Exploring big volume sensor data with Vroom

Oscar Moll
MIT
orm@csail.mit.edu

Aaron Zalewski
MIT
azalews@mit.edu

Sudeep Pillai
MIT
spillai@csail.mit.edu

Sam Madden
MIT
madden@csail.mit.edu

Michael Stonebraker
MIT
stonebraker@csail.mit.edu

Vijay Gadepally
MIT
vijayg@ll.mit.edu

ABSTRACT

State of the art sensors within a single autonomous vehicle (AV) can produce video and LIDAR data at rates greater than 30 GB/hour. Unsurprisingly, even small AV research teams can accumulate tens of terabytes of sensor data from multiple trips and multiple vehicles. AV practitioners would like to extract information about specific locations or specific situations for further study, but are often unable to. Queries over AV sensor data are different from generic analytics or spatial queries because they demand reasoning about fields of view as well as heavy computation to extract features from scenes. In this article and demo we present Vroom, a system for ad-hoc queries over AV sensor databases. Vroom combines domain specific properties of AV datasets with selective indexing and multi-query optimization to address challenges posed by AV sensor data.

1. INTRODUCTION

Autonomous vehicles (AVs) are equipped with a suite of sensors including multiple high-resolution cameras, lidar and GPS [7, 6, 4]. Timestamped readings from each of these instruments are logged on every trip. Table 1 profiles an example AV sensor log from an MIT robotics group. The table shows an aggregate data rate in the order of 10 MBps, with the dominant data rates coming from video frames and point clouds[1] Table 1 also shows that the remaining sensors add up to about 100 KBps, or 1% of the volume. This comparatively low data rate should not be confused with low value, however, as these lower throughput sensors effectively label the dataset with valuable metadata. For example, the controller area network (CAN) bus logs events in critical vehicle subsystems such as braking and steering wheel movements. Additionally, the inertial measurement unit (IMU) compass and GPS together establish a coordinate system and orientation for the rest of the data.

[1]The exact data rates depend on the compression used. Some datasets use video compression, others use frame level compression only, and yet other datasets such as [7] prefer lossless image formats for research purposes.

Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

While the primary goal of instrumenting these vehicles today is real-time navigation and collision avoidance, there are a number of natural queries on these sensor logs that have value to users beyond the real-time use cases. For example:

Q1 Compute basic statistics on recent trips such as data rates by sensor and location coverage.

Q2 Retrieve all forward-facing video frames of the corner of Vassar and Main St. in Cambridge, MA., ordered clockwise.

Q3 Retrieve lidar and video readings for all cameras in the vehicle, for intervals when any vehicle camera frame shows a bicycle. Group the data by trip, and order it by timestamp within each trip.

Q4 Retrieve all sensor readings in the minute leading up to an interesting event, such as a possible near miss. e.g., where a vehicle's CAN bus records a sudden brake or sharp steer, group the readings by trip and order them by timestamp within each trip.

These queries can be useful steps towards building 3D maps (query **Q2**) [1], preparing labeled training or testing sets for new perception algorithms (queries **Q4** and **Q3**), and their usefulness can extend beyond the autonomous vehicle navigation and control space.

Unfortunately, existing open source data collection tools today, such as ROS [10], only enable AV researchers to run a few tasks directly on the data files, such as replaying trip data or extracting readings from specific sensors. These tools often operate at the level of a single trip. More complex tasks over the data are solved with custom programs tied to the specific physical data format. As a result, queries such as the ones above are either cumbersome to write, prohibitively slow or both. AV researchers and other users would benefit from storing and querying this data via a database-like interface.

Previous work [8] has shown that relational databases can help manage AV datasets. However, there are several domain specific challenges that make traditional relational engines insufficient out of the box:

- *Computational intensity of UDFs:* State of the art convolutional neural networks (CNNs) for object classification on a single image often require in the order

Table 1: Profile of a 30 GB sensor log of a 40 min trip (13 MBps)

Sensor Type	Frequency	Data rate	Data type
Lidar	10 Hz	8 MBps	Point cloud
Lower-res Lidar (x4)	55 Hz	1 MBps	Point cloud
Lower-res Camera (x4)	20 Hz	4 MBps	JPEG frames
High-res Camera	4 Hz	1 MBps	JPEG frames
CAN bus	900 Hz	50 KBps	Custom struct
IMU	50 Hz	30 KBps	Custom struct
Compass	100 Hz	10 KBps	Custom struct
GPS	6 Hz	< 1 KBps	Custom struct

of 10 GFLOPs, and operate on roughly 200 KB images[11]. Today's 10 TFLOPS GPUs[9] and 200 MBps HDDs [13] have a similar compute-to-bandwidth ratio. Hence, for these UDFs, roughly one HDD can deliver enough data to saturate one GPU. The computational intensity of iterative algorithms over trajectory data [2] and over point clouds [12] is also higher than that of typical database workloads. Because of this shift in workload characteristics, queries over AV data can be heavily compute bound.

- *Big volumes:* At 10 MBps per car, even small organizations with fleets of a hundred cars will produce terabytes of data per hour. Archived data will be correspondingly larger. For ad-hoc querying over archived data to be feasible at all, the system needs to provide throughput orders of magnitude higher than real time.

- *Many features of interest:* obstacles such as bicycles, pedestrians and other cars are of interest to AV researchers. More generally, other features of the environment are often requested such as empty parking spaces, emergency vehicles, license plates, illegally parked cars, etc. Also, researchers are interested in variations in driving behavior, say when passing through deserts or corn fields or in the presence of a traffic jam. As a result, we expect to have to keep track of hundreds of features.

- *Interface and storage issues:* the datatypes of the readings are heterogeneous, as are the tools needed to access and operate on them efficiently. Point cloud data from lidar for example is different from trajectory data as well as images. In addition, querying this data may require expressing geometric predicates. For example, query **Q2** specifies a point of interest and retrieves video segments that look toward the location. The desired result does not include footage from nearby cameras facing the opposite way, only that of nearby cameras facing into the location of interest. The user needs to be able to express this.

The Vroom architecture incorporates multiple strategies to address the aforementioned challenges.

- *Sophisticated feature precomputation and indexing*: expensive feature extraction or object recognition filters do not need to run on every frame in a video captured at tens of frames per second. After all, how far can a bicycle move in a second? Feature recognition and indexing can use feature-specific sampling to cut down on the processing and index storage requirements. Like other data skipping techniques this tactic provides the illusion of higher than real time throughput. Besides sampling, we are also using selective indexing. Based on a query log of past requests we attempt to learn which features justify the cost of indexing. For example, attributes such as location correlate with the kinds of predicates being searched for: query activity on cornfields in Massachusetts does not justify the cost of indexing for cornfields in Massachusetts. In summary, features can be optionally precomputed and selectively indexed in both space and time.

- *Synthesizing cheap predicates*: Because UDF predicates on images such as CNNs are computationally costly and grow with the amount of data, it pays to account for both predicate cost as well and selectivity, and push cheap selective predicates closer to the scan. Using domain specific constraints we can go further, and better leverage predicate migration: in many domains we can synthesize semantically-redundant but cheap predicates. For example, Lidar sensors have a known limited range and location within the vehicle. Hence, a known sphere bounds all laser scan points within an interval. Similarly, image processing algorithms such as the Viola-Jones face detector apply filters in a sequence such that early cheap filters remove most of the work for later more expensive filters.[14].

- *Memoizing*: Often times intermediate results are small compared to the raw data used to generate them. For example object detection may output a few bounding boxes from of a frame, and point clouds can be represented using much less information than the raw points. In the case of object detection CNNs, for example, it almost always pays to preserve these intermediate results because the bandwidth and space costs of writing them back are small compared to the the space, bandwidth use and compute costs over the raw data. Any future queries over overlapping segments of data can reuse already computed results reducing both bandwidth and compute uses.

- *Storage clustering*: Depending on the workload, it will be profitable to store time-series trip data clustered by vehicle identifier, clustered by spatial location, or clustered by time. As with indexing, storage organization can be selective and workload driven.

- *Multi-query optimization*: In the worst case, queries to data which is not indexed will require a sequential search, which will take hours to days, and must be done via batch processes. We assemble such queries and run a cyclic scan to solve them in parallel. This optimization helps especially in the bandwidth bound cases, but also can help on the compute side if there is any overlap there as well.

- *A polystore data model*: There is no single data model or storage system that can accommodate all AV data types. This suggests the need for a polystore data model [3].

Our demo will allow users to interact with a Vroom prototype and visualize results for queries **Q2**, **Q3** and **Q4** against a real AV dataset, side by side with an external source like Google Street View to verify the functionality. Section 2 details the system components. We explain our demo in more detail in Section 3.

2. SYSTEM

There are three main components to the system: the interface, the storage engines and the query processor.

2.1 Query interface

Vroom Exposes a SQL interface over a set of tables, allowing columns with variant or nested types. For querying, the system offers a version of SQL adapted for this data model. For example, we use the following tables to represent the most basic elements necessary to query the data.

- **Raw data table** Each sensor reading is a row on this table. It's schema is `(reading id, timestamp, sensor id, sensor reading)`. Because sensor reading types are sensor dependent, we allow variant types.

- **Sensor metadata table** Each sensor has metadata such as range and field of view attributes. Schema: `(sensor id, sensor info)`. Sensor metadata is sensor dependent, so we allow variant types.

- **Vehicle configuration table** Each vehicle has a precisely specified sensor configuration. This allows us to know which cameras face out of the driver's seat and which out of the passenger's seat. This information is fully specified by an offset and orientation with respect to the vehicle. It's schema is `(vehicle id, sensor id, offset, orientation)`

- **Trip table**. Information about trip metadata. It's schema `(trip id, vehicle id, start, ...)`.

We can calculate basic stats on recently collected data as follows (**Q1**):

```
select sensor_id, sensor_type
 sum(byte_size(sensor_reading)) as data_volume,
 data_volume/trip_duration as data_rate,
 count(*)/trip_duration as frequency
from raw_data
where time.now - trip_start < 6 days
group by trip_id, sensor_id
order by trip_id, data_rate desc
```

Q2 Retrieve video frames facing the corner of Vassar and Main St. in Cambridge, MA., ordered clockwise.

```
let vassar_and_main =
 lat_lon_height(42.3628,-71.0915,7) in
select sensor_reading from raw_data where
sensor_reading.type in (VideoFrame) and
 let sensor_pose =
  pose_estimate(sensor_id, timestamp) in
  distance(sensor_pose,
    vassar_and_main) < 20 and
  angle(sensor_pose.x_axis,
        line(sensor_pose, vassar_and_main)) < 30
order by
 angle(line(sensor_pose, std.east),
       line(sensor_pose, vassar_and_main))
```

Q3 Make a list of trip intervals where a camera in the vehicle frame shows a bicycle. Group the data by trip, and order it by timestamp within each trip.

```
let bike_segments =
   select trip_id,
     timestamp - 5 as t_start,
     timestamp + 5 as t_end
   from raw_data
   where sensor_reading.type in (VideoFrame) and
       bike_detection_udf(sensor_reading) > 0.9
in
select distinct timestamp,
  trip_id, sensor_reading
from raw_data, bike_segments
where
  sensor_reading.type (PointCloud, Video) and
  raw_data.trip_id = bike_segments.trip_id and
  raw_data.timestamp between (t_start, t_end) and
  order by trip_id, timestamp
```

There are a few important aspects to note the queries above: users control the meanings of ambiguous notions like 'near', 'facing', as well as acceptable synchronization tolerances between sensor readings. The user also retains control over the classifiers they wish to use, and the confidence level they allow. If the bicycle feature is precomputed and indexed, then the query executor will convert a portion of query **Q3** to an indexed lookup. Functions to express geometric concepts such as distance, lines, angles, and pose estimation are built-ins. Put together, the functions and builtin tables help the query processor recognize feasible optimization opportunities.

2.2 Query processor

When a query arrives to the query processor, the engine makes an execution plan and applies a few optimization passes to improve it and then runs it over several more general purpose storage engines. Here is a brief description of how different queries get mapped to a desirable execution plan:

For Query **Q1**, a per trip aggregate triggers checks for existing per-trip memoized computations. Old trips that have memoized before match here, and the execution plan now specifies reading from this memoized data source.

Query **Q2** explicitly uses geometric builtins referring to sensor points of view, this cues the query processor to recognize the geometric predicates involved and bound which parts of trip trajectories to skip completely look at. We leverage existing trajectory indexing work for storage.

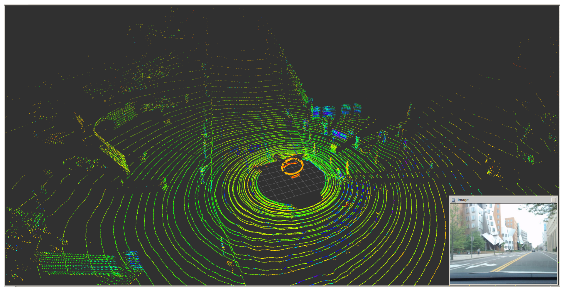

Figure 1: The user interacts with our system via a combination of an Rviz visualization for sensor playback and a web based GUI for query input.

Query **Q3** exemplifies a case where we expect processor to apply skipping techniques. The interval bound `(timestamp - 5, timestamp + 5)` we have chosen around the event of interest tells the optimizer it can prioritize looking at frames recorded at `timestamp - 10` or `timestamp + 10` . If there are any hits there, then there is no need to run the UDF on any frames in the interval `(timestamp - 10, timestamp)` .

For query batching, after the plan has gone through optimization passes we can establish if a full scan is required. If so, the query gets queued up. At that point, a new query plan is recomputed for all queries. Queries in the waiting batch are scheduled to run when the current long running batch finishes.

2.3 Storage engine

We make use of open source tools from the AV community for specialized operations such as noise filtering algorithms for trajectory data[10], as well as TensorFlow and open source object detection algorithms. Our demonstration prototype is implemented by combining a relational engine for metadata storage and file system based blob management. A python layer exposes a SQL-like query interface to the programmer.

3. DEMONSTRATION

We use a 200+ GB MIT AV dataset comprised of multiple trips from different vehicles, with the sensors shown in Table 1. The user will be able to with the database via both the Rviz data visualizer[10] and a web GUI, with the ability to vary query parameters such as the location, object of interest (e.g., car, person, bike), etc.

3.1 Applications/Queries

In our demonstration, we will allow users to run queries **Q1**, **Q2**, **Q3** and **Q4** from the introduction on the MIT AV data corpus, and allow them to change query parameters. We show a number of useful data visualizations generated from the query results. We let users verify the results by comparing results with existing references such as Google Street View.

4. CONCLUSION AND FUTURE WORK

Vroom is a system that addresses AV data challenges of interface, volume, computational intensity by providing a declarative SQL-like interface, and by employing a query engine that applies domain specific optimizations suitable for AV sensor data, as well as leveraging existing database techniques that are especially appropriate for this domain. The demonstration shows that Vroom can express important operations over AV sensor data as declarative queries. The queries we demo are both useful to AV researchers and challenging to implement otherwise.

We expect more types of storage engines to be necessary in the future, for example array specific storage to manage arrays of decompressed image data and point clouds, to enable and enable querying these data sources at finer granularity.

5. REFERENCES

[1] S. Agarwal, Y. Furukawa, N. Snavely, I. Simon, B. Curless, S. M. Seitz, and R. Szeliski. Building rome in a day. *Commun. ACM*, 54(10):105–112, Oct. 2011.

[2] S. Agarwal, K. Mierle, and Others. Ceres solver. `http://ceres-solver.org`.

[3] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker. The bigdawg polystore system and architecture. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–6. IEEE, 2016.

[4] V. Gadepally, A. Krishnamurthy, and U. Ozguner. A framework for estimating driver decisions near intersections. *IEEE Transactions on Intelligent Transportation Systems*, 15(2):637–646, 2014.

[5] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 267–276, New York, NY, USA, 1993. ACM.

[6] A. S. Huang, M. Antone, E. Olson, L. Fletcher, D. Moore, S. Teller, and J. Leonard. A high-rate, heterogeneous data set from the darpa urban challenge. *Int. J. Rob. Res.*, 29(13):1595–1601, Nov. 2010.

[7] W. Maddern, G. Pascoe, C. Linegar, and P. Newman. 1 year, 1000 km: The oxford robotcar dataset. *The International Journal of Robotics Research*, 2016.

[8] P. Nelson, C. Linegar, and P. Newman. Building, Curating, and Querying Large-scale Data Repositories for Field Robotics Applications. In *International Conference on Field and Service Robotics (FSR)*, Toronto, ON, Canada, June 2015.

[9] NVIDIA. Gp100 pascal whitepaper. [link].

[10] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[11] J. Redmon and A. Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[12] R. B. Rusu and S. Cousins. 3d is here: Point cloud library (pcl). In *2011 IEEE International Conference on Robotics and Automation*, pages 1–4, May 2011.

[13] Seagate. Enterprise capacity 3.5" hard drives (factsheet). [link].

[14] P. Viola and M. J. Jones. Robust real-time face detection. *Int. J. Comput. Vision*, 57(2):137–154, May 2004.