BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics

Daniel Kang, Peter Bailis, Matei Zaharia
Stanford InfoLab

ABSTRACT

Given recent advances in neural networks (NNs), it is increasingly feasible to automatically query large volumes of video data with high accuracy. While these deep NNs can produce accurate annotations of object's position and types in video, they are computationally expensive and require complex, imperative deployment code to answer queries. Prior work uses approximate filtering to reduce the cost of video analytics, but does not handle two important classes of queries, aggregation and limit queries, and still requires complex code to deploy. To address the computational and usability challenges of querying video at scale, we introduce BLAZEIT, a system that optimizes queries over video for spatiotemporal information of objects. BLAZEIT accepts queries via FRAMEQL, a declarative extension of SQL for video analytics that enables video-specific query optimization and does not require complex, imperative code to express queries. We implement two new query optimization techniques in BLAZEIT that are not supported by prior work. First, we develop methods of using NNs to quickly answer approximate aggregation queries with error bounds. Second, we present a novel search algorithm for cardinality-limited video queries. Through these these optimizations, BLAZEIT can deliver up to two orders of magnitude speedups over the recent literature on video processing.

PVLDB Reference Format:
Daniel Kang, Peter Bailis, Matei Zaharia. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. *PVLDB*, 12(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/TBD

1. INTRODUCTION

Two trends have caused recent interest in video analytics. First, cameras are now cheap and widely deployed, e.g., London alone has over 500,000 CCTVs [2], which could be used to study traffic patterns. Second, deep learning, in the form of deep neural networks (DNNs), can automatically produce annotations of video. For example, object detection DNNs [20] will return a set of bounding boxes and object classes given an image of frame of video. Analysts can use these DNNs to extract object position/type from every frame of video, a common analysis technique [63]. In this work, we study the batch setting, in which large quantities of video are collected for later analysis [6, 38, 47].

While DNNs are accurate [33], naively employing them has two key challenges. First, from a usability perspective, these methods require complex, imperative programming across many low-level libraries, such as OpenCV, Caffe2, and Detectron [26]—an ad-hoc, tedious process. Second, from a computational perspective, the naive method of performing object detection on every frame of video is infeasible at scale: state-of-the-art object detection, e.g., Mask R-CNN [33], runs at 3 frames per second (fps), which would take 8 GPU-decades to process 100 camera-months of video.

Researchers have proposed optimizations for video analytics, which largely focus on filtering via approximate predicates, e.g., NOSCOPE and TAHOMA train cheaper, surrogate models for filtering [6, 9, 47, 56]. However, these optimizations do not handle two key classes of queries, aggregate and limit queries. For example, an analyst may want to count the average number of cars per frame (aggregate query) or manually inspect only 10 instances of a bus and five cars (limit query) to understand congestion. Approximate filtering are inefficient for these queries, e.g., filtering for cars will not significantly speed up counting cars if 90% of the frames contain cars. Furthermore, these optimizations still require non-expert users to write complex code to deploy.

To address these usability and computational challenges of querying video, we present BLAZEIT, a system that allows users to express queries in a declarative query language and implements two novel optimizations for aggregation and limit queries. BLAZEIT's declarative query language, FRAMEQL, extends SQL with video-specific functions and allows users familiar with SQL to issue video analytics queries. Since queries are expressed in a declarative manner, BLAZEIT can *automatically* optimize them end-to-end with its query optimizer and execution engine. Finally, BLAZEIT provides two novel optimizations for aggregation and limit queries that enable it to outperform prior work, such as NOSCOPE [47], by up to 500×.

FRAMEQL can express selection queries in prior work [6, 9, 47, 56], along with new classes of queries, including aggregation and limit queries (§4). FRAMEQL allows users to query information of objects in video through a *virtual* relation. This relation represents the information of positions

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/TBD

and classes of objects in the video and is materialized on demand. Instead of fully materializing the FrameQL relation, BlazeIt uses a number of optimizations to invoke the object detection DNN as few times as possible while meeting an accuracy guarantee based on the structure of the FrameQL query (Figure 1).

To reduce object detection invocations, we introduce two novel optimizations for video analytics in BlazeIt's query plans. Both of these optimizations generate neural networks (NNs) to approximate the reference object detection network, but provide exact answers or accuracy guarantees. Furthermore, both of these optimizations can be extended to account for query predicates.

First, to answer aggregation queries, BlazeIt could filter frames without objects of interest or use approximate query processing (AQP) in the form of random sampling (r.s.). However, filtering is inefficient when objects appear frequently in video and r.s. does not take advantage of proxy models. Instead, BlazeIt optimizes approximate aggregates, e.g., counting the number of cars in a frame, by approximating the exact answer from object detection using a smaller, *specialized neural network*. By approximating the exact answer, BlazeIt can reduce the number of samples needed for a given error bound using the method of control variates or by forgoing the object detection method entirely (§6). Furthermore, control variates can give statistical guarantees on accuracy.

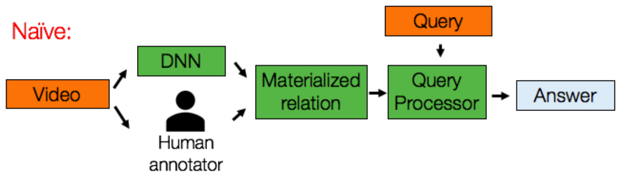
Second, to answer cardinality-limited queries, e.g., finding frames with at least three cars, BlazeIt could filter frames without certain objects or randomly sample. Similarly, filtering is inefficient when objects appear frequently and random sampling will perform poorly for rare events. Instead, BlazeIt evaluates object detection on frames that are more likely to contain the event using proxy models (§7). By prioritizing frames to search over, BlazeIt can achieve exact answers while speeding up query execution.

BlazeIt invokes these optimizations via its query optimizer and engine that efficiently executes FrameQL queries. Given query contents, BlazeIt employs a rule-based optimizer to generate optimized query plans that implements our two novel optimizations. BlazeIt's query plans will avoid executing object detection wherever possible, while maintaining the user-specified accuracy (relative to the object detection method as ground truth).

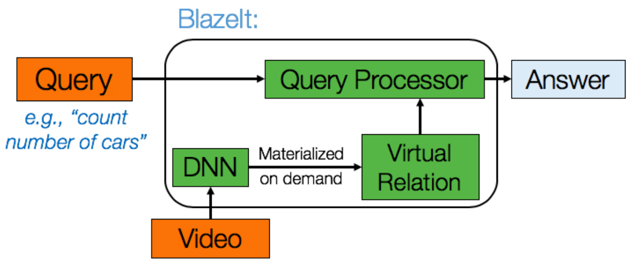
We evaluate BlazeIt on 6 query types (32 total queries) on four video streams that are widely used in studying video analytics [9, 38, 41, 47, 71] and two new video streams. We show that BlazeIt can achieve up to two orders of magnitude improvement over prior work in video analytics for limit queries and up to 8.7× improvement for aggregate queries.

In summary, we make the following contributions:

1. We introduce FrameQL, a query language for spatiotemporal information of objects in videos, and show it can answer a variety of real-world queries, including NoScope's selection queries, aggregation queries, and limit queries.
2. We introduce an aggregation algorithm that uses control variates to leverage specialized NNs for more efficient aggregation than existing AQP methods by up to 8.7×.
3. We introduce an algorithm for limit queries that leverages specialized NNs and can deliver up to 500× speedups over recent work in video analytics.



(a) Schematic of the naive method of querying video. Naively using DNNs or human annotators is too expensive for many applications.



(b) Schematic of BlazeIt. BlazeIt will create optimized query plans and avoid calling the expensive DNN where possible.

Figure 1: Schematic of the naive method of querying video and BlazeIt.

```
SELECT FCOUNT(*)
FROM taipei
WHERE class = 'car'
ERROR WITHIN 0.1
AT CONFIDENCE 95%
```

(a) The FrameQL query for counting the frame-averaged number of cars within a specified error and confidence.

```
SELECT timestamp
FROM taipei
GROUP BY timestamp
HAVING SUM(class='bus')>=1
   AND SUM(class='car')>=5
LIMIT 10 GAP 300
```

(b) The FrameQL query for selecting 10 frames of at least one bus and five cars, with each frame at least 300 frames apart (10s at 30 fps).

```
SELECT *
FROM taipei
WHERE class = 'bus' AND redness(content) >= 17.5
  AND area(mask) > 100000
GROUP BY trackid HAVING COUNT(*) > 15
```

(c) The FrameQL query for selecting all the information of red buses at least 100,000 pixels large, in the scene for at least 0.5s (15 frames). The last constraint is for noise reduction.

Figure 2: Three FrameQL example queries. As shown, the syntax is largely standard SQL.

2. USE CASES

Recall that we focus on the batch setting in this work. We give several scenarios where BlazeIt could apply:

Urban planning. Given a set of traffic cameras at street corners, an urban planner performs traffic metering based on the number of cars that pass by the intersections, and determines the busiest times [69]. The planner is interested in how public transit interacts with congestion [16] and looks for clips of at least one bus and at least five cars. Then, the planner seeks to understand how tourism affects traffic and looks for red buses as a proxy for tour buses (see Figure 3).

Autonomous vehicle analysis. An analyst studying AVs notices anomalous behavior when the AV is in front of a yellow light and there are multiple pedestrians in the crosswalk [23], and searches for such events.

Store planning. A retail store owner places a CCTV in

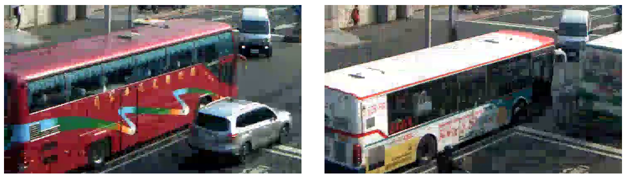

(a) Red tour bus. (b) White transit bus.

Figure 3: Examples of buses in `taipei`. A city planner might be interested in distinguishing tour buses from transit buses and uses color as a proxy.

the store [66]. The owner segments the video into aisles and counts the number of people that walk through each aisle to understand the flow of customers. This information can be used for planning store and aisle layout.

Ornithology. An ornithologist (a scientist who studies birds) is interested in understanding bird feeding patterns, so places a webcam in front of a bird feeder [1]. Then, the ornithologist puts different bird feed on the left and right side of the feeder and counts the number of birds that visit each side. Finally, as a proxy for species, the ornithologist might then select red or blue birds.

These queries can be answered using spatiotemporal information of objects in the video, along with simple user-defined functions (UDFs) over the content of the boxes. Thus, these applications illustrate a need for a unified method of expressing such queries.

3. BLAZEIT SYSTEM OVERVIEW

BlazeIt's goal is to execute FrameQL queries as quickly as possible; we describe FrameQL in Section 4. To execute FrameQL queries, BlazeIt uses an object detection method, an entity resolution method, and the optional user-defined functions (UDFs). We describe the specification of these methods in this section and describe our defaults in Section 8. Importantly, we assume the object detection class types are provided.

BlazeIt executes queries quickly by avoiding materialization using the techniques described Sections 6 and 7. Throughout, BlazeIt uses proxy models, typically specialized neural networks [47, 68] to avoid materialization (Figure 1b), which we describe below.

3.1 Components

Configuration. We assume the object detection method is implemented with the following API:

$$\texttt{OD(frame)} \rightarrow \texttt{Set<Tuple<class, box>>} \quad (1)$$

and the object classes (i.e., types) are returned. We assume the entity resolution takes two nearby frames and boxes and returns true if the boxes correspond to the same object. While we provide defaults (Section 8), the object detection and entity resolution methods can be changed, e.g., a license plate reader could be used for resolving the identity of cars. The UDFs can be used to answer more complex queries, such as determining color, filtering by object size or location, or fine-grained classification. UDFs are functions that accept a timestamp, mask, and rectangular set of pixels. For example, to compute the "redness" of an object, a UDF could average the red channel of the pixels.

Proxy models. BlazeIt can infer proxy models and/or filters from query predicates, many of which must be trained from data.

To train these proxy models, we assume that a small set of tuples are materialized by performing object detection, which we refer to as the *labeled set*. The labeled set is used to train the proxy models and is further split into training data and held-out data. This labeled set can be constructed once, offline, and shared for multiple queries later.

BlazeIt's optimizer must account for error rates in proxy models or filters. To account for this error rate, BlazeIt uses a held-out set of frames to estimate the selectivity and error rate. Given an error budget, BlazeIt's query optimizer uses rule-based optimization to select the fastest query plan. Finally, for limit queries, BlazeIt can always ensure no false positives by running the most accurate method on the relevant frames.

Specialized neural networks. Throughout, we use specialized NNs [47, 67], specifically a miniaturized ResNet [34] (§8), to improve query execution performance. A specialized NN is a NN that mimics a larger NN (e.g., Mask R-CNN) on a simplified task, e.g., on a marginal distribution of the larger NN. As specialized NNs predict simpler output, they can run dramatically faster.

Specialized NNs can be used as proxy models to improve aggregation and limit queries. BlazeIt will infer if a specialized NN can be trained from the query specification. For example, to replicate NoScope's binary detection, BlazeIt would infer that there is a predicate for whether or not there is an object of interest in the frame and train a specialized NN to predict this. Prior work has used specialized NNs for binary detection [31, 47], but we extend specialization to count and perform multi-class classification. Additionally, we apply statistical methods over the results of specialized NNs to accurately answer aggregation and limit queries.

3.2 Limitations

While BlazeIt can answer a significantly wider range of video queries than prior work, we highlight several limitations.

Labeled set. BlazeIt requires the object detection method to be run over a portion of the data for training specialized NNs and filters. Other contemporary systems also require a labeled set [38, 47].

Model Drift. BlazeIt assumes the labeled set follows the same distribution as the remaining video. If the distribution changes dramatically in the unseen video (e.g., a sunny day vs. a foggy day), BlazeIt will need a new labeled set. To our knowledge, tracking model drift in visual data has not been well characterized and existing systems do not handle model drift [38, 47]. We see the detection and mitigation of model drift as an exciting area of future work.

Object detection. BlazeIt depends on the object detection method and does not support object classes beyond what the method returns, e.g., the pretrained Mask R-CNN [26, 33] can detect cars, but cannot distinguish between sedans and SUVs. However, users can supply UDFs if necessary.

Scale-out. Our BlazeIt prototype is written for a single server. However, as the majority of BlazeIt's operators are data-parallel, subsets of the video can be processed on different servers, which could be implemented on a system

Field	Type	Description
`timestamp`	float	Time stamp
`class`	string	Object class (e.g., bus, car)
`mask`	(float, float)*	Polygon containing the object of interest, typically a rectangle
`trackid`	int	Unique identifier for a continuous time segment when the object is visible
`content`	float*	Content of pixels in mask
`features`	float*	The feature vector output by the object detection method.

Table 1: FrameQL's data schema contains spatiotemporal and content information related to objects of interest, as well as metadata (class, identifiers). Each tuple represents an object appearing in one frame; thus a frame may have many or no tuples. The features can be used for downstream tasks.

such as Scanner [62].

4. FRAMEQL: EXPRESSING COMPLEX VISUAL QUERIES

To address the need for a unifying query language over video analytics, we introduce FrameQL, a extension of SQL for querying spatiotemporal information of objects in video. By providing a table-like schema using the standard relational algebra, we enable users with only familiarity with SQL to query videos, whereas implementing these queries manually would require expertise in deep learning, computer vision, and programming.

FrameQL is inspired by prior query languages for video analytics [18, 49, 51, 55], but FrameQL specifically targets information that can be populated using computational methods. We discuss difference in detail at the end of this section.

FrameQL data model. FrameQL represents videos (stored and possibly compressed in formats such as H.264) as virtual relations, with one relation per video. Each FrameQL tuple corresponds to a single object in a frame. Thus, a frame can have zero or more tuples (i.e., zero or more objects), and the same object can have zero or more tuples associated with it (i.e., appear in several frames).

We show FrameQL's data schema in Table 1. It contains fields relating to the time, location, object class, and object identifier, the box contents, and the features from the object detection method. BlazeIt can automatically populate `mask`, `class`, and `features` from the object detection method (see Eq. 1), `trackid` from the entity resolution method, and `timestamp` and `content` from the video metadata. Users can override the default object detection and entity resolution methods. For example, an ornithologist may use an object detector that can detect different species of birds, but an autonomous vehicle analyst may not need to detect birds at all.[1]

FrameQL query format. FrameQL allows selection, projection, and aggregation of objects, and, by returning relations, can be composed with standard relational operators. We show the FrameQL syntax in Figure 4. FrameQL extends SQL in three ways: `GAP`, syntax for specifying an error tolerance, and `FCOUNT`. Notably, we do not support joins as we do not optimize for joins in this work, but joins could be easily added. We show FrameQL's extensions Table 2; several were taken from BlinkDB [5]. We provide the motivation behind each additional piece of syntax.

[1]BlazeIt will inherit any errors from the object detection and entity resolution methods.

```
SELECT * | expression [, ...]
  FROM table_name
  [ WHERE condition  ]
  [ GROUP BY expression [, ...]  ]
  [ HAVING condition [, ...]  ]
  [ LIMIT count  ]
  [ GAP count  ]
  [ ERROR WITHIN tol AT CONFIDENCE conf  ]
```

Figure 4: FrameQL syntax.

Syntactic element	Description
`FCOUNT`	Frame-averaged count (equivalent to time-averaged count), i.e., `COUNT(*) / MAX(timestamp)`
`ERROR WITHIN`	Absolute error tolerance
`FPR WITHIN`	Allowed false positive rate
`FNR WITHIN`	Allowed false negative rate
`CONFIDENCE`	Confidence level
`GAP`	Minimum distance between returned frames

Table 2: Additional syntactic elements in FrameQL. Some of these were adapted from BlinkDB [5].

First, when the user selects timestamps, the `GAP` keyword ensures that the returned frames are at least `GAP` frames apart. For example, if 10 consecutive frames contains the event and `GAP = 100`, only one frame of the 10 frames would be returned.

Second, as in BlinkDB [5], users may wish to have fast responses to exploratory queries and may tolerate some error. Thus, we allow the user to specify error bounds in the form of maximum absolute error, false positive error, and false negative error, along with a specified confidence level (e.g., Figure 2a). NoScope's pipeline can be replicated with FrameQL using these constructs. We choose absolute error bounds in this work as the user may inadvertanely execute a query with 0 records, which would require scanning the entire video (§6).

We also provide a short-hand for returning a frame-averaged count, which we denote as `FCOUNT`. For example, consider two videos: 1) a 10,000 frame video with one car in every frame, 2) a 10 frame video with a car only in the first frame. `FCOUNT` would return 1 in the first video and 0.1 in the second video. As videos vary in length, this allows for a normalized way of computing errors. `FCOUNT` can easily be transformed into a time-averaged count.

FrameQL examples. We describe how the some of the example use cases from §2 can be written in FrameQL. We assume the video is recorded at 30 fps.

Figure 2a shows how to count the average number of cars in a frame. The query uses `FCOUNT` as the error bounds are computed per-frame. Figure 2b shows how to select frames with at least one bus and at least five cars. This query uses the `GAP` keyword to ensure events are a certain time apart. At 30 fps, `GAP 300` corresponds to 10 seconds. Figure 2c shows how to exhaustively select frames with red buses. Here, `redness` and `area` are UDFs, as described in §3. The other example use cases can be answered similarly.

Comparison to prior languages. Prior visual query

Method	mAP	FPS
YOLOv2 [63]	25.4	80
Mask R-CNN [33]	45.2	3
Specialized NN	N/A	10,000
Color filter	N/A	100,000

Table 3: A comparison of object detection methods, filters, and speeds. More accurate object detection methods are more expensive. Specialized NNs and simple filters are orders of magnitude more efficient than object detection methods.

engines have proposed similar schemas, *but assume that the relation is already populated* [48, 53], i.e., that the data has been created through external means (typically by humans). In contrast, FrameQL's relation can be automatically populated by BlazeIt. However, as we focus on exploratory queries in this work, FrameQL's schema is *virtual* and rows are only populated as necessary for the query at hand, which is similar to an unmaterialized view. This form of laziness enables a variety of optimizations via query planning.

5. QUERY OPTIMIZER OVERVIEW

Overview. BlazeIt's primary challenge is executing FrameQL queries *efficiently*: recall that object detection is the overwhelming bottleneck and will likely become more expensive (Table 3). To optimize and execute queries, BlazeIt inspects query contents to see if optimizations can be applied. For example, BlazeIt cannot optimize aggregation queries without error bounds, but can optimize aggregation queries with a user-specified error tolerance.

BlazeIt leverages two novel optimizations to reduce the computational cost of object detection, targeting aggregation (§6) and limit queries (§7). As the filters and specialized NNs we consider are cheap compared to the object detection methods, they are almost always worth calling: a filter that runs at 100,000 fps would need to filter 0.003% of the frames to be effective (Table 3). Thus, we have found a rule-based optimizer to be sufficient in optimizing FrameQL queries.

BlazeIt also can optimize exhaustive selection queries with predicates by implementing optimizations in prior work, such as using NoScope's specialized NNs as a filter [47, 56]. As this case has been studied, we defer the discussion of BlazeIt's query optimization for exhaustive selection to an expended version of this paper [46].

BlazeIt's rule-based optimizer will inspect the query specification to decide which optimizations to apply. First, if the query specification contains an aggregation keyword, e.g., `FCOUNT`, BlazeIt will apply our novel optimization for fast aggregation. Second, if the query specification contains the `LIMIT` keyword, BlazeIt will apply our novel optimization for limit queries. Finally, for all other queries, BlazeIt will default to applying filters similar to NoScope's [47].

We describe the intuition, the physical operator(s), its time complexity and correctness, and the operator selection procedure for aggregates (§6) and limit queries (§7) below.

Work reuse. In addition to our novel optimizations, BlazeIt can reuse work by storing the specialized NN model weights and their results. The specialized NNs BlazeIt uses are small, e.g., < 2 MB, compared to the size of the video.

```
Data: Labeled set, unseen video,
$uerr \leftarrow$ user's requested error rate,
$conf \leftarrow$ user's confidence level
Result: Estimate of requested quantity
if training data has instances of object then
    train specialized NN on labeled set;
    $err \leftarrow$ specialized NN error rate;
    $\tau \leftarrow$ average of specialized NN over unseen video;
    if $P(err < uerr) < conf$ then
        return $\tau$;
    else
        $\hat{m} \leftarrow$ result of Equation 3 (control variates);
        return $\hat{m}$;
    end
else
    Return result of r.s.;
end
```
Algorithm 1: BlazeIt's Aggregation Query Procedure

6. OPTIMIZING AGGREGATES

Overview. In an aggregation query, the user is interested in some statistic over the data, such as the average number of cars per frame; an example is given in Figure 2a. To exactly answer these queries, BlazeIt must call object detection on every frame, which is prohibitively slow. However, if the user specifies an error tolerance, BlazeIt can leverage a range of optimizations for accelerated query execution.

We focus on optimizing counting the number of objects in a frame. BlazeIt requires training data from the labeled set (§2) of the desired quantity (e.g., number of cars) to leverage specialized NNs. If there is insufficient training data, BlazeIt will default to r.s. If there is sufficient training data, BlazeIt will first train a specialized NN to estimate the statistic: if the specialized NN is accurate enough, BlazeIt can return the answer directly. Otherwise, BlazeIt will use specialized NNs to reduce the variance of AQP via control variates, requiring fewer samples. We next describe the steps in detail.

Operator Selection. The intuition above is formalized in Algorithm 1. BlazeIt first determines whether there is sufficient training data ($> 1\%$ of the data has instances of the object) to train a specialized NN. In cases where the training data does not contain enough examples of interest (e.g., a video of a street intersection is unlikely to have bears), BlazeIt will default to standard AQP. We use a slightly modified adaptive sampling algorithm that respects the user's error bound, which is inspired by Online Aggregation [35] and BlinkDB [5].

When there is sufficient training data, BlazeIt will train a specialized NN and estimate its error rate on a held-out set. If the error is smaller than the specified error at the confidence level, it will then execute the specialized NN on the unseen data (subsampled by the Nyquist rate [57]) and return the answer directly. As specialized NNs are significantly faster than object detection, this procedure results in much faster execution.

When the specialized NN is not accurate enough, it is used as a control variate: a cheap-to-compute auxiliary variable correlated with the true statistic. Control variates can approximate the statistic with fewer samples than naive sampling.

Physical Operators. We describe the procedures for sampling, query rewriting, and control variates below.

Sampling. When the query contains a tolerated error rate and there is not sufficient training data for a specialized NN, BlazeIt samples from the video, populating at most a small number of rows for faster execution. Similar to online aggregation [35], we provide absolute error bounds. However, we present a sampling procedure that terminates based on the sampling variance and a CLT bound [54]. This procedure allows variance reduction methods to terminate early, which is useful for control variates.

For an absolute error bound of ϵ (irrespective of the confidence level), we require at least $\frac{K}{\epsilon}$ samples, where K is the range of the estimated quantity (derived from an ϵ-net argument [32]). For example, if the user queries for the average number of cars per frame, K would be the maximum number of cars over all frames plus one.

Thus, in BlazeIt's adaptive sampling procedure, we begin with $\frac{K}{\epsilon}$ samples. At step n, we increase the number of samples by $\frac{nK}{\epsilon}$. We terminate when the CLT bound gives that the error rate is satisfied at the given confidence level, namely,

$$Q(1-\tfrac{\delta}{2}) \cdot \hat{\sigma}_N < \epsilon \tag{2}$$

where δ is the confidence level, $\hat{\sigma_N}$ is the sample standard deviation at round N, and Q is the percent point function (i.e., the inverse of the cumulative distribution function) for the normal distribution [35]. We use the finite sample correction to compute the sample standard deviation.

Query Rewriting via Specialized NNs. In cases where the specialized NN is accurate enough (as determined by the bootstrap on the held-out set; the accuracy of the specialized NN depends on the noisiness of the video and object detection method), BlazeIt can return the answer directly from the specialized NN run over all the frames for dramatically faster execution and bypass the object detection entirely. BlazeIt uses multi-class classification for specialized NNs to count the number of objects in a frame.

To train the specialized NN, BlazeIt selects the number of classes equal to the highest count that is at least 1% of the video plus one. For example, if 1% of the video contains 3 cars, BlazeIt will train a specialized NN with 4 classes, corresponding to 0, 1, 2, and 3 cars in a frame. BlazeIt uses 150,000 frames for training and uses a standard training procedure for NNs (SGD with momentum [34]) for one epoch.

BlazeIt estimates the error of the specialized NN on a held-out set using the bootstrap [19]. If the error is low enough at the given confidence level, BlazeIt will process the unseen data using the specialized NN and return the result.

Control Variates. In cases where the user has a stringent error tolerance, specialized NNs may not be accurate enough to answer a query on their own. To reduce the cost of sampling from the object detector, BlazeIt introduces a novel method to take advantage of specialized NNs while still achieving high accuracy, by combining specialized NNs with AQP-like sampling. In particular, we adapt the method of control variates [28] to video analytics (to our knowledge, control variates have not been applied to database query optimization or video analytics). Specifically, control variates is a method of variance reduction (variance reduction is a standard technique in Monte Carlo sampling [65] and stochastic optimization [43]) which uses a proxy variable correlated with the statistic of interest. Intuitively, by reducing the variance of sampling, we can reduce the number of frames that have to be sampled and processed by the full object detector.

To formalize this intuition, suppose we wish to estimate the expectation m and we have access to an auxiliary variable t. The desiderata for t are that: 1) t is cheaply computable, 2) t is correlated with m (see time complexity). We further assume we can compute $\mathbb{E}[t] = \tau$ and $\text{Var}(t)$ exactly. Then,

$$\hat{m} = m + c(t - \tau) \tag{3}$$

is an unbiased estimator of m for any choice of c [28]. The optimal choice of c is $c = -\frac{\text{Cov}(m,t)}{\text{Var}(t)}$ and using this choice of c gives $\text{Var}(\hat{m}) = (1 - \text{Corr}(m,t)^2)\text{Var}(m)$. As an example, suppose $t = m$. Then, $\hat{m} = m + c(m - \mathbb{E}[m]) = \mathbb{E}[m]$ and $\text{Var}(\hat{m}) = 0$.

This formulation works for arbitrary t, but choices where t is correlated with m give the best results. As we show in §9.2, specialized NNs can provide a correlated signal to the ground-truth object detection method for all queries we consider.

As an example, suppose we wish to count the number of cars per frame. Then, m is the random variable denoting the number of cars the object detection method returns. In BlazeIt, we train a specialized NN to count the number of cars per frame. Ideally, the specialized NN would exactly match the object detection counts, but this is typically not the case. However, the specialized NNs are typically correlated with the true counts. Thus, the random variable t would be the output of the specialized NN. As our choice of specialized NNs are extremely cheap to compute, we can calculate their mean and variance exactly on all the frames. BlazeIt estimates $\text{Cov}(m,t)$ at every round.

Aggregation with query predicates. A user might issue an aggregation query that contains predicates such as filtering for large red buses (see Figure 3). In this case, BlazeIt will execute a similar procedure above, but first applying the predicates to the training data. The key difference is that in cases where there is not enough training data, BlazeIt will instead generate a specialized NN to count the most selective set of predicates that contains enough data.

For example, consider a query that counts the number of large red buses. If there is not enough data to train a specialized NN that counts the number of large red buses, BlazeIt will instead train a specialized NN that counts the number of large buses (or red buses, depending on the training data). If there is no training data for the quantity of interest, BlazeIt will default to standard sampling.

As control variates only requires that the proxy variable, i.e., the specialized NN in this case, be *correlated* with the statistic of interest, BlazeIt will return a correct answer even if it trains a specialized NN that does not directly predict the statistic of interest.

Correctness. BlazeIt's sampling-based methods will return approximate answers that respect the requested error bound and confidence level. While video is temporally correlated, as we assume the batch setting, shuffling the data will result in iid samples. Furthermore, control variates are an unbiased estimator for the statistic of interest [28], so standard proofs of correctness apply to control variates.

Query rewriting using specialized NNs will respect the requested error bound and confidence level under the assumption of no model drift (see §3.2).

Time and sample complexity. BlazeIt must take $c_\delta \frac{\sigma^2}{\epsilon^2}$ samples from a random variable with standard deviation σ (c_δ is a constant that depends on the confidence level, see Equation 2). Denote the standard deviation of r.s. as σ_a and from control variates as σ_c; the amortized cost of running a specialized NN on a single frame as k_s and of the object detection method as k_o; the total number of frames as F.

Control variates are beneficial when $k_s F < k_o \frac{c_\delta}{\epsilon^2}(\sigma_a^2 - \sigma_c^2)$. Thus, as the error bound decreases or the difference in variances increases (which typically happens when specialized NNs are highly accurate or when σ_a is large), control variates become more beneficial.

While σ_a and σ_c depend on the query, we empirically show in §9 that control variates and query rewriting are beneficial.

7. OPTIMIZING LIMIT QUERIES

Overview. In cardinality-limited queries, the user is interested in finding a limited number of events, (e.g., a bus and five cars, see Figure 2b), typically for manual inspection. Limit queries are especially helpful for rare events. To answer these queries, BlazeIt could perform object detection over every frame to search for the events. However, if the events occurs infrequently, naive methods of random sampling or sequential scans of the video can be prohibitively slow (e.g., at 30 fps, an event that occurs once every 30 minutes corresponds to a rate of 1.9×10^{-5}).

Our key intuition is to bias the search towards regions of the video that likely contain the event. To bias the search, we use specialized NNs, and combine them with importance sampling techniques from the rare-event simulation literature [44]. As an example of rare-event simulation, consider the probability of flipping 80 heads out of 100 coin flips. Using a fair coin, the probability of encountering this event is astronomically low (rate of 5.6×10^{-10}), but using a biased coin with $p = 0.8$ can be orders of magnitude more efficient (rate of 1.2×10^{-4}) [44].

Physical operator and selection. BlazeIt currently supports limit queries searching for at least N of an object class (e.g., at least one bus and at least five cars). In BlazeIt, we use specialized NNs to bias which frames to sample:

- If there are no instances of the query in the training set, BlazeIt will default to performing the object detection method over every frame and applying applicable filters as in prior work [47] (random sampling is also possible).
- If there are examples, BlazeIt will train a specialized NN to recognize frames that satisfy the query.
- BlazeIt rank orders the unseen data by the confidence from the specialized NN.
- BlazeIt will perform object detection in the rank order until the requested number of events is found.

For a given query, BlazeIt trains a specialized NN to recognize frames that satisfy the query. While we could train a specialized NN as a binary classifier of the frames that satisfy the predicate and that do not, we have found that rare queries have extreme class imbalance. Thus, we train the specialized NN to predict counts instead, which alleviates the class imbalance issue; this procedure has the additional benefit of allowing the trained specialized NN to be reused for other queries such as aggregation. For example, suppose the user wants to find frames with at least one bus and at least five cars. Then, BlazeIt trains a single specialized NN to separately count buses and cars. BlazeIt use the sum of the probability of the frame having at least one bus and at least five cars as its signal. BlazeIt takes the most confident frames until the requested number of frames is found.

In the case of multiple object classes, BlazeIt trains a single NN to predict each object class separately (e.g., instead of jointly predicting “car” and “bus”, the specialized NN would return a separate confidence for “car” and “bus”), as this results in fewer weights and typically higher performance.

After the results are sorted, the full object detector is applied until the requested number of events is found or all the frames are searched. If the query contains the `GAP` keyword, once an event is found, the surrounding `GAP` frames are ignored.

Limit queries with multiple predicates. As with aggregation queries, a user might issue a limit query with predicates. If there is sufficient training data in the labeled set, BlazeIt can execute the procedure above. If there is not sufficient training data, BlazeIt will train a specialized NN to search for the most selective set of predicates that contains enough data in a similar fashion to generating an aggregation specialized NN.

Correctness. BlazeIt performs object detection on all sampled frames, so it always returns an exact answer. All frames will be exhaustively searched if there are fewer events than the number requested.

Time complexity. Denote K to be the number of events the user requested, N the total number of matching events, and F the total number of frames in the video. We denote, for event i, f_i as the frame where the event occurred. Once an event is found, the `GAP` frames around the event can be ignored, but this is negligible in practice so we ignore it in the analysis.

If $K > N$, then every method must consider every frame in the video, i.e., F frames. From here on, we assume $K \leq N$.

For sequential scans, f_K frames must be examined.

For random sampling, consider the number of frames to find a single event. In expectation, random sampling will consider $\frac{F}{N}$ frames. Under the assumption that $K \ll N \ll F$, then random sampling will consider approximately $\frac{K \cdot F}{N}$ frames.

While using specialized NNs to bias the search does not guarantee faster runtime, we show in §9 that it empirically can reduce the number of frames considered.

8. IMPLEMENTATION

We implement a BlazeIt prototype of the above query optimizer. We implement the control plane in Python 3.5 as the deep learning frameworks we use for object detection require Python and, for efficiency purposes, we implement the non-NN filters in C++. We use PyTorch v1.0 for the training and evaluation of specialized NNs. For object detection, we use FGFA [74] using MXNet v1.2 and Mask R-CNN [33] using the Detectron framework [26] in Caffe v0.8. We modify the implementations to accept arbitrary parts of video. For FGFA, we use the provided pre-trained weights and for Mask R-CNN, we use the pretrained `X-152-32x8d-FPN-IN5k` weights. We ingest video via OpenCV.

Currently, BlazeIt uses a rule-based optimizer for query rewriting [61], which supports the queries in §4. Most queries follow the following general steps: 1) train specialized NNs and filters for the query at hand, 2) compute statistics on a held-out dataset to estimate the error or selectivity of the NNs and filters, 3) choose a plan for the unseen data, 4) execute the plan.

BlazeIt uses a Fluent DSL written in Python to specify FrameQL queries. The cost of storing and materializing the processed data is negligible, so we use Pandas dataframes for processing tuples.

Video ingestion. BlazeIt loads the video and resizes the frames to the appropriate size for each NN (65×65 for specialized NNs, short side of 600 pixels for object detection methods), and normalizes the pixel values appropriately.

Specialized NN training. We train the specialized NNs using PyTorch v1.0. Video are ingested and resized to 65×65 pixels and normalized using standard ImageNet normalization [34]. Standard cross-entropy loss is used for training, with a batch size of 16. We use SGD with a momentum of 0.9. Our specialized NNs use a "tiny ResNet" architecture, a modified version of the standard ResNet architecture [34], which has 10 layers and a starting filter size of 16, for all query types. As this work focuses on exploratory queries, we choose tiny ResNet as a good default and show that it performs better than or on par with the NNs used in [47] in an extended version of this paper [46].

Identifying objects across frames. Our default for computing `trackid` uses motion IOU [74]. Given the set of objects in two consecutive frames, we compute the pairwise IOU of each object in the two frames. We use a cutoff of 0.7 to call an object the same across consecutive frames.

9. EVALUATION

We evaluated BlazeIt on a variety of FrameQL queries on real-world video streams in two scenarios: 1) aggregate queries and 2) limit queries for rare events. We show that:

1. BlazeIt achieves up to 4000× increased throughput compared to a naive baseline, up to a 2500× speedup compared to a binary classification oracle, and up to a 8.7× speedup over AQP on aggregation queries (§9.2).
2. BlazeIt achieves up to 1000× speedup compared to a naive baseline and a 500× speedup compared to a binary oracle for video limit queries (§9.3).

9.1 Experimental Setup

Evaluation queries and videos. We evaluated BlazeIt on six videos shown in Table 4, which were scraped from YouTube. `taipei`, `night-street`, `amsterdam`, and `archie` are widely used in video analytics systems [9, 38, 41, 47, 71] and we collected two other streams. We only considered times where the object detection method can perform well (due to lighting conditions), which resulted in 6-11 hours of video per day. These datasets vary in object class (car, bus, boat), occupancy (12% to 90%), and average duration of object appearances (1.4s to 10.7s). For each webcam, we use three days of video: one day for training labels, one day for threshold computation, and one day for testing, as in [47].

We evaluate on queries similar to Figure 2, in which the class and video were changed.

Object detection methods. We labeled part of each video using Mask R-CNN [33], FGFA [74], and YOLOv2 [63], and manually selected the most accurate method for each video. Mask R-CNN and FGFA are significantly more accurate than YOLOv2, so we did not select YOLOv2 for any video.

In timing the naive baseline, we only included the GPU compute time and exclude the time to process the video and convert tuples to FrameQL format, object detection is the overwhelming computational cost.

Data preprocessing. The literature reports that state-of-the-art object detection methods still suffer in performance for small objects [33, 74]. Thus, we only considered regions where objects are large relative to the size of the frame (these regions are video dependent). Object detectors will return a set of boxes and confidences values. We manually selected confidence thresholds for each video and object class for when to consider an object present, shown in Table 4.

Evaluation metrics. We computed all accuracy metrics with respect to the object detection method, i.e., we treated the object detection method as ground truth. For aggregation queries, we report the absolute error. For limit queries, we guarantee only true positives are returned, thus we only report throughput.

We have found that modern object detection methods can be accurate at the frame level. Thus, we considered accuracy at the *frame level*, in contrast to to the one-second binning that is used in [47] to mitigate label flickering for NoScope.

We measured throughput by timing the complete end-to-end system excluding the time taken to decode video, as is standard [47, 56]. We assume the labeled set is computed offline once, so we excluded the time to generate the labeled set. Unlike in [47], we also show runtime numbers *when the training time of the specialized NN is included.* We include this time as BlazeIt focuses on exploratory queries, whereas NoScope focuses on long-running streams of data. We additionally show numbers where the training time is excluded, which could be achieved if the specialized NNs were indexed ahead of time.

Hardware Environment. We performed our experiments on a server with an NVIDIA Tesla P100 GPU and two Intel Xeon E5-2690v4 CPUs (56 threads). The system has 504 GB of RAM.

9.1.1 Binary Oracle Configuration

Many prior visual analytics systems answer binary classification queries [38, 47, 56] which are the closest systems to BlazeIt. These systems cannot directly answer queries in the form of aggregate or limit queries for multiple instances of an object or objects.

As binary classification is not directly applicable to the tasks we consider, where relevant, we compared against a *binary oracle*, namely a method that returns (on a frame-by-frame basis) whether or not an object class is present in the scene. We assume the oracle is free to query. Thus, this

Video Name	Object	Occupancy	Avg. duration of object in scene	Distinct count	Resol.	FPS	# Eval frames	Length (hrs)	Detection method	Thresh
`taipei`	bus	11.9%	2.82s	1749	720p	30	1188k	33	FGFA	0.2
	car	64.4%	1.43s	32367						
`night-street`	car	28.1%	3.94s	3191	720p	30	973k	27	Mask	0.8
`rialto`	boat	89.9%	10.7s	5969	720p	30	866k	24	Mask	0.8
`grand-canal`	boat	57.7%	9.50s	1849	1080p	60	1300k	18	Mask	0.8
`amsterdam`	car	44.7%	7.88s	3096	720p	30	1188k	33	Mask	0.8
`archie`	car	51.8%	0.30s	90088	2160p	30	1188k	33	Mask	0.8

Table 4: Video streams and object labels queried in our evaluation. We show the data from the test set, as the data from the test set will influence the runtime of the baselines and BLAZEIT.

oracle is strictly more powerful—both in terms of accuracy and speed—than existing systems. We describe how the binary oracle can be used to answer each type of query.

Aggregates. Binary oracles cannot distinguish between one and several objects, so object detection must be performed on every frame with an object to identify the individual objects. Thus, counting cars in `taipei` would require performing object detection on 64.4% of the frames, i.e., the occupancy rate.

Cardinality-limited queries. As above, a binary oracle can be used to filter frames that do not contain the objects of interest. For example, if the query were searching for at least one bus and at least five cars in `taipei`, a binary oracle can be used to remove frames that do not have a bus and a car. Object detection will then be performed on the remaining frames until the requested number of events is found.

9.2 Aggregate Queries

We evaluated BLAZEIT on six aggregate queries across six videos. The queries are similar to the query in Figure 2a, with the video and object class changed. We ran five variants of each query:

- Naive: we performed object detection on every frame.
- Binary oracle: we performed object detection on every frame with the object class present.
- Naive AQP: we randomly sampled from the video.
- BLAZEIT: we used specialized NNs and control variates for efficient sampling.
- BLAZEIT (no train): we excluded the training time from BLAZEIT.

There are two qualitatively different execution modes: 1) where BLAZEIT rewrites the query using a specialized NN and 2) where BLAZEIT samples using specialized NNs as control variates (§6). We analyzed these cases separately.

Query rewriting via specialized NNs. We evaluated the runtime and accuracy of specialized NNs when the query can be rewritten by using a specialized NN. We ran each query with a target error rate of 0.1 and a confidence level of 95%. We show the average of three runs. Query rewriting was unable to achieve this accuracy target for `archie`, so we excluded it. However, we show below that specialized NNs can be used as a control variate even in this case.

As shown in Figure 5, BLAZEIT can achieve up to 8500× speedup if the NN is cached and a 3200× speedup when including the training time and the time to compute thresholds. In contrast, [47] does not include this time in their evaluation. The binary oracle baseline does not perform well when the video has many objects of interest (e.g., `rialto`).

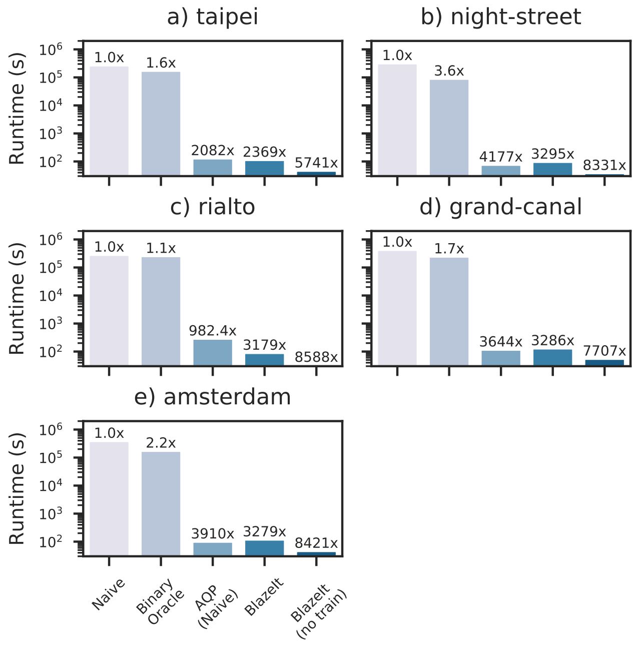


Figure 5: End-to-end runtime of baselines and BLAZEIT on aggregate queries where the query is rewritten with a specialized network, measured in seconds (log scale). All queries targeted an error of 0.1.

Video Name	Error
`taipei`	0.043
`night-street`	0.022
`rialto`	-0.031
`grand-canal`	0.081
`amsterdam`	0.050

Table 5: Average error of 3 runs of query-rewriting using a specialized NN for counting. These videos stayed within an error of 0.1.

In some cases, naive AQP outperforms BLAZEIT when BLAZEIT trains the specialized NNs from scratch. However, in all cases, BLAZEIT outperforms AQP when the NNs are cached.

While specialized NNs do not provide error guarantees, we show that the absolute error stays within the 0.1 for the given videos in Table 5. This shows that specialized NNs can be used for query rewriting while respecting the user's error bounds.

Video Name	Pred (day 1)	Actual (day 1)	Pred (day 2)	Actual (day 2)
`taipei`	0.86	0.85	1.21	1.17
`night-street`	0.76	0.84	0.40	0.38
`rialto`	2.25	2.15	2.34	2.37
`grand-canal`	0.95	0.99	0.87	0.81

Table 6: Estimated and true counts for specialized NNs run on two different days of video.

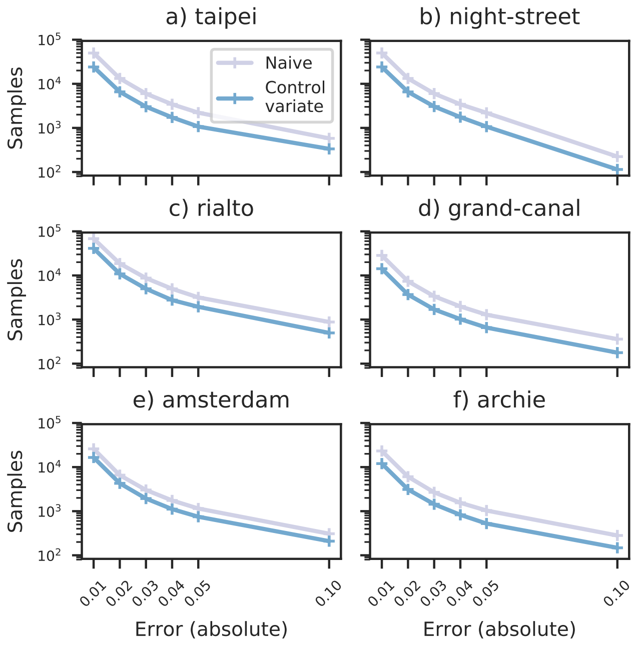


Figure 6: Sample complexity of naive AQP and AQP with control variates. Note the y-axis is on a log scale.

Sampling and control variates. We evaluated the runtime and accuracy of sampling with control variates. Because of the high computational cost of running object detection, we ran the object detection method once and recorded the results. Thus, the run times in this section are estimated from the number of object detection calls.

We targeted error rates of 0.01, 0.02, 0.03, 0.04, 0.05, and 0.1 with a confidence level of 95%. We averaged the number of samples for each error level over 100 runs.

As shown in Figure 6, using specialized NNs as a control variate can deliver up to a 2× reduction in sample complexity. As predicted by theory, the reduction in variance depends on the correlation between the specialized NNs and the object detection methods. Specifically, as the correlation coefficient increases, the sample complexity decreases.

Sampling with predicates. We evaluated the runtime of BlazeIt on aggregation queries with predicates. We evaluated on one query per video and counted the number of objects with a given color and at least a given size; full query details are give in an extended version of this paper [46]. We targeted an error rate of 0.001.

As shown in Figure 7, using specialized NNs as control variates can deliver up to a 2.4× speedup compared to naive AQP. While the absolute runtimes vary depending on the difficulty of the query, the relative gain of BlazeIt's control variates only depends on the reduction in variance. Finally, we note the gains are lower compared to queries with predicates as there is less training data.

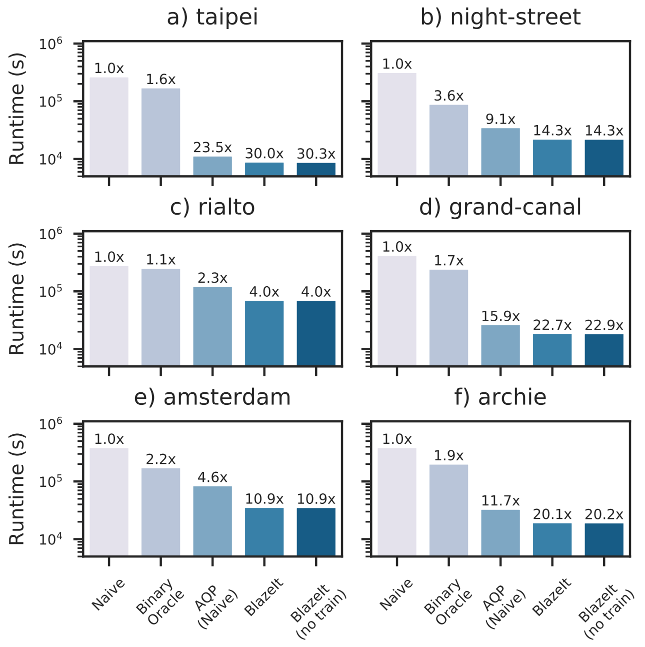


Figure 7: Runtime of BlazeIt and baselines for aggregation queries with predicates. Note the y-axis is on a log scale. As shown, BlazeIt can deliver up to 2.4× speedups over naive AQP.

Specialized NNs do not learn the average. Specialized NNs may perform well by simply learning the average number of cars. To demonstrate that they do not, we swapped the day of video for choosing thresholds and testing data. We show the true counts for each day and the average of 3 runs in Table 6. Notably, we see that the specialized NNs return different results for each day. This shows that the specialized NNs do not learn the average and return meaningful results.

9.3 Cardinality-limited Queries

We evaluated BlazeIt on limit queries, in which frames of interest are returned to the user, up to the requested number of frames. The queries are similar to the query in Figure 2b. We show in Table 7 the query details and the number of instances of each query. If the user queries more than the maximum number of instances, BlazeIt must query every frame. Thus, we chose queries with at least 10 instances.

BlazeIt will only return true positives for limit queries (§7), thus we only report the runtime. Additionally, if we suppose that the videos are indexed with the output of the specialized NNs, we can simply query the frames using information from the index. This scenario might occur when the user executed an aggregate query as above. Thus, we additionally report sample complexity.

We ran the following variants:

- Naive: we performed object detection sequentially until the requested number of frames is found.
- Binary oracle: we performed object detection over the frames containing the object class(es) of interest until the requested number of frames is found.

Video name	Object	Number	Instances
`taipei`	car	6	70
`night-street`	car	5	29
`rialto`	boat	7	51
`grand-canal`	boat	5	23
`amsterdam`	car	4	86
`archie`	car	4	102

Table 7: Query details and number of instances. We selected rare events with at least 10 instances.

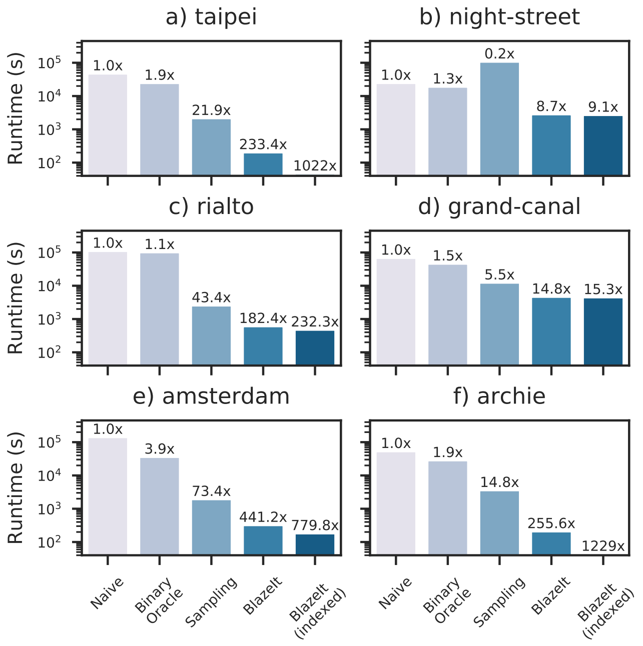


Figure 8: End-to-end runtime of baselines and BLAZEIT on limit queries. The y-axis is log-scaled. All queries looked for 10 events. The mean of three runs is shown.

- Sampling: we randomly sampled the video until the requested number of events is found.
- BLAZEIT: we use specialized NNs as a proxy signal to rank the frames (§7).
- BLAZEIT (indexed): we assume the specialized NN has been trained and run over the remaining data, as might happen if a user runs several queries about some class.

Single object class. Figure 8 shows that BLAZEIT can achieve over a 1000× speedup compared to baselines. We see that the baselines do poorly in finding rare objects, where BLAZEIT's specialized NNs can serve as a high-fidelity signal.

We also varied the number of cars in `taipei` to see if BLAZEIT could also search for common objects. As shown in Figure 9, the sample complexity increases as the number of cars increases for both the naive method and the binary oracle. However, for up to 5 cars, BLAZEIT's sample complexity remains nearly constant, which demonstrates the efficacy of biased sampling. While BLAZEIT shows degraded performance with 6 cars, there are only 70 such instances, and is thus significantly harder to find.

Multiple object classes. We tested BLAZEIT on multiple object classes by searching for at least one bus and at least five cars in `taipei`. There are 63 instances in the test set.

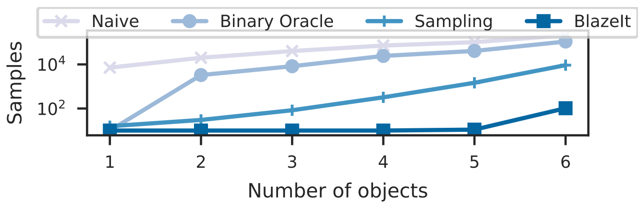


Figure 9: Sample complexity of baselines and BLAZEIT when searching for at least N cars in `taipei`. Note the y-axis is on a log-scale. All queries looked for 10 events.

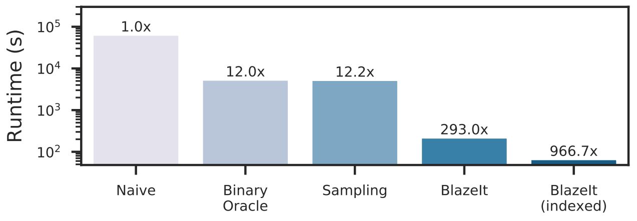


Figure 10: End-to-end runtime of baselines and BLAZEIT on finding at least one bus and at least five cars in `taipei`. Note the y-axis is on a log scale.

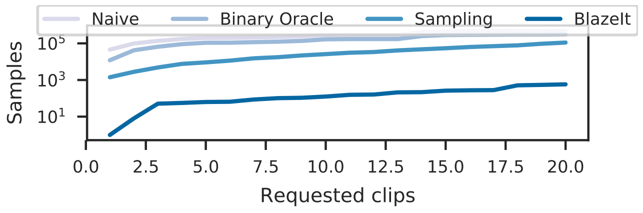


Figure 11: Sample complexity of BLAZEIT, the binary oracle, and the naive method when searching for at least one bus and at least five cars in `taipei`. The x-axis is the number of requested frames. Note the y-axis is on a log scale.

As shown in Figure 10, BLAZEIT outperforms the naive baseline by up to 966×. Searching for multiple object classes is favorable for the binary oracle, as it becomes more selective. Nonetheless, BLAZEIT significantly outperforms the binary oracle, giving up to a 81× performance increase.

Additionally, we show the sample complexity as a function of the `LIMIT` in Figure 11 of BLAZEIT and the baselines, for `taipei`. We see that BLAZEIT can be up to orders of magnitude more sample efficient over both the naive baseline and the binary oracle.

Limit queries with predicates. We evaluated BLAZEIT on limit queries with predicates by searching for objects with a specified color and at least a specified size. We present the full query details and total number of objects in an extended version of this paper for brevity [46].

As shown in Figure 12, BLAZEIT outperforms all baselines by up to 300×, even when including the time to train a proxy model. BLAZEIT will especially outperform baselines on queries that have few matches, as random sampling will perform poorly in these settings.

10. RELATED WORK

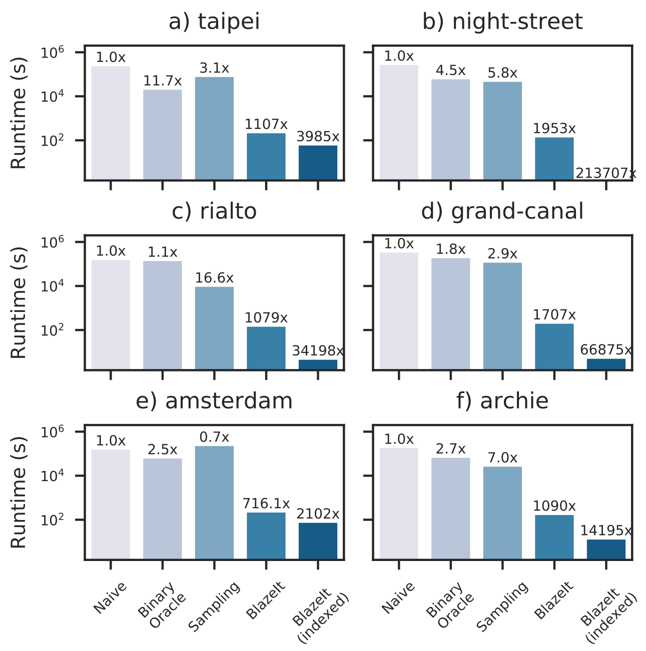


Figure 12: Runtime of BlazeIt and baselines on limit queries with predicates. As shown, BlazeIt's novel optimization outperforms all baselines, even when including the training time of the proxy model. BlazeIt especially outperforms baselines when the selectivity is high: random sampling will perform especially poorly on rare events.

BlazeIt builds on research in data management for multimedia and video, and on recent advances in computer vision. We outline some relevant parts of the literature below.

AQP. In AQP systems, the result of a query is returned significantly faster by subsampling the data [22]. Typically, the user specifies an error bound [5], or the error bound is refined over time [35]. Prior work has leveraged various sampling methods [4, 11], histograms [3, 15, 27, 60], and sketches [10, 36, 42].

We leverage ideas from this space and introduce a new form of variance reduction in the form of control variates [28] via specialized NNs. This form of variance reduction, and others involving auxiliary variables, does not apply in a traditional relational database: the cost of materializing a tuple must be large compared to computing the auxiliary variable.

Visual data management. Visual data management has aimed to organize and query visual data, starting from systems such as Chabot [58] and QBIC [21]. These systems were followed by a range of "multimedia" database for storing [8,52], querying [7,50,59], and managing [25,40,72] video data. The literature also contains many proposals for query languages for visual data [17, 39, 55].

Many of these systems and languages use classic computer vision techniques such as *low-level* image features (e.g color) and rely on textual annotations for semantic queries. However, recent advances in computer vision allow the *automatic* population of semantic data and thus we believe it is critical to reinvestigate these systems. In this work, we explicitly choose to extend SQL in FrameQL and focus on how these fields can be *automatically populated* rather than the syntax.

Modern video analytics. Systems builders have created video analytics systems, such NoScope [47], a highly tuned pipeline for binary detection: it returns the presence or absence of a particular object class in video. Other systems, e.g., Focus [38] and Tahoma [6], have also optimized binary detection. However, these systems are inflexible and cannot adapt to user's queries. Additionally, as NoScope does not focus on the exploratory setting, it does not optimize the training time of specialized NNs. In BlazeIt, we extend specialization and present novel optimizations for aggregation and limit queries, which these systems do not support.

Other contemporary work use filters with a false negative rate (called probabilistic predicates) that are automatically learned from a hold-out set [56]. These could be incorporated into BlazeIt for selection queries.

Other systems aim to reduce latency of live queries (e.g., VideoStorm [73]) or increase the throughput of batch analytics queries (e.g., Scanner [62]) that are pre-defined *as a computation graph*. As the computation is specified as a black-box, these systems do not have access to the semantics of the computation to perform certain optimizations, such as in BlazeIt. In BlazeIt, we introduce FrameQL and an optimizer that can infer optimizations from the given query. Additionally, BlazeIt could be integrated with VideoStorm for live analytics or Scanner for scale-out.

We presented a preliminary version of BlazeIt as a non-archival demo [45].

Speeding up deep networks. We briefly discuss two of the many forms of improving deep network efficiency.

First, a large body of work changes the NN architecture or weights for improved inference efficiency, that preserve the full generality of these NNs. Model compression uses a variety of techniques from pruning [30] to compressing [12] weights from the original NN, which can be amenable to hardware acceleration [29]. Model distillation uses a large NN to train a smaller NN [37]. These methods are largely orthogonal to BlazeIt, and reducing the cost of object detection would also improve BlazeIt's runtime.

Second, specialization [47, 67] aims to improve inference speeds by training a small NN to mimic a larger NN *on a reduced task*. However, specialization has typically been applied in *specific* pipelines, such as for binary detection. In BlazeIt, we extend specialization to counting and multi-class classification. Further, we show to how use specialized NNs as control variates and to bias searches for rare events.

11. CONCLUSIONS

Querying video for semantic information has become possible with recent advances in computer vision. However, these NNs run up to 10× slower than real-time and requires complex programming with low-level libraries to deploy. In response, we present a declarative language for video analytics, FrameQL, and BlazeIt, a system that accepts, automatically optimizes, and executes FrameQL queries. We demonstrate that FrameQL can answer a range of real-world queries, including queries in prior work, of which we focus on optimizing aggregation and limit queries. BlazeIt introduces new techniques based on AQP, Monte Carlo sampling, and rare-event simulation, and extends specialization to answer these exploratory queries up to two orders of magnitude faster than baselines. These results suggest that new classes of queries can be answered over large video datasets with orders of magnitude lower computational cost.

12. REFERENCES

[1] Cornell lab bird cams. `http://cams.allaboutbirds.org/`.

[2] Cctv: Too many cameras useless, warns surveillance watchdog tony porter, 2015.

[3] S. Acharya, P. B. Gibbons, and V. Poosala. Aqua: A fast decision support systems using approximate query answers. In *PVLDB*, pages 754–757, 1999.

[4] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *SIGMOD*, pages 481–492. ACM, 2014.

[5] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42. ACM, 2013.

[6] M. R. Anderson, M. Cafarella, T. F. Wenisch, and G. Ros. Predicate optimization for a visual analytics database. *ICDE*, 2019.

[7] W. Aref, M. Hammad, A. C. Catlin, I. Ilyas, T. Ghanem, A. Elmagarmid, and M. Marzouk. Video query processing in the vdbms testbed for video database research. In *International Workshop on Multimedia Databases*, pages 25–32. ACM, 2003.

[8] F. Arman, A. Hsu, and M.-Y. Chiu. Image processing on compressed data for large video databases. In *International Conference on Multimedia*, pages 267–272. ACM, 1993.

[9] C. Canel, T. Kim, G. Zhou, C. Li, H. Lim, D. Andersen, M. Kaminsky, and S. Dulloor. Scaling video analytics on constrained edge nodes. *SysML*, 2019.

[10] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP*, pages 693–703. Springer, 2002.

[11] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 2007.

[12] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. In *ICML*, pages 2285–2294, 2015.

[13] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Re, and M. Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *arXiv preprint arXiv:1806.01427*, 2018.

[14] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. *Training*, 100(101):102, 2017.

[15] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Space-and time-efficient deterministic algorithms for biased quantiles over data streams. In *SIGMOD*, pages 263–272. ACM, 2006.

[16] J. De Cea and E. Fernández. Transit assignment for congested public transport systems: an equilibrium model. *Transportation science*, 27(2):133–147, 1993.

[17] U. Demir, M. Koyuncu, A. Yazici, T. Yilmaz, and M. Sert. Flexible content extraction and querying for videos. In *FQAS*, pages 460–471. Springer, 2011.

[18] M. E. Dönderler, E. Şaykol, U. Arslan, Ö. Ulusoy, and U. Güdükbay. Bilvideo: Design and implementation of a video database management system. *Multimedia Tools and Applications*, 2005.

[19] B. Efron and R. J. Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

[20] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part-based models. *TPAMI*, 2010.

[21] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, et al. Query by image and video content: The qbic system. *computer*, 28(9):23–32, 1995.

[22] M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, 2001.

[23] A. Geiger, P. Lenz, and R. Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *CVPR*. IEEE, 2012.

[24] S. Geisser. *Predictive inference*. Routledge, 2017.

[25] S. Gibbs, C. Breiteneder, and D. Tsichritzis. Audio/video databases: An object-oriented approach. In *ICDE*. IEEE, 1993.

[26] R. Girshick, I. Radosavovic, G. Gkioxari, P. Dollár, and K. He. Detectron. `https://github.com/facebookresearch/detectron`, 2018.

[27] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, volume 30, pages 58–66. ACM, 2001.

[28] J. M. Hammersley and D. C. Handscomb. General principles of the monte carlo method. In *Monte Carlo Methods*, pages 50–75. Springer, 1964.

[29] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: efficient inference engine on compressed deep neural network. In *ISCA*, pages 243–254. IEEE, 2016.

[30] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[31] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys*, pages 123–136. ACM, 2016.

[32] D. Haussler and E. Welzl. -nets and simplex range queries. *Discrete & Computational Geometry*, 2(2):127–151, 1987.

[33] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn. In *ICCV*, pages 2980–2988. IEEE, 2017.

[34] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[35] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Acm Sigmod Record*, volume 26, pages 171–182. ACM, 1997.

[36] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. *External memory algorithms*, 50:107–118, 1998.

[37] G. Hinton, O. Vinyals, and J. Dean. Distilling the

knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[38] K. Hsieh, G. Ananthanarayanan, P. Bodik, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying large video datasets with low latency and low cost. *OSDI*, 2018.

[39] E. Hwang and V. Subrahmanian. Querying video libraries. *Journal of Visual Communication and Image Representation*, 1996.

[40] R. Jain and A. Hampapur. Metadata in video databases. *ACM Sigmod Record*, 23(4):27–33, 1994.

[41] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 253–266. ACM, 2018.

[42] C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou. Dynamically maintaining frequent items over a data stream. In *CIKM*, pages 287–294. ACM, 2003.

[43] R. Johnson and T. Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *NIPS*, pages 315–323, 2013.

[44] S. Juneja and P. Shahabuddin. Rare-event simulation techniques: an introduction and recent advances. *Handbooks in operations research and management science*, 13:291–350, 2006.

[45] D. Kang, P. Bailis, and M. Zaharia. Challenges and opportunities in dnn-based video analytics: A demonstration of the blazeit video query engine. CIDR, 2019.

[46] D. Kang, P. Bailis, and M. Zaharia. Optimizing declarative aggregation and limit queries for neural network-based video analytics. `https://ddkang.github.io/papers/2019/blazeit-vldb.pdf`, 2019.

[47] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. *PVLDB*, 10(11):1586–1597, 2017.

[48] T. C. Kuo and A. L. Chen. A content-based query language for video databases. In *ICMCS*, pages 209–214. IEEE, 1996.

[49] T. C. Kuo and A. L. Chen. Content-based query processing for video databases. *IJDTA*, 2(1):1–13, 2000.

[50] M. La Cascia and E. Ardizzone. Jacob: Just a content-based query system for video databases. In *ICASSP*. IEEE, 1996.

[51] T.-L. Le, M. Thonnat, A. Boucher, and F. Brémond. A query language combining object features and semantic events for surveillance video retrieval. In *MMM*. Springer, 2008.

[52] J. Lee, J. Oh, and S. Hwang. Strg-index: Spatio-temporal region graph indexing for large video databases. In *SIGMOD*, pages 718–729. ACM, 2005.

[53] J. Z. Li, M. T. Ozsu, D. Szafron, and V. Oria. Moql: A multimedia object query language. In *MIPR*, pages 19–28, 1997.

[54] S. Lohr. *Sampling: design and analysis*. Nelson Education, 2009.

[55] C. Lu, M. Liu, and Z. Wu. Svql: A sql extended query language for video databases. *IJDTA*, 2015.

[56] Y. Lu, A. Chowdhery, S. Kandula, and S. Chaudhuri. Accelerating machine learning inference with probabilistic predicates. In *SIGMOD*, pages 1493–1508. ACM, 2018.

[57] H. Nyquist. Certain topics in telegraph transmission theory. *Transactions of the American Institute of Electrical Engineers*, 47(2):617–644, 1928.

[58] V. E. Ogle and M. Stonebraker. Chabot: Retrieval from a relational database of images. *Computer*, 28(9):40–48, 1995.

[59] J. Oh and K. A. Hua. Efficient and cost-effective techniques for browsing and indexing large video databases. In *ACM SIGMOD Record*, volume 29, pages 415–426. ACM, 2000.

[60] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. *SIGMOD*, 1984.

[61] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *SIGMOD*, volume 21, pages 39–48. ACM, 1992.

[62] A. Poms, W. Crichton, P. Hanrahan, and K. Fatahalian. Scanner: Efficient video analysis at scale (to appear). 2018.

[63] J. Redmon and A. Farhadi. Yolo9000: better, faster, stronger. *arXiv preprint*, 2017.

[64] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *NIPS*, 2015.

[65] C. P. Robert. *Monte carlo methods*. Wiley Online Library, 2004.

[66] A. W. Senior, L. Brown, A. Hampapur, C.-F. Shu, Y. Zhai, R. S. Feris, Y.-L. Tian, S. Borger, and C. Carlson. Video analytics for retail. In *AVSS*. IEEE, 2007.

[67] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy. Fast video classification via adaptive cascading of deep models. *arXiv preprint*, 2016.

[68] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3646–3654, 2017.

[69] X. Sun, L. Muñoz, and R. Horowitz. Highway traffic state estimation using improved mixture kalman filters for effective ramp metering control. In *IEEE CDC*, volume 6, pages 6333–6338. IEEE, 2003.

[70] L. Wasserman. *All of statistics: a concise course in statistical inference*. Springer Science & Business Media, 2013.

[71] T. Xu, L. M. Botelho, and F. X. Lin. Vstore: A data store for analytics on large videos. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 16. ACM, 2019.

[72] A. Yoshitaka and T. Ichikawa. A survey on content-based retrieval for multimedia databases. *TKDE*.

[73] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, volume 9, page 1, 2017.

[74] X. Zhu, Y. Wang, J. Dai, L. Yuan, and Y. Wei.

Flow-guided feature aggregation for video object detection. *arXiv preprint arXiv:1703.10025*, 2017.

```
SELECT COUNT
  (DISTINCT trackid)
FROM taipei
WHERE class = 'car'
```

(a) Count distinct cars.

```
SELECT COUNT(*)
FROM taipei
WHERE class = 'car'
ERROR WITHIN 0.1
  CONFIDENCE 95%
```

(b) Example of error rate.

```
SELECT timestamp
FROM taipei
WHERE class = 'car'
FNR WITHIN 0.01
FPR WITHIN 0.01
```

(c) Replicating NoScope.

```
SELECT *
FROM taipei
WHERE class = 'car'
  AND classify(content)
    = 'sedan'
```

(d) UDF to classify cars.

Figure 14: Further examples of FrameQL.

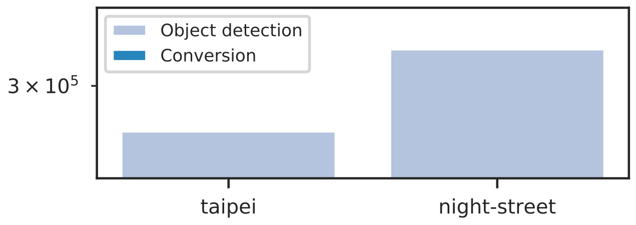


Figure 13: Time of object detection and conversion to FrameQL tuples, when processing the entire video. Zoomed in and log-scaled for visibility.

APPENDIX

A. FrameQL

A.1 Populating FrameQL

In this work, we focus on the batch analytics setting (as opposed to streaming analytics) so we assume the all the video is available for processing. To our knowledge, the majority of object detection and entity resolution methods are deterministic. Thus, once the object detection and entity resolution methods are fixed, the FrameQL tuples do not change.

We have empirically found that, even for busy scenes, a fully populated FrameQL table has around 100k rows per hour of video. We show the breakdown of time for detection and processing the data to FrameQL tuples for `taipei` and `night-street` in Figure 13 (`taipei` is the busiest video and thus takes the longest time to convert the output of object detection to FrameQL tuples). As converting the output of object detection to FrameQL tuples is a tiny fraction of total time, we ignore the transformation time.

A.2 Other Visual Query Languages

There are many other visual query languages in the literature, including SVQL [55], CVQL [49], BVQL [18], and SRQL [51]. FrameQL shares similarities with many of these query languages, but has several key differences. First, the majority of prior visual query languages either used low-level features, assumed the semantic data was provided, or used classical computer vision methods to extract semantic data. Second, many of these query languages focuses on events or specific instances of objects (e.g., specific actors in a movie). Third, many prior query languages implicitly focus on relatively short videos (e.g., minutes) as opposed to the long-running video streams we consider in this work.

In designing FrameQL, we specifically aimed to avoid the user having to annotate video. Thus, we restricted the language to only allow expressions that can be *automatically* computed with high accuracy (e.g., we do not allow querying for specific events). Researchers have only recently developed techniques to automatically extract this information from video with high accuracy [33].

A.3 FrameQL examples

We give further examples of FrameQL in Figure 14a.

First, counting the number of distinct cars can be written as in Figure 14a, which is not the same as counting the average number of cars in a frame, as this query looks for distinct instances of cars using `trackid` (cf. Figure 2a). Second, error rates can be set using syntax similar to BlinkDB [5], as in Figure 14b. Third, NoScope can be replicated in a similar manner to Figure 14c. Finally, a UDF could be used to classify cars into subtypes, as in Figure 14d.

A.4 Extensions to FrameQL

In this work, we focus on queries that can be optimized using current techniques. However, many use cases require two features that we do not currently support: 1) joins and 2) global identifiers for objects, which can easily be added to the language. Thus, we describe how to add joins and global identifiers to FrameQL.

Joins. Joins can be added to FrameQL as in standard SQL. Joins can be used to answer a wider range of queries. For example, computing the average count of cars on a single street camera can be used to determine when the given street is busiest, but not when traffic is the highest across a city. This query could be answered by joining on the timestamp and computing the average number of cars.

Global identification. While FrameQL can express a wide range of queries, several types of queries require a unique global identifier. For example, if two cameras were placed a mile apart on a highway, the flow rate of cars on the highway could be computed by the average time it takes for a car to leave one camera and arrive at the other.

Thus, FrameQL could be extended by adding a field `globalid`. In the case of cars, the license plate number could be used as a global identifier, but computing `globalid` is a difficult task in general.

B. AGGREGATION

B.1 Proof of Correctness

Correctness of AQP. As the correctness of AQP has been extensively studied, we sketch the proof outline here, which has been slightly modified from [35].

As BlazeIt does not currently support joins, all sampling strategies draw iid from the frames (or boxes). Furthermore, we assume the quantities of interest are bounded, which is trivially true, except in the case where a UDF returns an invalid value or infinity. We ignore this case.

Denote the random variable over n samples as Y_n. As the samples are drawn iid and are bounded, the central limit theorem applies and thus Y_n converges to the correct quantity [70].

Correctness of Control Variates. The proof for control variates follows directly from the above proof, as control variates are an unbiased estimator of the same quantity.

B.2 Extensions to Aggregation

In this work, we focus on extending aggregation beyond existing techniques, but we describe how aggregation can optimize queries over other statistics.

Aggregation for occupancy. Another statistic of interest is the percentage of frames that are occupied, which can be written in FrameQL as:

```
SELECT FCOUNT DISTINCT(*)
FROM taipei
WHERE class = 'car'
ERROR WITHIN 0.01 CONFIDENCE 95%
```

To answer this query, BlazeIt can perform Algorithm 1, but instead train a specialized NN that performs binary detection instead of counting the number of objects in the frame.

Aggregation for number of unique objects. Another statistic of interest is the number of unique objects, which can be written in FrameQL as in Figure 14a.

To answer this query, BlazeIt can use standard AQP with the following sampling procedure:

- Sample i iid from the number of frames.
- For each sampled frame f_i, perform object detection and entity resolution on frame f_i and f_{i+1}.
- Return the difference of the number of objects that are in frame f_i but not in f_{i+i}.

Aggregation for box statistics. Another class of statistics are statistics over the bounding boxes, e.g., an analyst might be interested in the average box area or the average position. For any numerical statistic over the bounding box, BlazeIt can leverage traditional AQP, using the following procedure:

- Sample a frame f_i iid.
- Perform object detection on f_i.
- Sample iid a single box matching the predicates of the query.
- Run the UDF over the box and return the answer.

To increase efficiency, BlazeIt can cache the box information if the frame f_i is sampled again.

C. OPTIMIZING CONTENT-BASED SELECTION

Overview. In content-based selection, the user is interested information about the mask or content of every instance of an event, e.g., finding red buses (Figure 2c). In these queries, the object detection method must be called to obtain the mask. As object detection is the overwhelming computational bottleneck, BlazeIt aims to perform object detection as few times as possible. For example, BlazeIt can filter frames that lack red *before* performing object detection to look for buses.

To reduce the number of object detection invocations, BlazeIt infers filters to discard frames irrelevant to the query before running object detection on them. BlazeIt currently supports four classes of filters: 1) label-based filtering, 2) content-based filtering, 3) temporal filtering, and 4) spatial filtering (described in detail below). Importantly, these filter types and parameters are automatically selected from the query and training data as described below.

While some filters can be applied with no false negatives, others filters are statistical in nature and may have some error rate. The error rate of these filters can be estimated on a held-out set, as in cross-validation [24]. However, as prior work [47] has considered how to set these error rates, we only consider the case where the filters are set to have no false negatives on the held-out set. Assuming the held-out set is representative of the unseen data (i.e., no model drift, see §3.2), this procedure will incur few false negatives on the unseen data, which is sufficient for exploratory queries or `LIMIT` queries.

Physical Operators. We present instantiations of each class of filter to demonstrate their effectiveness. We describe each class of filter and BlazeIt's instantiations of the filter class.

Label-based filtering. In label-based filtering, the video is filtered based on the desired labels. We leverage similar techniques to NoScope [47], namely training a specialized NN to discard frames without the label.

Content-based filtering. In content-based filtering, the video is filtered based on UDFs that return continuous values (e.g., the UDF `redness` returns a measure of redness of an image). BlazeIt will attempt to learn a threshold based on the training data or fall back to naive application of the UDF if it cannot learn a beneficial threshold.

Currently, BlazeIt supports automatically learning a filter from any UDF that only takes the box content, by applying the UDF over the entire frame. For example, if the query filters for `redness > R` and `area > A`, then `redness(frame) > A · R` is a safe threshold. In practice, even more conservative thresholds may work depending on the UDFs and video.

We describe when content-based filtering is effective below.

Temporal filtering. In temporal filtering, the video is filtered based on temporal cues. For example, the analyst may want to find buses in the scene for at least K frames. In this case, BlazeIt subsamples the video at a rate of $\frac{K-1}{2}$. BlazeIt additionally support basic forms of filtering such as range-based queries.

Spatial filtering. In spatial filtering, only regions of interest (ROIs) of the scene are considered. For example, a street may have cars parked on the side but the analyst may only be interested in vehicles in transit, so the analyst specifies in the query which parts of the scene contain moving vehicles. The ROI is specified by the user and can be used for faster object detection inference, and activity outside the ROI can be ignored, which can increase the selectivity of other filters.

BlazeIt crops the images to be more square if the given ROI allows such an operation. For example, if the query only looks for objects with `xmax(mask) < 720` in a 1280×720 video, BlazeIt will resize the frames to be 720×720. Standard object detectors run faster when the input is more square: in most existing detectors, the input image is resized so that the short-edge is a specific size and the aspect ratio is held constant [33, 64] (for a fixed short-edge size, reducing the long-edge size will make the image smaller). As

```
SELECT *
FROM VIDEO
WHERE class = OBJ
  AND area(mask) > AREA
  AND COLOR(content) > COLOR_VAL
GROUP BY timestamp
HAVING SUM(class = OBJ) >= NBOBJ
LIMIT NFRAMES
GAP 300
```

Figure 16: Base query for limit queries with predicates. The exact values for the variables are shown in Table 8.

```
SELECT FCOUNT(*)
FROM VIDEO
WHERE class = OBJ
  AND area(mask) > AREA
  AND COLOR(content) > COLOR_VAL
ERROR WITHIN 0.001
AT CONFIDENCE 95%
```

Figure 17: Base query for aggregation queries with predicates. The exact values for the variables are shown in Table 8.

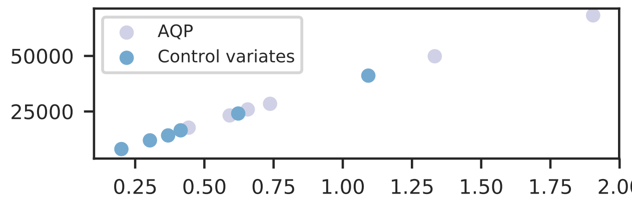


Figure 15: Variance of the estimator vs number of samples.

the computation scales with the resolution, square images result in the least computation.

Operator Selection. BlazeIt will infer which filters can be applied from the user's query. We describe how each class of filter can be inferred from the query.

First, if the user selects an area of the video, BlazeIt resizes the frame to be as square as possible, while keeping the area (along with some padding) visible, as described above.

Second, BlazeIt infers the times in the video and the subsampling rate from the query to achieve exact results. For example, if the user queries for objects present in the video for at least 30 frames (1 second), BlazeIt can sample once very 14 frames.

Third, if the user selects classes, BlazeIt trains a specialized NN to detect these classes, as in NoScope. Then, BlazeIt estimates the threshold on unseen data to attempt no false negatives.

Fourth, if the user provides a content-based UDF over the content (e.g., to determine the color of the object), BlazeIt can apply the UDF over the entire frame (as opposed to the box), and filter frames that do not satisfy the UDF at the frame level. BlazeIt sets UDF filter thresholds similar to how it sets thresholds for specialized NNs. However, for this procedure to be effective, the UDF must return a continuous value that can be scaled to a confidence. Consider two possible UDFs for redness: 1) a UDF which returns true if the over 80% of the pixels have a red-channel value of at least 200 (out of 256) and 2) a UDF that returns the average of the red-channel values. While both UDFs can be used as a filter, the second will be more effective.

Correctness. BlazeIt will perform object detection on all frames that pass its filters, so no false positives will be returned. Spatial and temporal filters are exact, but label-based and content-based filters are not. For probabilistic filters, we set the thresholds so there are no false negatives on the held-out set. The accuracy will be high, but possibly not perfect, and will return no false positives for `LIMIT` queries.

Time complexity. Before filter i is applied, denote the remaining frames F_i. Denote the cost of filter i on a frame to be c_i and the cost of object detection to be c_o. If $c_i \cdot F_i < c_o \cdot (F_i - F_{i+1})$, then the filter is beneficial to run. The filters we consider in this work are 3 to 6 orders of magnitude faster than object detection and are thus nearly always worth running.

D. FURTHER EXPERIMENTS AND EXPERIMENTAL DETAILS

D.1 Further Experimental Details

Query details for limit queries with predicates. We provide query details for the limit queries with predicates. The base query is shown in Figure 16. The exact values for the variables are shown in Table 8. As shown, each query searched for 5 or 10 instances of N objects with at least a certain area and color level.

Query details for aggregation queries with predicates. We show the base query for aggregates with predicates in Figure 17. The exact values for the variables are the same as the limit queries, with details shown in Table 8.

D.2 Content-based Selection Queries

To illustrate the effectiveness of content-based filters, we evaluate BlazeIt on the query shown in Figure 2c.

We run the following variants:

- Naive: we perform object detection on every frame.
- Binary oracle: we perform object detection on the frames that contain the object class of interest.
- BlazeIt: we apply the filters described in §C.

We do not include an AQP baseline, as sampling does not help for exhaustive queries.

For each query, BlazeIt's CBO trains, estimates the selectivity, and computes the threshold for each filter applicable to the query (which is determined by BlazeIt's rule-based optimizer). We include the time to train the filters and select the thresholds in the runtime. Due to the large computational cost of running the object detector, we extrapolate its cost by multiplying the number of calls by the runtime of the object detector.

End-to-end performance. The results for the end-to-end runtime of the baselines and BlazeIt are shown in Figure 18. As buses are relatively rare (12% occupancy, see Table 4), the binary oracle performs well on this query, giving a 8.4× performance improvement over the naive method. However, BlazeIt outperforms the binary oracle by 6.4×, due to its extended classes of filters. Furthermore, BlazeIt delivers up to 54× improved throughput over naive methods for this query.

Video name	Object	Number of objects	Number of frames	Area	Color type	Color value	Instances
`taipei`	bus	2	5	55000	Background red	13.0	5
`night-street`	car	2	10	90000	Luma	110.0	14
`rialto`	boat	3	5	40000	Luma	110.0	8
`grand-canal`	boat	2	5	60000	Background blue	6.5	6
`amsterdam`	car	2	5	18000	Background blue	15.0	5
`archie`	car	2	10	100000	Absolute red	15.0	20

Table 8: Query details for limit queries with predicates tested in this work.

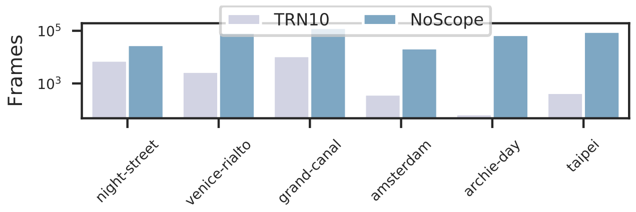


Figure 20: Number of samples to find 10 of the requested objects for each query, using TRN10 or a representative NoScope NN. As shown, TRN10 significantly outperforms the NoScope NN on all videos. The y-axis is on a log-scale. Average of 3 runs.

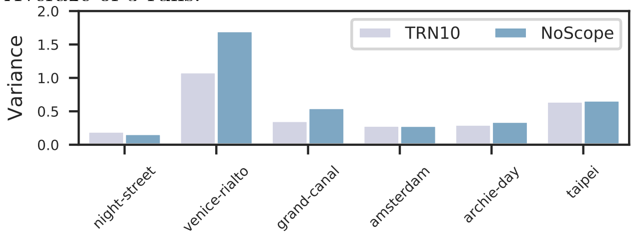


Figure 21: Variance of the control variates estimator when using TRN10 or a representative NoScope NN (lower is better). As shown, TRN10 typically matches or beats the NoScope NN, except for `night-street`.

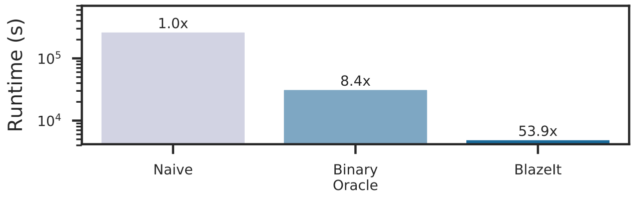


Figure 18: End-to-end throughput of baselines and BlazeIt on the query in Figure 2c. The y-axis log scaled.

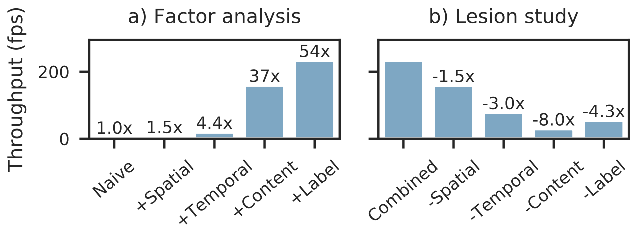


Figure 19: Factor analysis and lesion study of BlazeIt's filters on the query in Figure 2c.

Factor analysis. We perform a factor analysis (adding filters one at a time) and lesion study (individually removing filters) to understand the impact of each class of filter. Results are shown in Figure 19. As shown in the factor analysis, every filter adds a non-trivial speedup. Additionally, removing any class of filter reduces performance. Thus, every class of filter improves performance for this query.

D.3 Further Experiments

Variance of estimators vs number of samples. Theoretical analysis predicts that the variance of an estimator (either random sampling or control variates) linearly affects the number of samples to achieve a given error rate. To verify this hypothesis, we show a plot of the variance of the estimator vs the number of samples necessary to achieve an error of 0.01 at a confidence level of 95%. As shown in Figure 15, the number of samples is linear with the variance, as predicted by theory. We can also see that control variates have lower variance than traditional AQP, as predicted by theory.

Effect of NN type. In this work, we chose to use a tiny ResNet (referred to as TRN10) as the default specialized architecture. ResNets are an extremely popular architecture [13, 14]. To test our hypothesis that TRN10 is a good default, we compared TRN10 to a representative NoScope NN, parameterized by 32 base filters, 32 dense neurons, and 4 layers.

We used TRN10 and the NoScope NN on limit queries for each of the videos and computed the number of samples required to find the requested number of events in Table 7. As shown in Figure 20, TRN10 requires significantly fewer samples compared to the NoScope NN on all videos.

We additionally used TRN10 and the NoScope NN for the aggregation tasks for each video and computed the variance of the control variate estimator (the variance of the estimator is directly related to the number of samples; lower is better). As shown in Figure 21, TRN10 typically matches or beats the NoScope NN, except for `night-street`.

While TRN10 is a good default for BlazeIt, we have not fully explored model search for exploratory visual queries, which we view as an exciting area of future work.

E. RULE-BASED OPTIMIZER

BlazeIt currently uses a rule-based optimizer for queries, as we have found it sufficient to optimize a large class of queries. BlazeIt's rule-based optimizer attempts to classify queries as (approximate) aggregation, cardinality-limited queries, or content-based selection and applies the rules described in the main paper in these settings. For all other queries, BlazeIt falls back to materializing all the rows in the FrameQL table.

To determine if a query is an approximate aggregation query (exact aggregation requires all the relevant rows to be materialized), BlazeIt inspects the FrameQL query for the `ERROR` keyword and an aggregation keyword (e.g., `FCOUNT` or `AVG`). BlazeIt will then determine the necessary fields and perform approximate aggregation as described above.

To determine if a query is a cardinality-limited query, BlazeIt inspects the FrameQL query for the `LIMIT` keyword. BlazeIt then determines the necessary fields for the query and executes the cardinality-limited query as described above.

To determine if the query is a content-based selection query (with applicable filters), BlazeIt will inspect the query for the predicates as described in Section C and apply them as described above.

In all other cases, BlazeIt will default to applying object detection to every frame.