Demonstration of Apperception: A Database Management System for Geospatial Video Data

Yongming Ge*
University of California, Berkeley
yongmg@berkeley.edu

Vanessa Lin*
University of California, Berkeley
valin@berkeley.edu

Maureen Daum
University of Washington
mdaum@cs.washington.edu

Brandon Haynes
Gray Systems Lab, Microsoft
brandon.haynes@microsoft.com

Alvin Cheung
University of California, Berkeley
akcheung@cs.berkeley.edu

Magdalena Balazinska
University of Washington
magda@cs.washington.edu

ABSTRACT

Many recent video applications—including traffic monitoring, drone analytics, autonomous driving, and virtual reality—require piecing together, combining, and operating over many related video streams. Despite the massive data volumes involved and the need to jointly reason (both spatially and temporally) about these videos, current techniques to store and manipulate such data are often limited to file systems and simple video processing frameworks that reason about a single video in isolation.

We present Apperception, a new type of database management system optimized for geospatial video applications. Apperception comes with an easy to use data model to reason about multiple geospatial video data streams, and a programming interface for developers to collectively reason about the entities observed in those videos. Our demo will let users write queries over video using Apperception and retrieve (in real-time) both metadata and rendered video data. Users can also compare results and observe speedups achieved by using Apperception.

PVLDB Reference Format:
Yongming Ge, Vanessa Lin, Maureen Daum, Brandon Haynes, Alvin Cheung, and Magdalena Balazinska. Demonstration of Apperception: A Database Management System for Geospatial Video Data. PVLDB, 14(12): 2767 - 2770, 2021.
doi:10.14778/3476311.3476340

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://apperception-db.github.io/.

1 INTRODUCTION

Cameras are proliferating throughout our world (e.g., on mobile devices, city-wide camera networks, drones, and autonomous vehicles), and as a result we are accumulating an ever-increasing amount of geospatial video data.For users and service providers who operate on this data, managing and querying it is becoming increasingly challenging, especially given that existing mechanisms to do so are limited to video processing libraries (e.g., FFmpeg [9]) and recent video database management systems [1, 4, 7] that typically operate on single video streams.

Querying data from multiple video streams remains a challenging task as existing systems either operate on each video stream *independently* or attempt to operate on multiple videos by trading off resources and accuracy [8]. For instance, consider building a 3D scene reconstruction application that takes in video streams that capture the same object from multiple angles. To develop such an application using existing VDBMSs, users must write code that manually reads in multiple videos from the VDBMS, selects the video frames of interest from the various streams, merges frames using a reconstruction algorithm, and inserts the reconstructed result back into the VDBMS. Doing so requires developers to understand how the VDBMS represents video data once loaded into memory, devise their own data structures to store multiple video streams, and manually invoke object recognition and reconstruction algorithms. Furthermore, developers must tediously tune their data structures and implementation if application performance is suboptimal, which is likely the case given the many libraries and frameworks that are involved. Such a development flow is time-consuming and tedious.

In this paper, we describe the prototype and demonstration of a new geospatial video DBMS called Apperception. Building on our earlier vision paper [7], Apperception frees developers from the need to construct their own multi-video representation and processing algorithms. Apperception takes in videos containing geospatial information (i.e., when and where they were taken) and organizes them into a four-dimensional (i.e., spatial and temporal) *world*. Users manipulate and query video data stored in Apperception's worlds using a fluent Python application programming interface (API), and Apperception provides results in common video and image formats.

This paper presents a demonstration of Apperception. Our demo comes with a web interface representing an implementation of an AMBER Alert application for vehicle tracking (to be described in Section 2.1). Through this interface, users ingest different video datasets into Apperception and select a vehicle to locate and optionally specify spatiotemporal constraints. The interface will then submit the query to Apperception and (i) overlay the located vehicle's trajectory on a map, and (ii) show video cropped to the selected object. The key technical contribution of our system and the focus of this demonstration is the query language that provides the ability to easily query objects in a multi-camera world.

* Equal Contributions

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476340

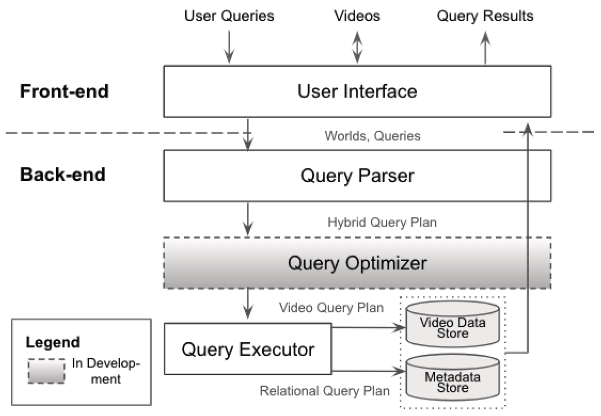


Figure 1: Apperception demo architecture illustrating the general flow of user queries and video data ingestion.

2 SYSTEM ARCHITECTURE

Apperception ingests multiple geospatial video files and organizes them into a four-dimensional spatiotemporal *world* consisting of x, y, z, and *time*. A world is the fundamental entity in Apperception. Each world contains cameras and objects. The cameras contain videos, which users specify, and objects are extracted from the video data. Users may query a world for different types of data. First, they may request objects that appear within the world, for example objects within a specified 3D volume, visible to a set of cameras, or at particular times. Second, they may retrieve video data, potentially filtered by camera, objects, or spatiotemporal constraints. Finally, they may query metadata associated with the entities in a world (e.g., an object's trajectory; see Fig. 2). The metadata stored to achieve this purpose consists of world and object metrics that include object types, trajectories, and bounding polygons.

The architecture of Apperception is shown in Fig. 1. Apperception's API is a Python interface that allows users to ingest video data and query for information of interest in videos or raw data. For the demonstration, users will interact with a front-end web user interface. A user initially creates an empty world and populates it by inserting cameras and video data. Apperception automatically handles the conversion between video coordinates and its 3D environment, allowing the user to focus on submitting declarative queries on objects in the 2D video domain. Apperception stores the data associated with each world in its metadata store, which is a hybrid relational and geographic information system (GIS) store, and sends video(s) to its video data store, which is a recently-developed video data store [3, 6]. World and camera metadata are stored as flat relations. Objects are partitioned into bounding polygons and trajectories, which are persisted as GIS trajectories. Apperception currently implements only 3D bounding boxes, but we plan to support arbitrary convex polygons in the future.

Apperception accepts user queries in the Python user interface. Queries are expressed in a fluent-like syntax, which allows predicates over cameras and objects (e.g., via `filter(...)` as shown in line 8), projections, and aggregates. Queries are composed lazily, and execution is deferred until explicitly requested by the user.

```
1   world = World(name='amber', units='metric')
2        .camera(position=p, orientation=o, lens=l)
3        .video('rtp://localhost/video')
4   world = world.recognize()
5   volume = Volume(world, ...)
6   query = world.filter(
7        lambda obj: obj.object_type = 'car' and
8        obj.location in volume and
9        timestamp in obj.times)
10  trajectories = query.select(lambda obj: obj.trajectory)
11                      .execute()
```

Figure 2: Code to retrieve car trajectories at an intersection.

Given a query, Apperception's query parser (QP) constructs a query plan for video data or metadata store. The query optimizer (QO) then pushes down operations to the Video Data Store.

Finally, Apperception's query executor (QE) takes video and metadata queries and executes them on its metadata and video data store (see subsection 2.2 for further detail). The data stores return query output as either metadata types, like trajectories and geometries, or videos, depending on what the user specified in their original Apperception query.

2.1 Usage Scenario

In this section we demonstrate a scenario where an AMBER Alert [5] has occurred and an analyst seeks to retrieve the trajectory of cars which appeared in an intersection at a certain time range. In Apperception, the query to retrieve these results is shown in Fig. 2. As mentioned previously, the query is constructed lazily, and execution is explicit, as shown in line 11.

Building the World: A user first defines a new world, as shown on line 1. Next, the user attaches a camera to the world at a given position (`p`) and orientation (`o`). A lens (`l`) is used to transform between 2D camera and 3D world coordinates; if not specified, a default transformation model, the pinhole camera model, is applied.

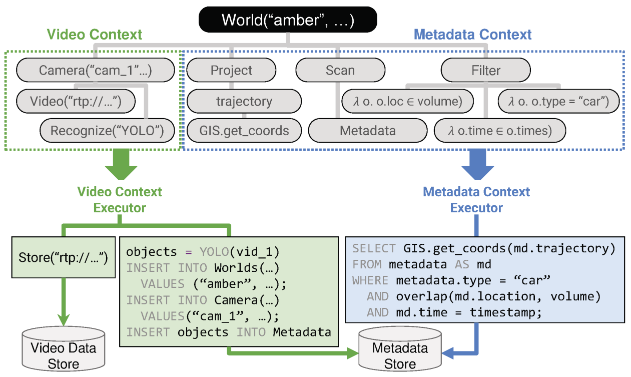
Users next associate one or more videos with each camera (line 3). While videos may be prerecorded on disk or (potentially live) streamed, our prototype currently supports only persisted video.

Having populated the world with video data, a user samples objects every few frames within the world (line 4). Recognized objects are incorporated into the 4D world at the places and times they are detected. Users may extend Apperception to support their preferred detection model or use a built-in model. Our prototype currently leverages YOLOv4 [2] as its default model, and we plan to incorporate support for other models.

Querying: To locate the cars that appear in the intersection in question, a user applies a predicate filtering on the desired object type ('car' in this example; see line 7) and the spatial and temporal region of interest (respectively `volume` at line 8 and `timestamp` in line 9 of Fig. 2). As mentioned, queries are accumulated lazily, and additional criteria may be specified prior to execution.

Like similar fluent frameworks (e.g., Pandas), the return type of each call is the associated world instance, allowing worlds (and queries over worlds) to be reused as needed.

Retrieving Results: Query results could include world objects, object metadata, or video data and or a combination of data types. The result shown in Fig. 2 is of metadata type and consists of a



```
objects = YOLO(vid_1)
INSERT INTO Worlds(…)
  VALUES ("amber", …);
INSERT INTO Camera(…)
  VALUES("cam_1", …);
INSERT objects INTO Metadata
```

```
SELECT GIS.get_coords(md.trajectory)
FROM metadata AS md
WHERE metadata.type = "car"
  AND overlap(md.location, volume)
  AND md.time = timestamp;
```

Figure 3: The top half structures the output from the Query Parser of the example query from Fig. 2. The bottom half visualizes the pipeline of the Query Executor.

sequence of trajectories of automobiles within the specified intersection at the specified time. Apperception surfaces these results as Python data types (e.g., NumPy arrays) to represent each trajectory. This approach allows for easy interoperation with other Python frameworks (e.g., a user might visualize trajectories on a map; see section 3). If a user is also interested in the actual video frames, she could issue a subsequent query, e.g., `query.video.execute()` to view complete frames or `query.select(lambda obj: obj.video)` for video cropped to the selected objects.

2.2 Implementation

Query Parser. The query parser (QP) parses world definitions and queries into a plan consisting of a *video context* and *metadata context.* The QP emulates an object-oriented programming setting in the context of a world, where the function calls are translated to SQL queries in the back-end. By abstracting away the data retrieval SQL syntax, the user can easily conceptualize objects as entities of the world, through calling easy-to-follow functions, like `recognize()` and `filter()`.

The top half of Fig. 3 displays the query plan that corresponds to the program in Fig. 2. When parsing world- and video-related data modification statements (e.g., world creation in line 1), the QP adds the relevant details into the plan's video context. The process is identical for camera creation, video ingestion, and object recognition.

For world queries (e.g., lines 6–10), the QP creates a filter operator for the predicate of the `filter` call in line 6. It also identifies the table (metadata table in this case) on which the filter's predicate acts on. The QP also maps properties and methods in the Python code to their GIS equivalent (e.g., line 10 maps `trajectory` to the GIS operation `get_coords` applied on the trajectory data in our metadata store.) Table scans and projections are handled similarly.

The example in Figure 2 requests object trajectories, which are retrieved from the metadata store. If the user instead requested a video (e.g., video of the automobiles in the intersection at a particular time), the QP would project on time and use this information to fetch video data.

Query Optimizer. Apperception's query optimizer is in early development. Apperception currently pushes temporal filtering and spatial cropping into the video data store to leverage these systems' optimized processing of compressed videos. We plan on exploring additional opportunities to increase the number of operations pushed down to both the video (e.g., framerate selections) and metadata (e.g., additional GIS-related operations) stores. We also plan on exploring optimizations over joins between the two constituent systems.

Query Executor. When users invoke `execute()` on a query, the query executor (QE) executes its parsed representation. When data modification is required (e.g., when a world or camera is created or video is ingested), the QE first checks to see if the query necessitates object recognition (i.e., a "Recognize" node is in the plan), and, if so, executes the specified algorithm (or YOLOv4 [2] as a default). It then persists any new video. Finally, the QE assembles and executes SQL batches to persist any new metadata (e.g., worlds, cameras, or objects). For data retrieval, the QE translates the plan into executable SQL queries. The QE traverses filter predicates in a top-to-bottom order and nests the operations applied to the column. It then translates the resulting abstract syntax tree for the predicates and translates it to SQL constraint syntax. For queries that retrieve video, the QE loads the relevant time data from the metadata store and uses this information to retrieve video frames for the requested timestamps. The result is outputted in a user-specified format.

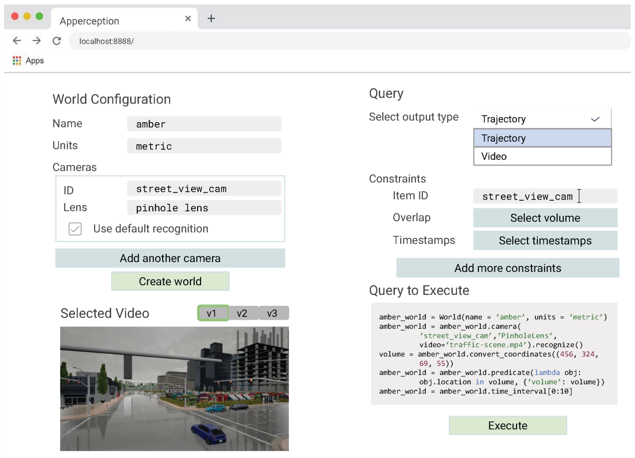
Storage. Apperception's storage system comprises the following:
Metadata Store. Apperception stores the metadata mentioned in section 2 in MobilityDB [10] which offers geometric convenience functions satisfying the needs of Apperception queries.
Video Data Store. Apperception leverages VSS [6] and TASM [3] for video storage and retrieval. Both offer a simple read/write interface over individual videos. VSS focuses on general video I/O performance, whereas TASM emphasizes efficient retrieval of video regions that contain objects of interest.

Apperception's video context executor ensures data consistency between the storage systems. Further, it invokes VSS or TASM depending on the type of query being executed. When a query requires (potentially spatiotemporally cropped) video frames, it issues calls to VSS for those regions. When possible, it pushes down operations into VSS (e.g., when only a particular time range is needed, Apperception reads only the relevant frames rather than the entire video). On the other hand, when a query requires video of specific objects (e.g., to view a particular object throughout its trajectory), it instead invokes TASM to execute the read.

3 DEMO WALK-THROUGH

For the demonstration, we will highlight the usage scenario on car trajectories—discussed in Section 2.1—using the user interface shown in Fig. 4. This interface is a web application consisting of a video panel, a world configuration view, and a query panel. The video panel shows the list of videos that the user has ingested into Apperception so far and allows the user to preview these videos. The world configuration view accepts user input to define and initialize a new world in Apperception, which contain cameras that captured the videos and objects that have been recognized. The query panel also accepts user input to form a query of their choice.

An end-user first ingests several traffic videos that she would like to analyze. These then appear in the "selected video" window. After



World Configuration

Name: amber

Units: metric

Cameras

ID: street_view_cam

Lens: pinhole lens

Use default recognition

Selected Video

Query

Select output type: Trajectory (Trajectory, Video)

Constraints

Item ID: street_view_cam

Overlap

Timestamps

Query to Execute

```
amber_world = World(name = 'amber', units = 'metric')
amber_world = amber_world.camera(
          'street_view_cam','PinholeLens',
          video='traffic-scene.mp4').recognize()
volume = amber_world.convert_coordinates((456, 324,
          69, 55))
amber_world = amber_world.predicate(lambda obj:
          obj.location in volume, {'volume': volume})
amber_world = amber_world.time_interval[0:10]
```

Figure 4: Demo user interface

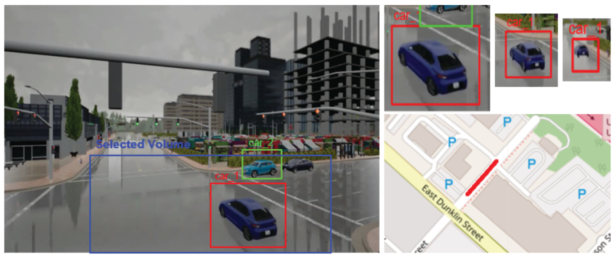


Figure 5: The left is a visualization of the user specified constraints. The top right are frames from the output video. The bottom right is a map overlay of the returned trajectory.

the user uploads the videos, they then define a new world, give it a name, and associate the videos with it. For example, Fig. 4 shows a user who has defined their world to be called `traffic_world`, selected `metric` units, associated one camera with it (called `cam_1`), and attach a video to the camera. The user can define cameras by a particular ID and lens type through the various text boxes. In Fig. 4, we show a camera named `cam_1` with a pinhole lens. After populating the world features, the user clicks "Create world" to execute the world creation process in Apperception.

Next, a user issues queries on the world she has created. To do so, the user enters an output type and defines various constraints in the query panel. Assume that the user is looking for the trajectory of a blue car that passed by the intersection shown. She first selects "trajectory" as the output type. She then specifies the blue car's identifier, selects the volume of interest (e.g., the traffic intersection) by drawing a polygon on a video frame, and indicates the time interval under consideration. Fig. 4 only shows the Trajectory, Geometry, Count, and Video as output types, but users can add additional custom output types to the query panel. The user can also include constraints on their query by pressing "Add more constraints." After adding all the desired constraints, the user obtains results by clicking the "Execute" button, which will in turn submit the query to Apperception and obtain the requested information or video results.

Fig. 5 shows the visualization for the resulting output in our example. If the user queries for the locations of the objects, the top right frames are from the resulting video that mark the bounding boxes of objects. In our demo for the example use case in Fig. 1, the returned trajectory of the object can be plotted on a real map as shown in the bottom right of Fig. 5.

4 RELATED WORK

Video data management—especially in the context of applied deep learning—has gained increasing popularity over the past few years. Many recent systems (e.g., [1, 4]) target efficient and expressive query processing over videos. So far, these systems target two-dimensional, independent video analytics, which forces developers to manually map 3D environments onto 2D videos and results in applications that are difficult to reason about, maintain, and evolve. Rekall [4] introduces a programming model to reason about video events but does not support geospatial queries and Vaas [1] supports the construction of efficient workflows to analyze videos. Apperception can also leverage Vaas to optimize the population of its metadata store.

5 CONCLUSION

This paper described a demonstration of Apperception, a novel video DBMS optimized for querying multi-perspective geospatial videos. Apperception captures a world of video streams that contain visual objects collected by many cameras. Through its world data model, users can express queries over these entities, visualize the results through our user interface, and compare query performance with a standard video processing library.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through grants IIS-1546083, IIS-1955488, IIS-2027575, CCF-1723352, CCF-1703051, DOE award DE-SC0016260, and ONR award N00014-21-1-2724; the Intel-NSF CAPA center, and gifts from Adobe, Facebook, Google, and VMware.

REFERENCES

[1] Favyen Bastani et al. 2020. Vaas: Video Analytics At Scale. *VLDB* 13, 12 (2020), 2877–2880.
[2] Alexey Bochkovskiy et al. 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. *CoRR* abs/2004.10934 (2020).
[3] Maureen Daum, Brandon Haynes, Dong He, Amrita Mazumdar, and Magdalena Balazinska. 2021. TASM: A Tile-Based Storage Manager for Video Analytics. In *37th IEEE International Conference on Data Engineering, ICDE*. IEEE, 1775–1786.
[4] Daniel Y. Fu et al. 2019. Rekall: Specifying Video Events using Compositions of Spatiotemporal Labels. *CoRR* abs/1910.02993 (2019).
[5] Timothy Griffin and Monica K Miller. 2008. Child abduction, AMBER alert, and crime control theater. *Criminal Justice Review* 33, 2 (2008), 159–176.
[6] Brandon Haynes, Maureen Daum, Dong He, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. 2021. VSS: A Storage System for Video Analytics. In *SIGMOD '21: International Conference on Management of Data*. ACM, 685–696.
[7] Brandon Haynes, Maureen Daum, Amrita Mazumdar, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. 2020. VisualWorldDB: A DBMS for the Visual World. *CIDR* (2020).
[8] Chien-Chun Hung et al. 2018. VideoEdge: Processing Camera Streams using Hierarchical Clusters. IEEE, 115–131.
[9] Suramya Tomar. 2006. Converting video formats with FFmpeg. *Linux Journal* 2006, 146 (2006), 10.
[10] Esteban Zimányi et al. 2020. MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS. *TODS* 45, 4 (2020), 19:1–19:42.