Lantern: A Query Language for Visual Concept Retrieval

Will Crichton
Carnegie Mellon University
Advised by Kayvon Fatahalian

Abstract

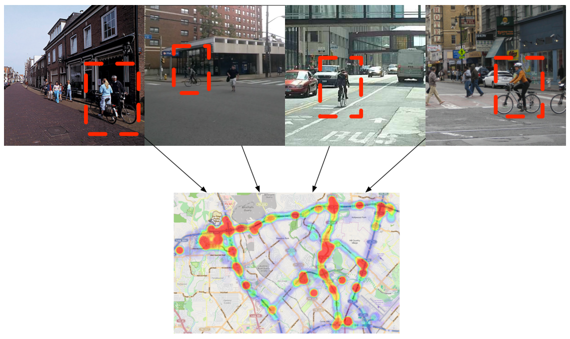
I present Lantern, a query language and database for finding spatiotemporal visual concepts in large datasets of images and video. Lantern addresses a rapidly growing need to efficiently explore and mine massive visual datasets for information, tasks like locating people in a video or determining similarity between images. A number of recent top-performing computer vision tools for these tasks rely on machine learning methods, specifically end-to-end training and evaluation which can take days or weeks to learn effective concept detectors. The language provides an abstraction, the spatial concept hierarchy, for combining existing vision algorithms with coarse grained rules for quickly developing new queries and interactively exploring visual data. Lantern compiles queries into operations on distributed collections to enable rapid execution on large clusters. I demonstrate the use of Lantern by building an interactive system for exploration of visual datasets, an object detector error analysis platform, and a tool to blur faces in videos. I show Lantern queries running across a cluster and heterogeneous hardware within a node. In each case, Lantern enabled me to rapidly construct queries and retrieve visual concepts on large visual datasets.

1 Introduction

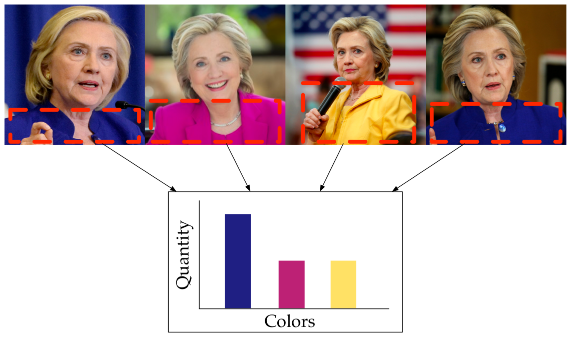
Visual data, or images and video, have become one of the primary means for capturing information about the world today. Always-on data sources like webcams, dashcams, and IoT sensors generate petabytes of video every day, and these datasets get used for a wide range of purposes ranging from planning cities to securing households. Visual data analyses for these kinds of purposes break down into two parts: *selection*, finding the visual concepts in the data that you want to analyze, and *aggregation*, using these concepts to answer questions or compute statistics. For example, questions like "where do people bike in my city?" or "what color pantsuit does Hillary Clinton usually wear?" can be decomposed as in Figure 1 on the following page. *Visual concepts* are things in images and videos which are uniquely identifiable based on their visual attributes. A person, a busy cafe, and dogs sitting around a table playing poker are all concrete concepts with visually explainable characteristics. Visual concepts always exist in the dimension of space, i.e. the image they they reside in, but concepts can also span time. A video of a person walking down a street is not a hundred frames of different people very close to each other, but instead a unique individual following a particular path through time.

Just as advances in relation databases and natural language processing have enabled text to be analyzed en masse, so too do we need tools for handling the enormous quantity of visual data to answer these kinds of questions. However, this type of data presents challenges from both computer vision and a systems perspective.

Vision challenges. Designing algorithms to understand images is hard. While you can look up a word in a dictionary, an RGB pixel does not map directly to any meaningful semantics. Tasks as simple as detecting a face in an image are still an active area of research within computer vision. Visual data is both noisy and highly ambiguous—interpreting a blob of pixels depends on the lighting, the camera perspective, the quality of the image encoding, and so on.



(a) Creating a heatmap of bicyclists from street cameras.



(b) Creating a histogram of pantsuit colors from head shots of Hillary Clinton.

Figure 1: Examples of visual analyses with selection and aggregation. Lantern tries to make the process of selection faster and easier. Dashed red boxes indicate concepts found with Lantern.

Over the last few years, deep neural networks have proven to be the most effective tool for performing most vision tasks. This trend started with the convolutional neural network AlexNet [1] for object recognition, or the task of finding certain categories of concepts like "person" or "potted plant" in images. DNNs have spread to take over most other vision tasks like captioning images [2] and analyzing roads for self-driving cars [3]. All of these networks operate via end-to-end training, where in order to find a visual concept you must provide many training examples and engage in a lengthy training process. Popular object recognition benchmarks like ImageNet [4] provide millions of images for training. And not only must all this data be collected, but all of these training examples must be labeled, usually by hand. For example, creating and labeling the Microsoft COCO dataset [5] of 2.5 million images reportedly took 70,000 man-hours. Even with the training data ready, the process of training a network can take days or weeks depending on the complexity of the network layout. We need vision tools that can quickly adapt to find novel visual concepts needed by analysts.

Systems challenges. Visual data requires intensive computation and large storage space. Tools like convolution neural networks that touch pixels not only have to intelligently manage the cache to avoid IO overhead but also multiply large matrices and repeatedly compute convolutions. An efficient visual data analysis must scale both horizontally across a cluster as well as vertically by using hardware acceleration like multicore and GPU processors.

Visual datasets, especially videos, take up enormous amounts of space even for what is considered relatively little data. A 480p video from a traffic camera streaming for 24 hours at 1 Mbit/s could take up to 10 GB, so an array of 30 cameras across a city for a month would consume upwards of 10 TB. For contrast, all of Wikipedia fits in only 10 GB. A given dataset may not fit on an entire disk, nevertheless in memory. Systems that process visual data must be designed with these resource constraints in mind.

In this thesis, I describe Lantern, a query language and database for finding visual concepts. The overall goal of the system is to combine modern vision techniques with a simple query model to enable rapid prototyping of visual concept detectors. The main contributions of this thesis are:

- A query model for concisely describing visual concepts with a *spatial concept hierarchy*. I show how to write simple and composable queries to find concepts in this hierarchy.
- A temporal extension of the concept model. Users can describe visual concepts that exist

across frames and can boost the accuracy of their query by using this knowledge.

- A system for compiling visual concept queries to parallel operations on distributed collections.
- A functioning prototype running on a cluster of dozens of machines.
- I demonstrate the use of Lantern to generate image analysis apps such as an interactive data explorer, a tool to blur faces in video, and a dashboard for assessing the quality of object detectors.

I show that that the language's abstractions are well suited to the domain and reduce the burden on the programmer in the quest for visual concepts.

2 Related Work

Visual query languages have a long history dating back to the 1990s when a lot of work was done on multimedia databases and content-based image retrieval (CBIR). The first major system in this area was QBIC [6], a consumer-focused IBM creation that allowed querying on example images and "user constructed sketches and drawings." Research continued in a strongly HCI-centric way with work focused on including human feedback in searches and designing GUIs for image search [7]. A number of researchers attempted to formalize the querying process into a query language, although none of these efforts broke into the mainstream like QBIC. However, these languages contained important concepts which influenced the design of Lantern, such as a grammar for describing spatial relations [8] and an approach for handling probabilistic queries [9]. These systems did not find widespread adoption largely because of, in the words of one paper, "the complexity of resolving atomic queries," i.e. tasks like object detection which were infeasible at the time. Some image searching tools like Google Reverse Image Search have made it into the mainstream, but most have fallen by the wayside.

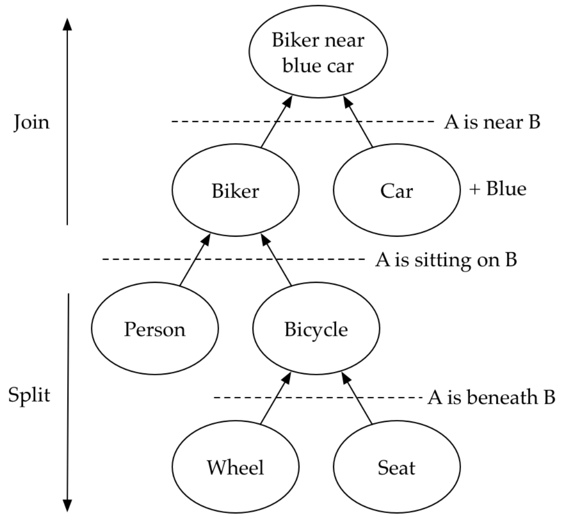
Only in the last few years, have tools emerged for doing these "atomic queries" with any reasonable accuracy, namely convolutional neural networks or convnets. Building upon AlexNet, tools like Faster-RCNN [10] and GoogLeNet [11] have innovated by introducing better network layouts and novel network layers as well as reducing evaluation and training times. More recent work has started to tackle the challenge of incorporating neural networks into interactive queries. Instead of composing the outputs of neural network as in Lantern, these systems use queries to compose networks on the fly and evaluate these amalgamated neural nets [12]. Other work has addressed this issue from the perspective of answering questions about images and captioning images. In the last year, some of this was spurred by the release of the Visual Question Answering (VQA) dataset [13] which contains tens of thousands of annotated images with questions and answers in various formats. Research in the VQA space has largely consisted of end-to-end training of recurrent nets and LSTMs [14] [15]. It is also worth noting that a non-goal of this thesis is to present a natural language interface for querying visual concepts. Various recent works have tried jointly training networks for processing natural languages and pixels [16] [12], but for simplicity we avoid involving any form of natural language.

3 Lantern Design

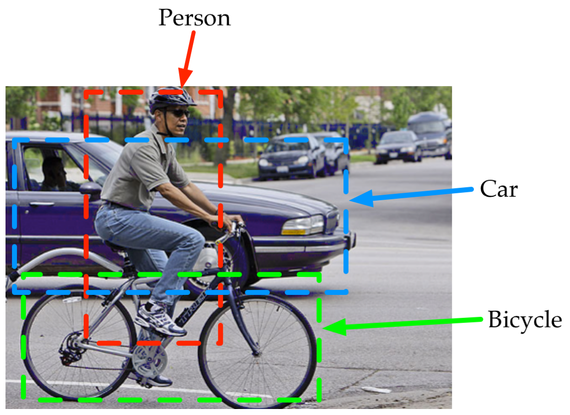
The design of Lantern is motivated by three goals:

- *Productivity.* Lantern should reduce the time between describing a concept and evaluating across visual datasets, so writing queries in Lantern should be easy and intuitive.
- *Efficiency.* Lantern should evaluate queries in parallel across a cluster and enable use of low-level hardware accelerators.
- *Interoperability with state-of-the-art computer vision techniques.* Lantern should complement, not replace, existing computer vision tools like neural networks, so it must support integrating these components into concept queries.

The Lantern system has two main components: a query language for describing visual concepts, and a database for ingesting data and evaluating concept descriptions. The primary abstraction of the query language is the *visual concept*, or a thing with visually distinct attributes that is captured in images or video. Lantern provides a model for constructing concept descriptions, or "queries" in database parlance, that find concept instances. This model mixes *attributes*, or functions that describe properties of visual concepts, with a *spatial hierarchy* for structuring and composing queries, shown in Figure 2a. To understand these abstractions, I will walk through how to search for an example visual concept, specifically "people riding bikes near blue cars" shown in Figure 2b.



(a) The spatial concept hierarchy corresponding to the visual concept. The dashed lines indicate the kind of spatial relationship connecting the two children, and the arrows on the left indicate the kind of operation that traverses the hiarchy.



(b) Example image labeled with the corresponding components of the visual concept.

Figure 2: Representing the visual concept "bikers near blue cars" in the concept hierarchy.

```
from rcnn import detect_objects
def describe(image):
  detected_objects = detect_objects(image)
  cars = filter(detected_objects, lambda obj: obj['label'] == 'car')
  people = filter(detected_objects, lambda obj: obj['label'] == 'person')
  bicycles = filter(detected_objects, lambda obj: obj['label'] == 'bicycle')

  # We have to write all the other logic ourselves...
  bikers = find_where_sitting_on(people, bicycles)
  blue_cars = find_where_blue(cars)
  return find_where_near(bikers, blue_cars)
```

(a) First pass at writing a concept description for our running example in Python.

```
from rcnn import detect_objects
def get_scores(detected, category):
  return map(detected, lambda obj: {'box': obj['bounding_box'],
                                    'score': obj['scores'][category])

def describe(image):
  detected = detect_objects(image)
  cars = get_scores(detected, 'car')
  people = get_scores(detected, 'person')
  bicycles = get_scores(detected, 'bicycle')

  # We have to write all the other logic ourselves...
  bikers = find_where_sitting_on(people, bicycles)
  blue_cars = find_where_blue(cars)
  bikers_near_cars = find_where_near(bikers, blue_cars)
  return bikers_near_cars.sort_on_score()
```

(b) Improvement on Figure 3a by describing a visual concept using scores. We can make use of the scores provided by the object detection algorithm. `find_where_blue` shows an example of combining scores together. With a simplifying independence assumption, we can score objects with attributes like "blue" by multiplying scores.

```
def blue_attribute(concept):
  # A heuristic for whether something is blue is the average of the
  # values in the blue channel on all the pixels in the concept's box.
  # We divide by 255.0 to normalize the score to the [0, 1] range.
  return blue_channel_average(image.box(concept['box'])) / 255.0
```

(c) The "blue" attribute function computes how blue a box of pixels is.

```
def find_where_sitting_on(sitting, sat_on):
  result_concepts = []
  for (sitting_concept, sat_on_concept) in zip(sitting, sat_on):
    # A heuristic for whether A is sitting on B could be the overlap between
    # their bounding boxes.
    sitting_score = overlap(sitting_concept['box'], sat_on_concept['box'])
    new_concept = {'components': [sitting_concept, sat_on_concept],
                   'score': sitting_score * sitting_concept['score'] * sat_on_concept['score']}
    # We can get rid of concepts whose score goes below a certain threshold.
    if new_concept['score'] > THRESHOLD:
      result_concepts.append(new_concept)
  return result_concepts
```

(d) The "sitting on" spatial relation function determines whether one concept is sitting on top of another.

Figure 3: Describing concepts without using the Lantern system.

3.1 Concept descriptions

Concept descriptions are a way of modeling the visual attributes of a concept in code. In order to understand the motivation for Lantern's concept description model (interchangeably its query model), I will derive its components from the ground up in general Python code and then show the explicit Lantern syntax. In its simplest formulation, a concept description looks like the interface to a computer vision tool for finding objects in images, i.e. a function `Describe` : `Image` $\rightarrow$ `List<BoundingBox>` that looks over an entire image and finds the concepts contained inside. Here, an instance of a concept, or the thing we're trying to describe, is represented as a box[1] of pixels in an image. Object recognition tools like RCNN [17] process an image and return a set of candidate bounding boxes with a corresponding probability distribution for each describing which category it most likely came from. Popular object detectors like RCNN have pre-trained and publicly available models for categories including people, bicycles, and cars, so we could apply that here. Figure 3a on the previous page shows how we could write a concept description in this style.

3.1.1 Scores

Because visual data is inherently ambiguous, visual concepts cannot be treated as a binary of exists/doesn't exist. For any blob of pixels, there is some likelihood as to whether it matches a given visual concept, so any part of the system that processes visual concepts must be able to assign a probability or score to the concepts. Not only does this more closely match how these vision tools approach data analysis, but this is also an essential building block for composition—if multiple components can express their confidence levels, the system can combine these together intelligently for the user.

Revising the code in Figure 3a on the preceding page, we can improve our query by moving it into a probabilistic space as shown in Figure 3b on the previous page.

3.1.2 Attributes

Next, we can provide structure to the query model by pulling out and codifying the reusable components of the concept query. Notably, concept descriptions have two main structural components: attributes and spatial relations. When we want to find "a biker near a blue car", attributes usually correspond to the adjectives in these phrases. The blue in "blue car", the busy in "busy cafe", and the happy in "happy person" are all attributes that describe categories of visual concepts. Specifically, an attribute is a function `Attribute` : `Concept` $\rightarrow$ `Score` which, given an instance of a concept, returns a score of how much that concept possesses a particular attribute. For instance, the "blue" attribute used in Figure 3b on the preceding page could be written as in Figure 3c on the previous page.

3.1.3 Spatial concept hierarchy

The next key insight is that visual concepts can be also described with *spatial relations*, or prepositions like "above" and "around." For example, a bicyclist could be described as "a person sitting

[1] A more precise model would use image segmentations, but these are difficult to compute and not as widely used in computer vision.

on a bicycle", or a busy cafe could be described as "a storefront with several tables containing an umbrella, at least three people, and a bowl of bread." This idea is used in several vision algorithms like the Constellation model [18] and the Deformable Parts model [19]. Like table joins in relational databases, spatial relations join together sets of concepts and evaluate them as pairs. For example, the "sitting on" relation, defined as `find_where_sitting_on` in Figure 3a on page 5, could be implemented as in Figure 3d on page 5.

If we join two visual concepts together, this starts to form a tree, as shown in Figure 2a on page 4 from our running example. I call this the *spatial concept hierarchy*, as it describes how concepts relate to each other in pixel space. Join allows us to traverse up the hierarchy, but we need an equivalent operation for the opposite direction, which I term a *split* as it means a query is splitting up a visual concept into its parts. The image itself lies at the root of the tree, so all concepts are ultimately derived by splitting the image up into smaller concepts.

3.1.4 Query constructors

Lantern enables expression of concept descriptions using a declarative programming style similar to languages like Datalog. Figure 4 shows how to describe the "blue car near bicyclist" example using Lantern. Figure 10 on page 15 shows several queries with their corresponding output.

```
from rcnn import detect_objects
from lantern import Query

images  = Query.all()
objects = images.split(detect_objects)
people  = objects.add_attribute(lambda concept: concept['score']['person'])
bikes   = objects.add_attribute(lambda concept: concept['score']['bicycle'])
cars    = objects.add_attribute(lambda concept: concept['score']['cars'])

blue_cars = cars.add_attribute(blue_attribute)
bicyclists = Query.join(people, bikes, sitting_on_score)
bikers_near_blue_cars = Query.join(bicyclists, blue_cars, near_score)
```

Figure 4: A concept description for the running example in Lantern syntax.

As shown in Figure 4, queries are constructed by one of four operations.

- `Query.all()`
  This represents the input to the query, similar to `SELECT *` in SQL. Usually this represents each image, but it could correspond to other sets of concepts.
- `Query.add_attribute(attribute_fn)`
  This adds an attribute function to the list of attributes for a particular visual concept query. Just as a visual concept can be described with multiple adjectives, so can a query have multiple attributes.
- `Query.join(query1, query2, relation_fn)`
  This combines two concept queries and scores all pairs with the relation function.
- `Query.split(query, split_fn)`
  This takes in a function `split_fn` : `Concept` $\rightarrow$ `List<Concept>` that produces sub-conceptse given a concept instance, and applies it across the input set of concepts.

This choice of syntax and abstraction provides a myriad of benefits over the manually-implemented Python in Figure 3 on page 5. First, the user-defined functions given to `join`, `split`, and `add_attribute` only have to focus on providing a single score for their relevant task, and Lantern can take care of combining scores from separate concepts together. Second, composition and reuse are simple in this system. If someone has already developed a implementation of `is_sitting_on` or `detect_bicycles`, a new Lantern query can easily swap them in and out of the concept model without any substantive code refactoring. Lastly, the user's kernels only need to be expressed as operations on a single concept instead of all concepts in an image, which reduces the amount of boilerplate.

3.2 Database

The Lantern database serves two primary functions. First, it handles the input and output of visual data to/from the system and also acts as a translation layer between concept queries and an efficient underlying representation for executing those queries. Figure 5 shows a sample usage of the database. To understand the interface, we will walk through the example step-by-step. The Lantern database is wrapped up in a single master object called `Database`, which can be accessed by providing credentials to the `connect` method—in our case these are for Google Cloud Storage and MySQL, although neither is fundamental to the database.

```
from lantern import Database, Query, QueryParams
import rcnn

db = Database.connect(<various credentials>)
tbl = db.table("lantern-test")

image_paths = ["frame0.jpg", "frame1.jpg", ...]
tbl.seed_images(image_paths)

people = Query.split(Query.all(), lambda concept: rcnn.find(concept, 'people'))

params = QueryParams()
params.table = tbl
params.column = "base"

# This creates a new column which can be the input to later queries
tbl.save("people", db.find(people, params))
```

Figure 5: Example session with loading data into Lantern, finding people in the input data, and saving the results.

3.2.1 Data storage and layout

Lantern adopts relational model of data, organizing it in tables with rows and columns. As an analytics engine, Lantern adopts the column store approach where table rows are largely immutable and table columns can be added on the fly. Each row maps to an image, and each column to data about the image. This could be the pixels, a set of concepts, or derived metadata. When a user calls `seed_images(path_list)`, it ingests the pixel data into the database, and then generates an initial set of concepts called "base" with each image having only its root concept (the image itself). Further sets of concepts can be created by executing queries and saving them back as a new column on the table.

Handling derived metadata, or things like scores, labels, and feature vectors, is also essential to an analytics system as many operations like computing the category of an object will often produce useful metadata like features that, even if not immediately useful, may come up later in the analysis if you want to do a nearest neighbor search (as in Figure 10 on page 15) or k-means clustering. Lantern allows the kernel functions to load and store metadata directly on the `Concept` object.

3.2.2 Query evaluation

The Lantern database interface provides two methods for evaluating a concept query: `find` and `track`. The `find` function runs a query across all images independently, combines all the results, and returns a `Map<ImageId, List<Concept>>` that maps each image to the corresponding concepts contained in that image. The translation from query to executable code will be discussed further in the next section.

The `track` method tracks visual concepts across time in video data. Although the concept description model given above assumes independence of each image, we can post-hoc ascribe temporality to concepts and treat them as such in the database. `track` attempts to find correspondences between concepts across the frames and produce *tracks*, or paths through time, of type `Map<FrameNumber, Map<TrackId, Concept>>` where each frame contains a map of which tracks were active in that frame. Figure 6 describes how the algorithm works.

```
def track(query):
  tracks := empty set of tracks
  for time t = 0 to n:
    for all active trackers in tracks:
      bounding_box, score := tracker.track(frame at time t)
      if score < SCORE_THRESHOLD: deactivate current track
      else add a new concept to frame t on the current track
  concepts = run_query(query, frame t)
  for concept in concepts:
    if concept sufficiently overlaps with any active track:
      ignore this concept
    else:
      create a new track starting at this frame and this concept
```

Figure 6: Pseudocode for the tracking algorithm

Figure 7 on the following page provides an example of this algorithm in action. Note that if correspondence cannot be established, e.g. because objects move sufficiently fast in the video and cannot be tracked, then we consider them to be on separate tracks.

The algorithm given in Figure 6 not only stitches tracks together from the query evaluator but also includes a separate video tracker to boost the accuracy of the overall query. Essentially, `track` not only ascribes temporality to a concept but also turns your query into an ensemble method that uses an concept-independent video tracking algorithm to complement the query evaluator. It uses the temporal nature of the data to produce better query results instead of treating each frame independently.

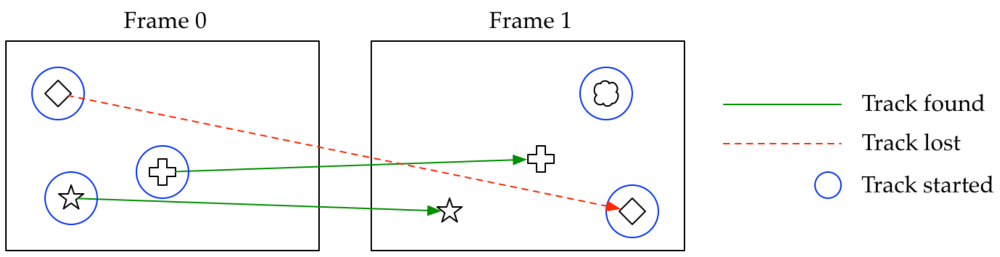


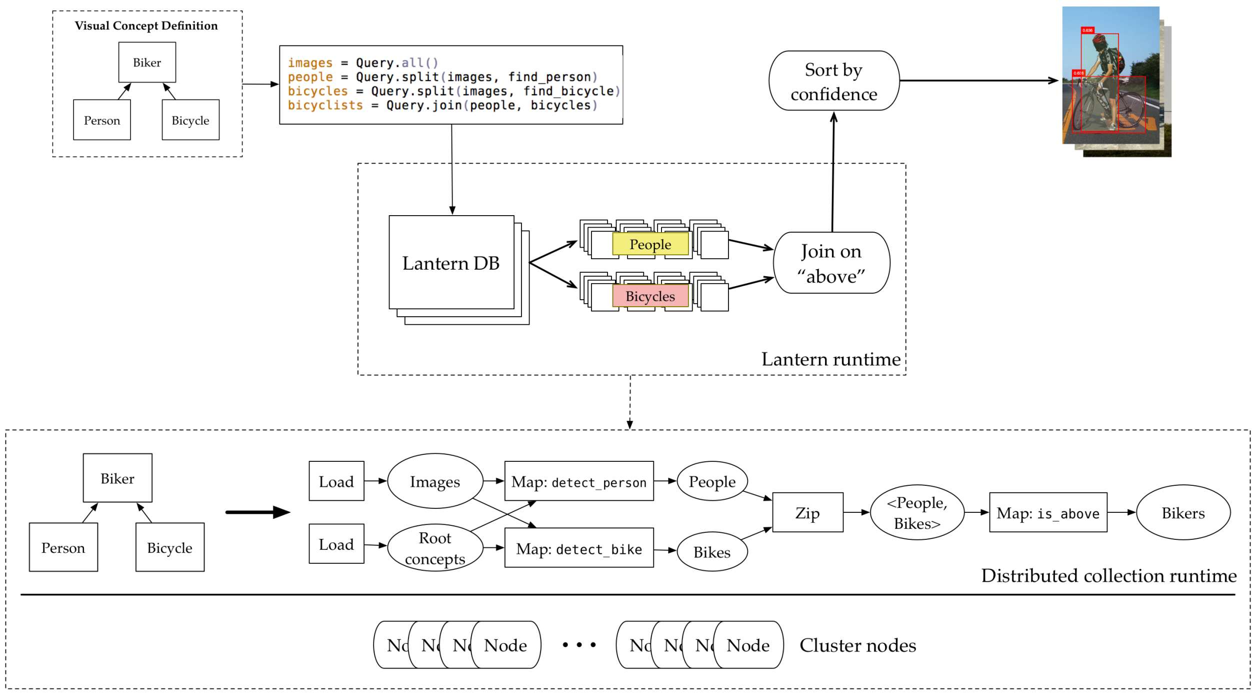
Figure 7: Application of the tracking algorithm between two frames. The star and cross are tracked across the frames, whereas the diamond is lost because it moves too far.

4 Implementation

The main challenges of the Lantern runtime implementation are:

- Compile queries into parallel operations on distributed collections.
- Combining confidence scores from the user across operations.
- Integrating video tracking into the temporal concept model.

The Lantern database is written in C++, although Python bindings like the ones shown in Figure 5 on page 8 are provided for convenience. Figure 8 provides an overview for how queries are processed and translated by the database.



```
images = Query.all()
people = Query.split(images, find_person)
bicycles = Query.split(images, find_bicycle)
bicyclists = Query.join(people, bicycles)
```

Figure 8: Finding bicyclists in the Lantern database from start to end. Compiling queries to distributed collection operations is shown in the box on the bottom.

4.1 Query compilation

The compilation step turns queries into operations on distributed collections in the same vein as Spark[2] [20]. The important operations are loads (retrieve concepts from the database), maps (apply a computation across the concepts), and zips (combine sets of concepts together). Lantern compiles queries into a directed acyclic graph (DAG) of collection operations. The user provides functions in their concept description, i.e. a split, join, or attribute, and these are used as kernels in the distributed collection operations. These collections are the length of the table they are loaded from, i.e. the number of images in our dataset, and each element of the collection is a list of concepts within that image, or a `Collection<List<Concept>>`. I choose to express data parallelism at the level of images and not concepts as this simplifies the runtime and there are empirically not enough concepts per image to justify needing a concept-parallel approach. Figure 9 shows the outline of the compilation algorithm, and Figure 8 on the previous page provides an example of this translation on the bicyclist concept.

```
def compile(query, params):
  table := params.table
  collection :=
    match query with
      | All () =>
        # Load in the concepts we're going query over and include the pixels
        zip(table.load(params.column), table.load("images"))
      | Split (sub_query, split_fn) =>
        parent_concepts := compile(sub_query, params);
        parent_concepts.map(split_wrapper(split_fn))
      | join (sub_query1, query2, join_fn)
        child1_concepts := compile(sub_query1, params);
        child2_concepts := compile(sub_query2, params);
        zip(child1_concepts, child2_concepts).map(join_wrapper(join_fn))
  for attribute in query.attributes:
    collection = collection.map(attribute_wrapper(attribute))
  return collection
```

Figure 9: Pseudocode for the query compilation algorithm.

The algorithm recursively steps through a concept query and constructs a DAG of operations. Because the elements of the collection are lists of concepts whereas the user's kernels operate over individual concepts, the algorithm uses higher order functions to generate new kernels which map the user's functions over lists. The generated wrapper functions also work with the scoring model described in the next section to filter out concepts with low scores.

4.2 Computing concept scores

A score allows the user to express a confidence level in the results of a particular attribute, split, or join. The implementation, however, must decide how to combine scores and how to eliminate concepts based on their scores.

Combining scores. Return again to the bicyclist example. Let's say the query evaluator claims that a particular set of two boxes are a person and a bicycle, and that they are close enough to

[2]The implementation actually uses an in-house system for this purpose written in C++ which allows for the runtime to more easily manage memory and interface with hardware like GPUs.

be called a bicyclist. If we ask "what probability that this concept instance is actually a bicyclist," there are three probabilities in play:

1. The probability that the person is a person.
2. The probability that the bicycle is a bicycle.
3. The probability that the person is riding the bicycle.

Each of these is a separate score in the system, the first two coming from whatever object detector you decide to use. The third sounds strange—how do you define the probability that a person is riding a bicycle? In the context of spatial relationships, Lantern has the user define fuzzy relations, not binary relations. If you want to express A is above B, then instead of writing that function as `A.y` $<$ `B.y`, we write the function as `A.y` $-$ `B.y`. Then objects A that are "more" above B will have a higher probability assigned to them.

Combining these scores now reaches into probability theory. Recent years have seen the rise of data structures like factor graphs and Markov random fields for expressing these kinds of probabilistic relationships, e.g. the Datalog-style Tuffy probability engine [21]. However, again in the interest of time I opted to make the heavily simplifying assumption that these probabilities are independent, so then combining scores is as simple as multiplying them all together. Hence, as shown in Figure 3d on page 5, when you apply an attribute or do anything that generates a score, it gets multiplied with the concept's current score to produce its new probability estimate.

Thresholding scores. If a concept has a low enough or zero score, then the runtime should eliminate it from consideration so as to reduce the execution time of the query. For example, if a concept description uses multiple attributes like a "rusty red old car," then once the runtime knows which concepts are rusty, the evaluation of the "red" attribute only needs to occur on that smaller subset of concepts. This is a kind of runtime query optimization, in contrast with the compile-time query optimizers in relation databases that do things like limit the elements of a `JOIN` clause based on a subsequent `WHERE` clause. The simple implementation of this optimization is to set a threshold and eliminate concepts with a score below that threshold. Fixing a single threshold wouldn't scale with the depth of the query, because given our previous model for probabilities, as the number of filters on a query grows, the average probability score gets smaller and smaller. Instead, Lantern allows the user to specify a tunable threshold inside the query parameters.

4.3 Parallel implementation of `track`

The last implementation of note returns us to the temporal concept. Most of the parallelism in the system comes from the distributed collections, but computing tracks presents a unique challenge because of its sequential nature. The algorithm presented in Figure 6 on page 9 for doing tracking on visual concepts operates sequentially, going through all the frames one by one. However, we still wish to make use of the distributed system sitting beneath Lantern, so the implementation of the tracking algorithm is a modified version of the original which partitions the input video into chunks, runs the tracker sequentially across each chunk, and then merges the resultant tracks down into a single track list. Figure 10 on the following page shows how this occurs. Merging tracks works similarly to adding new concepts into existing tracks—given two sequences A and B, it compares the tracks at the end of A and beginning of B and attempts to combine them if they are sufficiently close in pixel space.

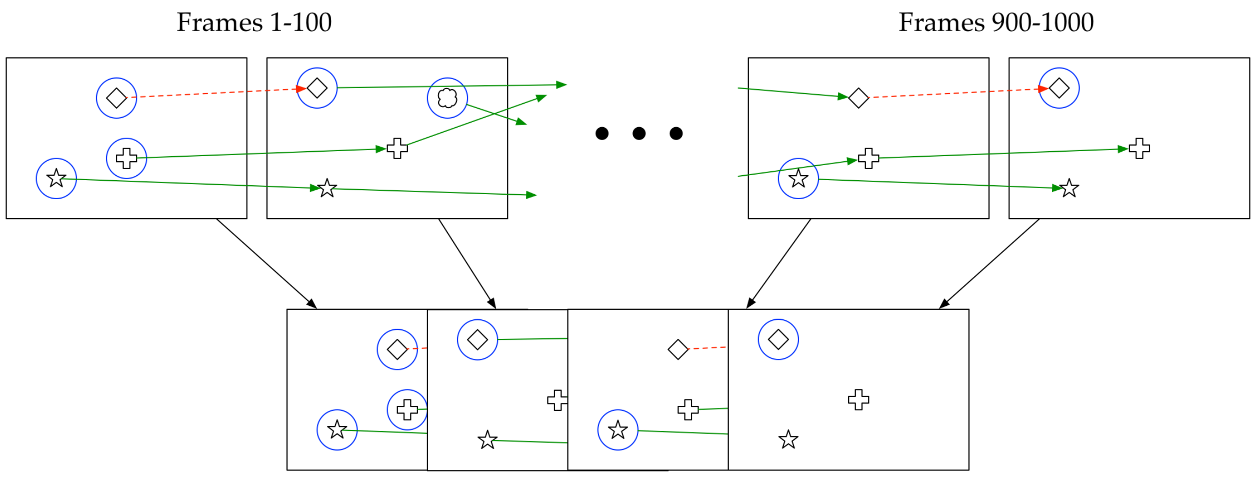


Figure 10: Parallel tracking example with 1000 frames and 10 chunks.

5 Evaluation

To evaluate how effectively Lantern achieved the stated design goals, I have implemented a number of applications using Lantern.

5.1 Interactive data explorer

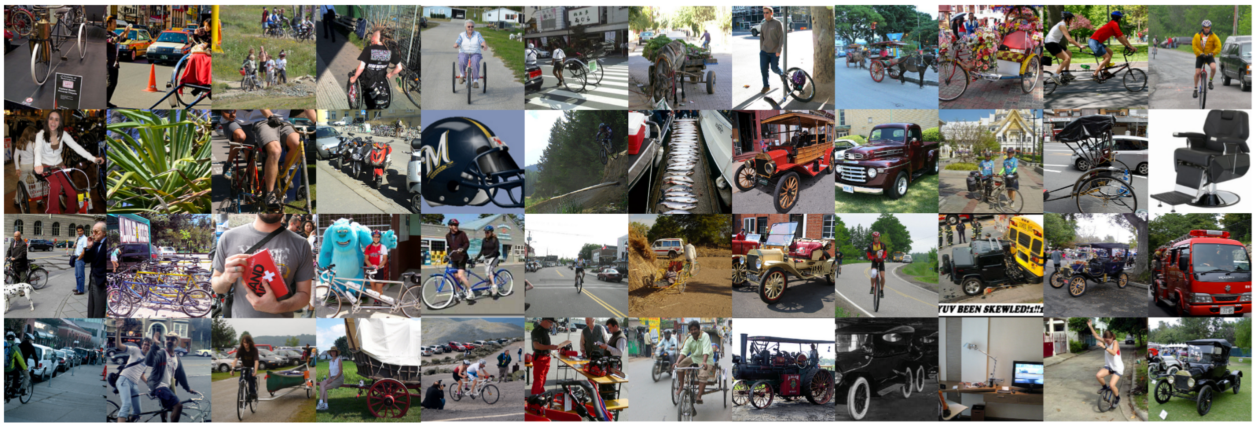
To test Lantern's ability to find novel concepts in unlabeled data, I created an interactive query page for searching various datasets including ImageNet [4], a diverse set of images commonly used for evaluating object detectors, and KrishnaCam [22], an ego-centric dataset filmed through Google Glass of many hours walking around in Pittsburgh. Figure 11 on page 16 shows a screenshot of the webpage set up to display the results, and Figure 10 on page 15 has several examples of queries run through the data explorer along with their results.

Through building the data explorer I learned threes valuable lessons about Lantern:

1. Python bindings substantially improve productivity. The utilities provided by Python are invaluable for post-processing the results of visual concept queries, and they can be written more fluently than in C++. This incurs the overhead of running a Python interpreter concurrently with the Lantern runtime, but the performance hit is worth the productivity boost.

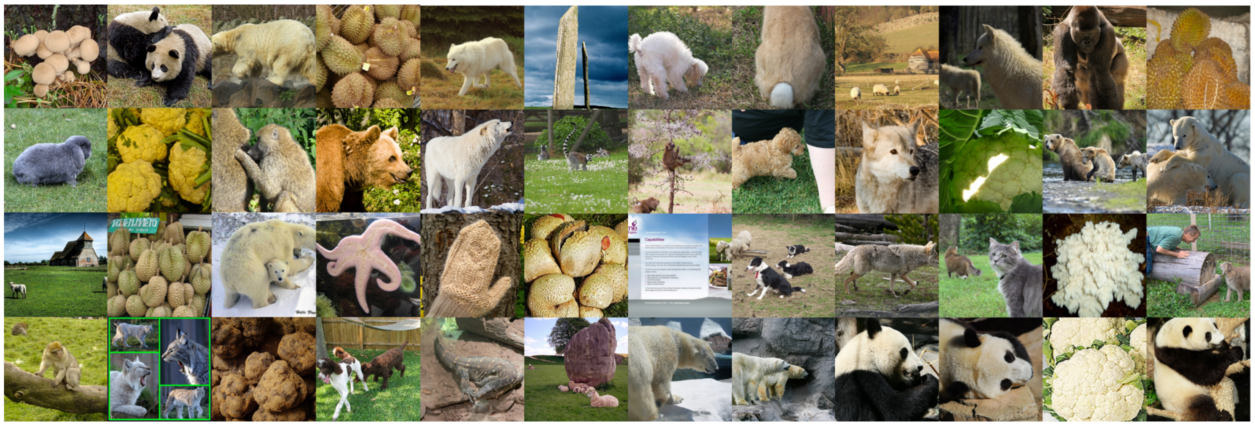
2. Spatial relations are an important building block of a query language, but it needs more operators to be expressive enough. Lantern currently cannot express grouping operations (e.g. "give me people sitting around a table") or counting operations (e.g. "show me pictures with at least five dogs"). Future work on Lantern should add more expressive power to the query language to increase usability.

3. The concept description model was well suited to the kinds of simple data exploration that I tried. When trying to find or generate ideas for novel concepts, it was not only easy to swap between queries but accurate enough to describe what I wanted to convey.



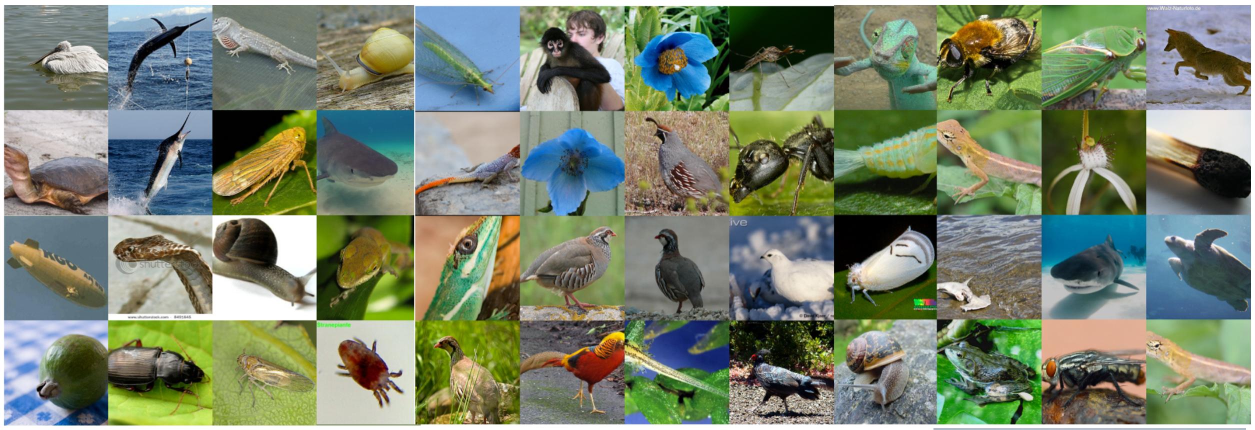
```
people = Query.all().add_attribute(lambda concept: concept['category'] == 'person')
bicycles = Query.all().add_attribute(lambda concept: concept['category'] == 'bicycle')
bicyclists = Query.join(people, bicycles, is_above)
cars = Query.all().add_attribute(lambda concept: concept['category'] == 'car')
cars_near_bikes = Query.join(cars, bicyclists, is_near)
```

(a) Searching for "bikers near cars." The query produced a fair number of false positives for cases where a biker was close to a car in pixel space but far away in the image due to perspective, but generally produced usable results.



```
sheep = Query.all().add_attribute(lambda concept, _: concept['category'] == 'sheep')
```

(b) Searching for sheep. This demonstrates that even the underlying object detectors, in this case RCNN, aren't perfect. A query for sheep returned pandas, dogs, wolves, bears, cauliflower, mushrooms, and even a starfish, but no sheep.



```
im = io.imread('whale.jpg')
features = get_image_features(im)
params.set_uniform(features)
def similarity_score(concept, reference):
    return 10.0 / np.linalg.norm(concept['features'] - reference)
similar = Query.all().add_attribute(similarity_score)
```

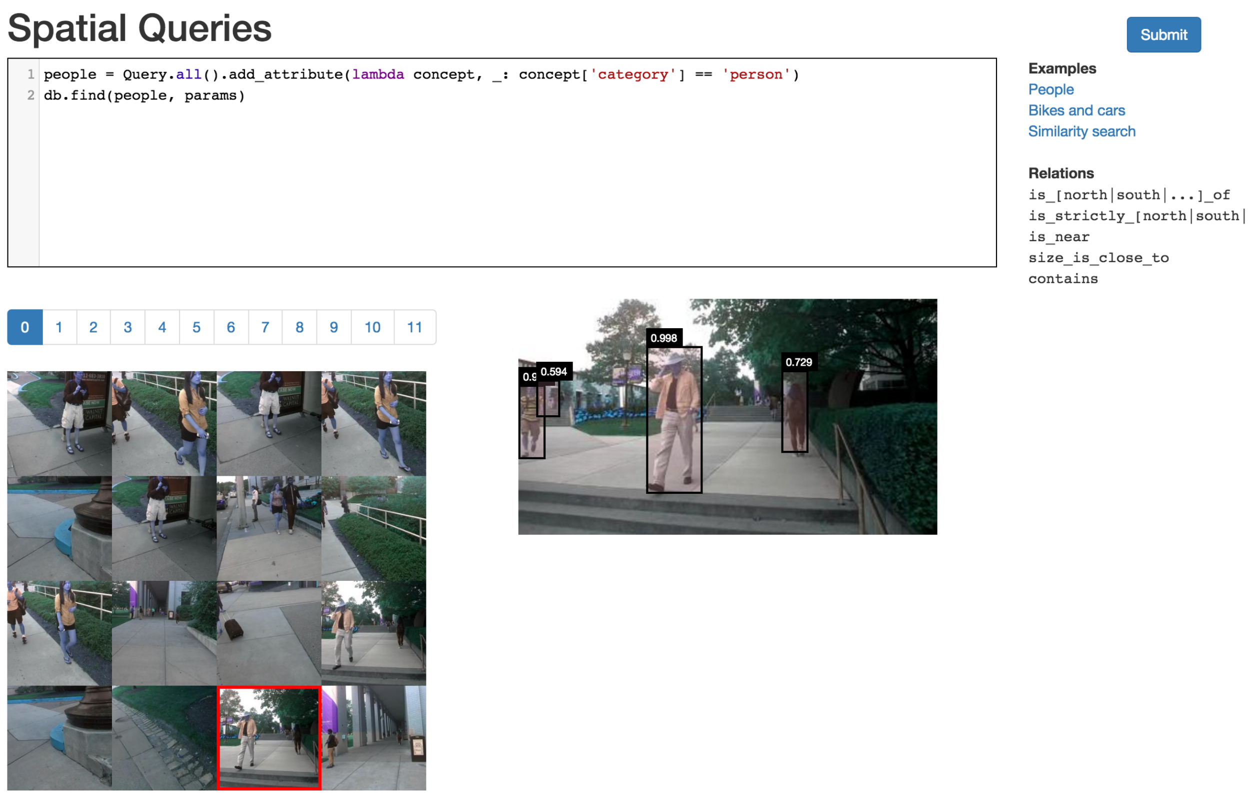
(a) This query implements a primitive reverse image search akin to Google's on the whale image above. To compare similarity of two images, we need a *feature vector*, or a list of numbers that semantically describe the content of the image. This is provided to us by RCNN. The similarity function is the Euclidean distance of the features for the example image and the feature vector for each concept in the database.

Figure 10: Examples of queries and their results from 100,000 images in ImageNet run through the interactive data explorer. These queries get executed over the set of concepts found by RCNN on these images.

5.2 Object detection error analysis

To explore how Lantern could play a part in a larger statistical analysis, I built an application for computer vision researchers at CMU that analyzed errors in the object detectors they were building. When vision researchers build/fine-tune models for detecting objects, they need a way to understand what conditions cause the model to fail. They will train a detector and then evaluate it on a validation set, or a set of prelabeled data separated from the training data for the purpose of validating the detector's model. Then they compare the "ground truth" bounding boxes and labels with the detected ones. Specifically, I implemented some of the analyses described by Hoiem [23] that categorize and cluster false positives, i.e. cases where an object detector classified a block of pixels with a label that was incorrect. These false positives were further classified into different kinds of error:

- loc, or localization: the detector classified an object with the right label, but the bounding box was not correct.
- sim, or similarity: the detector classified an object with the wrong label.
- bg, or background: the detector classified something that is not an object (i.e. the background).



```
people = Query.all().add_attribute(lambda concept, _: concept['category'] == 'person')
db.find(people, params)
```

```
is_[north|south|...]_of
is_strictly_[north|south|
is_near
size_is_close_to
contains
```

Figure 11: Screen capture of the data explorer web interface.

For this application, the primary benefits of Lantern were:

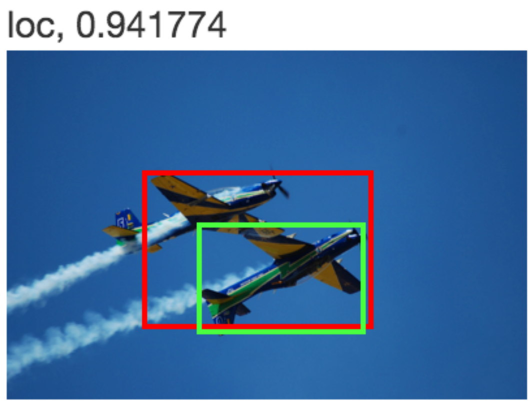
1. Finding detections in parallel, speeding up the process of gathering the data to analyze, and
2. Simplifying the ingest and comparison of the ground truth and detected concepts.

This also again reinforced the finding from before that shipping results to Python was essential, as doing any kind of analysis or plotting in C++ is enormously painful.

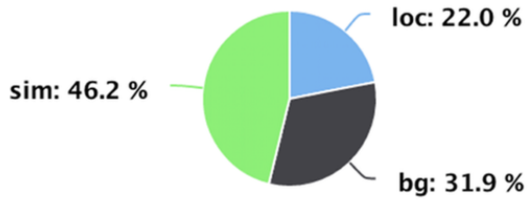
5.3 Face blurring in KrishnaCam

KrishnaCam was my primary video dataset, so it was a good testing ground for the temporal concept model. In order for the dataset to be released publicly, all the faces have to be blurred to protect the privacy of the individuals in the many videos. I was tasked with evaluating various methods for face tracking and finding the best way to blur faces. The face query is fairly simple, as shown in Figure 13 on the next page.

I ran the query with both the OpenCV default face detector based on Haar cascades as well as a more advanced DPM-based detector [24]. However, both of these failed to find most of the faces, likely because the images tend to be blurry since the camera is constantly moving and because the clearest faces are the ones closest to the camera, and those often tend to be occluded or obscured in some manner. Face detectors usually work best on front-facing faces. Instead, I opted to try doing person detection and then blurring the upper quarter of the person's bounding box as a rough



(a) A high confidence false positive. The detector classified two airplanes as a single plane, causing a localization error as the IoU of the two boxes is high enough.



(b) Breakdown of classification errors on the airplane category.

Figure 12: Examples of the kinds of errors shown in the error analysis application.

```
from lantern import Query
from shoddy_face_detector import shoddy_detect_faces
faces_query = Query.split(Query.all(), shoddy_face_detector)
faces = db.track(faces_query, ...)
faces.map(blur_faces).save()
```

Figure 13: Lantern query to blur faces in KrishnaCam. The face detector is shoddy because none of them worked particularly well.

approximation to the face. This method produced far better results, as shown in Figure 14 on the following page. Figure 15 on page 19 shows the execution times for this method, confirming that the queries scale with the number of nodes.

Here, Lantern's temporal concept extension proved highly useful on two levels. First, it improved the accuracy of the detector substantially, as any individual method would pick up some of the people some of the time but was rarely consistent. Second, the Lantern interface made it easy and quick to augment these baseline detectors with the video tracker and to swap out models as necessary.

6 Discussion

I view Lantern as the first step towards a more programmatic way of describing visual concepts for the purpose of visual data analysis. Lantern enables its users to easily describe concepts, efficiently evaluate those descriptions, and build on existing state-of-the-art computer vision tools to explore visual datasets. We will discuss the efficacy of Lantern specifically with respect each of these three goals.

Productivity. In all three applications, the Lantern query language enabled me to quickly describe the visual concepts I wanted to mine in my various datasets. Neither the object detector error

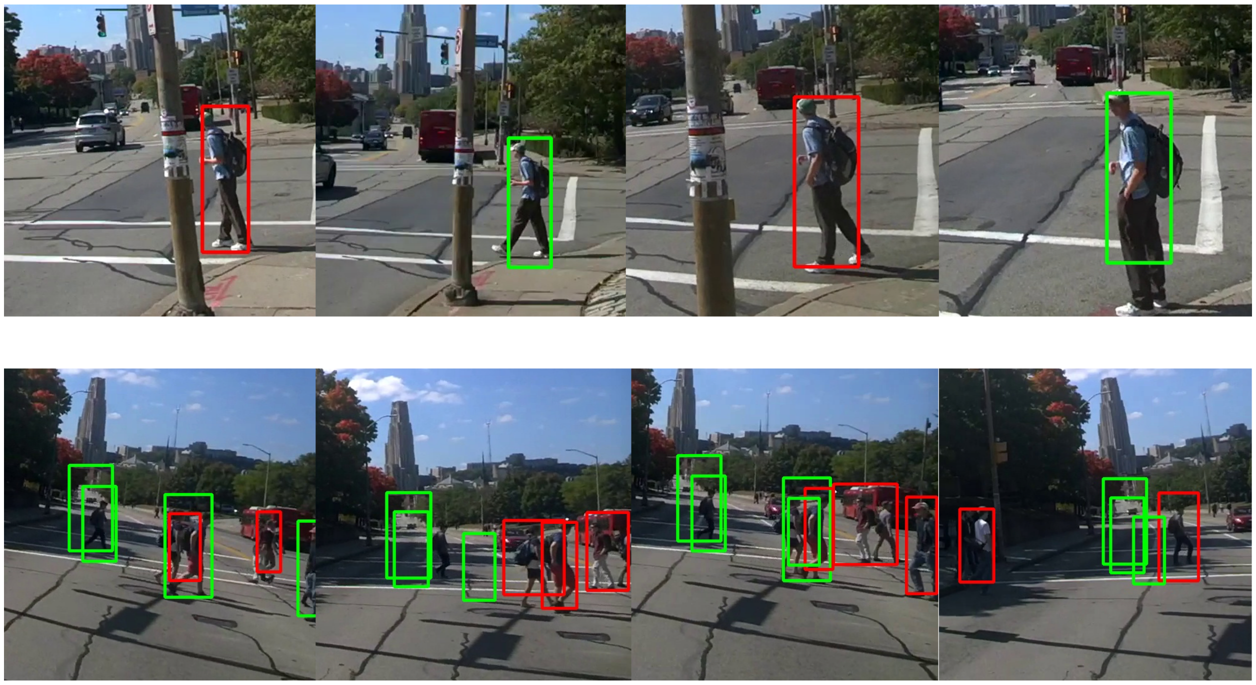


Figure 14: Tracking people in KrishnaCam. The red boxes are when the concept detector fired, and the green boxes are when the detector didn't fire but the concept was successfully tracked by the tracking algorithm. Video tracking generally made the results more accurate, but occassionally produced false positives as shown in the second frame from the left on the bottom.

analysis tool nor the face blurring program required particularly complex descriptions, but the data exploration interface allowed me to find novel concepts. The biggest limitation here, and relatedly an ideal area for future work, is the limited scope of operators in the query language. While the language supports attributes and spatial relations, it strongly needs a grouping operation to express concepts like "people sitting around a table" or "distinct families in a photo." Similarly, it needs simple functions like counting the number of concepts, part of a larger family of aggregative functions found in languages like SQL.

Additionally, the scoring model limits the ability of the user to express ambiguity in their data. For example, Datalog models like Tuffy [21] allow the user to write a number of weighted rules all that contribute to evaluating a claim like "this box is a bicycle." Lantern, by comparison, can describe concepts with multiple attributes, but these get composed together simply by multiplying all of their scores. Future work should explore more nuanced approaches to understanding weights and dependencies between the user's scoring functions.

Efficiency. Although requiring many hours of laborious debugging, writing the system in C++ and mapping to distributed collections enabled the queries to scale well with the capabilities of the cluster. Object detectors executed in sub-second time on GPUs and even inherently sequential algorithms like the video tracking parallelized well, as shown in Figure 15 on the following page. A big unresolved question here is how to allow interoperability with scripting languages while still preserving the speed of the system. The vast majority of open sourced computer vision tools are written in Python or Matlab, so supporting them is a must. However, when a user writes their kernels in Python, that slow, dynamically typed function gets executed in the innermost loop of the Lantern database, and the glue between the runtimes can require anything from hacky pointer

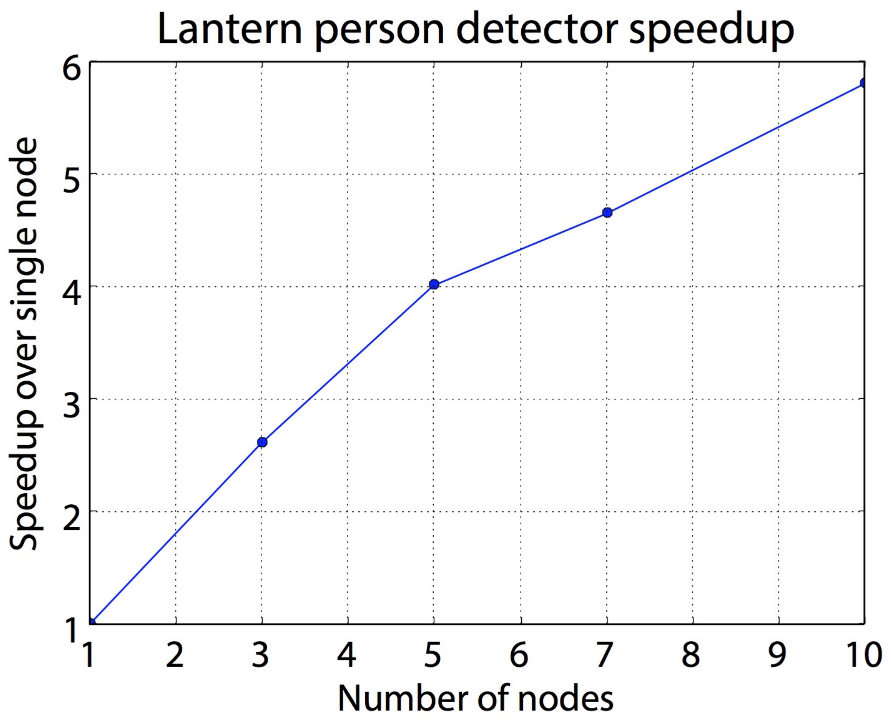


Figure 15: Execution times for face tracking with respect to the number of nodes in the cluster. The application succeessfully scaled as the number of nodes increased, and even though the speedup was not 1-to-1 (i.e. 10x speedup for 10 nodes), it still improved mostly linearly.

manipulation to full on serialization between the language barriers. An ideal visual query language would allow writing code in both a high-level and low-level fashion with tight integration between the parts, Terra [25] being a good example.

Interoperability with computer vision tools. Having both C++ and Python bindings enabled me to easily add in external tools, be it a face tracker, object detector, feature visualizer, etc. into Lantern. One issue with interoperability is that in the temporal concept model, Lantern does not allow concept descriptions to themselves make use of temporal priors when finding concepts in an image, it only augments the concept description evaluator with a generic video tracker that attempts to fill the holes missed by the query. Future work could explore how to, for example, incorporate probability distributions for pixels a given frame as a prior for detections on the next frame.

Overall, Lantern effectively reduced the overhead of writing novel concept detectors and rapidly executing them across large datasets. I hope that my experience building several data analysis applications can guide future work in extending Lantern or other systems for visual concept retrieval, and that Lantern will soon become a valuable tool for analysts to understand visual data.

7 Acknowledgements

I would like to thank Kayvon Fatahalian for being relentlessly helpful as my advisor on both this project and on the harrowing road to graduate school. I would also like to thank Alex Poms for all our good discussions and the late night tech support as well as Ravi Mullapudi for keeping me down to earth and Jonathan Ragan-Kelley for always providing valuable ideas.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[2] Justin Johnson, Andrej Karpathy, and Li Fei-Fei. Densecap: Fully convolutional localization networks for dense captioning. *arXiv preprint arXiv:1511.07571*, 2015.

[3] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[5] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[6] Myron Flickner, Harpreet Sawhney, Wayne Niblack, Jonathan Ashley, Qian Huang, Byron Dom, Monika Gorkani, Jurgen Hafner, Denis Lee, Dragutin Petkovic, et al. Query by image and video content: The qbic system. *Computer*, 28(9):23–32, 1995.

[7] Yong Rui, Thomas S Huang, Michael Ortega, and Sharad Mehrotra. Relevance feedback: a power tool for interactive content-based image retrieval. *Circuits and Systems for Video Technology, IEEE Transactions on*, 8(5):644–655, 1998.

[8] Christopher Town and David Sinclair. Ontological query language for content based image retrieval. In *Content-Based Access of Image and Video Libraries, 2001.(CBAIVL 2001). IEEE Workshop on*, pages 75–80. IEEE, 2001.

[9] Ronald Fagin. Fuzzy queries in multimedia database systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 1–10. ACM, 1998.

[10] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*, pages 91–99, 2015.

[11] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[12] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Learning to compose neural networks for question answering. *arXiv preprint arXiv:1601.01705*, 2016.

[13] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2425–2433, 2015.

[14] Mateusz Malinowski, Marcus Rohrbach, and Mario Fritz. Ask your neurons: A neural-based approach to answering questions about images. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[15] Haoyuan Gao, Junhua Mao, Jie Zhou, Zhiheng Huang, Lei Wang, and Wei Xu. Are you talking to a machine? dataset and methods for multilingual image question answering. *arXiv preprint arXiv:1505.05612*, 2015.

[16] Maaike de Boer, Laura Daniele, Paul Brandt, and Maya Sappelli. Applying semantic reasoning in image retrieval. *Proc. ALLDATA*, 2015.

[17] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Computer Vision and Pattern Recognition*, 2014.

[18] Michael C Burl, Thomas K Leung, and Pietro Perona. Face localization via shape statistics. In *Proc. First Int'l Workshop Automatic Face and Gesture Recognition*, pages 154–159, 1995.

[19] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(9):1627–1645, 2010.

[20] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[21] Feng Niu, Christopher Ré, AnHai Doan, and Jude Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *Proceedings of the VLDB Endowment*, 4(6):373–384, 2011.

[22] Krishna Kumar Singh, Kayvon Fatahalian, and Alexei A Efros. Krishnacam: Using a longitudinal, single-person, egocentric dataset for scene understanding tasks. *Chance*, 43(51.3):48–5, 2016.

[23] Derek Hoiem, Yodsawalai Chodpathumwan, and Qieyun Dai. Diagnosing error in object detectors. In *Computer Vision–ECCV 2012*, pages 340–353. Springer, 2012.

[24] Markus Mathias, Rodrigo Benenson, Marco Pedersoli, and Luc Van Gool. Face detection without bells and whistles. In *Computer Vision–ECCV 2014*, pages 720–735. Springer, 2014.

[25] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multistage language for high-performance computing. In *ACM SIGPLAN Notices*, volume 48, pages 105–116. ACM, 2013.