AFQLite: a user-friendly end-to-end video query system

Luka Govedic
lgovedic@mit.edu

Arman Dave
armdave@mit.edu

Abstract

In this paper, we propose AFQLite, an end-to-end system for video querying. AFQL allows users to specify custom object detection models by passing in the weights for the model and includes a python package and command line interface with video display capabilities for fast analytics. The language is based on SQL and is designed to be intuitive for data analysts familiar with SQL, rather than requiring a deep understanding of object detection algorithms. We present the AFQLite architecture, a case study of AFQLite performance on a corpus of videos, and discuss extensions of the system to meet future use cases.

1. Introduction

1.1. Motivation

The amount of video data created and published to the internet grows rapidly every year. Video analytics grows increasingly important, from self-driving cars to security to sports intelligence, and the need for human insight in the data collection and analysis process is diminishing as deep neural nets and object detectors grow increasingly sophisticated. There already exist systems that can answer "how many cars turned right at a given intersection" [5] or "give me all shots of Hermione between Harry and Ron" [3], but it is incredibly difficult to spin up these systems and format a proper query without understanding these system's internals. As the authors of VIVA [7] point out, the video database literature still lacks an end-to-end practical video analytics system. Video database systems have yet to define a model that decouples the language of querying data from a deep knowledge of how the system internals work.

1.2. Contribution

We propose the Abstract Frame Query Language (AFQL), an end-to-end database system. AFQL contributes the following to the video database field:

- We build AFQLite, a python package and CLI (like SQLite) with video display capabilities for users to do fast analytics.
- We abstract FrameQL to allow users to specify custom object detection models by passing in the weights for the model.
- We outline the full AFQL architecture - an end-to-end system that takes in SQL-like queries and outputs object detection tuples

2. Related Work

2.1. Multimedia Query Languages

In the 2000's, multimedia query languages (MQLs) generated considerable interest in the database systems domain Multimedia languages allow users to query not just data but "media," such as images, videos, etc. with syntax that is tailored to specify requirements on media's unique spatial-temporal features (e.g. finding all videos with a horse to the left of a man in a given time period). However, since video data had to be preprocessed, interest mostly faded due to the high human labor involved and the low accuracy of current computational methods.

In recent years, fast object-detection ML models have emerged, allowing for online querying and reigniting the interest in the field of video database systems. Despite this, various video database systems each possess their own unique query language, and understanding how to properly query the database often requires a non-trivial knowledge of the system's backend deep learning models. Data analysts are often familiar with SQL and could query video using the same principles instead of having to learn the intricacies of a object detection based backend. SVQL is an unimplemented query language proposed to meet this need, featuring a SQL-like syntax [8]. SVQL identified that spatiotemporal aspects are a key differentiator in video data from standard SQL data or even image data and built these aspects into the grammar. A unified query language similar to SVQL that could work with various different backend models in several different environments would be highly beneficial not just for reproducing research results but also in commercial use cases.

2.2. Object Detection Algorithms

In the past decade, advances in machine learning models, particularly deep learning, have proved so highly effective in the field of object detection (and thereby video database systems) that these models have rendered prior methods nearly obsolete. Two model architectures are widely used in the literature - YOLO [10] and Faster R-CNN [4]. While Faster R-CNN is slightly more accurate in terms of bounding boxes, especially for small objects, YOLO is faster at inference time: Depending on hardware architecture, YOLOv5 runs at rates of 60 FPS, while YOLOv7 has been shown to achieve speeds of 150 FPS. Despite these impressive speeds, it is still cost-prohibitive to process every frame on GPU even once for many datasets - let alone processing every frame per query. For instance, if one hour of GPU compute time costs 3 dollars, and we process 500 24-hr traffic cameras footage at 30 FPS over a month with YOLOv5 at 60 FPS, then our expenditure is 540,000 dollars.

While processing at query time naively without optimization is cost infeasible, a system that makes a single cheap pass over every frame is still attractive. Every year, the cost of hardware decreases and the efficiency of deep learning models increases rapidly. Furthermore, Neural Magic has released quantized versions of YOLO [9] that run on CPU and achieve similar accuracy to baseline YOLO while maintaining 60 FPS runtime. At the moment, the cost of CPU compute time is, on average, 1/100 the cost of compute time on a GPU. These CPU-performant models will play an important role in the preprocessing of video database systems due to the cost benefit.

2.3. End-to-end Video Database Systems

While integrating Deep Neural Nets (DNNs) into video analytics and video databases systems is a relatively new concept, the field has been rapidly growing since the first major system, NoScope [6]. OTIF [2] and MIRIS [1], two recent works, represent opposite ends of the spectrum for the key question of when processing is done with an object detection net. OTIF opts to preprocess the entire video while tuning frame rate and resolution such that the object detector is not used at query time (double check this). On the other hand, MIRIS processes video only at query time using a very accelerated frame rate and only ”slows down” to process frames at a finer grain when it detects objects moving quickly. In this respect, MIRIS is more similar to BlazeIt, a late materialization system that followed NoScope [5]. BlazeIt preprocesses video with cheap neural nets that act as filters and can run at 100,000 FPS; the system then applies a more specialized detector only on frames of interest according to the outputs of the preprocessing.

Apart from BlazeIt's contribution toward backend optimizations, the authors developed a frontend query language known as FrameQL. FrameQL implements many of the concepts from its predecessors, such as SVQL and MOQL (is this the right abbreviation), focusing on simplifying aggregations (e.g. counting the number of cars in a frame). FrameQL also enables a user to specify the error rate and confidence of the model's materialized results. However, FrameQL is not abstract enough in that users are either completely dependent on BlazeIt's built-in models and detectors - or users must understand the BlazeIt's backend well-enough to dig in and replace these models with their own. Furthermore, FrameQL lacks some of the spatiotemporal operators specified in SVQL.

In summation, while video database systems have made great strides, they face two key weaknesses:

- Video database systems do not have a unified query language, forcing the user to understand much of the implementation details about the backend - although FrameQL is close.

- Most importantly, there is no way in existing systems or languages to specify custom object detection/neural net models, which hamstrings advanced users who may possess fine-tuned and better-performing models in a given domain.

3. System Goals

The most important aspects of our system were user-friendliness and accessibility. We wanted to make sure that users without domain-specific knowledge in the area could still use it to run queries on videos. We also aimed to leverage approaches from existing systems to deliver good performance to the users. Finally, we wanted to allow experts in the area of object detection to leverage their custom-tuned models for video querying by plugging them into our system.

4. System Design

Our video database engine (called AFQLite) is an end-to-end system that allows users to perform structured queries, expressed in AFQL, on videos. The users can interact with the system through a command-line interface (CLI) or a Python interface. AFQLite follows a relational model in which each object detection is treated as a row in a relational table.

The high-level workflow in AFQLite goes as follows: the user first loads a video into AFQLite. The video then gets preprocessed with a lightweight object detector, and the detections get stored in the cache. When a user performs a query, the query plan scans the detections from the cache. Then it performs other structured query operations like filtering, joins, etc., including running the heavyweight object detector to increase the confidence of the results.

Each of the 4 main subsystems of AFQLite (User Interface, Caching, Query Plan Execution, and Detection) is described in its own subsection below.

4.1. User Interface

Users can import AFQLite as a Python package or use it through its CLI. The CLI is more convenient as it doesn't require writing any code (except for the queries), but the Python interface allows for more customization. The CLI is implemented as a wrapper around the Python interface that can be run as a standalone executable.

The Python package contains an AFQLite class, which is the global database instance for all AFQLite data. It supports adding custom detectors, loading videos, importing and exporting various caches, and most importantly running queries.

When the user loads a video, AFQLite will use a lightweight detector to preprocess all of the frames in the video and store the detections in the corresponding cache. The user can specify a custom lightweight detector to use for preprocessing, as well as a heavyweight detector. They can also import the cache directly if they had already run preprocessing on this video before. More on importing caches is at the end of the next subsection.

AFQLite displays the results of a query in a table format, but it also supports creating a video from the frames so that the users can visualize the results.

4.2. Caching

While detectors and videos are independent of each other, there exists exactly one cache for each (detector, video) pair. That allows detectors to only process each frame once, and store the detection results, which can then be reused next time. That allows us to bring the number of detector invocations (a bottleneck in the system) to a minimum.

Caches use a key-value store underneath, where the key is a tuple of the frame timestamp and object class. That allows efficient lookup by detectors, as well as guarantees a particular order during the scan. Specifically, the detections that share those two properties are going to be emitted in order, which is important for the part of the query plan that invokes a detector during execution.

Each video will have default lightweight and heavyweight caches. These correspond to the lightweight and heavyweight detectors for the video, which default to built-in detectors but can be manually specified as explained above. They are used in scans, described in the next section.

When importing a cache for a (detector, video) pair, the imported detections will be merged into the existing cache for the specified detector and video. If there's an overlap, the imported detections will replace existing detections in the cache. We chose this approach to give the user more flexibility, but they do need to be careful not to pollute their data.

4.3. Query Plan Execution

In AFQLite, the AFQL query is converted [1] to an intermediate representation, also called the query plan, which closely resembles a query plan for a relational database. Its only unique components are the Scan and DetectorFilter operations. Scan produces object detection tuples using the default light and heavyweight caches for the video. DetectorFilter runs the heavyweight (or custom) detector for any tuples that do not reach the confidence threshold, specified in the query. It then tries to match the detections from the detector with tuples coming from the nested query plan and replaces them in the final result[2].

4.4. Detection

We use the same detector architecture for built-in and custom object detectors. That makes it easier for the users to add custom detectors and choose default lightweight and heavyweight detectors for a video. Currently, detectors are loaded using Pytorch's `load` method, but in the future, we'd want to support users supplying custom detectors through an interface contract.

Each detector has a corresponding cache that stores its detections. When a query plan wants to run a detector on a frame, it first goes through a caching layer (CachedDetector) that checks if the results are available in the cache, and returns those. Otherwise, it runs the detector and stores the results in the cache before returning them to the query plan executor.

The built-in lightweight and heavyweight detectors are quantized pruned YOLOv5s[3] and YOLOv5s-base, respectively. The former runs using the DeepSparse engine, which achieves extremely high performance on a CPU while sacrificing some of the accuracy. We explore the performance of both in more detail in the Experiment section. For the latter, we considered both YOLO and Faster R-CNN. We ultimately chose YOLO as we did not want to sacrifice more speed than we already have. However, further evaluation would be required to find the optimal point on the speed-accuracy tradeoff curve. The ability to plug in custom detectors also lets users decide how much accuracy they want.

[1] We have not implemented the conversion yet. However, our query plan resembles the structure of AFQL closely so this is only a matter of implementation.

[2] This is another part we did not have time to implement. The matching process turns out to be very complex due to the number of unique scenarios like filtering and false positives and negatives.

[3] we got this to run independently but weren't able to integrate it into our system yet due to the interface discrepancy. We do not see this as a design challenge, it just wasn't implemented

5. Evaluation

We evaluate our system directly on our goals. While the accessibility and detector flexibility can only be evaluated qualitatively, we also provide a quantitative evaluation of the accuracy and performance of our approach.

5.1. Qualitative goals

Our system's intuitive and SQLite-like interface is very user-friendly in general. We paid special attention to making the CLI simple and made error messages expressive. The ability to display results of a query in a video further reduces the friction for users and lets them iterate on their queries much more quickly. On the other hand, we believe that the combination of caching and the unpredictable nature of object detection models could lead to some confusion in query results. We tried to mitigate that by giving the user a lot of control over the data in the system, but it could still be improved further. Our current implementation of detectors is not completely flexible, as we currently just load weights from a file using PyTorch. That means that users could load their fine-tuned own detectors, but only if they're in the same format as YOLO. Nevertheless, our architecture is designed in a way that makes this feature easy to add.

5.2. Datasets

As we began the project, we considered a variety of different datasets. Many pre-labelled video sets are traffic camera footage, which do not provide the variety of objects that necessitate a complete query language like AFQL. We also considered the ImageNet VID dataset, which contains a wide variety of objects, but each video clip is around 5 seconds long and tends to focus on only one object, meaning the video has little motion and few objects in the same frame; again, one of the purposes of AFQL is to make querying motion and variety of objects easier. We decided to create our own corpus to suit our needs, but due to the time constraints - particularly of decoding large quantities of video - our corpus is somewhat shorter than we liked, consisting of 4 videos: a scene from the movie Baby Driver, handheld footage of zoo animals, Lionel Messi soccer highlights, and a dog agility tournament.

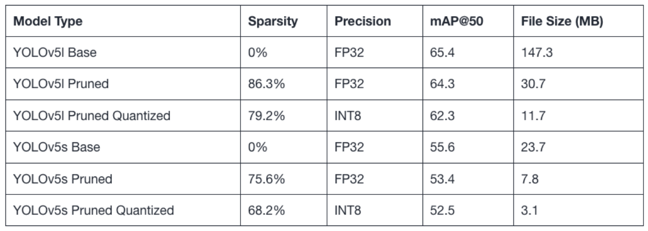
Each video contains more than 5 unique objects and contains a great deal of motion. The videos possess unique qualities that test the resilience of our object detection algorithm: Baby Driver features shots from a variety of angles, the zoo video (while markedly shorter) features a moving camera with lower resolution, the soccer video deals with small objects (due to an aerial view), and the dog agility video has banners of text that often obscure part of the objects in the video.

A note of interest is that our preprocessing algorithm considers all frames in the corpus. In contrast, most previous works devise an intelligent algorithm to skip many frames; for instance, MIRIS processes on average 1 frame per second, while we process 30 frames per second. This means that while we consider around 25 minutes of video, we generate as many frames as similar works considering 750 minutes of video.

5.3. Experiment

5.3.1 Comparing Quantized Pruned YOLO Model to YOLO Base

In our system, we preprocess all frames of the corpus with Neural Magic's YOLOv5s Pruned Quantized model on CPU. According to Neural Magic [9], their model has very nearly similar accuracy performance to the YOLOv5s baseline on the COCO dataset; on input batch size of 64, the pruned quantized model can process 410 items/sec on CPU, while YOLOv5s Base processes 94 items/sec.



Model Type	Sparsity	Precision	mAP@50	File Size (MB)
YOLOv5l Base	0%	FP32	65.4	147.3
YOLOv5l Pruned	86.3%	FP32	64.3	30.7
YOLOv5l Pruned Quantized	79.2%	INT8	62.3	11.7
YOLOv5s Base	0%	FP32	55.6	23.7
YOLOv5s Pruned	75.6%	FP32	53.4	7.8
YOLOv5s Pruned Quantized	68.2%	INT8	52.5	3.1

Figure 1. YOLOV5s quantized pruned results

The first step of the experiment is to evaluate the performance of quantized pruned YOLOv5s on every single frame since quantized pruned YOLOv5s will do this during preprocessing. Using the results of YOLOV5s Base as our ground truth, we measure the average IOU of the YOLOV5s quantized pruned model at confidence 50.

Our own results processing every frame in our video corpus are shown in the following figure. The quantized video model does very poorly on the soccer video, which suggests it is not as robust to small object detection as the base model (small object detection is a known weakness of YOLO models). Aside from this, the quantized model produces very similar results to the base (while running on a CPU at approximately 1/100 the cost).

Pruned Quantized Yolo IOU@50	
Video	**IOU@50**
Baby Driver Scene (6 min)	0.713
Zoo Handheld (10 sec)	0.963
Messi Soccer Vid (6 min)	0.257
Dog Agility Vid (12 min)	0.681

5.3.2 Performance on Example Queries

In the second phase of the experiment, we measure the performance of the AFQL architecture on accuracy and speed against a system that processes every frame of the corpus with YOLOv5s base; for each query, we define accuracy as the percent of true objects specified by the query returned by AFQL, and we define speed as the percent of frames in the video that AFQL heavyweight processed. As NoScope points out, in an architecture with cheap preprocessing first and a heavyweight detector for the second pass, a low false negative rate is highly desirable, since the heavyweight detector can always check false positives and discard them. In the AFQL architecture, false positives directly lead to a decrease in speed, while false negatives of course lead to less accuracy.

For Query 1, we select frames with a certain object of interest. For Baby Driver, Zoo, Messi soccer, and dog agility videos - we select for cars, zebras, sports (soccer) ball, and dogs.

Select * From Video Where Class=Object

Query 1		
Video	**Accuracy**	**Speed**
Baby Driver Scene	0.34	0.03
Zoo Handheld	0.98	0.45
Messi Soccer Vid	0.8	0.04
Dog Agility Vid	0.81	0.06

For Query 2, we test AFQL's spatiotemporal capabilities by executing a query that returns frames in which one object is detected as left of the other object - herein defined as the xmax coordinate of the first object being less than the xmin coordinate of the second object. For Baby Driver, we choose two cars; for Zoo Handheld, we choose a zebra left of a giraffe; for Messi Soccer Vid, we choose person left of ball; for Dog Agility vid, we choose dog left of person.

Select * From Video Where Obj1.bbxmax $<$ Obj2.bbxmin

Query 2		
Video	**Accuracy**	**Speed**
Baby Driver Scene	0.18	0.002
Zoo Handheld	0.625	0.017
Messi Soccer Vid	0.804	0.02
Dog Agility Vid	0.94	0.02

These results demonstrate that while quantized YOLOv5s speeds up the runtime of a query, it is not as accurate as YOLOv5s base. In Query 2, we see that the likelihood of error has a multiplicative effect: Baby Driver and Dog Agility suffer in particular because quantized YOLO often mistakenly marks dogs as cats and cars as trucks. In the AFQL architecture, the heavyweight detector will ignore these frames in the second pass - thus boosting speed beyond the true value. While running a lightweight detector on CPU is promising, future approaches may benefit from finetuning the model or quantization-aware training on datasets in the domain.

6. Limitations and Future Work

Bounding boxes could change after the detector replaces the results. That means that some predicates could now have a different value. We could mitigate by allowing users to choose one of three modes:

- default: bounding boxes are just replaced. This is the fastest way but the results could not satisfy the predicate in the query.
- strict: the filter is re-run after bounding boxes are replaced to eliminate detections that don't fulfill the predicate anymore. This takes some extra work but eliminates false positives. However, it does not help with false negatives.
- accurate: any filtering involving bounding boxes is only done after detection. This is a lot more computationally expensive as more detections have to be run, but it is the most accurate.

There are two more issues our system does not really solve directly. First, if the lightweight detector does not support a class, then that detection will never make it to the custom detector to check, even if that one does support it. This could potentially be solved by adding support for subclasses. Second, if preprocessing produces a false negative, the heavyweight detector again cannot help in recovering that detection. We could make our system more robust by using a model with fewer false negatives for preprocessing.

We do not consider these issues too detrimental, as a user could always plug in a custom model for any stage to eliminate them completely.

Next Steps. AFQL was built to be as abstract and extensible as possible. We have several future works on which to focus, but believe they should mesh with the existing system. The foremost goal is to implement the AFQL to specify queries. The next goal would be to abstract the detector class even further to support custom implementations of models. Once these steps are accomplished, we can publish AFQLite as a Python package. Further downstream tasks include query optimizations based on tuples in the preprocessing cache, training additional built-in detectors for specific domains, and integrating object track algorithms both to aid with cache replacement and to perform motion-based queries..

References

[1] Favyen Bastani, Songtao He, Arjun Balasingam, Karthik Gopalakrishnan, Mohammad Alizadeh, Hari Balakrishnan, Michael Cafarella, Tim Kraska, and Sam Madden. Miris: Fast object track queries in video. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1907–1921, 2020. 2

[2] Favyen Bastani and Samuel Madden. Otif: Efficient tracker pre-processing over large video datasets. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2091–2104, New York, NY, USA, 2022. Association for Computing Machinery. 2

[3] Daniel Y Fu, Will Crichton, James Hong, Xinwei Yao, Haotian Zhang, Anh Truong, Avanika Narayan, Maneesh Agrawala, Christopher Ré, and Kayvon Fatahalian. Rekall: Specifying video events using compositions of spatiotemporal labels. *arXiv preprint arXiv:1910.02993*, 2019. 1

[4] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015. 2

[5] Daniel Kang, Peter Bailis, and Matei Zaharia. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. *arXiv preprint arXiv:1805.01046*, 2018. 1, 2

[6] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: optimizing neural network queries over video at scale. *arXiv preprint arXiv:1703.02529*, 2017. 2

[7] Daniel Kang, Francisco Romero, Peter Bailis, Christos Kozyrakis, and Matei Zaharia. Viva: An end-to-end system for interactive video analytics. CIDR, 2022. 1

[8] Chenglang Lu, Mingyong Liu, and Zongda Wu. Svql: A sql extended query language for video databases. volume 8, pages 235–248, 2015. 1

[9] neural magic. Yolov5 on cpus: Sparsifying to achieve gpu-level performance and a smaller footprint. 2, 4

[10] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016. 2