful or stateless. Figure 3 demonstrates how to use Relation to construct a spatial relation between objects using simple python code.

```
relation SpatialRelation(vqpy.Relation):
  def __init__(self, vobj1, vobj2):
    pass

  @stateless(input1="center", input2="center"):
  def distance(centers):
    pass

  @stateful(input="distance", history_len=5):
  def getting_close(self, dists):
    return dists[0] - dists[4] < 0
```

Figure 3. VQPy spatial relation.

```
relation PersonBallInteraction(vqpy.HOIRelation):
  def __init__(self, person, ball):
    self.model = "UPT"
    self.object_class_name = "ball"

  @stateful(input="interaction", history_len=30):
  def serve(interactions):
    return hold_then_throw()
```

Figure 4. VQPy person ball relation.

Instead of hand-written python code, one could also build a property with a vision model. For example, in Figure 4, the PersonBallInteraction Relation uses the built-in human object detection model of "UPT" to compute the interactions between human and ball. The "interaction" property on RelationPersonBallInteraction builds a connection between a person and a ball. To construct such properties, users can select vision models from VQPy's library that can directly predict these properties on frames.

Query. Query is the main entry of a video query. With the Query construct, our goal is to enable query expressions to be semantically aligned with users' interests in video objects and their relationships. Figure 5 demonstrates how a police officer can construct a query of "retrieving the license plates of red cars" with VQPy. Figure 6 includes a more complex query on both video objects (a speeding car) and their spatial relationships (a car close to a person), where a city safety guard wants to query the traffic hazard case of a speeding car passing a person.

As shown in both examples, VQPy introduces two constructs, frame_constraint and frame_output. frame_constraint allows users to express their *filtering constraints* on video frames, while frame_output selects the output objects of interest. In particular, VQPy invokes frame_output to output a set of video objects whose

```
query FindRedCar(vqpy.Query):
  def __init__():
    self.car = Car()

  def frame_constraint():
    return self.car.color == "red"

  def frame_output():
    return self.car.license_plate
```

Figure 5. VQPy query for retrieving license plates of red cars.

```
query TrafficHazards(vqpy.Query):
  def __init__():
    self.car = Car()
    self.person = Person()
    self.relation = SpatialRelation(self.car, self.person)

  def frame_constraint():
    return (self.relation.distance < 0.1)
           & (self.car.speed > 60)
```

Figure 6. VQPy query for traffic hazards.

containing frames satisfy the constraints declared in frame_constraint.

Additionally, to enable queries over the *entire video*, we introduce video_constraint and video_output. Figure 7 depicts how to express a query of "counting the number of vehicles turning right throughout the video". A Query with video_constraint and video_output outputs the aggregated results, where the same object that appears in different frames will be regarded as one single entity.

```
query TrafficFlow(vqpy.Query):
  def __init__():
    self.vehicle = Vehicle()

  def video_constraint():
    return self.vehicle.motion == "turn_right"

  def video_output():
    return vqpy.count(self.vehicle)
```

Figure 7. VQPy query for traffic flow analysis.

Note that VQPy supports the use of logical operators (&, |, and ¬) to connect the predicates in a constraint. With logical operators, queries on video objects with the conjunction or disjunction of a number of predicates (*e.g.*, a person who wears jeans and whose hair is not black), as well as cocuring objects (*e.g.*, frames with a person and a red car) can all be easily expressed.

Inheritance. VQPy supports inheritance for VObj, Relation, and Query similar to standard python semantics. For example, a sub-VObj or sub-Relation can inherit a super-VObj/super-Relation, and thus all properties defined in the super-VObj/super-Relation are directly accessible in the sub-VObj/sub-Relation. A sub-Query can reuse the constraints of all its super-Query to construct a stricter constraint. Inheritance facilitates code reuse and allow VQPy to provide a library of basic VObj, Relation and Query for users to extend. Inheritance also provides a natural way to enable optimizations such as specialized NNs and frame filters, which we will detail in §4.4.

Event Composition with Higher-Order Queries. Event

composition allows users to express complicated queries by connecting basic queries. To support composition, VQPy provides higher-order queries that take other queries as input and extend the query dimension temporally or spatially. Specifically, VQPy offers three high-order queries: `DurationQuery`, `SpatialQuery`, and `TemporalQuery`.

A `DurationQuery` checks whether a condition defined in the base query continues to hold for a number of frames or a time period. It can express queries such as a person loitering for more than 20 mins, or a bag unattended for more than 5 mins. A `SpatialQuery` takes in two basic `Queries`, each containing a `frame_constraint` and a specific spatial relation. VQPy automatically generates a new `frame_constraint` for the `SpatialQuery` that checks whether the two video objects satisfy the specified spatial relationship. A `TemporalQuery` takes in two `Queries`, each containing a `frame_constraint` or `video_constraint` as well as a temporal relation. VQPy generates a `video_constraint` for the `TemporalQuery` that checks whether the two events satisfy the specified temporal relationship.

Query composition follows the following rules:
Rule 1: `SpatialQuery` takes in only basic queries;
Rule 2: `DurationQuery` takes in basic queries or `SpatialQueries`;
Rule 3: `TemporalQuery` takes in basic queries as well as all three higher-order queries (including itself).

Figure 8 depicts how a traffic safety analyst can use the higher-order query constructs of VQPy to implement a complicated query that searches for hit-and-run scenarios. The example includes two events, `car-hit-person`, and `car-run-away`, which happen sequentially. The `car-hit-person` query is concerned with the spatial relationship between a car object and a person object. The `car-run-away` query is interested in a car object with a cross-frame property ("speed").

```
from vqpy.lib.query import CollisionQuery, SpeedQuery
from vqpy.lib.query import SequentialQuery

query HitAndRun(vqpy.Query):
  def __init__():
    self.car = Car()
    self.person = Person()
    car_hit_person = CollisionQuery(
        subqueries = [self.car, self.person]
        dist_threshold = 0.1)
    car_run_away = SpeedQuery(car, velocity_threshold)
    self.sequential = SequentialQuery(
        subqueries=[car_hit_person, car_run_away],
        time_window="10s",
        id_match=(car_hit_person.car, car_run_away.car)
    )

  def video_constraint():
    return self.sequential

  def video_output():
    return self.car.license_plate
```

Figure 8. VQPy code for hit and run.

In the `HitAndRun` example, the user first employs `CollisionQuery`, a sub-`Query` of the higher-order `SpatialQuery` which checks whether the distance of the two input `VObjs` is smaller than a threshold indicating a potential collision, for building a car-hit-person query.

To construct this query, developers directly pass two `VObjs` Car and Person. They can then leverage VQPy's built-in `SpeedQuery`, to construct a `car_run_away` query, by specifying the Car `VObj` with a speed that exceeds `velocity_threshold`. With the two sub-queries defined, they can easily compose them into a `SequentialQuery`, a sub-`Query` of the higher-order `TemporalQuery`, by specifying a desired time window, which represents the maximum interval between the two events. The sequential query can used directly in the `video_constraint` of the `HitAndRun` query.

4 Backend: A Video-Object-Centric Optimization Framework

4.1 Plan Generation

As shown in Figure 1, VQPy's backend includes three components, *i.e.*, operators, query planner, and execution engine. The backend is designed with `VObjs` *and their relations* as its data model.

Data Model. Our query planner maintains a *graph* data structure, which flows across the operators on a DAG. Nodes in the graph represent `VObjs` and edges represent their relationships. There are four kinds of edges: (1) a motion edge connects two `VObjs` which represent an identical object from consecutive frames (to track stateful properties such as actions), (2) a spatial-relation edge connects two `VObjs` that are located in the same frame, (3) a duration-relation edge connects two `VObjs` in frames whose distance is within a given time constraint, and (4) a temporal-relation edge connects two `VObjs` such that the from-`VObj` is in a frame that precedes the frame that contains the to-`VObj`. The last three kinds of edges correspond to the three higher-order query types discussed earlier. Nodes and edges all carry properties.

Operators. VQPy supports queries with six types of operators: `video reader`, `frame filter`, `object detector`, `objector tracker`, `object filter`, and `projector`. `Object filter` includes `VObj` filter and `Relation` filter; `projector` includes `VObj` projector and `Relation` projector. `Video reader` reads in the video stream and passes the frame information to subsequent operators. `Frame filter` filters out irrelevant frames. For instance, a motion detector that filters out static frames can serve as a frame filter in the pipeline for answering queries including moving video objects; a texture-based filter can quickly eliminate frames that do not contain live objects such as humans or animals. `Object filter` filters outs `VObjs` or `Relations` that do not satisfy the user specified constraints.

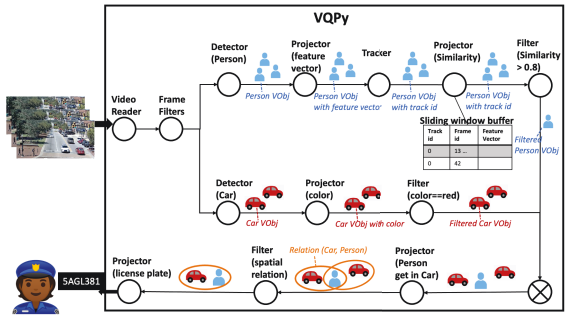


Figure 9. VQPy operator DAG for a query that searches for the suspect getting in a red car.

`Object detector` detects video objects and labels their classes. `Object tracker` identifies the same video object across different frames. Note that object tracker is used only when the query constraints involve `stateful` properties. `Projector` outputs the results for `VObj` or `Relation` involved in the query.

Operators are implemented as iterators. Each operator consumes the graph(s) passed from its previous operator, and outputs a new graph with updated nodes and/or edges. Specifically, `object detector` generates new nodes into the graph. Both `object tracker` and `projector` update edges. For example, `object tracker` adds motion edges between existing `VObjs`, and annotates them with `track_id`. `Object filter` removes nodes, spatial-relation edges, or temporal-relation edges that fail the user-specified constraints.

To support the computation of stateful properties on `VObj` or `Relation`, which requires the history data of their dependent properties, the stateful projector maintains a local sliding window of historical data of all of its dependencies. When executing the DAG, the executor generates frame batches (*i.e.*, the size of each batch is user-defined) and executes the pipeline on a per-batch basis.

Example DAG. Figure 9 shows how a pipeline DAG is constructed and executed to answer the query that searches for a suspect getting into a red car, and identifies the license plate of the car. The code for the query is illustrated in Figure 10.

A police officer uses the image of the suspect to find the person in the video. The example query has a spatial relation query of "person getting into car" (`PIntoC`), and two basic queries of a suspect person (`Suspect`) and a red car (`RedCar`), respectively. To find the suspect, our planner uses a human object detector that generates a graph with a single `Person VObj`, with two fields, a stateless `feature_vector` property that computes the feature vector of a person from each image, and a stateful `similarity` property that takes in the past 30 frames of the `feature_vector` property, to compare the distance between each person's feature vectors and those of the target suspect and determine whether they are the same person. To identify red cars, the planner uses a car

```
vobj Person(vqpy.VObj):
    @stateless(input="bbox")
    def feature_vector(self, bbox):
        pass

    @stateful(input="feature_vector", history_len=30)
    def similarity(self, feat_vectors):
        return dist(feat_vectors, SUSPECT_VEC).mean()

# red car query is omitted
query Suspect(vqpy.Query):
    def frame_constraint():
        return self.person.similarity > 0.8

query PIntoC(vqpy.Query):
    pass

query SuspectIntoCar(vqpy.Query):
    def __init__():
        self.spatial = (
            subqueries = [RedCar(), Suspect()],
            relation_q = PIntoC())

    def video_constraint():
        return self.spatial

    def video_ouput():
        self.car.license_plate
```

Figure 10. VQPy code for suspect getting in a red car.

detector that generates another graph with a single `Car VObj` containing a "color" property.

To construct the DAG, VQPy's planner first retrieves the dependencies of the nested query. The `PIntoC Query` depends on the `Suspect` and `RedCar Queries`, and therefore the planner places all the filters and projectors related to `PIntoC` after `Suspect` and `RedCar`. `Suspect` and `RedCar` have no dependencies between each other, so they can run in parallel. Next, the planner generates the detectors, trackers, projectors and filters for each query, according to the `VObjs` and `Relations` included in the frame/video constraints. Note that multiple projectors could be generated for one predicate, due to the dependencies between properties. In the example, two projectors for `feature_vector` and `similarity` property are generated for the `similarity > 0.8` predicate.

The planner generates multiple frame filters and places them before the computation-expensive detectors (car detector and person detector), including frame filters based on frame difference (*e.g.*, motion detector), and cheap filters corresponding to detectors (*e.g.*, texture-based car filter and person filter). Note that the `color==red` filter on the `car` path and the `similarity > 0.8` filter on the `person` path are `VObj` filters, that only filter out `VObjs` that falsify the constraints on properties, and cannot filter out frames. The join operator serves as a frame filter that filters out the frames without `RedCar` or `Suspect`, merges the information of person and car retrieved from the previous operators including computed properties and filtered `VObj` ids, and passes the graph with three (car and person) nodes to the spatial relation projector, which eventually adds two edges between these nodes.

Operators can be placed on different devices. For example, the compute-intensive object detector can be placed on a

GPU server while the low-cost object filter can be placed on an edge device (such as a camera). This design can easily support both offline batch and real-time streaming analytics.

4.2 Object-level Computation Reuse

The object-based data model used in our backend facilitates *object-level* computation reuse. In VQPy, each video object (`VObj`) represents a unique entity that appears across multiple frames, holding stateful and stateless properties. Reuse opportunities arise when a `VObj` possesses *intrinsic properties*, a special type of stateless property that remains constant across frames. Intrinsic properties are common in video queries—for example, an amber alert query may search for a red car with a license plate ending at "45" where both the red color and the license plate are intrinsic properties. VQPy enables users to annotate stateless properties as intrinsic using `intrinsic=True`, facilitating computation reuse at the video object level.

VQPy's backend tags each video object with a label indicating whether its intrinsic properties satisfy the query constraints. Due to the static nature of these properties, this label would never change once computed. When processing a new frame, VQPy uses a lightweight tracker based on the Kalman filter to identify `VObjs` on the frame. If a `VObj` has been detected before, VQPy directly utilizes the `VObj`'s intrinsic label for sub-queries that involve intrinsic properties while only sending newly-detected `VObjs` to the full computation pipeline. This optimization often leads to significantly improved computation efficiency.

In SQL-like video query frameworks, achieving such optimizations is unattainable because of the limitations imposed by their tabular data model. Under this model, each row in the tabular structure is treated as a separate entity, rendering the task of grouping rows based on objects challenging. Specifically, SQL frameworks lack a built-in concept of "objects", making it impossible to implement memoization strategies at the individual object level.

VQPy also supports query-level computation reuse where results from previous queries are materialized and reused when multiple queries are conducted on the same video, further improving efficiency.

4.3 DAG Optimization

Our planner performs three kinds of optimizations on the generated DAG: (1) operator fusion that fuses neighbor operators to reduce the overhead of executing each operator separately and minimize the intermediate data generation, (2) predicate pull-up that pulls filters to an early point of the pipeline, thereby reducing the amount of data that needs to be processed by the subsequent operators and saving computation costs, and (3) generating and comparing alternative optimization paths based on the *inheritance* relationships between video objects. Since the first two optimizations were used in prior systems such as (Xu et al., 2022) and (Xu et al., 2019), this subsection will focus specifically on the third optimization, which is a unique contribution of VQPy. To realize (3), our plan generates all possible execution DAGs from a given query, each corresponding to a potential execution pipeline. The planner then profiles each DAG using a short canary input video provided by the user —a technique also employed by systems like (Romero et al., 2022). The profiling helps compare the costs and accuracies of each DAG. During this process, we also identify the cost of each operator in each DAG, which can be used to perform intra-DAG optimizations such as (1) and (2). The planner selects the best plan that meets the target accuracy with the lowest cost (best runtime). This plan can be saved for future queries on similar datasets to save optimization time.

DAG optimizations require estimation of accuracy and performance. For accuracy estimation, we use common techniques from recent work (Kang et al., 2020; Bastani et al., 2020; Cao et al., 2021) that uses the original models to generate ground-truth labels. We use F1 score to estimate accuracy and compute an F1 score per DAG. To estimate each DAG's accuracy, VQPy runs the DAG with the most general models/filters and then other candidate DAGs over the canary input's frames and stores these results in a table. During query optimization, VQPy queries the table only with each DAG's predicates to produce a final set of labels. The results from the user's initial DAG are used as the ground-truth labels. Finally, the candidate DAG's F1 score is computed by comparing these labels.

We use a standard approach of estimating costs: we compute the latency of executing each DAG for a batch of input frames. The costs of different DAGs are compared to find the most efficient candidate (that still meets the accuracy).

4.4 Extension

We build our backend as an extensible optimization engine for easy integration of emerging video analytics optimizations. Basing VQPy off Python enables developers to easily integrate customized optimizations with Python annotations that can meet their desired performance-accuracy tradeoff. Here we showcase how to integrate three optimizations used commonly in previous works: (1) specialized NNs, (2) binary classifiers for objects, and (3) frame filters.

Specialized NNs. Specialized NNs are dedicated to the detection of specific objects and are often much less compute-intensive than general object detectors. Using specialized NNs first in the pipeline to filter out irrelevant frames is an effective optimization, which has been widely adopted in previous systems (Kang et al., 2021; 2017; Lu et al., 2018; Kang et al., 2022; Hsieh et al., 2018; Anderson et al., 2018).

```
vqpy.register(RedCarDetection, "my_red_car")

vobj RedCar(Car):
    def __init__(self):
        self.specific = {"color": "red"}
        self.model = "my_red_car"

    @filter():
    def no_red_on_road(self, frame_image):
        road_image = crop(frame_image, ROAD_REGION)
        return "red" in road_image
```

Figure 11. Optionally register models/filters on a `RedCar VObj`.

Figure 11 shows how to register a specialized NN for the `RedCar VObj`. Users need to first register their specialized NN (`RedCarDetection`) into VQPy's library by invoking the `register` function, so that they can use it in the `VObj` by referring to its name ("my_red_car"). Models from popular model zoos such as Huggingface, TorchVision, MMLab, and *etc.* are natively supported in VQPy and can be directly registered as specialized NNs.

Register specialized NNs in the super-`VObj` can lead to multiple query plans, thus providing additional optimization opportunities. For example, to detect a red car, we could directly employ the specialized red car detector registered with the red car `VObj`, or use the general car detector registered with its parent `VObj` car and then apply a red color filter. The planner can determine whether to use the specialized model by examining metrics such as the confidence scores of the output results and the cost of different execution paths.

Binary classifiers. Binary classifiers directly answer whether an object exists on a frame. With binary classifiers, frames with a low probability of including the target objects are discarded, improving computation efficiency. Binary classifiers have been used in multiple video optimization systems (Li et al., 2020; Lu et al., 2018; Yang et al., 2022).

Figure 11 also shows how to register a binary classifier (`no_red_on_road`) on the `RedCar VObj`, with a simple annotation `filter`. VQPy's planner takes this information to generate a corresponding filter operator and inserts it at the beginning of the pipeline. The filter inserted calls the `no_red_on_road` function to discard frames without any red cars at an early stage, improving computation efficiency.

Frame Filters. Differencing-based frame filters are an effective optimization adopted by many systems (Li et al., 2020; Kang et al., 2017; 2020); it filters out less informative frames that are close to the background or other frames.

```
vobj Scene:

    @filter():
    def similar_to_bg(self, frame_image):
        return diff(BACKGROUND, frame_image)<2.0

    @filter(n_prev=1):
    def similar_to_prev(self, frame_images):
        return diff(frame_image[0], frame_image[1])<1.0
```

Figure 12. Optionally register frame filters on the `Scene VObj`.

No.	NL Descriptions	CVIP standardized
Q1	"A green sedan is keeping straight."	"green sedan go straight"
Q2	"A green bus going straight down the street followed by a white car."	"green bus go straight"
Q3	"A red sedan runs down the street."	"red sedan go straight"
Q4	"A black sedan keeps driving forward."	"black sedan go straight"
Q5	"A large black SUV turns right."	"black suv turn right"

Table 1. Queries selected from CityFlow-NL.

Figure 12 shows an example for registering differencing-based frame filters to VQPy, which can be defined on VQPy's special `Scene VObj`. To define the `similar_to_prev` frame filter, which requires the results of a number of previous frames to compare against, users can specify such a number in the `filter` annotation.

5 Evaluation

We evaluated VQPy on 14 queries on 5 datasets from real-world surveillance video streams. Our evaluation results demonstrate that:

(1) VQPy achieves up to 12.6× query speedups compared to manually crafted pipelines on 5 complex vehicle retrieval queries (§5.1).

(2) VQPy achieves up to 12.3× speedups compared to EVA, the state-of-the-art SQL-based framework, on video object queries (§5.2).

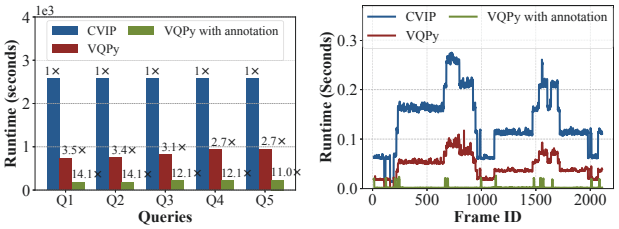
(3) Compared to a MLLM-based approach VideoChat, VQPy is 7× faster, requires 10× less GPU memory, and produces significantly (3-5×) higher accuracy (§5.3).

§5.4 briefly discusses how VQPy is adopted by a major tech company to help its customers develop complex queries.

5.1 Comparison with Handcrafted Pipelines

Dataset and Queries. Our experiments utilized the CityFlow-NL dataset (Feng et al., 2021) from the 2023 AI CITY CHALLENGE, Challenge Track 2 (Naphade et al., 2023), featuring 3.25 hours of traffic footage across 10 intersections from 40 cameras, with a minimum resolution of 960p at 10 frames per second. We evaluated on all 36 videos in the test set, which includes 184 vehicle tracks paired with natural language queries describing various vehicle attributes and scenarios. The original challenge was framed as a vehicle retrieval problem where vehicles for these queries are ranked. We repurposed the task into a video analytics problem which is concerned about the locations of the video frames that include the vehicles meeting the query constraints. Specifically, we randomly chose 5 queries from the CityFlow-NL dataset listed in Table 1.

Settings. We compared our method with CVIP (Le et al., 2023), the top prize winner in this Challenge Track. As detailed in Table 1, CVIP standardizes the natural language queries into a fixed format of *color-type-direction* during



(a) Runtime comparison. (b) Time spent on each frame.

Figure 13. Performance Comparison of VQPy and CVIP.

preprocessing. We adapted VQPy to use the same standlized query format, and evaluated the runtime performance of both VQPy and CVIP excluding the text standardization pre-processing step. Note that VQPy focuses on query development and optimization, *not* on model improvement (*i.e.*, for higher accuracy). For a fair comparison, we let VQPy use the same pre-trained vision models as used by CVIP in each query. We used two configurations for VQPy: the vanilla VQPy and VQPy with user-provided intrinsic annotations (see §4.2) for `color` and `type`. We evaluated both VQPy and CVIP on a Google Cloud virtual machine instance, equipped with one NVIDIA T4-16G GPU, 16 vCPUs (8 cores), and 104GB RAM.

Results. VQPy achieves the same accuracy as CVIP across all five queries, due to the use of the same pretrained models. Figure 13(a) compares the query execution time between VQPy and CVIP. Regardless of the query type, CVIP consistently requires around 850 seconds due to the necessity of processing *all* cropped images with all detection models (for color, type, and direction), resulting in a stable runtime. In contrast, VQPy employs filters after the computation of each property, efficiently filtering out vehicles that do not meet a property condition before computing other properties. For example, if the `color` property of a frame does not satisfy the condition (*e.g.*, red), VQPy will not proceed to the computation of the other properties. This approach significantly reduces unnecessary computations, resulting in an average **3.1**× speedup over CVIP, as shown in Figure 13(a). This performance gain stems from VQPy's object-centric approach (*e.g.*, properties can be computed and associated with individual video objects) as well as its use of *lazy evaluation*.

Specifically, VQPy demonstrates more pronounced performance gains for queries on green vehicles than black vehicles, as the former are less common in the dataset. This rarity results in more objects being filtered out early in the detection process, reducing unnecessary model inferences. Moreover, VQPy with user-specified intrinsic annotations outperforms vanilla VQPy by allowing reuse of computation results for `color` and `type`.

The detailed savings in per-frame computation time are reported in Figure 13(b), which depicts how the per-frame computation changes as frames are being processed. As seen, though spending much less time on each frame than CVIP (due to the use of lazy evaluation), the vanilla VQPy follows a similar computation curve while the intrinsic annotations flatten the curve, due to computation reuse. Overall, intrinsic annotations provide an additional **9.5**× speedup compared to the vanilla VQPy, bringing the improvement over CVIP up to **12.6**×.

5.2 Comparisons with SQL-based Frameworks

Baselines. We compared VQPy with the state-of-the-art SQL-based framework, EVA (Xu et al., 2022), whose optimizations subsume those in other frameworks such as (Kang et al., 2020; 2017; Lu et al., 2018). EVA is also the most well-developed framework, supporting the largest number of SQL primitives.

Queries. It is hard to construct video queries involving complex relationships between objects with EVA. Therefore we compared VQPy and EVA only with queries that count video objects with specific properties. We chose three types of queries: objects with stateless properties, objects with stateful properties, and objects with both kinds of properties.

Query type	Specific Query	Eva & VQPy expression
Stateless property	Red car	See Figures 18, 19
Stateful property	Speeding car	See Figures 20, 21
Stateless & stateful properties	Red speeding car	See Figures 22, 23

Table 2. Stateful and stateless properties to compare with EVA.

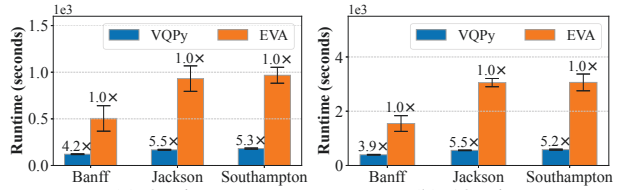
Table 2 summarizes the stateful and stateless properties we chose. For a fair comparison, we let VQPy use EVA's built-in YOLO detection model and nor-fair tracker. To detect color, we adopted an NN model from CVIP. The model can be easily annotated as a stateless property of color on the car `VObj`. To integrate the same model into EVA, we wrote a UDF to wrap the model around adapting the I/O (pandas Dataframes) formats required by EVA. To detect speed, we handcrafted a function using information from objects' bounding boxes. This function was used directly as a stateful property in VQPy; we turned it into a Python UDF to integrate it in EVA.

EVA lacks support for important SQL primitives such as window functions or group-by, which are necessary for forming the input, often a set of objects in consecutive frames, for stateful properties of video objects (*e.g.*, the speed of cars). To retrieve this historical data for each video object in Eva, we had to join the tables from different (*i-th*) lagged frames.

Appendix §B shows the actual query code under VQPy and EVA: it is clear that VQPy enables straightforward development of video queries while EVA presents users a relational structure requiring much more mental power to translate video objects and tables back and forth.

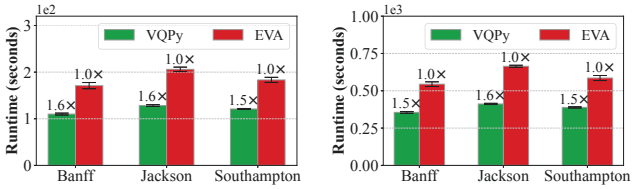
Camera location	FPS	Resolution
Banff, Candada (BanffLiveCam, 2019)	15	1280x720
Jaskson Hole, WY (SeeJacksonHole, 2019)	15	1920x1080
Southampton, NY (twinforkspestcontrol.com, 2019)	30	1920x1080

Table 3. Video datasets to compare with SQL-based frameworks.



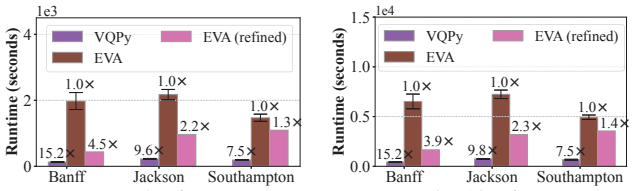
(a) 3 min. (b) 10 min.

Figure 14. Red Car Query: VQPy is averagely 4.9× faster.



(a) 3 min. (b) 10 min.

Figure 15. Speeding Car Query: VQPy is averagely 1.5× faster.



(a) 3 min. (b) 10 min.

Figure 16. Red Speeding Car Query: VQPy is averagely 11× faster.

Datasets. Table 3 summarizes our datasets, which come from public surveillance video streams deployed around North America and have been adopted by other works (Li et al., 2020; Xu et al., 2022; Kang et al., 2020; 2017) in evaluation. To evaluate how different video lengths affect the query execution time, we constructed the video datasets with two configurations: (1) 5 video clips of 3 minutes each, and (2) 5 video clips of 10 minutes each. Our evaluations were conducted on a Google Cloud virtual machine instance with an NVIDIA T4- 16G GPU, 16 vCPUs (8 cores), and 104GB RAM.

Results. Since EVA does not contain any frame filters or specialized NNs, we also disabled such optimizations in VQPy for a fair comparison. As such, query accuracy depends purely on the model chosen and the UDFs. By using the same models and UDFs, EVA and VQPy achieve the same accuracy. The performance comparisons are reported in Figure 14, Figure 15 and Figure 16, respectively, for the three queries. Each figure consists of the results for both 3-minute video clips (left) and 10-minute clips (right).

Stateless property. As shown in Figures 14(a) and 14(b), VQPy achieves an average of **5.0**× speedup on the short video clips and a **4.8**× speed up on the long clips. This is primarily due to the computation reuse across the frames for the same `VObj`, for its intrinsic `color` property.

Stateful property. As shown in Figure 15, VQPy is **1.5**× faster than EVA where the performance gain comes from EVA's requirement of using expensive table joins to compute the stateful property of `speed`.

Stateless and stateful property. EVA allows each query to contain only one single statement. As such, to implement queries on video objects with both stateless and stateful queries, we had to use query nesting. As shown in Figure 16, EVA is **7.5**-**15.2**× slower compared to VQPy. In addition to the aforementioned reasons, EVA does not support creating "VIEW" from queries, and hence filters used in later part of the query cannot be pushed to apply on earlier tables, leading to redundant executions of UDFs. We manually optimized EVA's SQL queries by pushing down the filters. Despite this optimization, EVA is still **3.3**-**5.7**× slower, due to its inability to perform object-level optimizations.

5.3 Comparisons with Multimodal LLMs

We have constructed a set of 6 queries to compare VQPy with VideoChat (Li et al., 2023), a state-of-the-art MLLM-based video understanding system. This set includes queries on specific video objects, aggregation queries, and those involving object interactions. VQPy achieves an average accuracy of **81.5%**, while VideoChat-13B of 42.6% and VideoChat-7B of 39.9%. In terms of the execution time, VQPy outperforms VideoChat by an overall of **7**×. Details of this evaluation can be found in §C.

5.4 Real-World Adoption and Use Cases

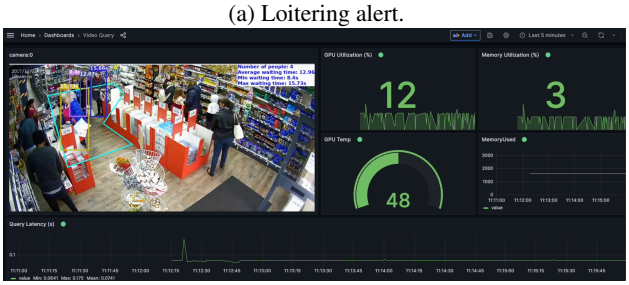
VQPy has been productized in Cisco as a query development/execution layer in its major vision product *DeepVision*, to enable its customers to develop and execute complex queries. DeepVision is a comprehensive video analytics system that enables users to monitor and analyze video streams from various sources with ease. It is a scalable and modular serverless open-source framework with state-of-the-art object detectors, trackers, and behavior detectors integrated.

VQPy was integrated into DeepVision as a service, executing queries over a real-time video stream from the company's video source service. VQPy's results are streamed back to DeepVision's dashboard for real-time monitoring. VQPy empowers DeepVision with the video query ability, orchestrating the underlying models' output, and automatically selecting the most efficient plan to construct and execute the pipeline, satisfying the real-time query latency requirements.

As of Oct 26 2023, VQPy has powered two critical user applications created for Cisco's Australian customers, *i.e.*, loitering and queue analysis. Loitering alerting is essential for smart city safety. Queue analytics plays a vital role

in retail store management. Figure 17 demonstrates how VQPy integrates with DeepVision and delivers the results on DeepVision's Graphana-based dashboard.

(a) Loitering alert.



(b) Queue analysis

Figure 17. Use cases of VQPy with DeepVision.

6 Conclusion

This paper presents VQPy, a video-object-oriented system we built for easy expression and optimization of complex video analytics queries. VQPy has an object-oriented frontend, enabling abstraction and modularity, as well as an extensible backend, allowing advanced developers to integrate optimizations into the VQPy pipeline. We have open-sourced VQPy, which is currently used in Cisco for a range of vision tasks.

Acknowledgments

We thank the anonymous MLSys reviewers for their thorough comments. This work is supported by NSF grants CNS-1907352, CNS-2007737, CNS-2006437, CNS-2106838, CNS-2128653, CNS-2330831, and two grants awarded by Cisco Research.

References

Anderson, M. R., Cafarella, M. J., Ros, G., and Wenisch, T. F. Physical representation-based predicate optimization for a visual analytics database. *CoRR*, abs/1806.04226, 2018. URL http://arxiv.org/abs/1806.04226.

BanffLiveCam. Banff live cam, alberta, canada, 2019. URL https://www.youtube.com/watch?v=9HwSNgcdQ7k.

Bastani, F. and Madden, S. Otif: Efficient tracker preprocessing over large video datasets. In *Proceedings of the 2022 International Conference on Management of Data*, pp. 2091–2104, 2022.

Bastani, F., He, S., Balasingam, A., Gopalakrishnan, K., Alizadeh, M., Balakrishnan, H., Cafarella, M., Kraska, T., and Madden, S. Miris: Fast object track queries in video. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pp. 1907–1921, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3389692. URL https://doi.org/10.1145/3318464.3389692.

Cao, J., Hadidi, R., Arulraj, J., and Kim, H. Thia: Accelerating video analytics using early inference and fine-grained query planning. *arXiv preprint arXiv:2102.08481*, 2021.

Cao, J., Sarkar, K., Hadidi, R., Arulraj, J., and Kim, H. Figo: Fine-grained query optimization in video analytics. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pp. 559–572, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3517857. URL https://doi.org/10.1145/3514221.3517857.

Chao, D., Koudas, N., and Xarchakos, I. Svq++: Querying for object interactions in video streams. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 2769–2772, 2020.

Chen, K., Wang, J., Pang, J., Cao, Y., Xiong, Y., Li, X., Sun, S., Feng, W., Liu, Z., Xu, J., Zhang, Z., Cheng, D., Zhu, C., Cheng, T., Zhao, Q., Li, B., Lu, X., Zhu, R., Wu, Y., Dai, J., Wang, J., Shi, J., Ouyang, W., Loy, C. C., and Lin, D. Mmdetection: Open mmlab detection toolbox and benchmark, 2019.

Chunduri, P., Bang, J., Lu, Y., and Arulraj, J. Zeus: Efficiently localizing actions in videos using reinforcement learning. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pp. 545–558, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3526181. URL https://doi.org/10.1145/3514221.3526181.

CityofAuburnAL. City of auburn toomer's corner webcam 2, 2022. URL https://www.youtube.com/watch?v=hMYIc5ZPJL4.

Feng, Q., Ablavsky, V., and Sclaroff, S. Cityflow-nl: Tracking and retrieval of vehicles at city scale by natural language descriptions. *arXiv preprint arXiv:2101.04741*, 2021.

Fu, D. Y., Crichton, W., Hong, J., Yao, X., Zhang, H., Truong, A., Narayan, A., Agrawala, M., Ré, C., and Fatahalian, K. Rekall: Specifying video events using compositions of spatiotemporal labels. *CoRR*, abs/1910.02993, 2019. URL http://arxiv.org/abs/1910.02993.

Ge, Z., Liu, S., Wang, F., Li, Z., and Sun, J. Yolox: Exceeding yolo series in 2021. *arXiv preprint arXiv:2107.08430*, 2021.

Gupta, S. and Malik, J. Visual semantic role labeling. *arXiv preprint arXiv:1505.04474*, 2015.

Hsieh, K., Ananthanarayanan, G., Bodik, P., Venkataraman, S., Bahl, P., Philipose, M., Gibbons, P. B., and Mutlu, O. Focus: Querying large video datasets with low latency and low cost. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 269–286, 2018.

Jocher, G. YOLOv5 by Ultralytics, 5 2020. URL https://github.com/ultralytics/yolov5.

Kang, D., Emmons, J., Abuzaid, F., Bailis, P., and Zaharia, M. Optimizing deep cnn-based queries over video streams at scale. *CoRR*, abs/1703.02529, 2017. URL http://arxiv.org/abs/1703.02529.

Kang, D., Bailis, P., and Zaharia, M. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. *Proc. VLDB Endow.*, 13(4):533–546, jan 2020. ISSN 2150-8097. doi: 10.14778/3372716.3372725. URL https://doi.org/10.14778/3372716.3372725.

Kang, D., Guibas, J., Bailis, P., Hashimoto, T., Sun, Y., and Zaharia, M. Accelerating approximate aggregation queries with expensive predicates. *Proc. VLDB Endow.*, 14(11):2341–2354, jul 2021. ISSN 2150-8097. doi: 10.14778/3476249.3476285. URL https://doi.org/10.14778/3476249.3476285.

Kang, D., Romero, F., Bailis, P., Kozyrakis, C., and Zaharia, M. Viva: An end-to-end system for interactive video analytics. In *Proceedings of the 12th Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org, January 2022. URL http://www.cidrdb.org/cidr2022/papers/p75-kang.pdf.

Le, H. D.-A., Nguyen, Q. Q.-V., Luu, D. T., Chau, T. T.-T., Chung, N. M., and Ha, S. V.-U. Tracked-vehicle retrieval by natural language descriptions with multi-contextual adaptive knowledge. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pp. 5510–5518, June 2023.

Li, K., He, Y., Wang, Y., Li, Y., Wang, W., Luo, P., Wang, Y., Wang, L., and Qiao, Y. Videochat: Chat-centric video understanding. *arXiv preprint arXiv:2305.06355*, 2023.

Li, Y., Padmanabhan, A., Zhao, P., Wang, Y., Xu, G. H., and Netravali, R. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pp. 359–376, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379557. doi: 10.1145/3387514.3405874. URL https://doi.org/10.1145/3387514.3405874.

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pp. 740–755. Springer, 2014.

Liu, X., Ghosh, P., Ulutan, O., Manjunath, B., Chan, K., and Govindan, R. Caesar: cross-camera complex activity recognition. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pp. 232–244, 2019.

Lu, Y., Chowdhery, A., and Kandula, S. Optasia: A relational platform for efficient large-scale video analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pp. 57–70, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450345255. doi: 10.1145/2987550.2987564. URL https://doi.org/10.1145/2987550.2987564.

Lu, Y., Chowdhery, A., Kandula, S., and Chaudhuri, S. Accelerating machine learning inference with probabilistic predicates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pp. 1493–1508, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3183751. URL https://doi.org/10.1145/3183713.3183751.

Naphade, M., Wang, S., Anastasiu, D. C., Tang, Z., Chang, M.-C., Yao, Y., Zheng, L., Rahman, M. S., Arya, M. S., Sharma, A., Feng, Q., Ablavsky, V., Sclaroff, S., Chakraborty, P., Prajapati, S., Li, A., Li, S., Kunadharaju, K., Jiang, S., and Chellappa, R. The 7th ai city challenge. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2023.

Poms, A., Crichton, W., Hanrahan, P., and Fatahalian, K. Scanner: Efficient video analysis at scale. *ACM Transactions on Graphics (TOG)*, 37(4):1–13, 2018.

Romero, F., Zhao, M., Yadwadkar, N. J., and Kozyrakis, C. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–17, 2021.

Romero, F., Hauswald, J., Partap, A., Kang, D., Zaharia, M., and Kozyrakis, C. Optimizing video analytics with declarative model relationships. *Proc. VLDB Endow.*, 16(3):447–460, nov 2022. ISSN 2150-8097. doi: 10.14778/3570690.3570695. URL https://doi.org/10.14778/3570690.3570695.

SeeJacksonHole. Jackson hole wyoming usa town square live cam, 2019. URL https://www.youtube.com/watch?v=1EiC9bvVGnk.

twinforkspestcontrol.com. Southampton traffic cam, 2019. URL https://www.youtube.com/watch?v=y3NOhpkoR-w.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Huggingface's transformers: State-of-the-art natural language processing, 2020.

Wu, Y., Lentz, M., Zhuo, D., and Lu, Y. Serving and optimizing machine learning workflows on heterogeneous infrastructures. *arXiv preprint arXiv:2205.04713*, 2022.

Xarchakos, I. and Koudas, N. Querying for interactions. *IEEE Transactions on Knowledge and Data Engineering*, 35(2):1977–1990, 2023. doi: 10.1109/TKDE.2021.3094997.

Xu, G. H., Veanes, M., Veanes, M., Musuvathi, M., Mytkowicz, T., Zorn, B., He, H., and Lin, H. Niijima: Sound and automated computation consolidation for efficient multilingual data-parallel pipelines. In *SOSP*, SOSP '19, pp. 306–321, 2019.

Xu, Z., Kakkar, G. T., Arulraj, J., and Ramachandran, U. Eva: A symbolic approach to accelerating exploratory video analytics with materialized views. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pp. 602–616, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3526142. URL https://doi.org/10.1145/3514221.3526142.

Yang, Z., Wang, Z., Huang, Y., Lu, Y., Li, C., and Wang, X. S. Optimizing machine learning inference queries with correlative proxy models. *arXiv preprint arXiv:2201.00309*, 2022.

Zhang, F. Z., Campbell, D., and Gould, S. Efficient two-stage detection of human-object interactions with a novel unary-pairwise transformer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 20104–20112, 2022.

Zhang, H., Ananthanarayanan, G., Bodik, P., Philipose, M., Bahl, P., and Freedman, M. J. Live video analytics at scale with approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 377–392, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang.

A Detailed Related Work

SQL-like languages. Treating video frames as relational tables, prior work on video analytics often uses SQL-like languages to express video queries. Optasia (Lu et al., 2016), BlazeIt (Kang et al., 2020), and EVA (Xu et al., 2022) allow users to register NN UDFs and query frames whose properties satisfy given predicates. These frameworks, however, cannot express complex queries that involve multiple objects across multiple frames.

Miris (Bastani et al., 2020) and OTIF (Bastani & Madden, 2022) are frameworks specially designed for queries requiring cross-frame positional information (of bounding boxes), like a speeding car or a car moving toward a person. However, they cannot process queries on actions involving posture details (inside a bounding box), like people falling. SVQ++ (Chao et al., 2020) and (Xarchakos & Koudas, 2023) are frameworks targeting interactions between objects; they can express queries like "a person throw a ball". Zeus (Chunduri et al., 2022) is another specialized framework supporting queries on object interactions.

EVA (Xu et al., 2022) and VIVA (Romero et al., 2022; Kang et al., 2022) support queries on actions, but *not* spatial and temporal relationships between video objects. Miris (Bastani et al., 2020) uses a customized `Speed` UDF to express queries involving cross-frame information such as `Speed(car) > 30 km/h`; SVQ++ (Chao et al., 2020) requires a customized a `THROW` UDF to express interactions between different objects (*e.g.*, a person throws a ball). If the predefined UDFs and models cannot satisfy a user's need, the user must hand-write UDFs to deal with raw frames, a task that typically requires an in-depth understanding of the underlin-

ing systems. VIVA (Romero et al., 2022) provides relation hints that allow users to express relationships between their own models and existing models in the framework.

Other query languages. Rekall (Fu et al., 2019) offers a Python library for video event specification, which allows users to iteratively specify and refine the temporal and spatial relationships of video segments. It cannot express queries including complex activities within or between objects, such as a person walking or riding a bicycle. Caesar (Liu et al., 2019) employs a text-based specification for users to express queries. It defines a vocabulary of basic actions such as "approach", "near", and "stop", and users can use these basic actions to compose a more complex action to use in a query. However, the vocabulary system is hard to extend and customize. Queries are constrained by the supported vocabulary.

Specialized models. Several video analytics systems use specialized models to either filter out frames to reduce the work of using expensive machine learning models or directly provide answers. Existing works mostly target specific query types such as selection (*e.g.*, NoScope (Kang et al., 2017) and PP (Lu et al., 2018)) or aggregation (*e.g.*, Blazelt (Kang et al., 2020) and ABAE (Kang et al., 2021)).

Model selection. Other works propose using multiple models with different performance-accuracy trade-offs to adapt to the dynamic nature of the videos. TAHOMA (Anderson et al., 2018) and NoScope (Kang et al., 2017) employs a cascading approach that can skip compute-expensive models by answering the query with cheap models on easy frames. THIA(Cao et al., 2021) shares the same idea but uses a single model with multiple exit points to explore the performance-accuracy trade-off. Figo (Cao et al., 2022), instead of cascading the models, uses an ensemble of models with the same architecture but different sizes and dynamically selects the most cost-efficient model for the current video chunk.

Runtime optimizations. Systems like Scanner (Poms et al., 2018), Jellybean (Wu et al., 2022), VideoStorm (Zhang et al., 2017), and LLAMA (Romero et al., 2021) focus on optimizations of end-to-end analytics pipelines via efficient use of heterogeneous hardware. These techniques do not focus on query expressions, and thus are also orthogonal to and can be integrated into VQPy.

B Query Code Comparisons between VQPy and EVA

This section compares three pairs of query programs (illustrated in Figure 18-19, Figure 20-21, and Figure 22-23) used to evaluate VQPy and EVA (§5.2). Programs in each pair are written in EVA and VQPy, respectively, to express the same query. As seen, the abstractions we provide in VQPy make it much easier to develop and understand a query while developers must "think like a table" when using a SQL-based framework, which is counter-intuitive and leads to code that is hard to understand and maintain.

C Comparisons with Multimodal LLMs

No.	Query Type	Statement
Prompt		You are an AI assistant. A human gives a video about traffic and asks questions about the humans and cars on it. You should give helpful, detailed, and polite answers.
Q1	Boolean	Are there any people passing the crosswalk?
Q2	Boolean	Are there any cars turning left at the crossing?
Q3	Boolean	Are there any red cars in the video?
Q4	Aggregation	Tell me the average number of cars on the crossing.
Q5	Aggregation	Tell me the average number of people that are walking.
Q6	Boolean	Is anyone hitting the ball in the image? Answer by yes or no.

Table 4. Three query sets (Q1-Q3, Q4-Q5, Q6) used to compare VQPy with VideoChat, and their natural language statement.

Queries and Datasets. We have constructed a set of 6 queries to compare VQPy with MLLM-based methods, summarized in Table 4. This set includes queries on specific video objects, aggregation queries, and those involving object interactions. We used Auburn (CityofAuburnAL, 2022), a public surveillance video that monitors the traffic at a crossroad, as our dataset for the first two queries. Table 3 includes some of its properties, and a sample frame from the video can be found in Figure 24. Since MLLM models consume a large amount of computation resources, we could only use a 10-minute clip in the daytime in the experiment. The first query set (Q1-Q3) contains three boolean queries asking for Yes/No responses. The second query set (Q4 and Q5) consists of aggregation queries asking for a floating value.

For Q6, we used an image dataset V-COCO (Gupta & Malik, 2015) to query on object relationships. This dataset, which contains 4532 images for testing, selects a subset of MS-COCO (Lin et al., 2014) images and annotates the human-object interaction. In our evaluation, we considered only one specific human-object pair and queried whether this interaction exists in each clip.

VideoChat. We compared VQPy with VideoChat (Li et al., 2023), a state-of-the-art multimodal LLM designed for video analytics, which can support both boolean queries and aggregation queries. However, it can only answer questions regarding the entire video, *not* individual frames. When asking questions regarding individual frames, VideoChat often provided irrelevant responses; an example of that is illustrated in Figure 25. When the length of the video grows, the consumption of GPU memory rapidly increases: we need at least 40GB GPU memory to process VideoChat-7b for a video clip as short as 540 frames of a 1920x1080 resolution. Moreover, the video embedding computation is slow, and this problem becomes more pronounced when we offload

```
import evadb

cursor = evadb.connect().cursor()
cursor.query("""
    LOAD VIDEO
        './video.mp4'
    INTO
        MyVideo;
""").df()
cursor.query("""
    CREATE FUNCTION
        Color
    IMPL
        './color.py';
""").df()
cursor.query("""
    SELECT
        id, YOLO.bbox
    FROM
        MyVideo
    JOIN LATERAL
        UNNEST(Yolo(data))
    AS
        YOLO(label, bbox, score)
    WHERE
        Color(Crop(data, YOLO.bbox)) = 'red'
    AND
        YOLO.label = 'car';
""").df()
cursor.query("DROP FUNCTION IF EXISTS Color;").df()
cursor.query("DROP TABLE IF EXISTS MyVideo;").df()
```

Figure 18. EVA SQL expressions for querying red cars.

```
import vqpy

vobj Car(vqpy.VObj):
    def __init__(self):
        self.model = "yolov8m"
        self.class_names = ["car"]

    @stateless(model="color_detect", intrinsic=True)
    def color(self, images):
        pass

query QueryRedCar(vqpy.Query):
    def __init__(self):
        self.car = Car()

    def frame_constraint(self):
        return (self.car.score > 0.6) \
               & (self.car.color == 'red')

    def frame_output(self):
        return (self.car.bbox,)
```

Figure 19. VQPy expressions for querying red cars.

parts of the model to the CPU to save GPU memory.

Consequently, we had to split the 10-minute video dataset into 600 one-second clips and query VideoChat on each clip.

Evaluation Setup. We evaluated both VQPy and VideoChat on a Google Cloud virtual machine instance, equipped with one A100-40G GPU, 12 vCPUs (6 cores), and 85GB RAM. We consiered two versions of VideoChat in our experiments: VideoChat-13B and VideoChat-7B. Since our GPU memory is insufficient to store both the entire VideoChat-13B model and the intermediate results, we enabled the low-resource mode when running the VideoChat-13B model, which uses 8-bit weight and offloads part of the video embedding computation to CPU memory.

Since VideoChat can only provide natural language replies, we had to analyze the query results ourselves. We carefully designed the clear statements (as shown in Table 4 to make the responses more regular. We used a pattern-based analyzer to resolve most of the responses and annotated the remaining manually. For unclear responses, we simply dropped these data points when computing accuracy.

For the first two sets of queries Q1-Q5, we used YOLOX (Ge et al., 2021) as VQPy's detection model. For the interaction query Q6, we used UPT (Zhang et al., 2022), one of the best two-stage models in the V-COCO dataset as VQPy's baseline model.

No.	VideoChat-7B	VideoChat-13B*	VQPy	VQPy-Opt
Pre	38.4	1071.0	N/A	N/A
Q1	72.4	656.3	**34.4**	
Q2	80.7	637.3	**32.9**	
Q3	85.1	563.7	**48.2**	**52.6**
Q4	116.9	848.6	**31.9**	
Q5	137.3	836.8	**35.4**	
Q6	3503.8	8183.5	112.4	**30.0**

Table 5. Execution time for VideoChat and VQPy (millisecond per frame). Note that VideoChat has a pre-computation phase that loads the video and computes its embedding. VQPy-Opt combines Q1-Q6 in a single execution with computation reuse enabled. VideoChat-13B*: Low resource mode due to GPU memory limit.

No.	Pr(positive)	VideoChat-7B	VideoChat-13B*	VQPy
Q1	21.7%	0.412	0.422	**0.902**
Q2	37.5%	0.382	0.360	**0.591**
Q3	46.1%	0.674	0.685	**0.915**
Q6	4.9%	0.130	0.237	**0.867**

Table 6. F-1 score for boolean queries. We also provide the positive sample rate in the dataset to reflect the structure of data. VideoChat-13B*: Low resource mode due to GPU memory limit.

Results. Table 5 reports the execution time for VideoChat and VQPy under these queries. VQPy outperforms VideoChat under all the settings and tests.

Table 6 shows the F-1 score for the boolean queries Q1, Q2, Q3 and Q6. For Q1-Q3, we obtained the ground truth by labeling them manually. For Q6, we used the V-COCO annotations to generate the result. We also provided the percentage of queries that have a Yes response, which can affect the F-1 score if the tested program has a fixed probability of misprediction on each type of response. To summarize,

```
import evadb

cursor = evadb.connect().cursor()
cursor.query("""
    LOAD VIDEO
        'video.mp4'
    INTO
        MyVideo;
""").df()
cursor.query("""
    CREATE FUNCTION
        Add1
    IMPL
        './add1.py';
""").df()
cursor.query("""
    CREATE FUNCTION
        Velocity
    IMPL
        './velocity.py';
""").df()
cursor.query("""
    CREATE TABLE
        TrackResult
    AS
        SELECT
            id, data, T.iid, T.bbox, T.score, T.label
        FROM
            MyVideo
        JOIN LATERAL
            UNNEST(EXTRACT_OBJECT(
                data, Yolo, NorFairTracker))
        AS
            T(iid, label, bbox, score);
""").df()
cursor.query("""
    CREATE TABLE
        TrackResultAdd1
    AS
        SELECT
            Add1(id, iid, bbox)
        FROM
            TrackResult;
""").df()
cursor.query("""
    SELECT
        trackresult.id, trackresult.iid, trackresult.bbox
    FROM
        TrackResult
    JOIN
        TrackResultAdd1
    ON
        trackresult.id = trackresultadd1.added_id
    AND
        trackresult.iid = trackresultadd1.cur_iid
    WHERE
        trackresult.label = 'car'
    AND
        Velocity(trackresult.bbox,
            trackresultadd1.last_bbox) > 1;
""").df()
cursor.query("DROP TABLE IF EXISTS MyVideo;").df()
cursor.query("DROP TABLE IF EXISTS TrackResult;").df()
cursor.query("DROP TABLE IF EXISTS TrackResultADD1;").df()
cursor.query("DROP FUNCTION IF EXISTS Add1;").df()
cursor.query("DROP FUNCTION IF EXISTS Velocity;").df()
```

Figure 20. EVA SQL expressions for querying speeding cars.

```
import vqpy
from getvelocity import get_velocity

vobj Car(vqpy.VObj):
    def __init__(self):
        self.model = "yolov8m"
        self.class_names = ["car"]

    @stateful(input="bbox", history_len=1)
    def velocity(self, bboxes):
        velocity = get_velocity(bboxes[0], bboxes[1])
        return velocity

query QuerySpeedingCar(vqpy.Query):
    def __init__(self):
        self.car = Car()

    def frame_constraint(self):
        return (self.car.score > 0.6) \
               & (self.car.velocity > 1.0)

    def frame_output(self):
        return (self.car.track_id, self.car.bbox,)
```

Figure 21. VQPy expressions for querying speeding cars.

VQPy exhibits superior accuracy, achieving an average F-1 score of **81.87%** across queries Q1, Q2, Q3, and Q6, far surpassing the F-1 scores of VideoChat-13B and VideoChat-7B, which are 42.6% and 39.95%, respectively.

Model	Average Response	Maximum Response
VideoChat-7B	Q4: 6.87; Q5: 6.78	Q4: 250; Q5: 414
VideoChat-13B*	Q4: 4.86; Q5: 4.95	Q4: 65; Q5: 100
VQPy	Q4: 0.89; Q5: 0.66	Q4: 3.3; Q5: 5.28

Table 7. Evaluation results for Q4 and Q5.
VideoChat-13B*: Low resource mode due to GPU memory limit.

Table 7 summarizes the evaluation results for the aggregation queries Q4 and Q5, revealing that VideoChat's answers are rather inaccurate. For VideoChat-13B, 73.7% queries are preserved for Q4 and 60.1% for Q5. For VideoChat-7B, these percentages are 64.2% and 53.2%, respectively. For Q4, there are never more than 4 cars on the crossing at the same time, but under both models, the average number reported by VideoChat exceeds this. For Q5, there are never more than 10 walking people in the video, but there are at least 15% of responses that return a value larger than 10.

Available Optimizations. Similar to VideoChat which allows users to ask multiple queries after uploading a video, we can also support this in VQPy to reduce unnecessary computations. We tested the VQPy pipeline that executes Q1-Q5 in a single execution, which results in an overall 3.4× speedup compared to executing them individually; details are shown in Table 5.

Another optimization we performed on Q6 is to extend our backend with specialized NNs and filters. We used a cheap detector (Jocher, 2020) to filter out frames without target objects, and trained a specialized model (following the ideas in (Xarchakos & Koudas, 2023)) to drop the frames that are unlikely to contain the required action. With these optimizations, we were able to obtain a further gain of **3.7**× with a 0.08 loss in the F1-score.

```
import evadb

cursor = evadb.connect().cursor()
cursor.query("""
    LOAD VIDEO 'video.mp4'
    INTO MyVideo;
""").df()
cursor.query("""
    CREATE FUNCTION Add1
    IMPL './add1.py';
""").df()
cursor.query("""
    CREATE FUNCTION Velocity
    IMPL './velocity.py';
""").df()
cursor.query("""
    CREATE FUNCTION Color
    IMPL './color.py';
""").df()
cursor.query("""
    CREATE TABLE
        TrackResult
    AS
        SELECT
            id, Color(Crop(data, bbox)), T.iid,
                T.bbox, T.score, T.label
        FROM
            MyVideo
        JOIN LATERAL
            UNNEST(EXTRACT_OBJECT(
                    data, Yolo, NorFairTracker))
        AS
            T(iid, label, bbox, score);
""").df()
cursor.query("""
    CREATE TABLE
        TrackResultAdd1
    AS
        SELECT
            Add1(id, iid, bbox)
        FROM
            TrackResult
""").df()
cursor.query("""
    CREATE TABLE
        TrackResultJoin
    AS
        SELECT
            trackresult.id, trackresult.iid,
            trackresult.color, trackresult.bbox,
            trackresult.label, trackresult.score,
            trackresultadd1.last_bbox
        FROM
            TrackResult JOIN TrackResultAdd1
        ON
            trackresult.id = trackresultadd1.added_id
        AND
            trackresult.iid = trackresultadd1.cur_iid;
""").df()
cursor.query("""
    SELECT
        id, iid, bbox
    FROM
        TrackResultJoin
    WHERE
        Velocity(bbox, last_bbox) > 1
    AND
        color = 'red' AND label = 'car';
""").df()
cursor.query("DROP TABLE IF EXISTS MyVideo;").df()
cursor.query("DROP TABLE IF EXISTS TrackResult;").df()
cursor.query("DROP TABLE IF EXISTS TrackResultAdd1;").df()
cursor.query("DROP TABLE IF EXISTS TrackResultJoin;").df()
cursor.query("DROP FUNCTION IF EXISTS Add1;").df()
cursor.query("DROP FUNCTION IF EXISTS Velocity;").df()
cursor.query("DROP FUNCTION IF EXISTS Color;").df()
```

Figure 22. EVA SQL expressions for querying red speeding cars.

```
import vqpy
from getvelocity import get_velocity

vobj Car(vqpy.VObj):
    def __init__(self):
        self.model = "yolov8m"
        self.class_names = ["car"]

    @stateless(model="color_detect", intrinsic=True)
    def color(self, images):
        pass

    @stateful(input="bbox", history_len=1)
    def velocity(self, bboxes):
        velocity = get_velocity(bboxes[0], bboxes[1])
        return velocity

query QueryRedSpeedingCar(vqpy.Query):
    def __init__(self):
        self.car = Car()

    def frame_constraint(self):
        return (self.car.score > 0.6) \
                & (self.car.color == 'red') \
                & (self.car.velocity > 1.0)

    def frame_output(self):
        return (self.car.track_id, self.car.bbox,)
```

Figure 23. VQPy expressions for querying red speeding cars.

Figure 24. Sample image from Auburn video.

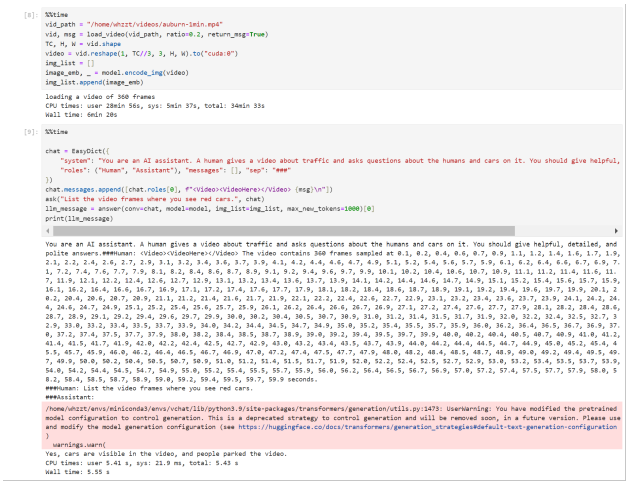


Figure 25. VideoChat is not able to solve long video on-frame queries.