# IOT Traffic Simulation: *Predictive Analysis*

2016 Computer Science Senior Thesis

Neeraj Asthana
University of Illinois at Urbana-Champaign
Laboratory for Cosmological Data Mining
Urbana, Illinois
neeasthana@gmail.com

Robert J. Brunner[*]
University of Illinois at Urbana-Champaign
Laboratory for Cosmological Data Mining
Urbana, Illinois
bigdog@illinois.edu

## ABSTRACT

The abstract for your paper for the PVLDB Journal submission. The template and the example document are based on the ACM SIG Proceedings templates. This file is part of a package for preparing the submissions for review. These files are in the camera-ready format, but they do not contain the full copyright note. Note that after the notification of acceptance, there will be an updated style file for the camera-ready submission containing the copyright note.

## 1. INTRODUCTION

### 1.1 Problem Statement

Traffic congestion and associated effects such as air pollution pose major concerns to people who are on the move. Congestion has increased dramatically during the past 20 years in U.S. cities. During this time, the number of hours lost each year by the average driver to congestion has increased by 300 percent. According to Newsweek, the average United States commuter spends an average of 42 hours stuck in traffic every year, resulting in an estimated 160 billion dollar loss from loss of productivity. Currently, drivers are only informed by current and historical traffic conditions in their area. Additionally, increasing the capacity of the roadways is expensive and, in some areas where land is scarce, is not an option.

### 1.2 Proposed Solution

Many popular methods predict traffic states by aggregating streamed GPS data (coordinates, velocities, and timestamps) of many vehicles. With the recent advances in sensor technology, infrastructure for estimating temperature, pressure, humidity, precipitation, etc. are readily available on roadways and can be used to determine driving conditions. The goal and novelty of our project is to combine streamed GPS traffic estimation algorithms with dynamic road sensor data to accurately predict future traffic states and efficiently advise and direct drivers. We will make use of the time decay model for aggregating stream GPS data suggested in Aggregating and Sampling Methods for Processing GPS Date Streams for Traffic State Estimation and include methods to incorporate road sensor data. By using both and current traffic conditions and sensors on the road, we hope to understand how traffic may change on a single road segment in the near future (30 minutes, 1 hour, etc.).

[*]Senior Thesis Advisor

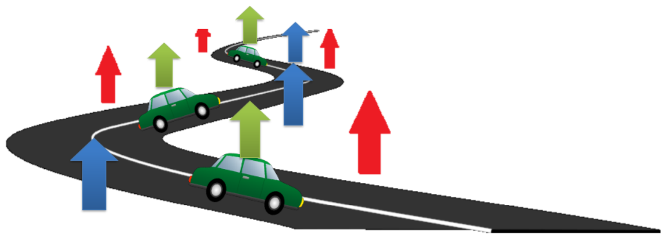

**Figure 1: Cars and Sensors along a road segment generating data that can be processed by our system.**

## 2. DATA SOURCES

There are two major data sources necessary for this project: streamed GPS data and road sensor data. These data sets are continually generated and streamed to our backend clusters and these portions of the project were produced by my fellow project collaborators. Rishabhs work focused on the generation of streamed GPS data while Vaishalis work focused on generating road sensor data.

### 2.1 Streamed GPS Data

There will be many vehicles travelling along the roads of our model and each vehicle will be continuously be emitting data to our backend server at a predefined interval (i.e. every second, every minute, etc.). Each emitted data packet from a vehicle will be of the form (vehicle id, location, velocity, timestamp) which can be abbreviated as $(id, v_i, l_i, t_i)$. The vehicle id is a unique identifier for each vehicle. The location consists of a geodesic latitude and longitude coordinate. The velocity field consists of the vehicles current velocity which will be measured in meters per second. The timestamp references the time at which the location and velocity data was recorded in the form %Y-%m-%d %H:%M:%S.

**Table 1: Network Resources setup on the NCSA ACX Cluster**

| Cluster | Workers | Cores | RAM |
|---|---|---|---|
| KAFKA | 2 | 8 | 8 GB |
| CASSANDRA | 2 | 4 | 4 GB |
| SPARK | 6 | 24 | 17.2 GB |

## 2.2 Streamed Road Sensor Data

Our model allows us to generalize to any number or type of sensor, however for now we will only focus on a single type of sensor which measures road surface temperature. Sensors will be placed along a roadway at a predefined interval ((i.e. every mile, every five miles, etc.) and I will assume that these locations are known. Each sensor will periodically emit information to our backend server at a predefined interval (i.e. every second, every minute, etc.). Each sensor data packet will be in the form: (sensor id, temperature, timestamp). The sensor id is a unique identifier for each sensor. The temperature defines a road surface temperature at the location of the sensor in Celsius degrees. The timestamp references the time at which the temperature was recorded in the form %Y-%m-%d %H:%M:%S. In the future we will focus on extending our model to incorporate different types of sensors. Other potential sensor types include humidity, wind speed, water precipitation, etc.

## 3. TECHNICAL SPECIFICATIONS

Our backend will be implemented on the NCSA ACX cloud computing cluster. The purpose of these systems is to efficiently stream our data sources to allow for real time traffic state prediction. The focus of this paper will mostly be centered on the stream processing/cloud computing system by implementing the model mentioned in sections below with the data streamed from our message broker. A basic network diagram of the data flow is included in Figure 2 for reference. This section describes each of the systems and explains the actions taken at each step.

## 3.1 Network Resources

The specifications for each cluster are defined in Table 1.

### *3.1.1 Message Broker (Kafka)*

All vehicle and sensor data will be aggregated and ordered by a single message broker for efficient processing. Large amounts of real time data will continually be streaming into our backend and a message broker like Apache Kafka will help us efficiently manage large amounts of messages. All the data captured at this step will be aggregated and ordered so that it can be sent to the data storage and stream processing units. The Kafka cluster contains two major queues, traffic and weather, known as topics. The traffic topic contains streamed vehicle data and each entry is of the form (latitude, longitude, velocity, time, vehicle id). The weather topic contains streamed vehicle data and each entry is of the form (segment id, time, measurement).

### *3.1.2 Stream Processing (Spark)*

The most recent data from the cars and the sensors will be sent to this unit to update the current road condition models using the time decay model. The newest data will

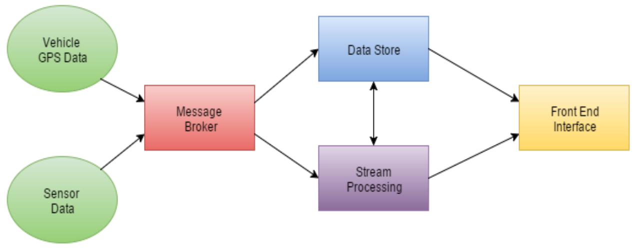

**Figure 2: Network Diagram highlighting the major tasks occuring at each level.**

be streamed into the system and be used (along with historical GPS data from the data store) to generate a new model describing the road conditions. Local variables will be stored on all of the nodes in the Spark Cluster that persist and update the parameters from the time decay model (details about the time decay model are provided in the following section). More details and specific code segments are provided about Spark Streaming are provided in the section below. There are considerations to also use or benchmark Apache Flink or Apache Storm in place of Spark as these may scale better to larger data streams.

### *3.1.3 Database (Cassandra)*

All data will be stored in this unit to either be included in future models or displayed on the front end interface. HDFS, HBase, and MongoDB have also been considered as replacements for this unit.

## 3.2 Data Pipeline

Figure 3 details the general flow of the data between different clusters. Data is generated by the Vehicles and Road Sensors at a specified time interval (i.e. every second, mintute, etc.). This large stream of data is then aggregated and sorted by Kafka, placing the data into two separate streams. The data is then sent to the Web Dashboard to display the current traffic conditions and it is also sent to the Spark Streaming instance to be processed into future speed predictions. The current conditions and predicted speeds are stored in a Cassandra database to be analyzed in depth at a later point. Data is also sent to the Web Dashboard which can then be displayed on consoles in vehicles.

## 4. TIME DECAY MODEL

Our model incorporates the time decay model suggested in Aggregating and Sampling Methods for Processing GPS Date Streams for Traffic State Estimation with road sensor data to estimate traffic states on specific road segments. We implement the aggregation based method mentioned in the paper as it requires minimal resources to implement and is efficient at estimating average speeds on road segments. We further adjusted the calculated average speed on each road segment based on data from road sensors from that same road segment.

The message broker (Kafka) sends vehicle packets of the form (vehicle id, location, velocity, timestamp) and each road sensor sends packets of the form (sensor id, temperature, timestamp), which all will be ordered and aggregated by the message broker to form a stream and sent to the Spark Streaming system.

## 4.1 Velocity Estimation

Previous models of estimating traffic states from GPS data streams focus on either historical data or a sliding window (SW) sampling method. Historical methods attempt to aggregate previous speeds of vehicles on a particular road (each of which has an equal weight); however this method is not extremely accurate in the short term as GPS data streams are volatile and evolve rapidly based on different circumstances. The sliding-window sampling method conversely saves the most recent GPS data stream and uses only a select few point to estimate new traffic states. This also may not be completely accurate when estimating traffic states as it creates unstable traffic conditions that may not reflect the average behavior of a specific road. Therefore, I plan to implement a time decay model which highly values new data records but also factors old data records. This method is also online and real time, allowing us to process data quickly and return current information.

The key idea of the time decay model is to weight new records higher than old records in that same location. This is accomplished by weighting each record with a value w using the following exponential decay function where i represents the ith GPS record, t is the processing time, and ti is the timestamp of the current record:

$$w(i,t) = e^{\beta(t_i - t)} \tag{1}$$

where $\beta > 0$ and $t \geq t_i$.

Using this model, our system calculates weights for each of the data records on a specific road segment and older records are weighted less than newer records.

We must also notice that high velocities are more useful than low velocities as low velocities can signal events that are not related to traffic flow such as parked vehicles or stopping at a red light. Therefore, we must weigh high velocity data as more important than low velocity data which is accomplish by assigning another weight to each record using a function g that is positive, nondecreasing, and monotonic as follows:

$$w^v(i) = g(v_i) \tag{2}$$

Our model then assigns a full weight to a specific GPS data point by the following equation:

$$w^*(i,t) = w(i,t) \cdot w^v(i) \tag{3}$$

For each predefined road segment, defined as $r$ (specifying a latitude and longitude radius), the estimated average velocity at time $t$ can be found using the following equation:

$$\overline{V}(r) = \frac{\sum_{n=1}^{m} f(t_i - L) \cdot g(v_i) \cdot v_i}{\sum_{n=1}^{m} f(t_i - L) \cdot g(v_i)} \tag{4}$$

## 4.2 Model Updates

Equation 4 would be ideal for a batch processing system, however, it does not scale well to a fully streamed system that must constantly be updated with new values. The model must be easily updated with newly streamed data points as well. In order to accomplish this, we will break $\overline{V}(r)$ into its numerator and denominator and define two new values $X$ and $Y$:

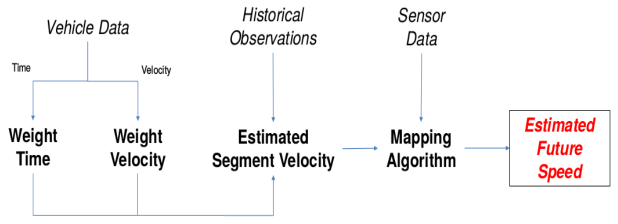

**Figure 3: Describes how data flows through our modified Time Decay model to produce an estimated future speed.**

$$X = \sum_{n=1}^{m} f(t_i - L) \cdot g(v_i) \cdot v_i \tag{5}$$

$$Y = \sum_{n=1}^{m} f(t_i - L) \cdot g(v_i) \tag{6}$$

The values $X$ and $Y$ will be persisted for each road segment and new observations can incrementally be added to these specific components. Then, whenver a velocity estimate ($\overline{V}(r)$) is needed, the values can be divided as $\overline{V}(r) = \frac{X}{Y}$.

## 4.3 Incorporating Road Sensors

The model described above is used only to get current estimates of speeds on roads and is not novel. Instead, our model instead incorporates road sensor data from road segments along with the estimated current velocity of those segments to project a future velocity on that same road segment (ex. In the next 30 mins, hour, etc.). Our current model only uses road surface temperatures with known parameters to predict a future speed. However, in the future when multiple sensors are used, we will use a machine learning algorithm (i.e. simple linear ridge regression) to estimate future velocities using all of these sensor data as inputs.

## 5. DYNAMIC MAP MATCHING ALGORITHM

Car data and road sensor data each have locations associated with each observation and must be matched to road segments in order to update specific model parameters. Therefore, we must translate a latitude and longitude observation to a road segment.

## 5.1 Code

The current algorithm matches a road segment by find the closest road segment to the observation. The algorithm casts a radius of .25 kilometers to accommodate for slight errors in GPS streamed locations. The code contains 2 key functions. The first function (distance) takes two latitude and longitude pairs and calculates the distance between them.

```
#Distance in kilometers between two latitude and long
def distance(obsLoc, segmentLoc):
    lat1, lon1 = obsLoc
    lat2, lon2 = segmentLoc
    radius = 6371 # km

    dlat = math.radians(lat2-lat1)
    dlon = math.radians(lon2-lon1)
    a = math.sin(dlat/2) * math.sin(dlat/2)\
     + math.cos(math.radians(lat1)) \
        * math.cos(math.radians(lat2)) * \
        math.sin(dlon/2) * math.sin(dlon/2)
    c = 2 * math.atan2(math.sqrt(a), \
        math.sqrt(1-a))
```

The second function (findSegment) attempts to find the closest segment to an observations location within a .25 km radius.

```
#match an observation to a road segment (within .25 km of a road segment)
def findSegment(latitude, longitude):
    obsLoc = (latitude,longitude)

    closest = None
    for segment in segments:
        segmentLoc = (segment.latitude, \
            segment.longitude)
        dist = distance(obsLoc, segmentLoc)
        if closest == None or \
            dist <= closest:
            if dist <= .25:
                closest = segment
```

## 5.2 Future Considerations

In the future, we hope to be able to better distinguish between road segments by including a direction of travel field within each observation. In this way, we will be able to accurately match road segments if vehicles are at an intersection or close to another road. We will also be able to keep data for two way traffic (incoming and outgoing for a road segment).

## 6. SPARK STREAMING

Spark Streaming is an open source cluster computing engine for efficient and large-scale data processing (in real time). It also has simple integration libraries for Cassandra and Kafka, as well MLib (machine learning library). We used pyspark within an IPython Notebook to run our Spark commands. Applications are easily submitted to the cluster using a submit script.

## 6.1 Mechanics

Spark holds its data in objects know as Resilient Distributed Datasets (RDDs) which in our case are created from data streamed from Kafka. Each RDD can then be transformed by operations (such as mapping, reducing, filtering, etc.) to generate the desired results. Spark Streaming creates a DStream for each Kafka queue which is essentially a series of related RDDs to represent a continuous stream of data. Each RDD in the DStream represents a specified time interval in the stream (a single fetch from Kafka).

Each Spark program contains a driver that handles organization and structure of each job submitted to the cluster. Other nodes, workers, then execute the transformational code in parallel and eventually accumulate a result.

## 6.2 State Preservation

Each road segment is instantiated using the Segment class and are held in the segments data structure, which is implemented as shared variables in Spark. This data structure will be updated and available on all Spark Nodes for efficient computing.

Each Segment object has an id and a location (latitude and longitude). In addition, each segment object contains the parameters X and Y for that road segment, the estimated speed and the number of observations the model is updated with. Lastly, the segment holds a landmark time (time of the first observation) and the time of the most recent observation, which are both used to weight observations in the time decay model.

Code:

```
class Segment:
    def __init__(self, latitude, longitude,\
     landmark_time, identification):
        self.latitude = latitude
        self.longitude = longitude
        self.landmark = landmark_time
        self.current_time = landmark_time
        self.id = identification
        self.obs = 0
        self.X = 0
        self.Y = 0
        self.speed = 0
```

## 6.3 Model Update

Vehicle and temperature data are streamed from Kafka every second and each stream is stored as a single Resilient Distributed Dataset (RDD) on a random worker node in the Spark cluster. Each observation is then matched to a road segment using the dynamic map matching algorithm (if possible). Map, Reduce, and Filter Python queries in batch and streaming mode are then used by PySpark to implement time decay traffic algorithm. Each observations velocity and time weights are calculated and then aggregated to update the $X$ and $Y$ values of the road segment.

Steps:

1. Mapping Each node operates on each observation in an RDD individually by associating a location using a dynamic matching algorithm

2. Reducing Each key is then aggregated using the time decay model update functions

The mapping and reducing is managed by a single master node (driver) that schedules and combines map-reduce tasks from different nodes.

Code:

```
def f(time):
    return time

def g(velocity):
    return (velocity ** 2)

def update(lat, longitude, \
    velocity, time):
    #dynamic map matching algorithm
    seg = findSegment(lat,longitude)

    #weight time and velocity
    wtime = f(time-self.landmark)
    wvel = g(velocity)

    #produce X and Y for this observation
    #and add these to the segment speed
    X = wtime*wvel*velocity
    Y = wtime*wvel
    seg.X += X
    seg.Y += Y
    seg.speed = seg.X/seg.Y
    seg.obs += 1
    seg.current_time = time
```

## 6.4 Spark-Kafka Integration

Spark contains Kafka integration libraries that are easily managed. The important step is to include the jar in the submit script used to run the command on the cluster. We simply provided a list of the brokers and the topics to our Spark Streaming Context in order to receive data from Kafka on our Spark cluster. The Streaming Context fetches data from the Kafka queues every 30 seconds however this time interval is easily adaptable as the application scales.

Code:

```
%%writefile timedecaykafka.py

from __future__ import print_function

import pyspark
from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils

# create SparkContext
conf = SparkConf().setAppName("Kafka-Spark")
sc = SparkContext(conf=conf)

# create StreamingContext
ssc = StreamingContext(sc, 30)

# new approach (w/o receivers)
topic = ["mytopic"]
brokers = "141.142.236.172:9092,141.142.236.194:9092"
directKafkaStream = KafkaUtils.createDirectStream(ssc, topic, {"metadata.broker.list": brokers})

lines = directKafkaStream.map(lambda x: x[1])
counts = lines.flatMap(lambda line: line.split(" ")) \
        .map(lambda word: (word, 1)) \
        .reduceByKey(lambda a, b: a+b)
counts.pprint()

ssc.start()
ssc.awaitTermination()
```

## 7. FUTURE WORK

In the future we would like to incorporate many features into our models which were not possible to include during the spring semester. There are many optimizations and large scale features we hope to work on that could improve the system.

## 7.1 Dropped/Missing Data

Not all data is properly delivered to our backend servers as receivers could lose service or turn off. We must adapt our model to successfully predict traffic conditions even when every piece of data is not completely available.

## 7.2 Security

How can we protect our network from hackers? The data and inferences provided by this model are extremely sensitive and proper authentication would be necessary to access this information.

## 7.3 Unidentified Vehicles

Some vehicles are not equipped with GPS devices and we must adapt our models to incorporate these vehicles as well. These vehicles could skew different results from the Time Decay model (especially if there are large concentrations of them) and we must remain agile amongst this uncertainty.

## 7.4 Variable Sensors

Many types of sensors are available to monitor weather and traffic conditions. Currently, our model only uses road surface temperature sensors. However, we must incorporate other types of sensors into this model to try and see how different combinations of road conditions change vehicle behavior.

## 7.5 Sliding Window

Delete insignificant and outdated data inside the database.

## 8. ACKNOWLEDGEMENTS

We acknowledge support from the National Science Foundation Grant No. AST -1313415, the National Center for Supercomputing Applications, the University of Illinois and Microsoft Azure.

## 8.1 References

Laboratory for Cosmological Data Mining
National Center for Computing Applications ACX Cluster
Apache Spark 1.5.0 Documentation http://spark.apache.org/docs/lates
Ipython Notebooks http://ipython.readthedocs.org/en/stable/
Docker https://docs.docker.com/
Spark-Ipython Notebook Integration http://www.kuntalganguly.com/2
ipython-notebook-with-apache.html
Cassandra http://docs.datastax.com/en/cassandra/2.0/cassandra/gett
Time Decay Scholarly Paper Intelligent Transportation Systems, IEEE Transactions on, Issue Date:Dec. 2013, Written by:Jia-Dong Zhang; Jin Xu; Liao, S.S.
Kafka http://kafka.apache.org/documentation.html
https://arrayofthings.github.io/

## 9. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author's Guide* and the **.cls** and **.tex** files that it describes.

## 9.1 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references).

## APPENDIX

You can use an appendix for optional proofs or details of your evaluation which are not absolutely necessary to the core understanding of your paper.

## A. FINAL THOUGHTS ON GOOD LAYOUT

Please use readable font sizes in the figures and graphs. Avoid tempering with the correct border values, and the spacing (and format) of both text and captions of the PVLDB format (e.g. captions are bold).

At the end, please check for an overall pleasant layout, e.g. by ensuring a readable and logical positioning of any floating figures and tables. Please also check for any line overflows, which are only allowed in extraordinary circumstances (such as wide formulas or URLs where a line wrap would be counterintuitive).

Use the `balance` package together with a `\balance` command at the end of your document to ensure that the last page has balanced (i.e. same length) columns.