ABSTRACT
State of the art sensors within a single autonomous vehicle (AV) produce video and lidar data at more than 30 GB/hour. Not surprisingly, even small AV research teams easily 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 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 rise to the 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 [8, 7, 4]. For example, Table 1 profiles an example AV sensor log from the MIT robotics group. The table shows an aggregate data rate of 10 MBps, with the dominant data types by byte volume being point clouds and video frames from multiple sensors. Table 1 also shows that the remaining sensors put together add up to about 100 Kbps, or about 1% of the volume. The comparatively low volume should not be confused with low value, however, as these sensors hold key metadata that makes the overall dataset much more queryable. For example, the inertial measurement unit (IMU), compass and GPS together help establish a coordinate system and orientation for the rest of the data. The car-area network (CAN) bus records important events from critical systems such as brakes and steering wheel. Note that each record in the sensor logs has a timestamp.

Besides the in-car use of these sensors for real-time navigation and collision avoidance, there are many uses for archives of this data, giving rise to a number of natural queries users would like to ask over accumulated trip data, for example:

Q1 Compute basic stats on recent trips such as data collected by sensor, rates.
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 are useful to sanity check sensor configurations (query Q1), building maps (query Q2), offline training of collision avoidance algorithms (query Q3), and understanding vehicle dynamics in emergency situations (query Q4).

Unfortunately, existing open source data collection tools, like ROS [11], limit AV researchers to running a few tasks directly on the data files, such as replaying data directly from a log, 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 data formats. 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 on managing AV datasets [9] has demonstrated the usefulness of database techniques to the problem. However, there are several domain specific challenges that make traditional relational engines insufficient out of the box:

- Interface issues: the datatypes of the readings are heterogeneous, as are the tools needed to operate on them efficiently. In addition, querying this data requires expressing geometric predicates. For example, query Q2

1 The exact data rates depend on the levels of compression used. Some datasets use video compression, others use frame level compression only, and yet other datasets such as [8] prefer raw imagery for research purposes.
Table 1: Sensor data profile for a sample trip in the MIT dataset. File size: 30 GB. Trip duration: 40 min. Data rate 13 MBps.

<table>
<thead>
<tr>
<th>Sensor Type</th>
<th>Frequency</th>
<th>Data rate</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>lidar</td>
<td>10 Hz</td>
<td>8 MBps</td>
<td>Point cloud</td>
</tr>
<tr>
<td>Lower-res Camera (x4)</td>
<td>20 Hz</td>
<td>4 MBps</td>
<td>JPEG frames</td>
</tr>
<tr>
<td>Laser scanner (x4)</td>
<td>55 Hz</td>
<td>1 MBps</td>
<td>Point cloud</td>
</tr>
<tr>
<td>High-res Camera</td>
<td>4 Hz</td>
<td>1 MBps</td>
<td>JPEG frames</td>
</tr>
<tr>
<td>CAN bus</td>
<td>900 Hz</td>
<td>50 KBps</td>
<td>Custom struct</td>
</tr>
<tr>
<td>IMU</td>
<td>50 Hz</td>
<td>30 KBps</td>
<td>Custom struct</td>
</tr>
<tr>
<td>Compass</td>
<td>100 Hz</td>
<td>10 KBps</td>
<td>Custom struct</td>
</tr>
<tr>
<td>GPS</td>
<td>6 Hz</td>
<td>&lt; 1 KBps</td>
<td>Custom struct</td>
</tr>
</tbody>
</table>

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.

- Computational intensity of UDFs: State of the art convolutional neural networks (CNNs) for object classification on a single image often require in the order of 10 GFLOPs, and operate on roughly 200 KB images\[12\]. Today’s 10 TFLOP GPUs\[10\] and 200 MBps HDDs\[14\] 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\[1\] and over point clouds\[13\] 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.

In this demo, we will show our data processing system for AV data, Vroom. Vroom incorporates the following strategies to address the aforementioned challenges:

- **Sampling.** Complex scene analysis ("feature recognition") does not need to be done on every frame in a video captured at 30 or 60 frames per second. After all, how far can a bicycle move in a second? Indexing features can use feature-specific sampling to cut down on the processing required. This technique helps address both bandwidth and compute pain points, and like other data skipping techniques provides the illusion of higher than real time throughput.

- **Synthesizing cheap predicates:** Because the compute costs of some UDF predicates are high 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: we can synthesize semantically-redundant but cheap predicates. For example, a camera pointed out the driver’s side can’t see something on the passenger side, and lidar has a known limited range. Hence, a known sphere bounds all laser scan point clouds. Collected datasets already specify the locations and orientations of their sensors, so Vroom exploits that metadata to more cheaply filter or skip data.

- **Memoizing and Indexing:** 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\[6\]. 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. Because raw data inputs are immutable, many intermediate results remain valid after inserting more data.

- **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. Point cloud data from lidar for example is different from geospatial data, and from images. We leverage our MIT polystore (BigDAWG) to facilitate data storage and query processing.

Our demo will allow users to interact with Vroom and visualize the results of queries such as Q2, Q3 and Q4 against a real AV dataset, side by side with an external source like Google Street View. Section 2 details the system components. We propose our demo 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 like 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

- **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, trip start, trip end, ...).

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

```sql
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
order by trip_id, sensor_id, data_rate desc
```

On the other hand, existing tooling runs on the input format and suffers from recurrent overheads due to the nature of that format. By using a query interface, the first query benefits from basic optimizations like physically clustering streams by sensor type. Because the query above does not need lidar data, the system reads substantially less data. While the above examples are simple, in our experience, researchers directories are full of manually cached views of this data in the filesystem to avoid re-running these simple commands. Now we can provide less ambiguous descriptions for queries in the introduction:

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

```sql
let vassar_and_main =
l 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, VideoFrame) and
let sensor_reading =
pose_estimate(sensor_id, VideoFrame) 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.

```sql
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. Functions to express geometric concepts such as distance, lines, angles, and pose estimation are built-ins. Put together, the functions and built-in 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.

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 close to timestamp - 10 and timestamp + 10 . If there are any hits there, then we no longer need to run the UDF on any frames in the interval (timestamp - 10, timestamp). The engine has hardcoded knowledge about interval algebra to reason about time intervals.

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
2.3 Storage engine

The raw AV data is normally collected in a handful of binary formats. Collected data is naturally clustered physically by trip, in chronological order. Several practical heuristics are applied at load time, such as splitting the bundled sensor streams into data type specific columns. This simple technique helps querying the lower volume (KBps) sensors much faster. Additionally, we make use of tools from the open source AV community for specialized operations such as applying localization algorithms\[1\] at trip load time, as well as trajectory indexing libraries.

UDFs get registered with the engine, and their results become extra (sparse) columns associated with the data objects they have computed results on.

3. DEMONSTRATION

3.1 Dataset

We use a 10 TB dataset of sensor data collected by the robotics group at MIT. The composition of the data is similar to that shown in Table\[1\].

The proposed demonstration will rely on a number of storage engines orchestrated by the BigDAWG polystore. We employ Vertica for some of the metadata and SciDB for array data such as point clouds and images. The proposed architecture is given in Figure\[1\]. Specifically for this demonstration, we also employ open source visualization tools for AV data \[11\].

3.2 Applications/Queries

In our demonstration, we let users run the queries \[Q1\], \[Q2\], \[Q3\] and \[Q4\] from the introduction the MIT AV data corpus we described, and allow them to change some of the specific 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

Vroom addresses the AV data challenges of interface, volume, computational intensity by creating a declarative sql-like interface, and by deploying 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 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.

5. ADDITIONAL AUTHORS

6. REFERENCES


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


