AVQS:Advanced Video Querying System for Machine Learning Applications

Vikas Kumar, University of Windsor, Ontario, Canada, kumar3s@uwindsor.ca
Brendan Gignac, University of Windsor, Ontario, Canada, gignac31@uwindsor.ca
Abdulrauf Gidado, Algoma University, Ontario, Canada, abdulrauf.gidado@algomau.ca
Naman B. Shah, University of Windsor, Ontario, Canada, shah362@uwindsor.ca
Krish J. Shah, University of Windsor, Ontario, Canada, shah662@uwindsor.ca

***Abstract*—The increasing volume of video data, exemplified by datasets such as YouTube-8M, presents significant challenges to data engineering processes, particularly in efficiently extracting relevant information for model training. Existing video querying systems often fall short in providing a comprehensive solution, lacking the ability to fetch relevant video sections and frames, analyze videos for effective searching, and store data in a centralized location. These limitations are particularly problematic for enterprises with large datasets, often composed of multiple sources, where new projects frequently emerge. Such projects require specific types of videos, such as those featuring cars or animals. Currently, meeting these changing needs requires reprocessing existing datasets or relying on previously assigned tags, which may not capture the full range of complex interactions. In response to these challenges, we propose a video querying system that enables precise searches within large-scale video datasets. Our solution employs Yolo11 for object detection and tracking, generating detailed metadata that includes bounding box coordinates and timestamps. This metadata is stored in a MongoDB database, allowing users to perform complex queries, such as identifying when a person and a car are within 10 pixels of each other. By optimizing the retrieval of relevant video segments and enhancing the tagging process, our system aims to meet the evolving needs of enterprises while leveraging the current technological capabilities of the Yolo11 model.**

I. INTRODUCTION

With the increasing volume of video data, massive datasets like YouTube-8M [1] present significant challenges to data and machine learning engineering processes due to their size and complexity. Processing entire datasets for training artificial intelligence (AI) models is highly resource intensive and often impractical. This underscores the need for an advanced system that can effectively filter and retrieve only the most relevant video segments for targeted machine learning purposes.

Current video querying solutions offer basic features like object detection and simple temporal tagging but lack sophisticated capabilities such as spatial querying and detecting interactions between multiple objects. They are unable to handle complex queries, such as identifying when a person and a car are within a specific proximity or when two objects appear in a scene together within a specific time frame. This project addresses these limitations by creating an efficient system that filters massive datasets and extracts specific segments needed for AI model training.

The growing complexity of video data requires precise and efficient querying tools to facilitate the training of AI models. Such tools are essential in domains like autonomous driving, surveillance, and content recommendation. By enabling precise data extraction, our system aims to reduce computational resources and accelerate AI development cycles, leading to more efficient and accurate machine learning processes.

This work aims to develop a comprehensive video querying system capable of performing precise spatial-temporal queries and detecting complex object interactions. This system is designed to serve as a targeted data filtering mechanism, enhancing the efficiency of AI training workflows and reducing resource consumption.

A. Contributions

The primary objectives of the work are:

- Enable Precise Temporal and Spatial Queries: Allow users to search for moments in videos based on time intervals and spatial locations.
- Support Complex Object Interaction Queries: Facilitate searches involving multiple objects and their interactions.
- Provide Efficient Data Filtering for Machine Learning: Serve as a system to filter data segments needed for training, conserving resources, and optimizing workflows.

The system integrates state-of-the-art object detection and tracking technologies into a queryable video database application. It involves:

- Video Ingestion: Videos are uploaded and processed using Yolo11 [2] to detect objects.
- Metadata Generation: Detailed metadata, including spatial and temporal tags, is stored in MongoDB [3].
- Query Engine: Supports temporal and spatial searches based on the stored metadata.
- Feature Engineering: Detailed metadata, bounding box coordinates, and timestamps are generated, with spatial indexing (using R-trees) and temporal indexing implemented to enable efficient querying.

The AVQ video management system involves:

- Video Upload and Storage: Video is divided into smaller chunks and stored in MongoDB GridFS [4], with metadata saved in MongoDB [3].
- Metadata and Querying: Users can query the video data through a query processor that responds with relevant metadata and video access links.

- Video Retrieval: The user can fetch specific video chunks based on the query.

By providing a system that performs complex temporal and spatial queries and efficiently retrieves video data, this project addresses a critical open-sourced need in managing large-scale datasets for machine learning, making it a valuable tool for engineers and researchers alike.

II. Literature Review

Currently, there is no open source framework that efficiently retrieves relevant video segments based on content analysis. Existing closed solutions focus mainly on video analysis [5], [6], returning metadata that users must store and process independently. This requires the creation of a pipeline to segment videos into required portions and convert them into frames, which is a repetitive and resource-intensive process [7]. Without such pipelines, significant computational resources may be wasted processing irrelevant content, often forcing users to restart the analysis from scratch for each new use case.

The closest tool to address this issue is Scanner [8], which provides flexibility and allows the integration of various machine learning models. Scanner [8] efficiently processes large video datasets, making it suitable for applications such as 3D pose estimation, VR video synthesis, and large-scale video data mining. However, it still requires reprocessing the entire video when use cases change, despite having techniques to minimize irrelevant frames.

Major platforms like Google and IBM offer video analysis services but lack integrated storage, querying, and streaming solutions. Clients must upload videos to these platforms and expend resources developing their own solutions for these tasks.

In the realm of video database management systems (VDBMS), there are several legacy systems that allow content-based querying of video frames [2], [9]–[15] (explored in this Section II of this work). However, many of these platforms have become obsolete due to advances in video analysis techniques and codecs that offer superior compression. Consequently, they have failed to adapt to evolving technological standards.

- VDBMS [9]: This platform introduces video as a fundamental data type, supporting image similarity search and video streaming, but primarily focuses on storage and does not serve data in formats conducive to machine learning, returning the entire video instead.
- BilVideo [10]: Their analysis process, termed ”fact-extraction,” is semi-automatic, requiring users to manually specify objects in video frames using minimum bounding rectangles (MBRs). However, as an older platform, it lacks modern analysis capabilities.
- VIMS [11] This system employs a completely manual process for analysis and object detection, making it less efficient.
- YOLO11 [2] leverages transformer-based attention mechanisms to enhance spatial-temporal feature extraction, boosting accuracy in complex video analysis. Its anchor-free design improves object localization, while neural architecture search (NAS) fine-tunes the model's structure for optimal performance. Multi-scale and multi-query processing further elevate detection across diverse scenes, ensuring robustness in varying contexts.
- TransVOD [12] utilizes spatial-temporal transformers for object detection and tracking, emphasizing relevant feature extraction through attention mechanisms.
- TransVOD++ [16] enhances feature fusion, improving spatial and temporal integration for better contextual understanding. Its refined attention mechanisms dynamically focus on critical features, while multi-scale processing enhances detection of varying object sizes. Advanced training methodologies further improve generalization.
- TransMOT [13] specializes in multi-object tracking using graph transformers, effectively capturing spatial relationships but lacking comprehensive querying capabilities.
- BoostTrack [14] emphasizes real-time tracking using efficient algorithms like DeepSORT, excelling in accuracy but limited in broader detection tasks.
- AQATrack [15] focuses on single-object tracking, achieving high accuracy but struggling with interactions among multiple objects.

Fig. 1. Screenshot Demonstration of BilVideo System

Table I provides the summary of the differences between the systems explored above and our proposed AVQS system.

YOLO11 leverages neural architecture search (NAS) and transformer-based models to enhance object detection accuracy in video queries. By integrating attention mechanisms, YOLOv11 ensures precise localization and efficient real-time video analysis. Its flexible, anchor-free detection system, combined with transformers for improved query handling, sets a new benchmark for performance in dynamic video and image analysis tasks [2].

III. Proposed AVQS System

A. Explanation of the Model

This work aims to develop a comprehensive video querying system designed to enable precise spatial-temporal queries

TABLE I. THIS TABLE SHOWS THE DIFFERENT FEATURES PROVIDED BY THE EXISTING PROJECTS THAT ARE RELEVANT TO OUR PROBLEM SPACE

	TransVOD	TansVod++	TransMOT	BilVideo	VIMS	VDBMS	Online Platforms- Video Analysis	Our Solution
Analysis	Y	Y	Y	N(Manual)	(TAg Based)	Y	Y	Y
Storage	N	N	N	Y	Y	Y	N	Y
Metadata Query	Y	Y	Y	Y	Y	Y	N	Y
Metadata storage	N	N	N	Y	Y	Y	N	Y
Streaming	N	N	N	Y	Y	Y	N	Y
Chunk Streaming	N	N	N	N	N	N	N	Y

and support complex multi-object interactions in large-scale video datasets. The core idea of this system is to extract and index relevant metadata for each video frame, including object locations, timestamps, and interactions, making it possible to search for specific events based on user-defined criteria.

B. Workflow Overview

The system workflow consists of the following key stages.

- Video Ingestion: Users upload videos to the system through APIs. Videos are pre-processed and segmented for transfer efficiency and relevant segment retrieval.
- Object Detection and Metadata Generation: Using YOLO11 [2], the system detects objects in each frame, identifies their spatial coordinates (bounding boxes) and timestamps their occurrences. Then we process this data to track the object over the video using our Algorithm described in the next section.
- Metadata Storage: The generated metadata, including spatial-temporal data and object identifiers, is stored in MongoDB [3].
- Query Processing: Users can input spatial-temporal queries through a query engine. This engine processes queries by searching the stored metadata and retrieving relevant video segments based on user-defined conditions.
- Result Presentation: The system provides access to the retrieved video segments or specific frames that match the query criteria.

C. Workflow Explanation

The workflow diagram illustrates the architecture and operation of the Video Management System.

- User Requests to Upload a Video: The user initiates the process by requesting to upload a video. The user provides a title description of the video.
- Save Video Details in MongoDB [3]: The program saves the user-provided data (such as video title, upload date, etc.) in the MongoDB [3] database in collection videos. The server responds to the user with the generated ID.
- Upload: The user uploads the video to the upload link and provides the video ID.
- S3: Once the video is uploaded, it is sent to s3 so that other services like Analyzer and then Fragmenter can access it.
- Analyzer: Processes the video to extract meaningful metadata using YOLO11. The extracted metadata only contains information about object locations and relative positions. We created our algorithm that uses *Intersection* over *union* as the backbone to track objects over time.
- More processing pipelines can be added here along with an Analyzer that specializes in different categories based on the needs and processing power available.
- Fragmenter: This divides the video into smaller 5-second chunks and stored efficiently in MongoDB GridFS [4]. GridFS further breaks these fragments into 5 MB chunks, which minimizes the loss.

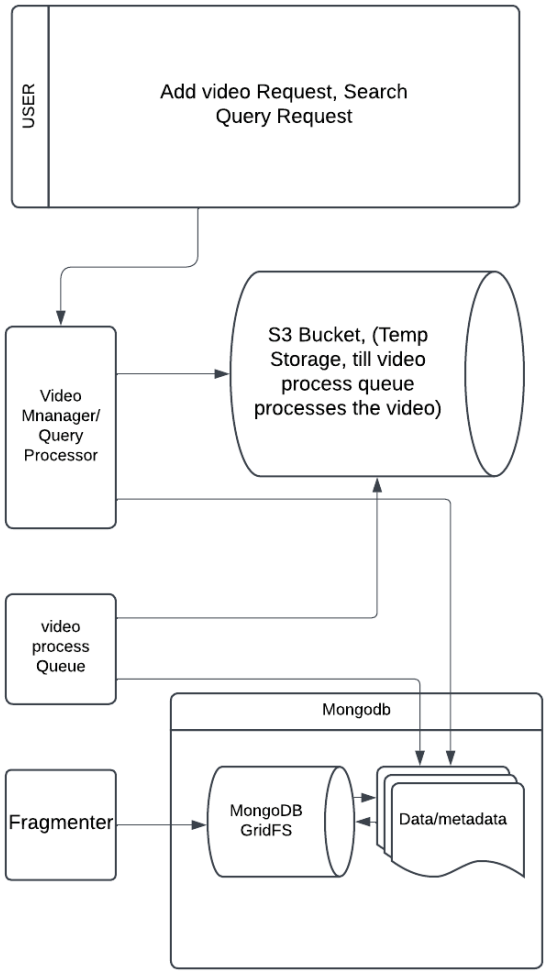

Fig. 2. Architecture Diagram

Thereafter, the video is available to query. We created five

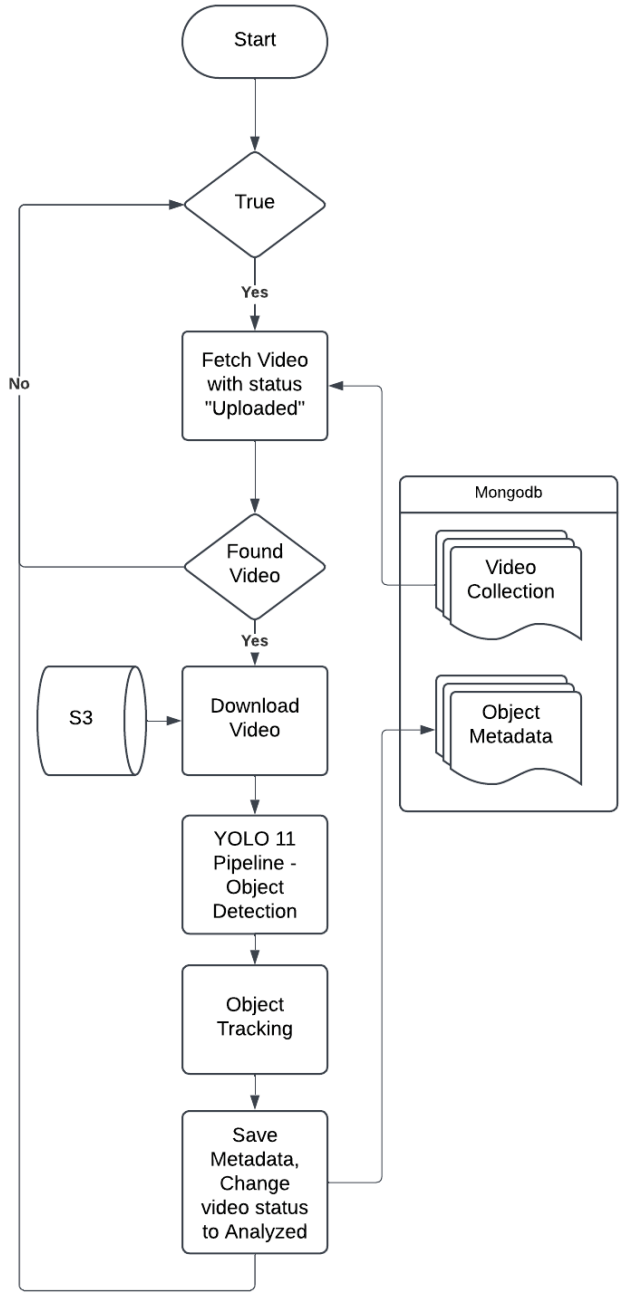

Fig. 3. Analysis Pipeline Program Flowchart

types of queries that can solve many kinds of problems related to searching in video. itemsep=5pt

- User Queries through APIs: We provide REST APIs that the user has to use in order to get the relevant timestamps of the video.
- The queries utilize objects collection of MongoDB which contains the metadata. This metadata is processed on both MongoDB and query processor which is written in Nodejs.
- Download video: The user can use this timestamp value to download the videos, only the relevant section of the video will be returned to the user.

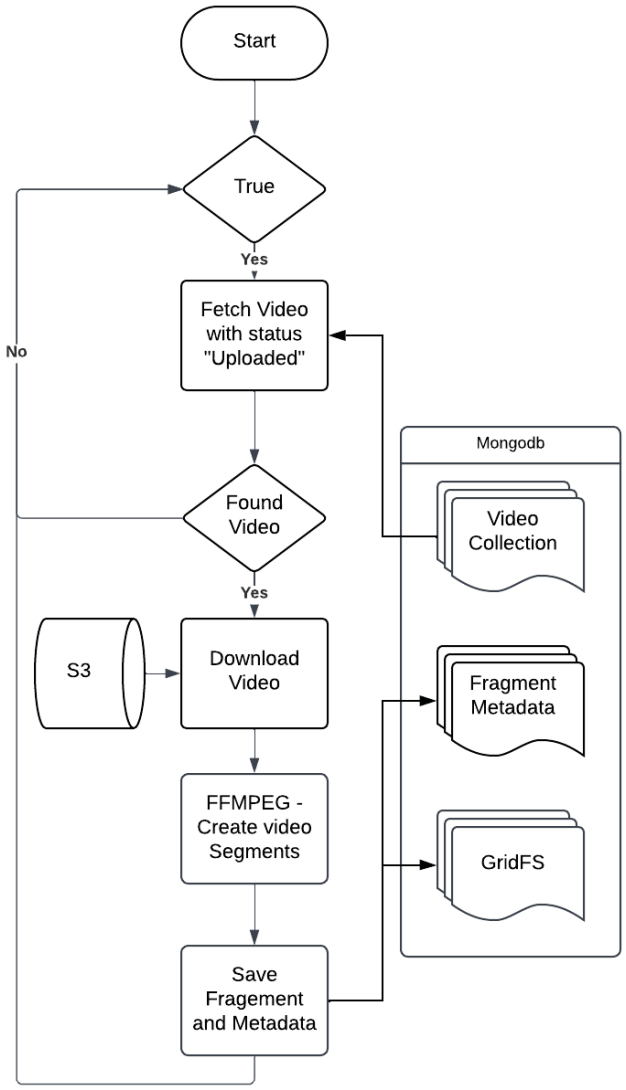

Fig. 4. Fragmenter Pipeline Program Flowchart

The diagram emphasizes the integration of video storage, efficient metadata management, and a robust querying process to achieve precise and fast video retrieval based on user-defined spatial and temporal conditions. This architecture ensures that large videos are managed efficiently, enabling seamless upload, storage, and querying operations

D. Differences from Existing Works

Existing commercial video querying services, notably Google Cloud Video Intelligence API [17] and IBM Watson Video Enrichment [18], primarily focus on basic general object detection and simple tagging but do not offer advanced querying capabilities. They do not support user-defined complex queries involving multi-object interactions, precise

TABLE II. OBJECTS COLLECTION SCHEMA STORING DETECTED OBJECT INSTANCES AND THEIR TEMPORAL METADATA

Field Name	Type	Short Explanation
_id	String	Unique identifier for each object instance
video_id	String	Reference to the source video containing the object
event_name	String	Class name of detected event (e.g., person, dog, bird)
event_type	String	Based on the model used for query (e.g., Object, Speech-to-text, Posture)
start_time	Number	Start timestamp of object appearance in seconds
end_time	Number	End timestamp of object disappearance in seconds
frames	Array	Collection of frame data with timestamps and bounding boxes

TABLE III. FRAME DATA STRUCTURE WITHIN OBJECTS COLLECTION STORING PER-FRAME DETECTION DETAILS

Field Name	Type	Short Explanation
frame	Number	Frame number within the video sequence
timestamp	String	Timestamp in HH:MM:SS.mmm format when object appears
box	Array	Bounding box coordinates [x1, y1, x2, y2] in pixels
relative_position	Array	Normalized center position [x_center, y_center] (0-1 scale)
confidence	Number	Detection confidence score from YOLO model (0-1 range)

spatial proximity, or movement tracking across regions and time intervals among more specialized complex use cases.

The proposed AVQS model differs by providing:

- Customizable Spatial-Temporal Queries: Users can search for moments based on both spatial regions within video frames and specific time intervals.
- Multi-Object Interaction Detection: The system allows users to define and search for interactions between multiple objects.
- Context-Aware Spatial Region Queries: Enables context-aware searches based on specific regions and actions within those regions.
- Efficient Data Filtering for ML Applications: It filters and extracts only the required video segments, reducing the computational load for targeted AI training.

E. AVQS Features

- Precise Spatial and Temporal Querying: The system can identify objects based on user-defined spatial coordinates within a video frame and specific timestamps. Example: ”Find moments when a car enters the top-left quadrant between 2 and 3 minutes.”
- Multi-Object Interaction Detection: Enables searches for specific interactions between objects, such as ”Find all moments when a person and a bicycle are within 10 pixels of each other.”
- Context-Based Spatial Region Queries: Supports querying based on objects’ movements and actions within specific regions of the frame. Example: ”Identify moments when a person enters the bottom-right quadrant and remains there for 5 seconds.”
- Efficient Indexing and Metadata Storage: Uses a combination of spatial (R-trees) and temporal (B-tree) indexing for efficient data storage and retrieval.

F. Metadata Table for Complex Spatial-Temporal Queries

The metadata table serves as a crucial feature of the video querying system, as it efficiently organizes and indexes information about events detected within the videos. Each row in the table represents a specific event, object, or action that occurs in a video, along with key details necessary for precise spatial-temporal querying.

Here’s an explanation of the table II and III components: itemsep=5pt

- Event Name: Identifies the type of event or object (e.g., ”ball” or ”jump”).
- Event Type: Specifies whether the entry is an ”object” (stationary or moving) or an ”action” (a specific event like jumping).
- Video ID: A unique identifier for the video where the event occurs, allowing for cross-referencing between different videos in the database.
- Start Time and End Time: These columns denote the time interval in the video where the event is observed. For example, the ”ball” appears between seconds 1 and 3 in the video, and the ”jump” action is observed between seconds 5 and 7.
- Metadata: Provides additional detailed context about the event, such as ”subtitles,” which could include descriptions or dialogue related to the event.
- Relative Position Start and End: Records the spatial coordinates of the event at the start and end times. This information is essential for tracking movements or interactions within the frame. For instance, the coordinates (X, Y) help the system determine where in the frame the ”ball” appears and whether it moves or remains stationaryc.

In the example Metadata Schema table, we see two events: A ball is seen from 1 to 3 seconds in the video, located at specific coordinates within the frame. A jump action occurs from 5 to 7 seconds, representing an action that can be queried based on time. The metadata table enables the system to perform complex queries such as: ”Find all instances where a ball is within a specific region of the frame between seconds 1 and 3.” The screenshot of the table effectively showcases the

TABLE IV. FRAGMENTS COLLECTION SCHEMA FOR GRIDFS-STORED VIDEO SEGMENTS WITH TEMPORAL METADATA

Field Name	Type	Short Explanation
_id	ObjectId	Unique identifier for each video fragment file
filename	String	Original name of the video fragment file
contentType	String	MIME type of the fragment (e.g., video/mp4)
length	Number	Size of the fragment file in bytes
chunkSize	Number	Size of individual chunks for GridFS storage
uploadDate	Date	Timestamp when fragment was uploaded to GridFS
metadata	Object	Custom metadata including video ID and timing information
metadata.videoID	String	Reference to parent video for fragment association
metadata.duration	Number	Duration of fragment in seconds
metadata.startTime	Number	Start timestamp within parent video
metadata.endTime	Number	End timestamp within parent video

organized structure of the event metadata. By linking spatial-temporal data with detailed object and action information, this feature allows the system to efficiently handle and retrieve specific video segments based on user-defined conditions. This approach significantly enhances the accuracy and flexibility of the video querying process.

IV. TECHNICAL DETAILS AND ALGORITHMS

A. Object Tracking

1) Overview : The provided implementation performs object tracking in videos using the YOLO (You Only Look Once) object detection model. The key objectives are:

- Detect objects in each video frame.
- Maintain a consistent identity for each detected object across frames.
- Handle multiple instances of the same object type.

Key Components:

2) Object Detection: The YOLO model detects objects in each frame and provides the following:

- Bounding boxes B represent the location of detected objects in the frame.
- Class labels L indicating the type of object (e.g., person, car).
- Confidence scores C for each detection.

3) Active Object Tracking: Detected objects are stored in an `active_objects` dictionary, which tracks each object's state, including:

- A unique identifier instance_id.
- The last frame and timestamp the object was seen.
- The bounding box B of the object.

4) Intersection over Union (IoU): To associate detections across frames, the Intersection over Union (IoU) metric is used. IoU measures the overlap between two bounding boxes B_1 and B_2. Mathematically, it is defined as:

$$\text{IoU} = \frac{|B_1 \cap B_2|}{|B_1 \cup B_2|}, \tag{1}$$

where $|B_1 \cap B_2|$ is the area of intersection and $|B_1 \cup B_2|$ is the area of the union of the two bounding boxes. An IoU threshold τ is used to determine if two bounding boxes represent the same object.

5) Timeout and Object Expiry: To handle objects disappearing from the frame temporarily, a timeout threshold T_{timeout} is used. If an object is not detected for a duration exceeding T_{timeout}, it is removed from the `active_objects` dictionary.

6) Handling Multiple Instances of the Same Object Type: When multiple objects of the same class are present, the implementation distinguishes them using the following steps:

7) Label-Based Grouping: Detected objects are grouped by their class label L. Each class label maintains its own list of active objects.

8) IoU-Based Association: For each detected object in the current frame, the IoU is computed with all active objects of the same label. Let B_t be the bounding box of a detected object at time t, and B_{t-1} be the bounding box of a previously tracked object. If:

$$\text{IoU}(B_t, B_{t-1}) > \tau, \tag{2}$$

then B_t is associated with the same object as B_{t-1}.

9) New Object Creation: If no existing object satisfies the IoU threshold, a new object is created with a unique identifier instance_id and added to `active_objects`.

10) Relative Position Calculation: The relative position of an object in the frame is calculated as:

$$\text{Relative Position} = \left(\frac{x_{\text{center}}}{W}, \frac{y_{\text{center}}}{H}\right), \tag{3}$$

where $x_{\text{center}} = \frac{x_1+x_2}{2}$ and $y_{\text{center}} = \frac{y_1+y_2}{2}$. Here, x_1, y_1, x_2, y_2 are the coordinates of the bounding box, and W, H are the frame width and height, respectively.

11) Data Storage and Updates: Each object's data is stored in a MongoDB collection, including:

- Bounding box coordinates B.
- Confidence score C.
- Relative position.
- Start and end timestamps.

When an object is matched in a new frame, its entry in the database is updated with the latest data.

V. RESULTS

For the proposed AVQS, we implement query endpoints that enable precise video contents retrieval. We also report the algorithms and data structures used (QuadTree, Interval Tree, sweep line, greedy sequence) with their threshold

policies, MongoDB indexing strategy, and caching behavior for reproducible performance. Users can specify where the object should be present in the video by using the predefined positions: ”top-half”, ”bottom-half”, ”left-half”, ”right-half”, ”top-third”, ”middle-third-horizontal”, ”bottom-third”, ”left-third”, ”middle-third-vertical”, ”right-third”, ”top-left”, ”top-right”, ”bottom-left” and ”bottom-right”.

A. Spatial Object Queries

Endpoint:
`/query/spatialObjects`

This endpoint enables querying for objects within specified spatial regions. It implements a logical OR operation to find instances of any specified objects in the defined area.

Complexity and thresholds: For each instance, frame scanning is O(F). For large frame sets (F¿100), a per-instance QuadTree is built in O(F) and queried in approximately O(k) where k is matches; for smaller sets, a linear scan is faster. Windows are merged by sorting and scanning.
Algorithm (Threshold-Based QuadTree):

```
SpatialQueryOR(objects[], area):
  results ← []
  FOR EACH obj IN objects DO
    instances ← DB.find(
                  {object_name: obj})
    FOR EACH inst IN instances DO
      frames ← inst.frames
      IF frames.length > 100 THEN
        quadTree ← BuildQuadTree(frames)
        matches ← quadTree.query(area)
      ELSE
        matches ← LinearScan(frames, area)
      windows ← BuildTimeWindows(matches)
      results.append({inst, windows})
  RETURN MergeWindows(results)
```

Data structures used: QuadTree for spatial pruning when F>100; otherwise linear scan.
Time complexity: Build O(F) per instance; query ≈ O(k); merge O(W log W). Space O(F) transient per instance (tree discarded after query).
Parameters:

- `objects`: Array of object names (e.g., [”truck”, ”handbag”])
- `area`: Predefined area (halves/thirds/quadrants) or coordinate array [x1, y1, x2, y2]

Example queries:

```
/query/spatialObjects?objects=["bird"]
&area=top-half
Response: [
    {
        "video_id": "689cd3905c1945e3f7602f42",
        "object_name": "bird",
        "windows": [
            {
                "start_time": "00:00:01.840",
                "end_time": "00:00:04.920"
            }
        ]
    },
    {
        "video_id": "689cd3905c1945e3f7602f42",
        "object_name": "bird",
        "windows": [
            {
                "start_time": "00:00:01.880",
                "end_time": "00:00:02.680"
            }
        ]
    }
]
  /query/spatialObjects?objects=["person"]
  &area=bottom-right
  Response: [
      {
          "video_id": "689cd3905c1945e3f7602f42",
          "object_name": "person",
          "windows": [
              {
                  "start_time": "00:00:01.040",
                  "end_time": "00:00:09.200"
              }
          ]
      },
      {
          "video_id": "689cd3905c1945e3f7602f42",
          "object_name": "person",
          "windows": [
              {
                  "start_time": "00:00:21.560",
                  "end_time": "00:00:34.560"
              }]}]
```

B. Spatial Objects AND Query

Endpoint:
`/query/spatialObjectsAnd`

This endpoint implements logical AND operations to find video segments where multiple objects appear simultaneously in the specified area.

Complexity: Intersecting time windows is performed via sorting and two-pointer scan: O(W log W) for sorting + O(W) scan.
Algorithm:

```
SpatialQueryAND(objects[], area):
  individualResults ← []
  FOR EACH obj IN objects DO
    windows ← SpatialQueryOR([obj], area)
    individualResults.append({obj, windows})
  intersection ← IntersectTimeWindows(
                      individualResults)
  RETURN intersection
```

Method: Interval intersection via sort-and-scan; avoids $O(W^2)$ pairwise checks.
Complexity details: Sort O(W log W); scan O(W) overall; space $O(O \times W)$ to hold per-object windows.
Parameters:

- `objects`: Array of object names (minimum 2 objects required)

- `area`: Predefined area or custom box coordinates

Example queries:

```
/query/spatialObjectsAnd?objects=["person","dog"]
&area=bottom-half
Response: [
    {
        "video_id": "689cd3905c1945e3f7602f42",
        "objects": [
            {
                "object_names": [
                    "person",
                    "dog"
                ],
                "windows": [
                    {
                        "start_time": "00:00:56.440",
                        "end_time": "00:06:02.200"
                    }
                ]
            }
        ]
    }
]
```

```
/query/spatialObjectsAnd?objects=["person","bird"]
&area=right-half
Response: [
    {
        "video_id": "689cd3905c1945e3f7602f42",
        "objects": [
            {
                "object_names": [
                    "person",
                    "bird"
                ],
                "windows": [
                    {
                        "start_time": "00:01:00.120",
                        "end_time": "00:05:01.960"
                    }
                ]
            }
        ]}]
```

C. Instance-Based Queries

1) Distinct Instances Query: **Endpoint:** `/query/queryDistinctInstances`

Retrieves all individual instances of a specified object class across all videos.

Example query:

```
/query/queryDistinctInstances?object=person
Response: {
    "person": [
        {
            "_id": "67c7456d-b4d4-4311-b28f-5a7f8fb64ecc",
            "video_id": "689cd3905c1945e3f7602f42",
            "object_name": "person",
            "start_time": 0.01,
            "end_time": 9.2
        },
        {
            "_id": "a8805091-c22e-40b6-b8f9-caf325571808",
            "video_id": "689cd3905c1945e3f7602f42",
            "object_name": "person",
            "start_time": 13,
            "end_time": 21.16
        },
        ]}
```

2) Instance Overlaps Query: **Endpoint:** `/query/queryInstanceOverlaps`

Identifies temporal windows where multiple instances of the same object class appear simultaneously.

Algorithm: Sweep line over start/end events (end-before-start tie-break). Complexity O(I log I) for sorting, O(I) scan.

```
QueryInstanceOverlaps(object, count):
  instances ← DB.find(
                {object_name: object})
  events ← []
  FOR EACH inst IN instances DO
    events.append({time: inst.start_time,
                   type: 'start', inst})
    events.append({time: inst.end_time,
                   type: 'end', inst})
  events.sort(by time; tie: end first)
  active ← ; overlaps ← []
  current ← NULL
  FOR EACH e IN events DO
    IF e.type = 'start' THEN
      active.add(e.inst)
      IF |active|  count AND
         current = NULL THEN
        current ← {start: e.time,
                   instances: copy(active)}
    ELSE
      IF |active|  count AND
         current  NULL THEN
        current.end ← e.time
        overlaps.append(current)
        current ← NULL
      active.remove(e.inst)
  RETURN MergeContiguousOverlaps(overlaps)
```

End-before-start tie-break prevents counting abutting intervals as overlaps. Space O(I) for events.

Parameters:

- `object`: Object class name
- `count`: Minimum number of simultaneous instances

Example queries:

```
/query/queryInstanceOverlaps?object=bird&count=2
Response: {
    "bird": [
        {
            "video_id": "689cd3905c1945e3f7602f42",
            "merged_overlaps": [
                {
                    "start_time": 1.88,
                    "end_time": 2.68
                },
                {
                    "start_time": 21.24,
                    "end_time": 23.36
                },
            ]}]}
```

```
/query/queryInstanceOverlaps?object=person&count=2
Response: {
    "person": [
        {
            "video_id": "689cd3905c1945e3f7602f42",
            "merged_overlaps": [
                {
                    "start_time": 41.88,
                    "end_time": 44.68
                },
                {
                    "start_time": 55.24,
                    "end_time": 59.36
                },
            ]}]}
```

3) Instance Overlaps in Area Query: **Endpoint:** `/query/queryInstanceOverlapsInArea`

Extends the instance overlaps query by adding spatial constraints, finding multiple instances of the same object class within a specified area.

For each overlap window, frames are verified against the specified area; success intervals are returned when the count threshold is satisfied across frames within the area.

Example queries:

```
/query/queryInstanceOverlapsInArea?object=bird
&count=3&area=bottom-half
Response: {
    "success": true,
    "data": [
        {
            "video_id": "689cd3905c1945e3f7602f42",
            "success_intervals": [
                {
                    "start_time": 21.24,
                    "end_time": 23.36
                }
            ]}]}
  /query/queryInstanceOverlapsInArea?object=person
  &count=2&area=left-half
  Response: {
      "success": true,
      "data": [
          {
              "video_id": "689cd3905c1945e3f7602f42",
              "success_intervals": [
                  {
                      "start_time": 41.88,
                      "end_time": 44.68
                  },
              ]}]}
```

D. Temporal Queries

Endpoint:

`/query/queryInstancesAtTime`

Retrieves all instances of specified objects at a particular timestamp.

Algorithm and thresholds: Instances are grouped by video. If instances per video ≥ 20, a center-based Interval Tree is used with point query $O(\log I + k)$; otherwise a linear scan is used. The closest frame to the target time is found via binary search $O(\log F)$.

```
QueryInstancesAtTime(object_name, time):
  instances ← DB.find({object_name})
  groups ← GroupByVideo(instances)
  results ← []
  FOR EACH (video_id, arr) IN groups DO
    IF arr.length  20 THEN
      tree ← intervalTreeCache.getOrBuild(
                video_id, arr)
      matches ← tree.queryPoint(time)
    ELSE
      matches ← LinearFilter(arr, time)
    FOR EACH m IN matches DO
      frame ← BinarySearchFrame(
                 m.frames, time)
      results.append({m, frame})
  RETURN results
```

Data structures: Interval Tree (cached for up to 100 videos), Binary Search for frame lookup.
Complexity: Build O(I log I) per video (amortized via cache); query O($\log I + k$) + frame lookup O($\log F$). Space O(I) per tree; cache bounded.

Parameters:

- `object`: Object class name
- `time`: Timestamp in seconds

Example queries:

```
/query/queryInstancesAtTime?object=person
&time=15
Response: {
    "object": "person",
    "time": 15,
    "instances": [
        {
            "video_id": "689cd3905c1945e3f7602f42",
            "instance_id":
                "a8805091-c22e-40b6-b8f9-caf325571808"
        }
    ]
}
  /query/queryInstancesAtTime?object=person
  &time=51
  Response: {
      "object": "person",
      "time": 51,
      "instances": [
          {
              "video_id": "689cd3905c1945e3f7602f42",
              "instance_id":
                  "910921dc-4f83-4d1a-9f60-e7dfa62e690c"
          } ]}
```

Query Name	Query Time
spatialObjects	438 ms
spatialObjectsAnd	700 ms
queryDistinctInstances	389 ms
queryInstanceOverlaps	382 ms
queryInstanceOverlapsInArea	803 ms
queryInstancesAtTime	441 ms
objects	1101 ms
spatialObjectsTemporal	416 ms

TABLE V
QUERY PERFORMANCE AVERAGE TIMES FOR VARIOUS QUERY TYPES

Systems	Prediction Accuracy
Google Cloud Video Intelligence API	80-90%
Yolo 11	79%

TABLE VI
PREDICTION ACCURACY COMPARISON WITH SYSTEMS.

These query endpoints demonstrate the system's ability to perform complex spatial-temporal searches across video content, enabling precise retrieval of relevant video segments based on object presence, location, and timing requirements.

E. Spatial-Temporal Query

Endpoint:

`/query/spatialObjectsTemporal`

Combines spatial OR filtering with a temporal window [`start_time`,`end_time`]. Windows overlapping the temporal range are clipped to the range and returned.

Parameters:

- `objects`: Array of object names
- `area`: Predefined area name or coordinate array
- `start_time`, `end_time`: Temporal window in seconds

Algorithm (Temporal Window Filter):

```
FilterByTimeWindow(results, start, end):
  FOR EACH r IN results DO
    filtered ← []
    FOR EACH w IN r.windows DO
      overlap ← ComputeOverlap(
                   w, [start, end])
      IF overlap   THEN
```

```
        filtered.append(overlap)
    r.windows ← filtered
  RETURN results
```

Complexity: O(R × W) time and space (R results; W windows per result).

F. Sequence Query

Endpoint:
`/query/temporal/objects` (alias supported: `/query/querySequence`)

Finds video segments where objects appear in a specified sequential order using a greedy closest-match strategy (each next object starts at or after the previous ends).

Parameters:

- `sequence`: Array of object names in order
- `window_size` (optional): Maximum allowed duration

Algorithm (Greedy Sequential Matching):

```
QuerySequence(sequence[], windowSize):
  grouped ← {}
  FOR EACH o IN sequence DO
    inst ← DB.find({object_name: o})
    FOR EACH x IN inst DO
      grouped[x.video_id].append(x)
  results ← []
  FOR EACH (video_id, data) IN grouped DO
    IF NOT AllObjectsPresent(
              data, sequence) THEN CONTINUE
    firstSet ← data[sequence[0]]
    FOR EACH a IN firstSet DO
      end ← a.end_time; path ← [a]
      FOR i ← 1 TO sequence.length-1 DO
        b ← FindClosestAfter(
                data[sequence[i]], end)
        IF b = NULL THEN BREAK
        path.append(b); end ← b.end_time
      IF path.length = sequence.length THEN
        dur ← end - a.start_time
        IF windowSize = 0 OR
           dur  windowSize THEN
          results.append({video_id, path})
  RETURN results
```

Complexity: Fetch/group O(M × N); matching ≈ O($I_1 \times M \times \log I_2$) with sorted sets.

G. Objects Co-occurrence Query

Endpoint:
`/query/objects`

Finds time windows per video where all specified objects co-appear (temporal overlap only; no spatial constraint). Optionally enforces a maximum window size.

H. Additional Endpoints

- `/query/spatialObjectsPaginated`, `/query/queryInstancesPaginated`
- `/query/stream`

I. Algorithmic Complexity Summary

1) Notation:

- F: number of frames in an instance
- I: number of object instances (per video when noted)
- k: number of results returned
- W: number of time windows
- M: sequence length; N: avg instances per object
- I_1: instances of first object; I_2: avg instances of subsequent objects

J. Indexing Strategy (MongoDB)

Objects collection (5 indexes):

- `object_name: 1`
- `video_id: 1`
- `object_name: 1, video_id: 1`
- `object_name: 1, start_time: 1, end_time: 1`
- `video_id: 1, object_name: 1, start_time: 1`

Videos collection (2 indexes):

- `status: 1, createdAt: -1`
- `createdAt: -1`

Rationale: Compound indexes order the most selective fields first to maximize pruning. B-tree lookups provide O($\log N$) access versus O(N) scans.

K. Caching and Thresholds

Query result cache: LRU with capacity-based eviction (100 MB size limit) and 1-hour TTL; manual invalidation endpoints are provided to clear entries by video or object.

Interval Tree cache: LRU for up to 100 videos (center-based tree reused across queries).

Thresholds: QuadTree used when frames per instance > 100; Interval Tree when instances per video ≥ 20; Binary search always used for frame lookup.

Spatial semantics: Spatial queries primarily use normalized centers (`relative_position`); QuadTree queries approximate inclusion via small boxes around centers. If full per-frame `bounding_box` is available, it can be used for higher precision.

L. Advanced Optimizations

1) LRU Query Result Cache: **Algorithm:**

```
CLASS LRUCache(capacity, maxSize, ttl):
  cache: Map<key, {value, ts, size}>
  keys: Array (LRU order)
  FUNCTION query(qType, params, fn):
    key ← generateKey(qType,
              normalizeParams(params))
    IF cache.has(key) AND
       notExpired(cache[key]) THEN
      moveToEnd(key)
      RETURN cache[key].value
    result ← fn()
    size ← estimateSize(result)
    IF size > maxSize THEN RETURN result
```

TABLE VII. ALGORITHMIC COMPLEXITY SUMMARY FOR CORE QUERIES

Query Type	Simple Approach	Our Approach	Data Structures
Find objects in area	O(F) per instance	O(F) build + O(k) query	QuadTree, Cache
Find all objects in area	$O(W^2)$ compare	O(W log W) sort + scan	Time Window Intersection
Instances at time	$O(I \times F)$	O(I log I) build + O(log I + k) + O(log F)	Interval Tree, Binary Search, Cache
Overlapping instances	$O(I^2)$ pairs	O(I log I) sort + scan	Sweep Line
Sequence pattern	$O(I^m)$ combinations	$O(M \times N \log N) + O(I_1 \times M \times \log I_2)$	Greedy Sequence

```
    evictToFitSize(size)
    cache[key] ← {value: result,
                  ts: now(), size}
    keys.append(key); RETURN result
```

Performance: Hits O(1); misses equal original query time. Size bounded to 100 MB, TTL 1 hour; manual invalidation supported.

2) QuadTree (Spatial Search): **Structure and operations:**

```
CLASS QuadTreeNode:
  bounds; children; objects
  maxObjects=10; maxLevels=4; level
FUNCTION BuildQuadTree(frames):
  root ← new QuadTreeNode([0,0,1,1])
  FOR EACH (f, i) IN frames DO
    Insert(root, f.relative_position, i)
  RETURN root
FUNCTION Query(node, area):
  IF NOT Intersects(node.bounds, area)
    THEN RETURN []
  IF node.isLeaf() THEN
    RETURN FilterByArea(node.objects, area)
  results ← []
  FOR child IN node.children DO
    results.extend(Query(child, area))
  RETURN results
```

Complexity: Build O(F); query $\approx$ O(k); space O(F). Used only when F>100.

3) Interval Tree (Time Search): **Structure and point query:**

```
CLASS IntervalTreeNode:
  center
  intervals{left,right,overlap(sorted)}
  left; right
FUNCTION BuildIntervalTree(intervals):
  // choose median center
  // partition into left/right/overlap
  // recurse
FUNCTION QueryPoint(node, time):
  // scan overlap (early exit by start)
  // then recurse left/right by time
```

Complexity: Build O(I log I); query O(log I + k); space O(I). Cached for up to 100 videos.

M. When to Use Each Data Structure

Threshold-based selection:

- QuadTree for spatial search when frames per instance $>$ 100; otherwise linear scan
- Interval Tree for time search when instances per video $\geq$ 20; otherwise linear scan
- Binary Search always for frame lookup

Decision logic:

```
IF searching by location AND
   frames > 100:
  Use QuadTree
ELSE:
  Use simple loop
IF searching by time AND
   instances >= 20 per video:
  Use Interval Tree (with caching)
ELSE:
  Use simple loop
For frame lookups:
  Always use Binary Search
```

Memory usage: Interval Tree cache ($\approx$1–2 MB per video, up to 100 videos); result cache up to 100 MB; QuadTree built per query and discarded.

N. Efficient Video Fragmentation and Retrieval

The proposed system incorporates a novel approach for video fragmentation, using keyframes to divide videos into smaller, approximately 5-second chunks. This method is essential for optimizing the storage and retrieval of video data, particularly for large-scale datasets.

1) Keyframe-Based Fragmentation: Keyframes, or intra-frames, are self-contained frames that do not rely on other frames for decoding. The fragmentation process ensures that video chunks are split at keyframes, avoiding visual artifacts and ensuring compatibility with playback tools. The system uses FFmpeg to achieve this, with the following characteristics:

- **No Transcoding/Re-Encoding**: By copying the codec (`-codec copy`), the video quality remains intact, and computational overhead is minimized.
- **Variable Chunk Lengths**: Chunks align with the nearest keyframe after the specified duration (e.g., 5 seconds), ensuring the integrity of each segment.

2) Metadata for Precise Retrieval: Each video chunk is associated with metadata stored in MongoDB GridFS, including:

- **Video ID**: Identifies the original video.
- **Start and End Timestamps**: Denote the temporal boundaries of the chunk.
- **Duration**: Specifies the length of the fragment.

This metadata facilitates temporal and spatial queries, enabling efficient access to specific segments without processing the entire video.

3) Advantages of Fragmentation: The fragmentation approach offers several benefits:

- **Efficient Bandwidth Usage**: Only relevant chunks are transmitted, significantly reducing network load compared to retrieving whole video files.
- **Scalable Storage**: Parallel processing of chunks improves system scalability for large datasets.
- **Optimized Querying**: Metadata enables rapid temporal and spatial searches, essential for applications requiring precise video segment retrieval.
- **Improved User Experience**: Users can access desired segments quickly, enhancing interactivity and responsiveness.

O. Scalability

1) Microservices Architecture: The system is designed as a loosely-coupled microservices architecture where each component operates independently and can be deployed on specialized hardware optimized for its workload:

- **Service Isolation:** Python ML pipeline, Node.js API service, MongoDB database, and FFmpeg processing operate as independent services with well-defined interfaces
- **Containerization:** Docker-based deployment enables consistent environments across development, staging, and production; Kubernetes orchestration supports autoscaling and load balancing
- **Heterogeneous Hardware:** ML service on GPU-enabled nodes (NVIDIA Tesla/RTX); query service on CPU-optimized instances; database on storage-optimized instances
- **Message Queue Integration:** Asynchronous video processing via job queues (Redis/RabbitMQ) decouples upload from analysis pipeline
- **API Gateway:** RESTful API (Fastify framework) serves as unified entry point; supports rate limiting, authentication, and request routing
- **Cost Optimization:** GPU instances used only during video analysis; query service runs on cost-effective CPU instances; MongoDB replica sets provide read scaling without GPU overhead

2) Python ML Pipeline:

- **GPU Requirements:** YOLO11 model inference benefits from CUDA-enabled GPUs (NVIDIA RTX 3060 or higher recommended); CPU-only mode supported with reduced throughput
- **Processing Rate:** GPU: ≈30-60 FPS for 1080p video; CPU: ≈5-10 FPS
- **Memory:** 2-4 GB GPU VRAM for batch processing; 4-8 GB system RAM per worker process
- **Scalability:** Horizontal scaling via multiple worker processes; video processing queue supports distributed deployment

3) Node.js Query Service:

- **CPU Requirements:** Multi-core CPU (4+ cores recommended) for concurrent query processing
- **Memory:** 2-4 GB per Node.js instance; cache footprint up to 200 MB (100 MB result cache + 100 MB interval tree cache)
- **Throughput:** Handles 100-500 queries/second depending on query complexity and cache hit rate
- **Scalability:** Stateless design enables horizontal scaling behind load balancer; cache warming strategies improve cold-start performance

4) MongoDB Database:

- **Storage Growth:** ≈10-50 MB metadata per hour of analyzed video (varies with object density and frame rate)
- **Index Overhead:** 5 compound indexes on Objects collection; total index size ≈20-30% of collection size
- **Query Performance:** Indexed lookups O($\log N$) scale to millions of object instances; compound indexes optimize multi-field queries
- **Scalability:** Supports replica sets for read scaling and sharding for horizontal partitioning on `video_id`

5) FFmpeg Video Processing:

- **CPU Requirements:** Multi-threaded encoding/decoding; 4-8 cores for real-time processing
- **Hardware Acceleration:** Supports NVENC (NVIDIA), Quick Sync (Intel), VideoToolbox (macOS) for GPU-accelerated encoding
- **Memory:** 1-2 GB per concurrent FFmpeg process depending on resolution
- **Scalability:** Process isolation enables parallel processing of multiple videos; containerization (Docker) facilitates deployment

6) Storage Efficiency:

- **Video Storage:** Original videos stored in GridFS; fragments stored separately with temporal metadata
- **Metadata Compression:** Per-frame detections stored with normalized coordinates (8 bytes per detection vs. full bounding boxes)
- **Deduplication:** Instance-level aggregation reduces redundancy; shared object tracks minimize storage
- **Total Footprint:** For 100 hours of analyzed video: ≈1-5 GB metadata + original video size

7) Deployment Patterns:

- **Cloud-Native Deployment:** Compatible with AWS (EC2, S3, DocumentDB), Google Cloud (Compute Engine, Cloud Storage, Atlas), Azure (VMs, Blob Storage, Cosmos DB)
- **Hybrid Architecture:** On-premise GPU cluster for ML processing; cloud-hosted query service and database for global accessibility
- **Edge Computing:** Lightweight query service deployable on edge nodes; ML processing offloaded to centralized GPU resources
- **Auto-Scaling Policies:** Query service scales based on request rate (target: 70% CPU utilization); ML workers scale based on queue depth

- **Geographic Distribution:** Multi-region MongoDB replica sets reduce query latency; video storage replicated across regions for redundancy

8) Performance Optimization Strategies:

- **Batch Processing:** ML pipeline processes multiple video frames in batches (8-16 frames) to maximize GPU utilization
- **Lazy Loading:** Fragment metadata loaded on-demand; full video data fetched only when required
- **Connection Pooling:** MongoDB connection pool (50-100 connections) reduces overhead for concurrent queries
- **CDN Integration:** Processed video fragments served via CDN (CloudFront, Cloudflare) for low-latency delivery
- **Cron-Based Fragmentation:** Background worker periodically fragments uploaded videos during off-peak hours
- **Read Replicas:** Read-heavy queries distributed across MongoDB replicas; write operations directed to primary node

9) Monitoring and Observability:

- **Metrics Collection:** Prometheus-compatible metrics for query latency, cache hit rates, GPU utilization, and queue depth
- **Distributed Tracing:** Request tracing across microservices for bottleneck identification
- **Logging:** Centralized logging (ELK stack or CloudWatch) for error tracking and audit trails
- **Health Checks:** Kubernetes liveness/readiness probes ensure service availability; automatic restart on failure

VI. Challenges and Limitations

A. Challenges

This project faced several challenges, particularly in transitioning from TransVOD++ to YOLOv11. TransVOD++, while optimized for spatiotemporal video analysis, exhibited significant limitations in object detection across diverse scenarios. For instance:

- **Narrow Object Detection Scope:** TransVOD++ reliably detected specific objects (e.g., a ”husk”) but struggled with generic object classes such as an ”apple,” limiting its utility in broader applications.
- **Complexity of Integration:** Its focus on temporal and spatial modeling made it challenging to align with our metadata generation pipeline, which prioritizes explicit object recognition.

In contrast, YOLOv11 was chosen for its robust neural architecture search (NAS) capabilities and transformer-based enhancements, which significantly improved detection accuracy and query responsiveness. However, integrating YOLOv11 into a scalable framework required careful adjustments to the pipeline, including adapting it to handle varying video resolutions and object densities.

Additionally, handling metadata storage and retrieval in MongoDB GridFS posed logistical hurdles. Optimizing the chunk size to 5 MB minimized overhead, but this approach still poses challenges with larger video files, as chunk size may need further re-evaluation to maintain efficiency.

B. Limitations

The current approach, while demonstrating significant advancements, has notable limitations:

- **Limited Object Detection Scope:** The reliance on a pretrained YOLO11 model constrains the system to detecting a predefined set of objects from the COCO dataset. This limitation restricts applicability in domains requiring the identification of custom or specialized object types.
- **Pipeline Integration Challenges:** Existing video processing pipelines like Scanner [8], which could enhance video processing efficiency, are no longer actively developed. Attempts to integrate these tools revealed compatibility issues with our workflow.
- **GridFS Chunk Size Management:** While the default 255 KB chunk size in GridFS was optimized to 5 MB to reduce metadata overhead, this solution may not scale well for significantly larger video files. Dynamic evaluation of chunk sizes is necessary to balance storage and retrieval efficiency.
- **Absence of Advanced Features:** The system does not yet handle cross-model collaboration or provide real-time updates. Implementing these features would require substantial architectural enhancements.

Despite these limitations, the system lays a solid foundation for future enhancements. Addressing these areas will unlock its potential for diverse applications, including autonomous systems, large-scale surveillance, and AI-driven content recommendation.

VII. Code & Data Availability

A. GitHub:

- The code is available in this repository https://github.com/vikasdimaniya/VidMetaStream
- The complete api documentation with inputs and output schema is available in Postman JSON format.

VIII. Conclusions and Future Work

A. Conclusions

In this work, we presented a novel video querying system designed to address the challenges of processing and analyzing large-scale video datasets. Our approach integrates advanced object detection and metadata generation techniques to enable precise spatial-temporal queries and multi-object interaction detection. Key achievements of the system include:

- Leveraging MongoDB for metadata storage, enabling efficient querying and retrieval.
- Utilizing FFmpeg for keyframe-based video fragmentation to optimize storage and scalability.
- Enhancing the querying process to support diverse use cases, such as autonomous driving and surveillance.

While the system demonstrates significant advancements in video analysis and querying, certain limitations remain. These

include reliance on a pre-trained YOLO11 model constrained by its training dataset, the absence of parallel processing for multiple machine learning models, and challenges in leveraging pipelines such as Scanner. Addressing these constraints offers exciting directions for future development.

The proposed framework lays the foundation for scalable and efficient video analysis systems, with potential applications in autonomous driving, surveillance, and content recommendation. Future enhancements, such as incorporating dynamic chunking strategies, spatial indexing, and cross-model collaboration, can further elevate the system's capabilities, making it a valuable tool for researchers and engineers working with large-scale video datasets.

B. Future Work

To further enhance the system, future work will focus on:

- **Advanced Scene-Text Grounding:** Integrating neural architectures and contextual embeddings for precise text extraction from video frames, enabling applications like automated indexing, real-time text recognition, and semantic video search.
- **Parallel Model Integration:** Enabling the concurrent execution of multiple machine learning models by introducing additional status states within the video processing pipeline, thereby generating richer metadata.
- **Spatial and Temporal Indexing:** Employing advanced indexing techniques to accelerate spatial-temporal queries, particularly in large datasets requiring complex object interaction analysis.

These improvements will significantly enhance the system's utility, making it a more versatile and impactful tool for researchers and engineers working with large-scale video datasets.

1) Conclusion: This implementation effectively tracks objects across video frames using a combination of:

- Detection via YOLO.
- IoU-based matching for consistent identification.
- Timeout thresholds for handling occlusions.
- MongoDB for persistent storage.

These methods enable reliable tracking of multiple instances of the same class and ensure accurate object identification throughout the video.

REFERENCES

[1] S. Abu-El-Haija, S. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan, "Youtube-8m: A large-scale video classification benchmark," *arXiv preprint arXiv:1609.08675*, 2016. [Online]. Available: https://arxiv.org/pdf/1609.08675v1

[2] Ultralytics, "Yolov11 documentation," *Ultralytics Official Documentation*, 2024. [Online]. Available: https://docs.ultralytics.com/models/yolo11/

[3] I. MongoDB, *MongoDB Documentation*, 2024, available: https://www.mongodb.com/docs/. [Online]. Available: https://www.mongodb.com/docs/

[4] ——, *GridFS - MongoDB Manual*, 2024, available: https://www.mongodb.com/docs/manual/core/gridfs/. [Online]. Available: https://www.mongodb.com/docs/manual/core/gridfs/

[5] P. Michael, Z. Hao, S. Belongie, and A. Davis, "Noise-coded illumination for forensic and photometric video analysis," *ACM Trans. Graph.*, vol. 44, no. 5, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3742892

[6] R. Pal, A. A. Sekh, D. P. Dogra, S. Kar, P. P. Roy, and D. K. Prasad, "Topic-based video analysis: A survey," *ACM Comput. Surv.*, vol. 54, no. 6, Jul. 2021. [Online]. Available: https://doi.org/10.1145/3459089

[7] Z. He, Z. Ling, J. Li, Z. Guo, W. Ma, X. Luo, M. Zhang, and G. Zhou, "Short video segment-level user dynamic interests modeling in personalized recommendation," in *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1880–1890. [Online]. Available: https://doi.org/10.1145/3726302.3730083

[8] A. Poms, W. Crichton, P. Hanrahan, and K. Fatahalian, "Scanner: efficient video analysis at scale," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 1–13, August 2018.

[9] W. G. Aref, A. C. Catlin, A. K. Elmagarmid, J. Fan, M. Hammad, I. Ilyas, M. Marzouk, S. Prabhakar, Y.-C. Tu, and X. Zhu, "Vdbms: A testbed facility for research in video database benchmarking," in *Proceedings of the 9th International Workshop on Multimedia Information Systems (MIS)*, Isle of Skye, Scotland, 2003, pp. 41–47. [Online]. Available: https://cse.usf.edu/~tuy/pub/DMS03.pdf

[10] G. Özsoyoglu, Z. M. Özsoyoglu, T. Bozkaya, A. Budak, Y. Özden, G. Sahin, D. Singer, and M. Unal, "Bilvideo: A video database management system," *IEEE Multimedia*, vol. 10, no. 1, pp. 66–70, 2003. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1167924

[11] J. C. Lee, Q. Li, and W. Xiong, "Vims: A video information management system," *Multimedia Tools and Applications*, vol. 4, no. 1, pp. 7–28, 1997. [Online]. Available: https://doi.org/10.1023/A:1009655814781

[12] Q. Zhou, X. Li, L. He, Y. Yang, G. Cheng, Y. Tong, L. Ma, and D. Tao, "Transvod: End-to-end video object detection with spatial-temporal transformers," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–16, 2022.

[13] P. Chu and H. Ling, "Transmot: Spatial-temporal graph transformer for multiple object tracking," *arXiv preprint arXiv:2104.00194*, 2021. [Online]. Available: https://arxiv.org/abs/2104.00194

[14] V. D. Stanojevic and B. T. Todorovic, "Boosttrack: boosting the similarity measure and detection confidence for improved multiple object tracking," *Machine Vision and Applications*, vol. 35, no. 3, 2024.

[15] J. Xie, B. Zhong, Z. Mo, S. Zhang, L. Shi, S. Song, and R. Ji, "Autoregressive queries for adaptive tracking with spatio-temporaltransformers," 2024. [Online]. Available: https://arxiv.org/abs/2403.10574

[16] Q. Zhou, L. He, X. Li *et al.*, "Transvod++: Enhancing end-to-end video object detection with improved feature fusion and multi-scale processing," 2023, available at: https://github.com/qianyuzqy/TransVOD_plusplus.

[17] G. Cloud, *Google Cloud Video Intelligence API Documentation*, 2024, available at: https://cloud.google.com/video-intelligence. [Online]. Available: https://cloud.google.com/video-intelligence

[18] I. Corporation, *IBM Watson Video Enrichment Documentation*, 2024, available at: https://cloud.ibm.com/docs/services/Watson-Video-Enrichment. [Online]. Available: https://cloud.ibm.com/docs/services/Watson-Video-Enrichment