The Edward S. Rogers Sr. Department of Electrical and Computer Engineering

University of Toronto

ECE496Y Design Project Course
Group Final Report

Title: Applications for Intelligent Transport – History Timeline Reporting Engine

Team #:	2016616	
Team members:	Name:	Email:
	Joohyun Lee	joohyun.lee@mail.utoronto.ca
	Eric Deng	eric.deng@mail.utoronto.ca
	Terry Shi	terry.shi@mail.utoronto.ca
	Shafaaf Hossain	shafaaf.hossain@mail.utoronto.ca
Supervisor:	Alberto Leon Garcia & Ali Tizghadam	
Section #:	9	
Administrator:	Nick Burgwin	
Submission Date:	March 23rd 2017	

Group Final Report Attribution Table

This table should be filled out to accurately reflect who contributed to each section of the report and what they contributed. Provide a **column** for each student, a **row** for each major section of the report, and the appropriate codes (e.g. 'RD, MR') in each of the necessary **cells** in the table. You may expand the table, inserting rows as needed, but you should not require more than two pages. The original completed and signed form must be included in the hardcopies of the final report. Please make a copy of it for your own reference.

Section	Student Names			
	Eric Deng	Joohyun Lee	Shafaaf Hossain	Terry Shi
Executive Summary	RD	ET	ET	MR
Summary of Group Progress	RD	ET	ET	MR
Background and Motivation	ET	RD	ET	ET
Project Goal	ET	MR	RD	RD
Project Requirements	ET	MR	RD	RD
Technical Design	RD	RD, MR	RD	ET
Assessment of Final Design	RD	MR	ET	RD
Testing and Verification	ET	RD, MR	RD	ET
Conclusion	RD	ET	ET	ET
Appendix	RD	RD, MR	ET	RD
All	FP	CM	FP	FP

Abbreviation Codes:

Fill in abbreviations for roles for each of the required content elements. You do not have to fill in every cell. The "**All**" row refers to the complete report and should indicate who was responsible for the final compilation and final read through of the completed document.

RS – responsible for research of information
RD – wrote the first draft
MR – responsible for major revision
ET – edited for grammar, spelling, and expression
OR – other
"All" row abbreviations:
- FP – final read through of complete document for flow and consistency
- CM – responsible for compiling the elements into the complete document
- OR - other

If you put OR (other) in a cell please put it in as OR1, OR2, etc. Explain briefly below the role referred to:
OR1: enter brief description here
OR2: enter brief description here

Signatures

By signing below, you verify that you have read the attribution table and agree that it accurately reflects your contribution to this document.

Name	**Eric Deng**	**Signature**		**Date:**	**March 23rd 2017**
Name	Joohyun Lee	**Signature**		**Date:**	**March 23rd 2017**
Name	Shafaaf Hossain	**Signature**		**Date:**	**March 23rd 2017**
Name	Terry Shi	**Signature**		**Date:**	**March 23rd 2017**

Voluntary Document Release Consent Form[1]

To all ECE496 students:

To better help future students, we would like to provide examples that are drawn from excerpts of past student reports. The examples will be used to illustrate general communication principles as well as how the document guidelines can be applied to a variety of categories of design projects (e.g. electronics, computer, software, networking, research).

Any material chosen for the examples will be altered so that all names are removed. In addition, where possible, much of the technical details will also be removed so that the structure or presentation style are highlighted rather than the original technical content. These examples will be made available to students on the course website, and in general may be accessible by the public. The original reports will not be released but will be accessible only to the course instructors and administrative staff.

Participation is completely voluntary and students may refuse to participate or may withdraw their permission at any time. Reports will only be used with the signed consent of all team members. Participating will have no influence on the grading of your work and there is no penalty for not taking part.

If your group agrees to take part, please have all members sign the bottom of this form. The original completed and signed form should be included in the hardcopies of the final report.

Sincerely,
Khoman Phang
Phil Anderson
ECE496Y Course Coordinators

Consent Statement

We verify that we have read the above letter and are giving permission for the ECE496 course coordinator to use our reports as outlined above.

Team #: _____ Project Title: _____

Supervisor: _____ Administrator: _____

Name	_____	**Signature**	_____	**Date:**	_____
Name	_____	**Signature**	_____	**Date:**	_____
Name	_____	**Signature**	_____	**Date:**	_____
Name	_____	**Signature**	_____	**Date:**	_____

[1] This form will be detached from the hardcopy of the final report. Please make sure you have nothing printed on the back page.

Executive Summary (Author: E. Deng & T. Shi)

Connected Vehicles Smart Transportation (CVST) provides a platform for novel applications and innovations to improve the efficiency and safety of transportation systems. The system mines and stores a large variety of information related to traffic. However, given the sheer size of the data available, the current platform makes it challenging for users to rapidly draw conclusions from the data. This project's primary objective is to improve the CVST platform by providing a reporting web application to create visualizations of CVST in the form of charts and graphs.

Users cannot gather information at a glance from the current CVST system. Without a page to display any general information, a user currently must come to CVST with very specific requests to search for it. It is also difficult to analyze the historical data available, as the user would need to navigate to the location he is interested in and click on the nodes to generate past reports.

The design consists of four major parts: a data processing engine, a database for processed data, a server, and a web app interface that allows users to select and filter data and displays graphs. The data processing engine is built from several components. Data is retrieved from the CVST and stored within HDFS. From HDFS, Spark will process the data by aggregations and analysis and store the results into Elasticsearch. As clients request data, the server will query Elasticsearch to process the requests. Compared to the original design, a few components have been shuffled around and added. HDFS was used to replace the original plan to have Elasticsearch provide the raw CVST data store. Spark is still used to provide analytics but the pre computed results are now stored into Elasticsearch.

Functionality of the design was tested by stepping through the report generation process for a variety of different data types and time ranges. In addition, unit tests were performed on the module level to ensure functionality. The design was also tested against the functions, objectives and constraints originally laid out during the proposal. Test results showed that the design did not fully meet all of the system level tests layout for the design. In particular, the design was not able to create all of the different types of charts stated in the design proposal. It also did not support the data in CVST originating from Highway and Traffic cameras. The design incorporates all other data types within CVST and was successful in creating visuals for said data types.

In conclusion, the design was able to successfully visualize the data within CVST. However, since the design ultimately did not support the full range of data types within CVST and the full range of visuals initially proposed, it can only be considered a partial success.

Group Highlights and Individual Contributions

The major accomplishments of the group are listed in the table below. Some features proposed in the early stages of the project are partially complete, due to certain changes in the system design that will be elaborated in later parts of this document. Overall, the most critical and challenging components of the design, such as Spark Streaming, Elasticsearch component, and heatmap data visualization were completed and successfully operating.

Key Accomplishments	Status	Key Challenges	Key Decisions
Spark Streaming	Complete	- Learning to use Spark streaming and finding the best way to save data to HDFS - Deploying Cluster mode	- Integrating CVST's pub-sub system in addition to the web API - Use local mode on different machines to run jobs
Spark Analytics	Partially complete	- Learning to use Spark SQL - Could not figure out how to run analytics on real time data	- Instead of real time processing, we use intervals of one hour
Heatmaps	Complete	- Learning how to use Google Heat Maps APIs to plot the data. - How to optimize performance to reduce load time for large data sets.	- Since initially the heatmaps show present time data., to load faster, the server would just use the CVST apis to get the latest data instead of querying the Elasticsearch database.
Elasticsearch	Complete	- Making background scripts which keeps storing data for all data types to be up to date. - Scaling performance for large data queries.	- The scripts only store the fields from the CVST APIs which are needed for aggregations. This is done to make the storing faster and consume less space.
Scheduled report sent via email	Partially complete	- Learning how to make reports for different time intervals. These include daily, weekly monthly and yearly reports. - Email system still in progress as SMTP server needs to be setup on top of existing server.	- Elasticsearch has a date time query aggregation feature where it can take in the interval as either day, month, week or year. Using this ensured the reports were properly formatted.

Individual Contributions - Eric:

Hadoop/HDFS

- Installed Hadoop/HDFS onto the virtual machines
- Configured HDFS for cluster mode

Spark

- Installed Spark onto the virtual machines
- Configured Spark to run in cluster mode

CVST Publish-Subscribe

- Wrote scripts to subscribe to the existing CVST pubsub service
    - Constantly stream live data from CVST
    - Parse data into a json format that will be used in future Spark processing
    - Pipe to Spark Streaming module
    - Currently running for Bixi, TTC and road traffic
- Wrote Spark Streaming job to receive the stream from the script described above
    - Accept stream from the scripts mentioned above
    - Save each json object into HDFS line by line so that it is in a compatible format for querying data

Spark Job Analytics

- Wrote Spark jobs to run analytics on the data stored in HDFS
    - Take the json files stored in HDFS and transform then into a Spark object called Dataframes and extracting information from these using SQL queries
    - Supports the ongoing analysis of data, by running jobs hourly
        - Hourly jobs were chosen to maintain human readability in HDFS, while still providing reasonably “live” data.
    - Supports running the analytics on past data stored in HDFS
        - User can enter the time in the for which analytics should be run. The purpose of this is to populate Elasticsearch with data from the past
    - SQL queries were used to manipulate the data stored
- Format out to be compatible with Elasticsearch database
    - Worked with Shafaaf and Terry to integrate Elasticsearch and Spark

Individual Contributions - Shafaaf:

Setup Python Programming environment.

- Setup Python's package management system pip which is used to install and manage software packages written in Python.
- Used Python's virtualenv library to setup a virtual environment to easily port libraries and dependencies to other operating systems and environments. Then setup git repository in Gitlab and individual branches for version control.

Installed and setup ElasticSearch and other related utilities

- Downloaded ElasticSearch and configured nodes, clusters and ports to set up environment.
- Setup browser to make HTTP GET requests to verify data that will be stored.

Store data into Elasticsearch from CVST APIs

- Different APIs in the current CVST system bring in new data at different rates. The rate at which new data is coming in is calculated using Python scripts for all data types including bixi, weather, etc.
- Made the script to make requests to the existing CVST apis and store the data into ElasticSearch.
- Some APIs require authentication and so modified the scripts to also automate the login process and then fetch the data.

Designed new APIs for ElasticSearch

- To fetch the data from our ElasticSearch database, new APIs were made.
- Querying based on type of data, date, intervals, etc were implemented.
- Multiple aggregations APIs for different data types including weather, TTC, Bixi were made. These aggregations include averages, max and minimum values, sorted top 10 and trending data..
- APIs were designed to return in JSON format to be compliant with the usual REST API format.

Designing Tornado server

- Made Python server using Tornado to fetched from ElasticSearch and serve to the front end component.

Individual Contributions-Terry:

Elasticsearch:

- Created various mappings for the various data-types in CVST
- Wrote python scripts to insert data from CVST API into Elasticsearch for road closures and road traffic
- Tested performance of Elasticsearch database using various query settings
- Worked with Shafaaf to set up mapping deciding on file storage structure within Elasticsearch

Tornado Server Setup and Configuration:

- Configured and set up logging for the server
- Wrote various unit tests for the web server testing functionality of the handlers
- Setup file structure within server. Separating handlers and files better accessibility and readability.

Tornado Server Performance Testing:

- Performance tested server using various user inputs including:
    - Various date ranges for requests
    - Various data-types for requests
    - Various spatial aggregations for requests

Tornado Server Other:

- Designed querying API for road traffic as well as road closures
- Created report generation for Road traffic and Road Closures
- Setup asynchronous method calls for Elasticsearch queries
- Error checking for user inputs as well as various error messages for the server to provide more descriptive error messaging to users.

Front End:

- Created districts for spatial aggregation of data using the wards in GTA
- Generated heatmaps using said districts

Individual Contributions - Joohyun Lee:

Aggregation and Statistics Design

- Designed types of aggregation and statistics to visualize for timeline and report generator

Frontend Web Application Development

- Designed the web app using Bootstrap template to create simple, all-in-one layout
- Configured input fields and options for users to submit and send request
- Created interface design for front page timeline dashboard and report generator
- Applied Date-Time-Picker Bootstrap application in HTML, JavaScript to handle date/time inputs
- Customized Cascade-Style-Sheet to arrange layout and simplify the frontend aesthetics

Connecting Frontend to Backend Server

- Fetched and parsed user inputs submitted from the web app and sent them to the backend server in JavaScript
- Routed HTTP requests to appropriate handlers in the server
- Implemented Ajax calls to retrieve queried data from the server

Spark Data Aggregation

- Worked with Eric and Shafaaf to write Spark SQL aggregation queries to aggregate meaningful data from raw data obtained from CVST API

Data Visualization

- Used Google Charts API to visualize data in line charts, bar charts, and table charts
- Formatted JSON data retrieved from the server into data array used to graph Google Charts
- Worked with Shafaaf and Terry to visualize heatmap data using Google Map API

Acknowledgements (Author: E.Deng & J.Lee)

Foremost, we would like to acknowledge our supervisor, Dr. Ali Tizghadam, for meeting with us nearly every week and keeping us on track. We would also like to thank Dr. Hamzeh Khazaei for joining us in many of these meetings and giving us much needed technical advice. They have given us invaluable guidance and opportunities to acquire many new technical skills.

Moreover, we would like to thank our administrator, Mr. Nick Burgwin, for providing us with useful feedback for our design documents and presentations throughout this course. We would also like to extend our thanks to Bahareh Najafi, Morteza Moghaddassian, and Daiqing Li for helping us in other various technical aspects of project.

Table of Contents

1.0 Introduction (Author: J. Lee)

This document details the implementation of the History Timeline Report Engine as part of the Design project for the course ECE 496. The report begins with the gap identified with the current CVST portal system. It then details the functions the final project should perform as well as the design chosen to achieve said functions. It completes with the implementation and testing of the Reporting Engine as well as suggestions of improvement and future work that could be done to the design.

1.1 Background and Motivation

Connected Vehicles Smart Transportation (CVST) is a platform that serves real-time and historic data related to transportation in the Greater Toronto Area (GTA). These data types consist of the following: traffic incidents, road closure, information on TTC transportation, BIXI bikes, weather information, air quality, and traffic sensors. This data is gathered with a network of sensors provided by the university, government and industry and presented through the CVST portal[1].

In the current CVST portal, data regarding traffic sensors, road closures, and weather are presented on a map as a large number of nodes. At a glance, these nodes provide the current data taken from these sensors. For some sensors, a graph of the data for the past week is provided. However, for most data types (as listed in Appendix D), CVST does not provide the ability to generate reports on historical data. This severely hinders the usefulness of CVST for users looking to analyze trends in data such as traffic, road closures and Bixi. In addition, this functionality being missing reduces a user's ability identify correlations between data sets. These correlations include the impact of road closures on traffic, weather on traffic and closures on TTC.

Another shortcoming of CVST is that users cannot gather information at a glance. Upon entering the portal, a user is overwhelmed with a map of Toronto with a large quantity of nodes and several options on the side. Without a page to display any general information, a user currently must come to CVST with a very specific request and search for it on his own.

To improve the current CVST system, the project will begin by extending the report generating functionalities of the TTC data to all data types within CVST. Users will be able to sort and aggregate data by time and location to generate reports such as trending graphs on traffic and road closure frequencies in different areas. In addition, functionalities with regards to data

integration between different data types will be provided to utilize the scope of CVST in its entirety. The design team also plans to include a way to display some general data so that users have somewhere to start before they explore the vast amount of data in CVST. This data will be presented in visual formats akin to a dashboard so users are able to quickly draw conclusions on the current status of the transportation network in the GTA.

These improvements will hopefully improve CVST's purpose as a feedback mechanism for the transit system within the GTA. Users such as city planners or transit officials will be able to use CVST in their decision making process.

1.2 Project Goal

The goal of this project is to implement a comprehensive dashboard that allows for the detailed report generation and all-in-one analytics summary for the various data types in CVST. The report engine will provide users with a platform that can navigate the enormous collection of data hosted by CVST by using meaningful data aggregations and comparisons. Finally, the report system will present them in different visual formats including but not limited to bar chart, line chart, table chart, and heatmaps.

1.3 Project Requirements

The following project requirements are listed based on importance.

1.3.1 Functional Requirements:

- F1: The system shall allow users to filter and aggregate the data within CVST based on time and location.
- F2: The system shall allow users to select the output format of the data visualization as either line graphs, pie charts, bar graphs, or scatter plots.
- F3: The system shall access data within the CVST database.
- F4: The system shall be integrated into the existing CVST portal.

1.3.2 Constraints:

- C1: The system must work with all data types (specified in Appendix D) within the scope of CVST.

- C2: Any third party software must be open source to allow modification of source code for specific tasks in the future.
- C3: The data aggregation and analysis presented must be accurate to 95% of the original data. Any uncertainties and assumptions, particularly with predictive analysis, must be clearly indicated.

1.3.3 Objectives:

- O1: Analysis of data must be fast to ensure parallelism with real life events. Report generation should complete in 10s, otherwise indicators will be used to notify progress. [2]
- O2: The reporting engine should be scalable for changes with CVST. As CVST grows, new data types as well as the its hardware system will increase. Integration of new data types should be as simple as possible.
- O3: The CVST reporting engine should be fault tolerant. The engine should not contain any single point of failure and faults would be isolated by having a modular design and not allowing them to be propagated to other components.
- O4: The system should provide a scheduling mechanism for users to automatically receive generated reports at set intervals.

2.0 Technical Design (Author: J. Lee, T. Shi, E. Deng & S. Hossain)

2.1 System-Level Overview (Author: E.Deng & J. Lee)

The proposed design will consist of 4 modules. A Spark/Python engine will be used to fetch the data from CVST using either the existing CVST API or CVST's publish-subscribe system. In addition, the engine will perform analytics and computation in order to derive the information being presented to the users. Elasticsearch will be used to query and filter the data based on the inputs received from the user. The query results is then sent to the user and is visualized using Google Maps and Google Charts API.

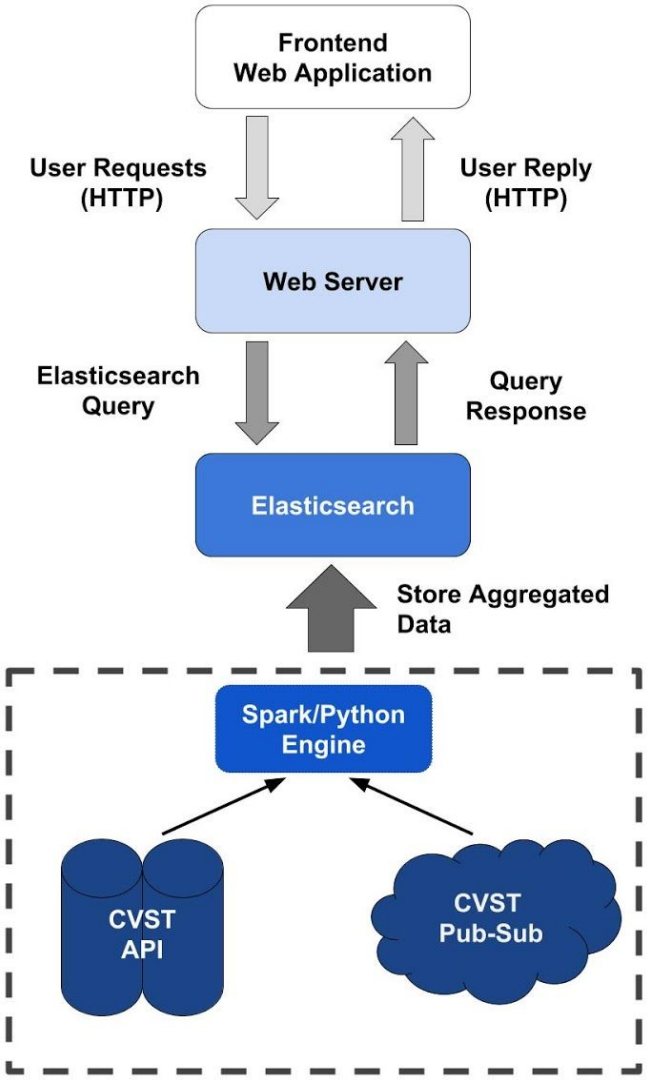


Figure 2.1.1 - System block diagram of the final design, consisting all 4 modules.

After meeting and discussing with our project advisor, Hamzeh, the interaction between Spark and Elasticsearch modules has been changed. As depicted in Figure 2.1.2 below, our original design assumes that a Spark job would be created to fetch data when a user requests data that is not currently residing in the Elasticsearch cluster. Our project advisor hinted that this is not an viable method to retrieve data, because it would basically require a map-reduce job each time. This would mean returning information to the user would be much too slow. In the altered final design, the Spark and Elasticsearch components are swapped.

The CVST Pub-Sub (Publish-Subscribe) system is now also integrated to provides a reliable and more frequently updated stream of data. This data is processed line by line using a Spark streaming job for each different data type. The streaming job saves everything into HDFS in a friendly json file for further processing. An example of this can be found in Appendix F. Next, every hour, the new data is processed by Spark analytic jobs. These jobs take the json files in HDFS and convert them using to Spark Dataframes which can be manipulated with standard SQL queries. This feature of Spark made it stand out from other computation engines. The output of this step is saved into Elasticseach and an example can also be found in Appendix F. This design takes advantage of Elasticsearch's extremely powerful queries to serve users quickly, while still using Spark's powerful computational abilities.

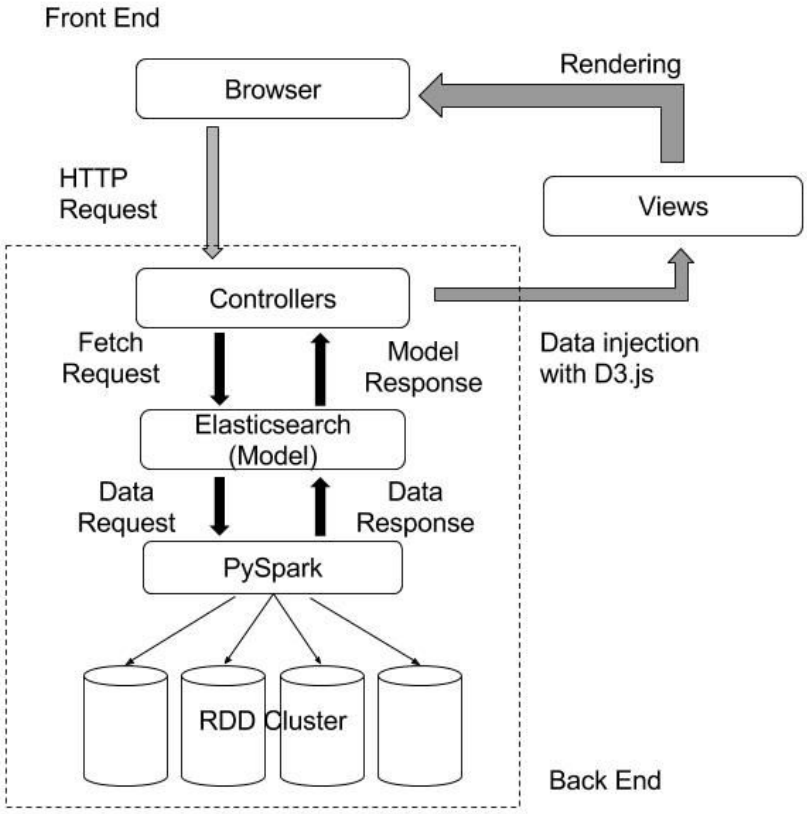


Figure 2.1.2 - Block diagram of the initial system design proposed in the Design Proposal. The Elasticsearch module will no longer send requests to the Spark module. Instead data is directly inserted into Elasticsearch from Spark.

2.2 Module-Level Description of Final Design (Author: J. Lee, T. Shi, E. Deng & S. Hossain)

The following sections describes each module in detail. In addition, design decisions which were made during the implementation of the modules are also included.

Web Frontend
Inputs: ● User Input Parameters for: ○ Data Type (Road Traffic, Road Incident, TTC, Weather, Bixi or Air Quality) ○ Aggregation Type (Top 10 data,) ○ Start time and end time for time aggregation ○ Scope of location for spatial aggregation
Outputs: ● Google Charts, Google Maps data visualization ● JSON formatted data
Function: The web application will be the interface between the timeline report engine and the user. It will display two components: historical timeline dashboard, which will be on the main front page, and the report generator. User inputs will allow users to filter the data they wish to generate reports for by data type, aggregation type, time, and location. Upon input submission by the user, the web application will return data analytics and visualization.
Implementation: The frontend web application was designed using a Bootstrap template. The Date/Time Picker application and Google Maps API was implemented to allow user to select time intervals and geographical space upon which the data is aggregated. When the user submits request for data, an Ajax call is made to the server to retrieve the requested data in JSON format. The data is also parsed into an array that is used by Google Charts and Google Maps API functions to graph the data in forms of line chart, bar chart, table chart, and heatmap. At a glance, the web application shows timeline heatmaps and aggregations for all data types in an all-in-one dashboard format. The front page showing the timeline looks as follows:

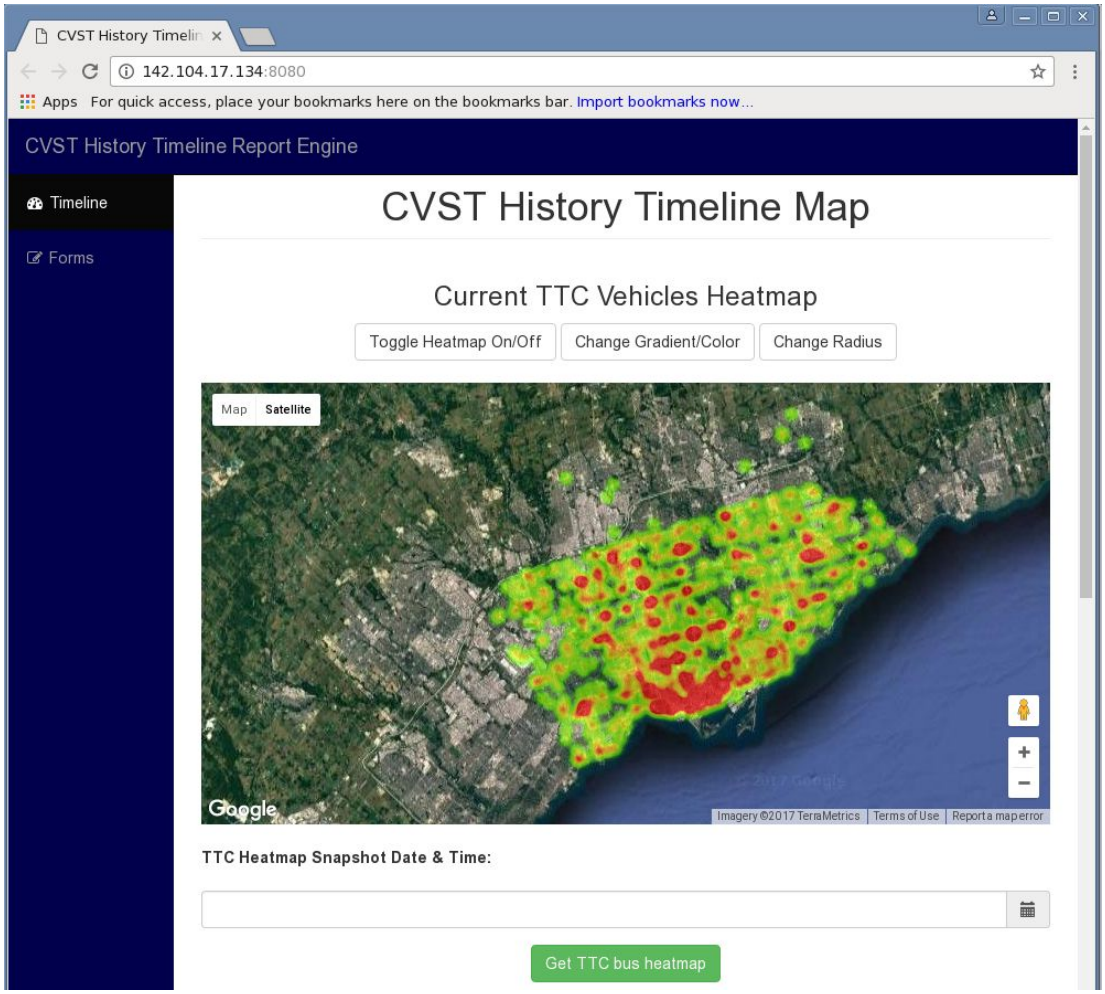


Figure 2.2.1 - Front page timeline of the web application, currently showing the latest heatmap of TTC vehicles across GTA at the time the screenshot was taken. Red indicates highly concentrated number of vehicles in the area, yellow indicates mediocre concentration, and green indicates low concentration.

More screenshots of the timeline and report generator are available in Appendix E.

Server
Inputs: ● HTTP requests
Outputs: ● HTTP responses ● JSON formatted data
Description/Functions: The web server, built using the tornado framework, will take in the user parameters in the HTTP requests. It will parse the request and format Elasticsearch queries based on the inputs. The web server will send the query to Elasticsearch and upon format the results back to the user.
Implementation: During the implementation of the web server, decisions were made in regards to the balancing of the work load between the front end, the web server as well as the Elasticsearch database. Tornado was chosen as the framework for the server as it asynchronous; the majority of the work load during a server request being the Elasticsearch query, the server needs to handle other requests during that time. Logging and testing for the web server were completed using Python's logging functionality and unittest. The input format for the user request was formatted to allow for the server to act as an API. Code snippets of the server are included in Appendix H.

Elasticsearch

Inputs:

- Json formatted data
- Query Strings

Outputs:

- Query Results(Json format)

Description/Functions:

Elasticsearch will serve as the database for the information used to generate the reports. Using a mixtures of queries and filters, the database will perform spatial as well as time aggregations on the data. Data inputs will be pre-parsed and computed so complex computations is not performed.

Implementation:

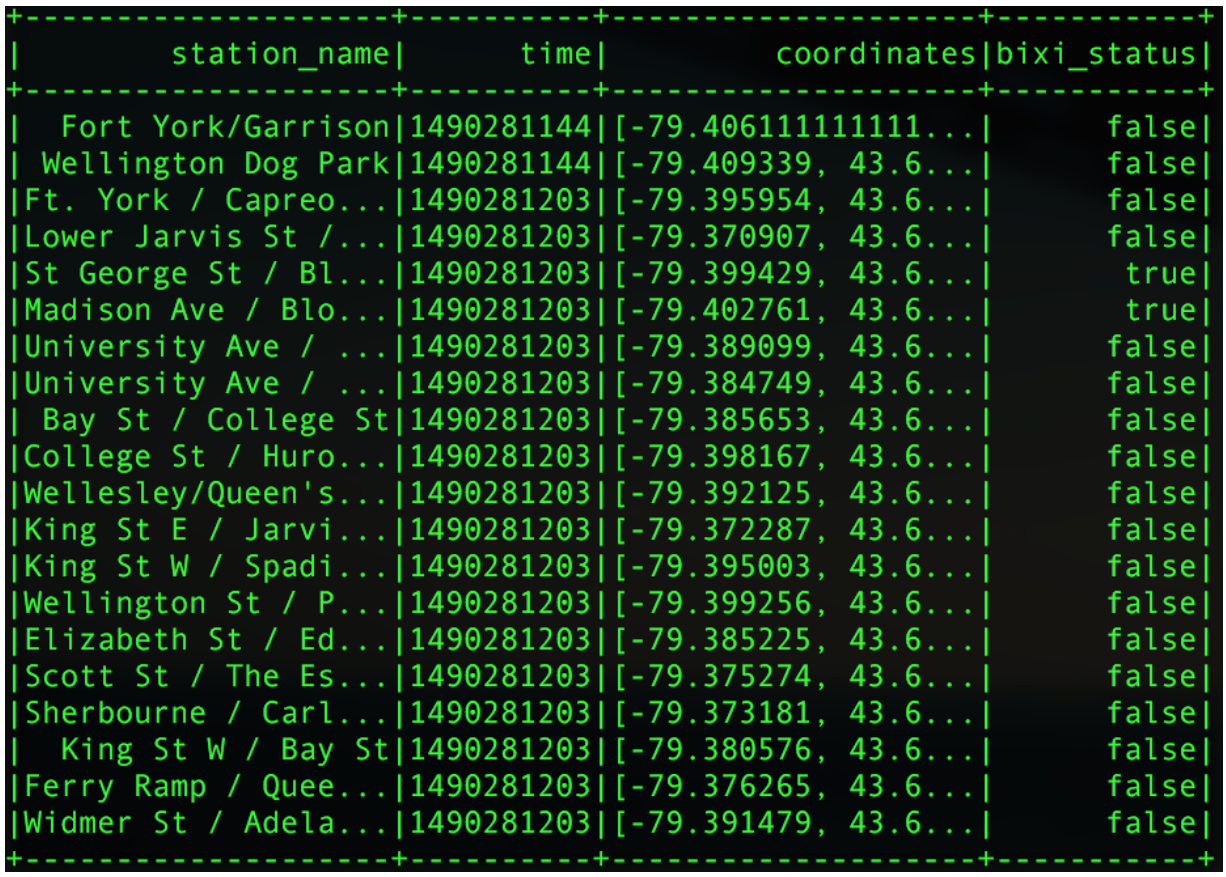
The nature of Elasticsearch required set up of nodes, replicas as well as mapping the doc-types during the creation of Elasticsearch index. These settings cannot be easily changed as changing mapping and the number of shards in an index requires re-indexing of all data within the index.

Decisions on the mapping for the data was made based on the aggregations needed. For example, coordinates are stored as geo_points for spatial aggregation, i.e aggregating for all data points within a region.

Configuration of nodes and shards relate directly to the design's ability to scale. Larger number of shards leads to better parallelism of queries and faster responses. [16] However, each shard consumes resources in memory, CPU and file handles. As each data type requires documents of different sizes, the data types are structured within Elasticsearch as different indexes and configuration for each index was done separately.

Refer to Appendix F for an overview of Elasticsearch usage.

Spark Engine
Inputs: ● CVST Data
Outputs: ● JSON formatted data
Description/Functions: The Spark/Python engine retrieves the data needed to perform the analytics and data visualization required for the design. A combination of Spark and Python are used as some data types require more complex computation, thus the use of Spark, whereas others only require simple parsing, using only python. The data is pulled from both CVST apis and pub-sub system. The engine is runned continuously to keep the data in the design up to date. It formats the data and inserts it into Elasticsearch.
Implementation: Spark is an engine for distributed data processing. It allows for adding more nodes to the cluster in order to process more data. It also has function These properties were why Spark was chosen to be used for our data processing. To start, the cluster we are using will consist of two machines, just in order to ensure Spark will work in cluster mode. Other reasons Spark was chosen as were because of its speed and its compatibility with Elasticsearch. It also provides powerful functions that allow for relatively easy-to-implement aggregations on the data in the form of SQL queries. An example of the one complete job:



station_name	time	coordinates	bixi_status
Fort York/Garrison	1490281144	[-79.406111111111...	false
Wellington Dog Park	1490281144	[-79.409339, 43.6...	false
Ft. York / Capreo...	1490281203	[-79.395954, 43.6...	false
Lower Jarvis St /...	1490281203	[-79.370907, 43.6...	false
St George St / Bl...	1490281203	[-79.399429, 43.6...	true
Madison Ave / Blo...	1490281203	[-79.402761, 43.6...	true
University Ave / ...	1490281203	[-79.389099, 43.6...	false
University Ave / ...	1490281203	[-79.384749, 43.6...	false
Bay St / College St	1490281203	[-79.385653, 43.6...	false
College St / Huro...	1490281203	[-79.398167, 43.6...	false
Wellesley/Queen's...	1490281203	[-79.392125, 43.6...	false
King St E / Jarvi...	1490281203	[-79.372287, 43.6...	false
King St W / Spadi...	1490281203	[-79.395003, 43.6...	false
Wellington St / P...	1490281203	[-79.399256, 43.6...	false
Elizabeth St / Ed...	1490281203	[-79.385225, 43.6...	false
Scott St / The Es...	1490281203	[-79.375274, 43.6...	false
Sherbourne / Carl...	1490281203	[-79.373181, 43.6...	false
King St W / Bay St	1490281203	[-79.380576, 43.6...	false
Ferry Ramp / Quee...	1490281203	[-79.376265, 43.6...	false
Widmer St / Adela...	1490281203	[-79.391479, 43.6...	false

Figure 2.2.2 This is an example of a snippet of Bixi data analytics, specifically the downtime which returns whether or not there are bikes available at a given station. Rather than printed to console, this is normal saved to Elasticsearch.

Additional screenshots are available in Appendix G.

2.3 Assessment of Final Design (Author: T. Shi, E. Deng, J. Lee & S.Hossain)

The following sections describe the successes and failures of each module in detail. Reflections on the impact of the design decisions are also included as well as an overall assessment of the modules.

2.3.1 Frontend

*Successes***:**

The frontend meets all the functional tests required for the module. The user inputs are formatted into the correct HTTP requests and sent to the server. The response is retrieved from the server and outputted to the user. All data results are retrieved and displayed on the frontend in less than a few seconds, even when the user requests several months of aggregated data. The end-to-end response time of the system design demonstrated to be outstanding.

Failures/Difficulties:

To eliminate extra complexity in the design and timing constraints, the visualization of the data was done using Google Charts and Google Maps APIs. The final output formats were in line charts, bar charts, table charts, and heatmaps. While the frontend functions properly, implementation of the frontend concluded with a UX design that we felt was unsatisfactory for the final product that we had envisioned.

2.3.2 Web Server

Successes:

The web server meets the functional requirements of the module. It correctly parses the user inputs and formats them into a query string. Using the Tornado framework made the initial setup of the server very simple. Including other libraries for tasks, such as json formatting and parsing was also easier in Python as compared to other programming languages.

Failures/Difficulties:

There were no major complications that arose during the implementation of the web server.

2.3.3 Elasticsearch

Successes:

Choosing Elasticsearch as the database for queries made various aggregations much simpler. In particular, the date data type and geo-point (geographical map point) data type within Elasticsearch allowed for easier aggregations for time and location. The non-relational aspect of Elasticsearch meant the queries were much faster than traditional relational databases and due to the nature of the project, where data only share timestamps and locations, the drawbacks of

non-relational databases were a non-issue. This module functioned correctly and it meet all the performance and validation tests which were performed.

Failures/Difficulties:
Elasticsearch did not allow any fallback mechanism when mistakes were made in initial configuration of indexes. In particular, changing the mapping for data already stored inside the database was not possible. This required either completely reloading the data or re-indexing which proved difficult and time consuming.

2.3.4 Spark/Python Engine

Successes
The Spark analytic and streaming modules have been deployed smoothly and provides Elasticsearch with the data it requires to serve users. Performance for Spark was vastly superior to the equivalent python script due to the added parallelism.

Failures:
However, Spark is not currently working in its cluster mode. As of now, the Spark component is running on one node, meaning on only one machine. Because the Spark programs are currently working, if the configuration bugs can be worked out, cluster mode can be deployed immediately.

3.0 Testing & Verification (Author: J. Lee, T. Shi, E. Deng & S.Hossain)

This section describes the results of testing and verification of the final design by comparing with the target requirements specified in section 1.3, Project Requirements.

Requirement: F1: Spatial and Time aggregation & filtering of data
Target Specification: Time Aggregation: Start and end dates for data as well as the interval for each data point. Spatial Aggregation: Aggregating either with a specific region inputted by the user or a pre constructed area, using municipal Wards.
Final Result: Success: ● Filtering by start and end dates ● Aggregating based on a given interval (day, hour, week) ● Filtering based on sensor names ● Aggregating based on sensor location. Images are located in Appendix E for frontend output, and also Appendix F for queried output from Elasticsearch.
Compliance (Pass/Fail): Pass.
Comments & Documentation: The design is able to successfully perform spatial and time aggregations. The user is able to specify the start and end dates, and the period/interval to aggregate for; i.e interval of weeks with the start and end dates being last month. The user is able to aggregate by space by specifying the sensors to include/exclude by name and providing polygonal regions to aggregate over.

Requirement: F2: Output Format Selection
Target Specification: Allow users to request data visualizations in forms of line graphs, pie charts, bar graphs, or scatter plots.
Final Result: Currently returns results either in the form of heatmaps, line charts, bar charts, or table charts. It does not support pie graphs or scatter plots as proposed.

Compliance (Pass/Fail): Conditional Pass
Comments & Documentation: See Appendix E for the different types of data visualizations that generated on the front-end web application.

Requirement: F3: Connect to CVST
Target Specification: The system connects to CVST to retrieve the data
Final Result: Connects to CVST's web API and pub-sub (publish-subscribe) system in order to retrieve the data for the design. The web application is able to display user requested data derived from this pub-sub system, as shown in Appendix E.
Compliance (Pass/Fail): Pass
Comments & Documentation:

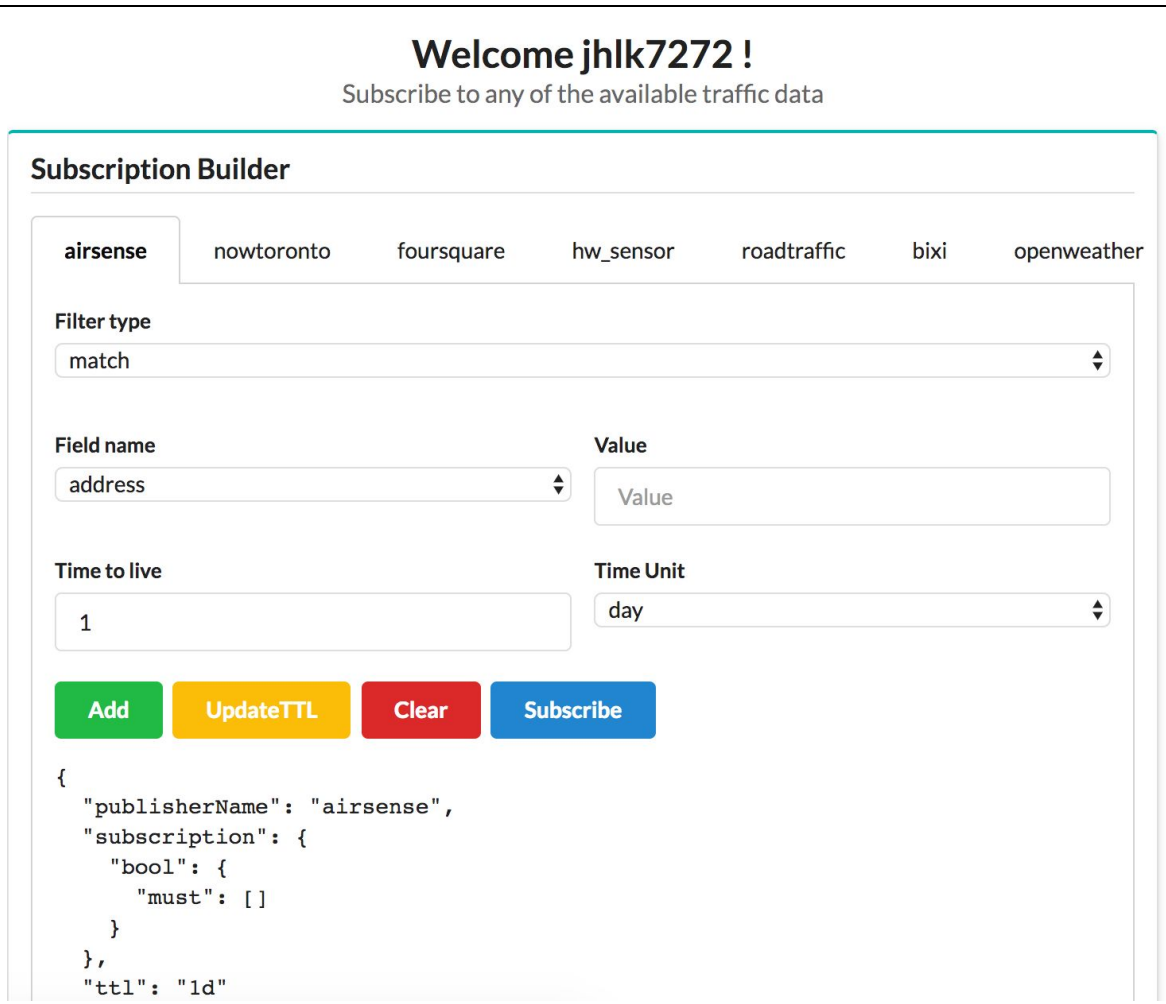


Figure 3.1 - This subscription builder application generates the message to send to the CVST pub-sub service. The message tells the pub-sub service which raw data you would like to receive. This raw data is passed to Spark, which then proceeds to aggregate the data.

Requirement: F4: Integration into CVST Portal
Target Specification: The timeline as well as report generator should be integrated into the existing CVST portal and accessible through the portal website.
Final Result: The design is currently hosted on a virtual machine, accessible through this IP address: 142.104.17.134:8080 . It is not accessible directly through the CVST portal yet.
Compliance (Pass/Fail):

Fail
Comments & Documentation: Upon receiving approval from our supervisor, the web application will be linked in the main page of CVST portal to provide public access.

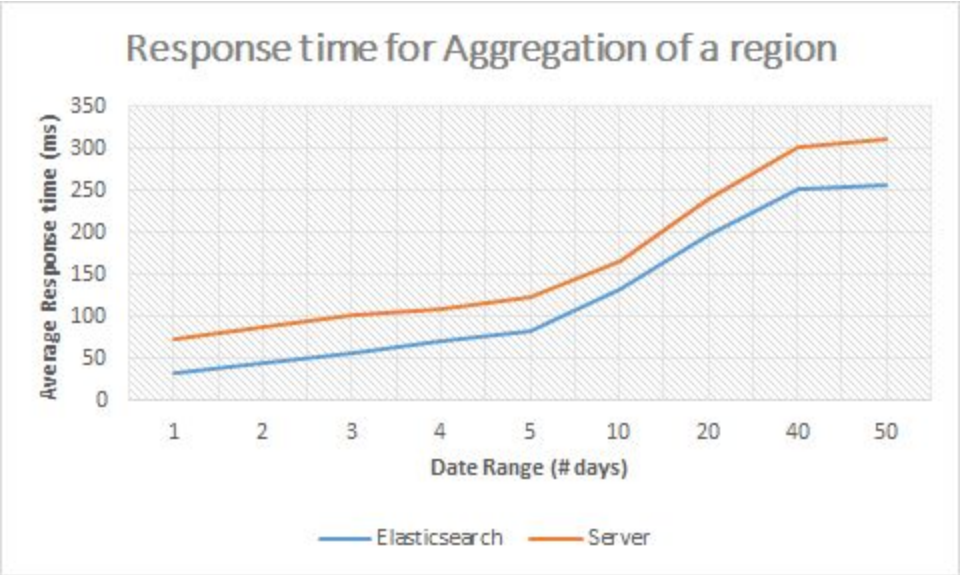
Requirement: C1: Work with all data types within CVST
Target Specification: Work with all Data types within CVST: ● Road Traffic ● Highway Cameras ● Drone Cameras ● Road Incidents/Closures ● Bixi ● Air Sensors ● Weather
Final Result: The system works with all of the following data types: ● Road Traffic ● Road Incidents/Closures ● Bixi ● Air Sensors ● Weather
Compliance (Pass/Fail): Conditional Pass
Comments & Documentation: The final design incorporates the use of data from all data types except for data obtained from highway/drone cameras. It was decided in early discussions with our supervisor that the complexity of doing image processing was far beyond the scope of this project. This data type was left in as a potential/optional objective in case there was extra time between design completion date and the final project deadline. Unfortunately, these optional objectives were not implemented.

Requirement: C2: Open source third party software

Target Specification: Any third party software must be open source
Final Result: All components used to complete the design are open source. This includes Apache Spark, Elasticsearch, Tornado, jQuery, Google Charts, and Google Maps API.
Compliance (Pass/Fail): Pass

Requirement: C3: Accuracy
Target Specification: Accuracy of the design must be within 95% of the original data. Any inaccuracy or assumptions must be clearly presented to the user.
Final Result: To test accuracy, the aggregations were coded in Python scripts to be done manually in order to do cross-comparison with the queried results from Elasticsearch. The results from these scripts matched the results from Elasticsearch, which ensures the level of accuracy required.
Compliance (Pass/Fail): Pass
Comments & Documentation: Ensuring accuracy within Elasticsearch brought about some particular challenges. Due to the nature of Elasticsearch and its structure, aggregations and filtering often produce some form of inaccuracy to optimize performance. [17] To meet our accuracy standards, the Elasticsearch queries had to enforce doc_count_error to be 0. This affected the performance of our queries; however the design was still able to satisfy its performance metrics.

Requirement: O1: Performance Timing
Target Specification: Report generation should be completed within end-to-end transaction time of 10 seconds.
Final Result:



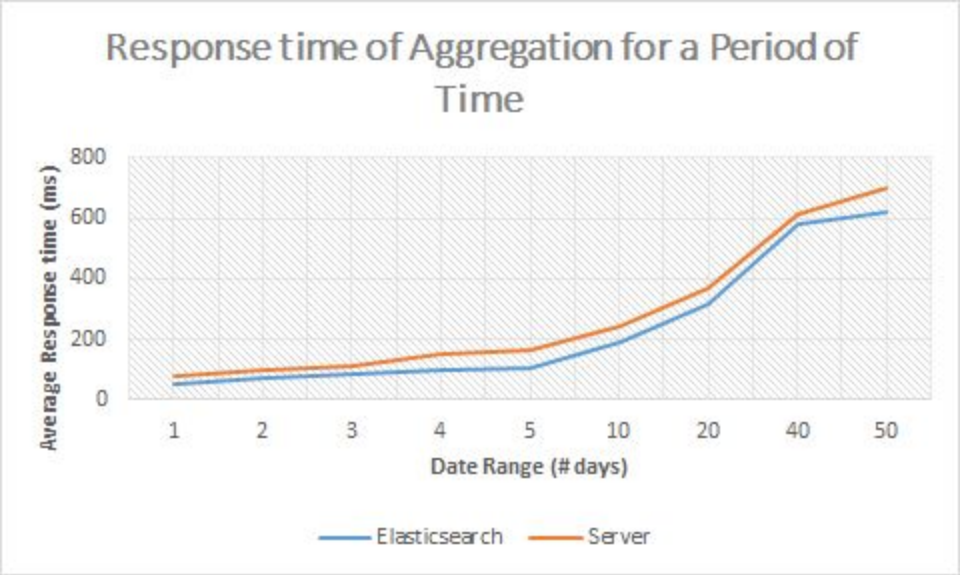


Figure 3.2 - Report generation timing slightly varies between data types as well as the time scale. However, all functionalities take significantly less than 10 seconds to complete. The maximum recorded response time is under 0.8 seconds, as shown above.

Compliance (Pass/Fail):
Pass.

Comments & Documentation:
Various input parameters were attributed for performance testing. These parameters include different sizes of geographical region, different date/time intervals, different numbers of stations and locations to include and/or exclude, as well as the different data types to query.

Requirement: O2: Scalability
Target Specification: The design should be scalable into the future. It should accommodate new data being included for the existing data types as well as be able to integrate new data types.
Final Result: The configuration of the current system allows uninhibited insertion of new data types into the Elasticsearch database. Current set up accommodates appending latest data to Elasticsearch via pub-sub system and Spark Streaming for the existing data types.
Compliance (Pass/Fail): Pass
Comments & Documentation: As described in Appendix E, the timeline syncs with the latest data derived from CVST API's to conduct real-time data analytics.

Requirement (# & Title): O3: Fault Tolerance
Target Specification: The reporting engine should be fault tolerant without single points of failure.
Final Result: The Elasticsearch database currently runs on a single VM (virtual machine) along with the server. Failure of the VM would result in users being unable to access the web application and/or acquire requested data.
Compliance (Pass/Fail): Fail
Comments & Documentation: Reflecting on the design, the single point of failure for the Elasticsearch and backend server can be eliminated through use of cluster mode for Elasticsearch and running the server on multiple machines. Implementation of this change was not done due to time constraints.

Requirement (# & Title): O4: Scheduling Mechanism
Target Specification: Scheduling mechanism for users to receive periodic information based on preset inputs. Example: user wishes to receive the time of day where Dundas and University has the most

traffic for the past month.
Final Result: There is no report scheduling feature that allows users to subscribe to the report generator via email subscription.
Compliance (Pass/Fail): Fail
Comments & Documentation: During the late developments of the backend server, simple timed events and email subscription functionality were implemented and tested. However, the added complexity of keeping track of timed events for cancelled subscription could not be complete within the time constraint.

4.0 Summary & Conclusions (Author: T. Shi)

The purpose of our design was to present the large amounts of data within the CVST portal in a form of a visual summary for users. The design hoped to achieve this through the use of spatial and time aggregations for the various data-types as well as providing a detailed reporting engine to help analyze trends and changes to the CVST system.

Changes were made to the initial design with respect to the functionalities of the Spark and Elasticsearch components and its interactions with the other modules. Spark was changed to generate the data used in our design from existing CVST API's, while Elasticsearch was designated as the query database of our design.

Testing of the individual modules were conducted using unit tests for functionality. Performance tests were conducted for the web server, Elasticsearch and frontend modules. As demonstrated by the results of the tests, the design is able to successfully achieve most of the goals and requirements for the project. The design is able to retrieve data through CVST, perform spatial and time aggregation, querying based on user inputs, and finally visualize the results of the queried aggregations into a visual format under the timing constraints.

Design and implementation of the project exposed the challenges that are present in dealing with large amounts of data in real time. Due to this challenge, our design became complex and difficult to implement, with most of the project time spent in getting the full frontend and backend pipelining to function properly. Most of the data the design used to generate visuals for the user had to be precomputed. This forced us to limit the complexity of the queries users are able to perform in order to meet timing constraints.

Future work include adding support for more graphs for the data types that the design currently supports; inclusion of pie graphs and scatter plots. Improvements can be made to the timeline UI in terms of visuals as well as UX for the user inputs. Additionally, future work can incorporate adding support for additional data types when they are added to CVST.

References

[1] "Connected Vehicles And Smart Transportation", [Online] Available: http://portal.cvst.ca/ . N.p., 2016. Web. 8 Sept. 2016.

[2] Nielsen, Jakob, "Response Time: The 3 Important Limits", [Online] Available: https://www.nngroup.com/articles/response-times-3-important-limits/ [Accessed: 8 Sept. 2016].

[3] ISO/IEC 9126-1:2001, Software Engineering -- Product quality -- Part1: Quality model

[4] ISO/IEC 25010:2011, C.2 Software Quality Measurement

[5] Kirahowski, J, "The Use of Questionnair Methods for Usability Assessment", [Online] Available: http://sumi.uxp.ie/about/sumipapp.html, [Accessed: 15 Oct. 2016]

[6] Chavi Gupta, "Advantages Of Elastic Search", [Online] Available: http://www.3pillarglobal.com/insights/advantages-of-elastic-search [Accessed: 25 October 2016]

[7] "Solving Concurrency Issues", [Online] Available: https://www.elastic.co/guide/en/elasticsearch/guide/current/concurrency-solutions.html [Accessed: 25 October 2016]

[8] John Mack, "Five Advantages & Disadvantages Of MySQL", [Online] Available: https://www.datarealm.com/blog/five-advantages-disadvantages-of-mysql/ [Accessed: 25 October 2016]

[9] "Flare Data Visualization For The Web", [Online] Available: http://flare.prefuse.org/tutorial [Accessed: 25 October 2016]

[10] Miller, Michael, "Are You Ready for Computing in the Cloud," in Cloud Computing: Web-Based Applications that change the way you work and collaborate online. Que Publishing, 2008, pp. 24-25.

[11] Armbrust, Michael, et al, "Scaling Spark in the Real World: Performance and Usability" [Online], Available: https://cs.stanford.edu/~matei/papers/2015/vldb_spark.pdf [Accessed: 15 Oct. 2016].

[12] Bostock, Mike. "D3.Js - Data-Driven Documents". D3js.org, [Online] Available: https://d3js.org/. [Accessed: 21 Sept. 2016].

[13] Bouwkamp, Katie, “What Salary Can you Expect As An Entry Level Software Developer?”, [Online] Available: http://www.codingdojo.com/blog/entry-level-software-developer-salary/ [Accessed: 17 Oct. 2016]

[14] ECE496, “Project Proposal Guidelines”, [Online] Available: https://internal.ece.toronto.edu/ece496.1516/pages/guides/ProjectProposalGuidelines.pdf [Accessed: 22 Sept 2016]

[15] “Amazon RDS Pricing”, [Online] Available: https://aws.amazon.com/rds/pricing/ [Accessed: 26 Oct 2016]

[16] “Optimizing Elastic Search” . [Online] Available: https://qbox.io/blog/optimizing-elasticsearch-how-many-shards-per-index [Accessed: March 15th 2017]

[17] “Term Aggregation“. [Online] Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket-terms-aggregation.html#_document_counts_are_approximate [Accessed: March 15th 2017]

Appendix A: Gantt Chart History

Below are the three versions of Gantt Chart: first figure is the final Gantt Chart reflecting the complete schedule that was carried out for this project, second figure is the Gantt Chart proposed in the Progress Report, reflecting the schedule adjustments from milestone setback, and the last figure is the first version of Gantt Chart initially proposed in the Project Proposal.

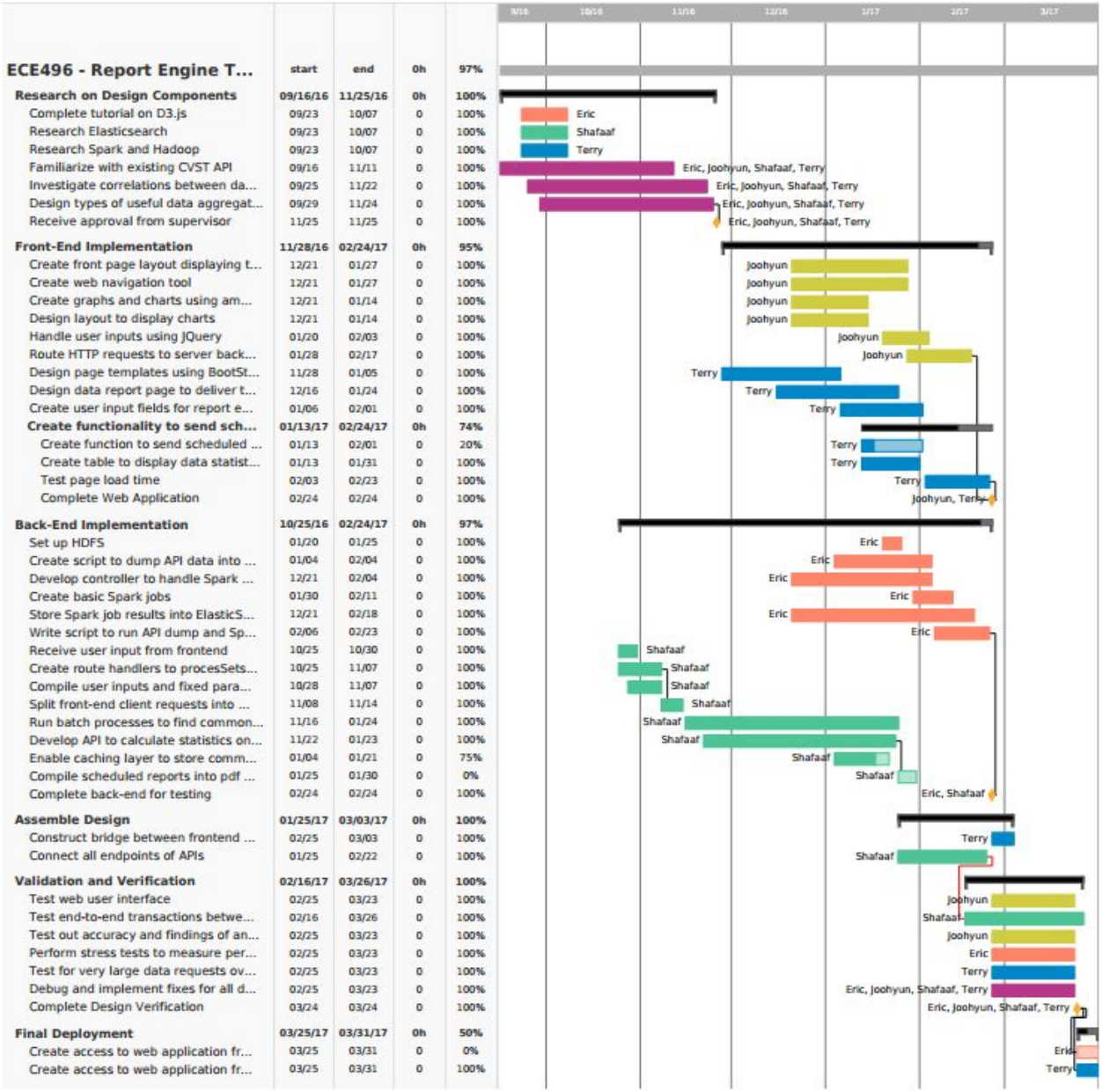


Figure A.1 - Final version of Gantt Chart that displays the final schedule of work plan completed by far.

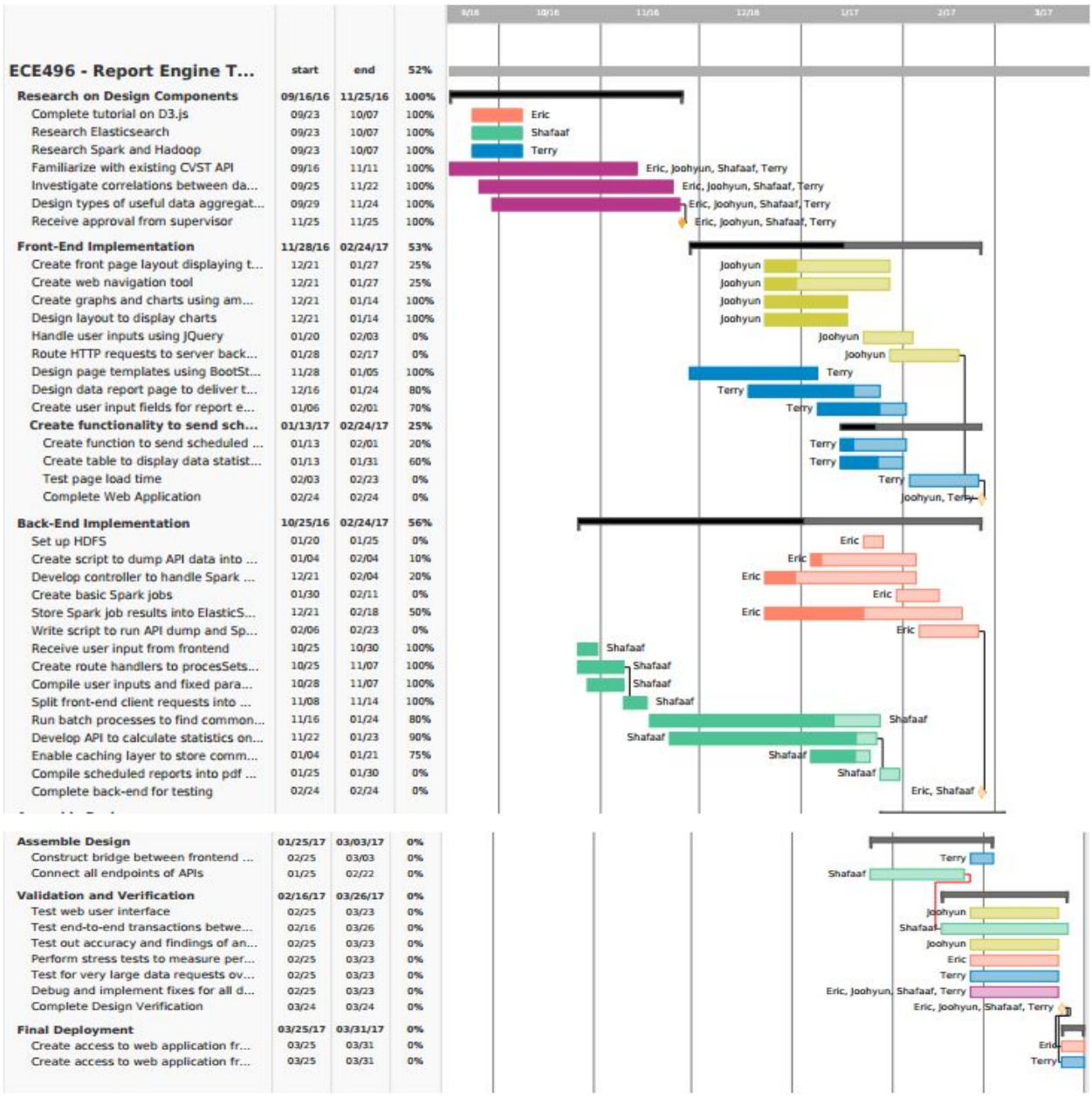


Figure A.2 - Second version of Gantt Chart that displays the adjusted work plan schedule proposed in the Progress Report.

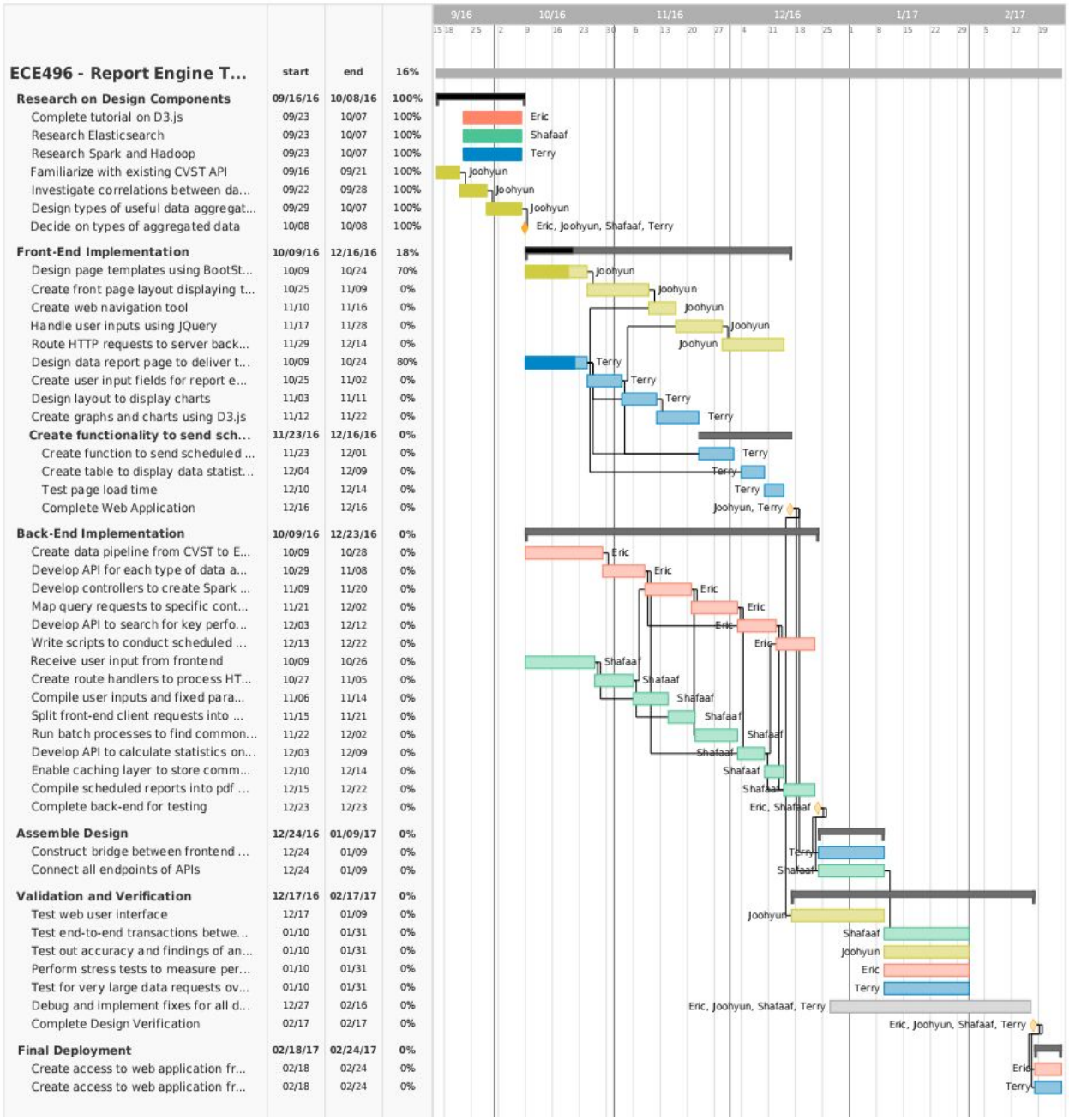


Figure A.3 - First version of Gantt Chart that shows original work plan schedule from the Project Proposal document.

Appendix B: Financial Summary

Below are the two versions of financial summary: first table is the complete financial summary reflecting the final budget for this project and second table is the initial version of the financial summary, where all costs were estimates based on research and guidelines as indicated.

Table B.1 - Final Project Budget

Student Labor				
Item	**Quantity (hrs)**	**Cost/Unit**	**Total Cost**	
Student 1	350	$26.44	$9,254.00	
Student 2	350	$26.44	$9,254.00	
Student 3	350	$26.44	$9,254.00	
Student 4	350	$26.44	$9,254.00	
Total Student Labor			**$37,016.00**	
Cloud Hosting Cost				
Item	**Priority**	**Quantity**	**Cost/Unit**	**Total Cost**
VMs	1	3	$0.00	$0.00
Total Hosting Cost				**$0.00**
Total Cost of Project				**$37,016.00**
Total Funding				**$0**

The project budget is complete with the following baseline:

- Hourly wages is derived from the average salary of entry level software developers [13]
- The ECE496 Project Proposal Guidelines assumed 500hrs of work will be put in per students for the entire Capstone project [14]. 70% of that is assumed to be time spent working on the project, as well as meetings whereas 30% of it accounts for the administration tasks, such as document writing.
- All developer tools used are open-source, hence free of cost.

- Virtual machine for web hosting and storing data file system is provided by the supervisor, as in using the existing inventory for the ongoing CVST project.

Table B.2 - Proposed Project Budget

The existing infrastructure that is currently being used to the the CVST system will be used to implement our system. However, should that not be possible, the cost for hosting the website and application will require additional funding laid out in the budget table

Student Labor				
Item	**Quantity (hrs)**	**Cost/Unit**	**Total Cost**	
Student 1	350	$26.44	$9,254.00	
Student 2	350	$26.44	$9,254.00	
Student 3	350	$26.44	$9,254.00	
Student 4	350	$26.44	$9,254.00	
Total Student Labor			**$37,016.00**	
Cloud Hosting Cost				
Item	**Priority**	**Quantity**	**Cost/Unit**	**Total Cost**
DB	1	1 Year	$1842.00	$1842.00
Computations	1	1 Year (2 hrs/day)	$0.266/Hour	$194.18
Developer Tools	2	~	$150.00	$150.00
Total Hosting Cost				**$2186.18**
Total Cost of Project				**$39,202.18**
Total Funding				**$500**

The project budget is made with the following assumptions:

- Hourly wages is derived from the average salary of entry level software developers [13]
- The ECE496 Project Proposal Guidelines assumed 500hrs of work will be put in per students for the entire Capstone project.[14] 70% of that is assumed to be time spent working on the project, as well as meetings whereas 30% of it accounts for the administration tasks, such as document writing.
- Hosting Costs are retrieved from AWS (Amazon Web Services). The database selected was db.m3.large which should accommodate a project of this size.
- Total funding currently available is from $100 per student as well as $100 from the supervisor. [15]

Appendix C: Validation and Acceptance Tests

The following is the original ‘Validation and Acceptance Tests’ section described in the Project Proposal:

Testing and verification will be conducted to check if the design meets all of the functional requirements and constraints. In addition, the effectiveness of the design will be measured using the following criteria.

- ➢ Functionality:
  - Use a large variety of input options and requests sizes to simulate real user usage.
  - Test sorting and aggregation functionality against precomputed results to ensure numerical correctness.
  - Test numerical values inputted into the graphing module will output the expected graphs.
  - Test to ensure scheduled reports will be delivered on time and correctly.

- ➢ Performance:
  - Measure the time it takes to retrieve data in the back-end over timespan of years, months and weeks with all data types.
  - Measure the time it takes to generate graphs with all types of data aggregation. Results should be no more than three seconds as not be the bottleneck in the system.
  - Record end-to-end transaction time from when users requesting data by submitting inputs, to when output of data visualizations are generated for users to view. The total transaction time should be under 10 seconds [2].

- ➢ Usability:
  - Recruit users of various technical backgrounds to make various report requests.
  - Survey user experience from test users.
  - Evaluated with metrics, such as completion rate, number of errors and task times, covered in ISO/IEC 9126 [3].
  - Assess the quality of the system using Software Usability Measurement Inventory. [5]

The following is a list of more specific tests which were conducted during the testing and validation phase of our design:

#	Test	Description
Frontend		
1	Accepting User Inputs	
2	Formatting User Inputs	
3	Data Visualization	
Web Server		
1	Parse User Inputs	Correctly read user inputs from the HTTP request. Catch invalid inputs and HTTP requests and respond with appropriate messages.
2	Format Elasticsearch Queries	Given user inputs, does the web server format the correct query strings for Elasticsearch.
3	Query Output formatting	
4	Performance Testing Query Strings	Various query strings were used to test the performance of Elasticsearch. These strings varied in terms of time span, aggregations required as well as data types.
Elasticsearch		
1	Geo Location Query	Tests that aggregation by geo_polygon functions correctly
2	Date Histogram Query	Tests that aggregating into intervals functions correctly
3	Filtering Query	Data is filtered correctly based on inputs
4	General Query	Mish-mash of different possible user inputs
Spark		
1	Retrieve CVST data from pub-sub	Compare data retrieved from the pub-sub system with data retrieved from web API
2	Store json in HDFS	Confirm validity of json using json verification program
3	Spark job output	Confirm by verify small data sets by hand

Appendix D: General Overview of CVST Portal

The CVST portal is the web application platform for displaying the data contained within CVST.

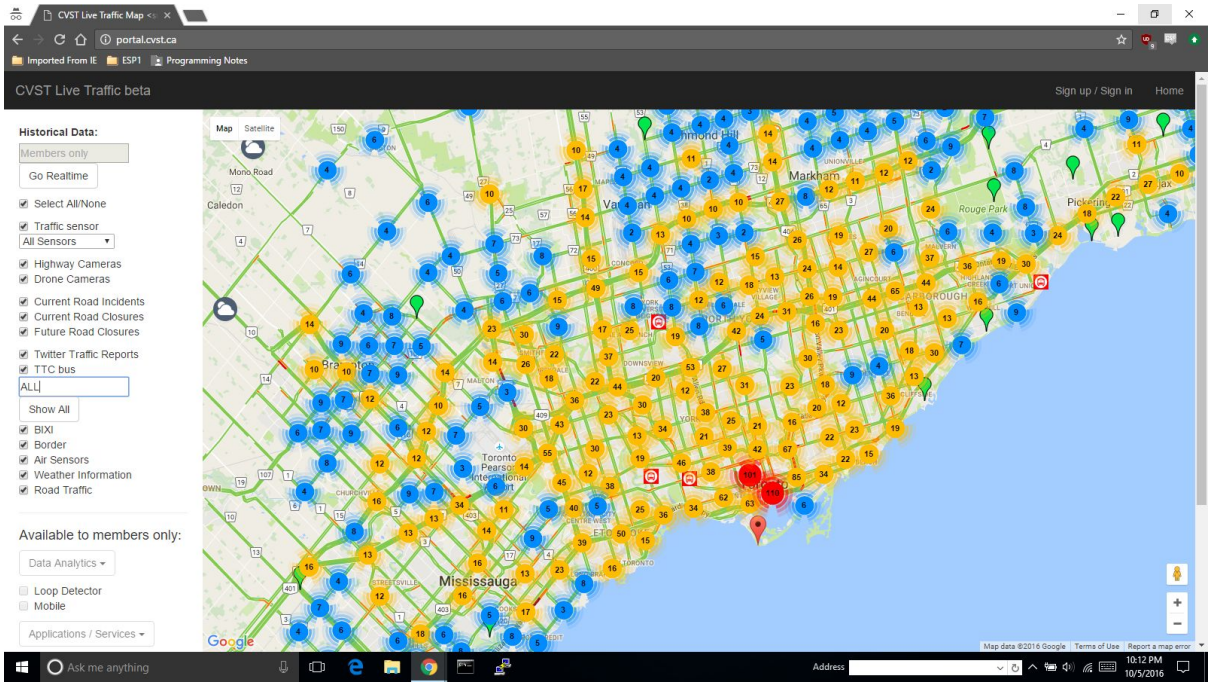


Figure D.1 - This is a screenshot of the CVST portal. It displays its data on top of a map of the GTA. The blue, red and yellow circles indicate current road traffic. The green balloons show traffic sensor data for major roadways. Clouds indicate weather sensors and red bus signs indicate current bus locations.

Figure D.2 - This is the information that is brought up when one of the traffic sensor nodes are selected. It displays information about the sensor as well as the data collected within a set period.

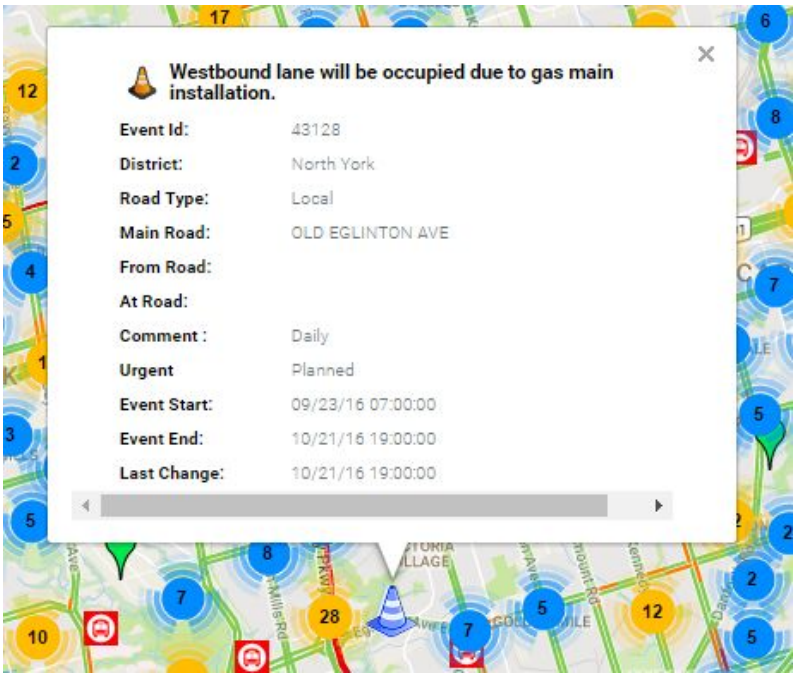


Figure D.3 - This is the information displayed for road closures.

Appendix E: Overview of History Timeline Report Engine Web Application

The frontend web application consists of two pages: ‘Timeline’, which is the main front page containing a dashboard of heat maps and visualizations of aggregated data, and ‘Forms’, which contains the report generator where the user specifies data type, location, and time intervals to obtain specific data analytics. To access the web application, go to this IP address: 142.104.17.134:8080 . The images of frontend designs are listed below.

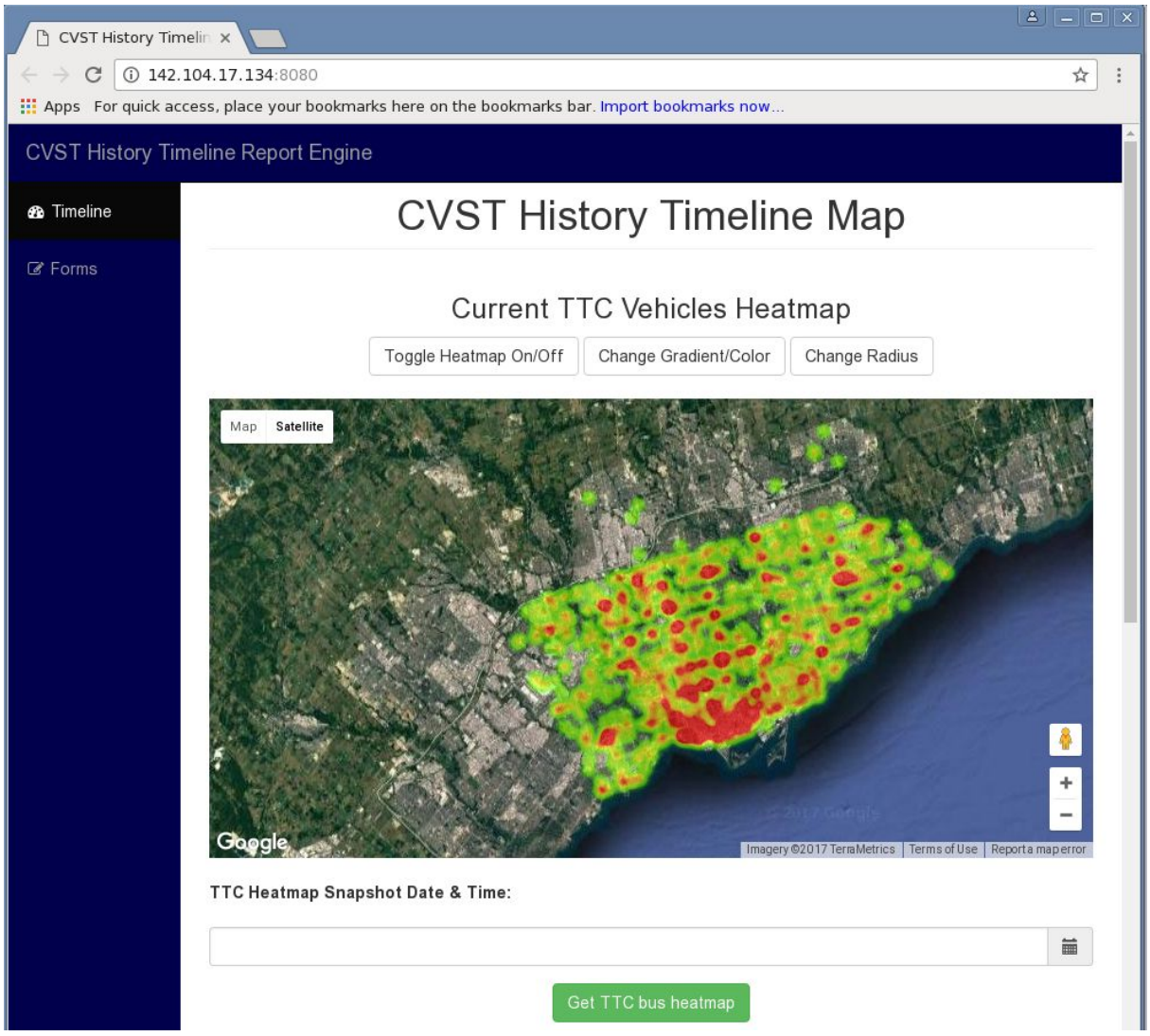


Figure E.1 - Front page timeline of the web application, currently showing the latest heatmap of TTC vehicles across GTA at the time the screenshot was taken. Red indicates highly concentrated number of vehicles in the area, yellow indicates mediocre concentration, and green indicates low concentration.

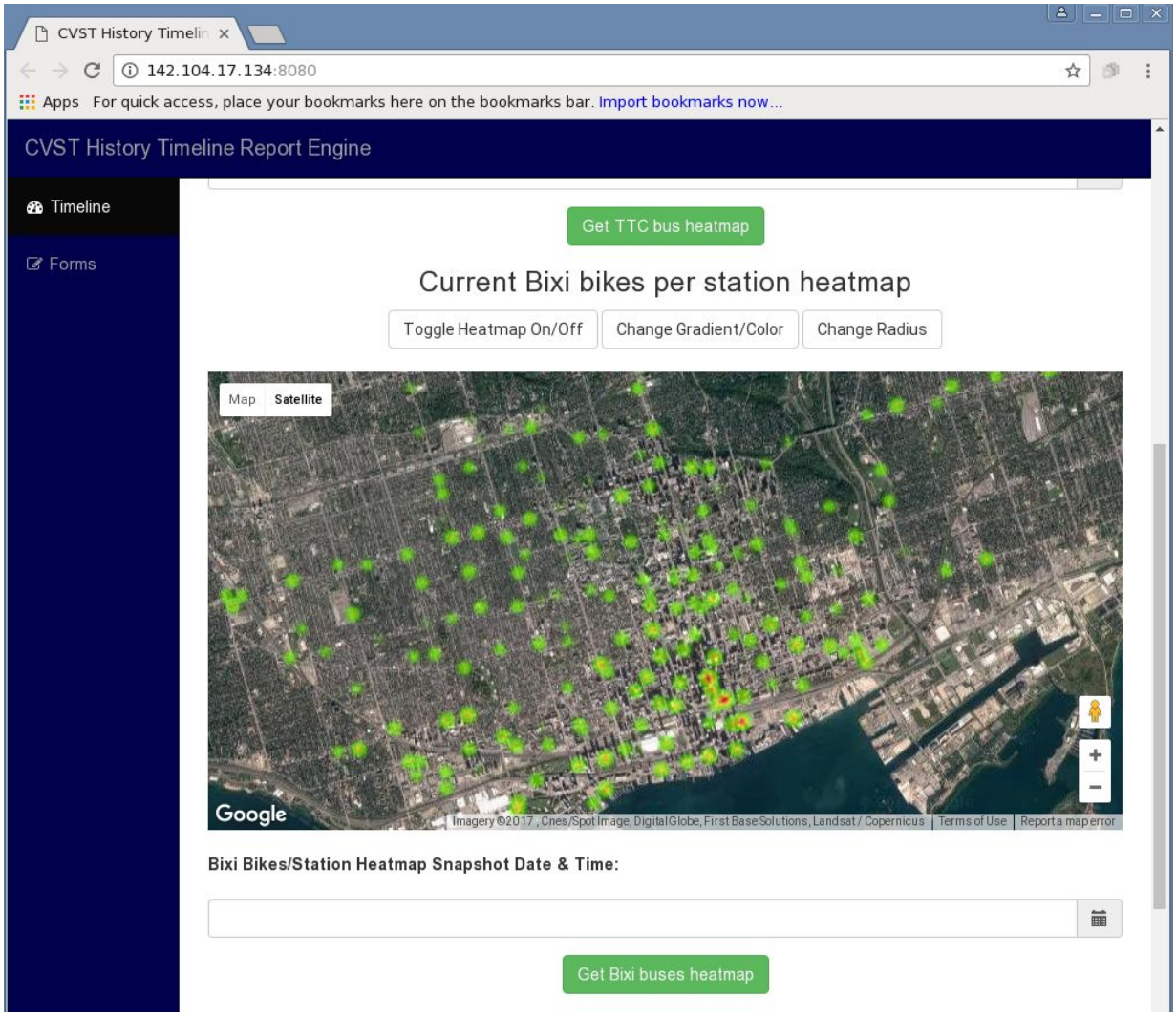


Figure E.2 - Front page timeline scrolled down, displaying the latest heatmap of BIXI bikes across GTA at the time the screenshot was taken. Red indicates highly concentrated number of BIXI bikes in the stations within the highlighted area, yellow indicates mediocre concentration, and green indicates low concentration.

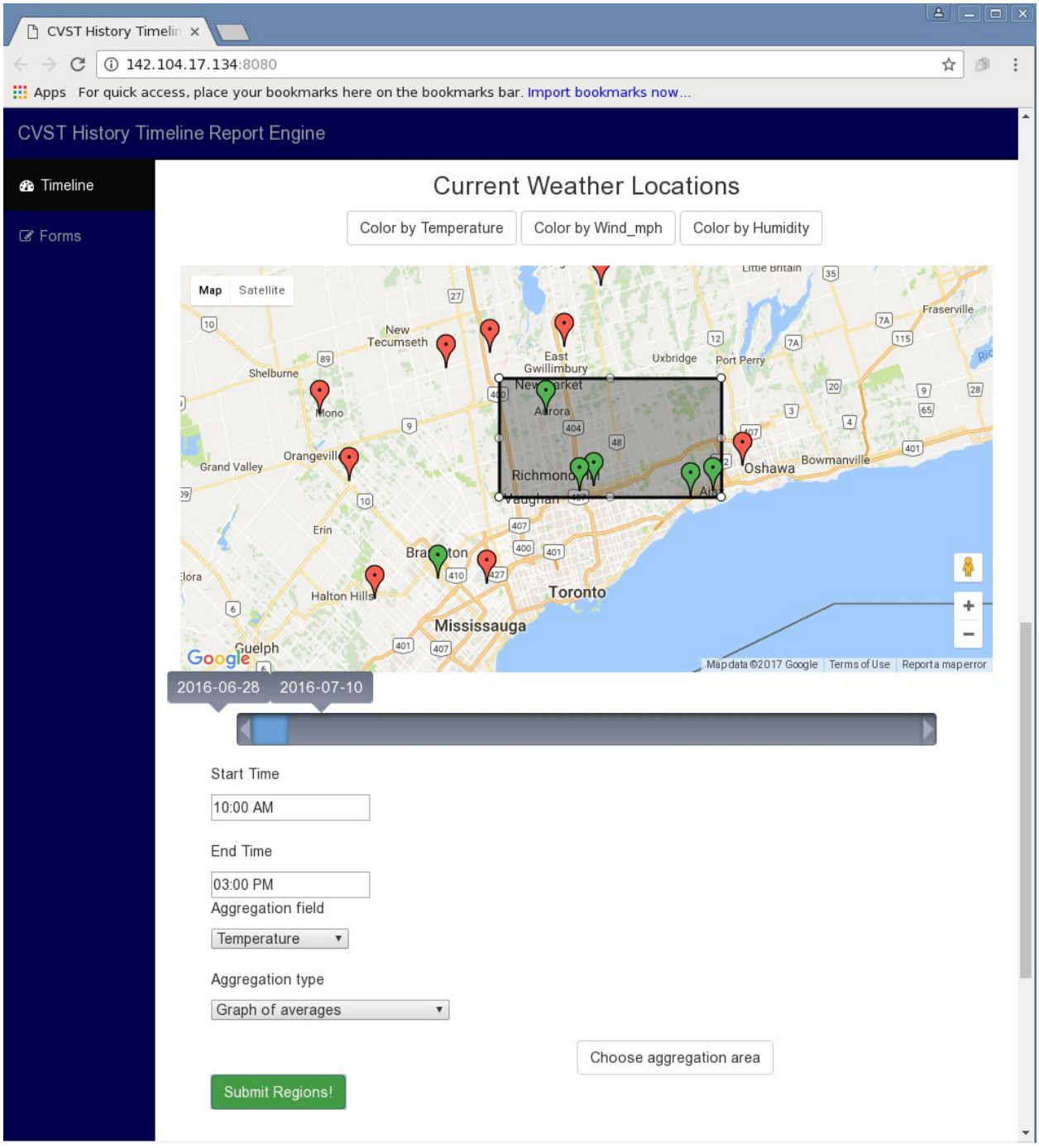


Figure E.3 - Front page timeline scrolled down, displaying the input fields for weather aggregation. The user can select area and time intervals, as well as aggregation types (top 10 averaged locations, top 10 snapshots of data, line charts, and values of max/min/avg).

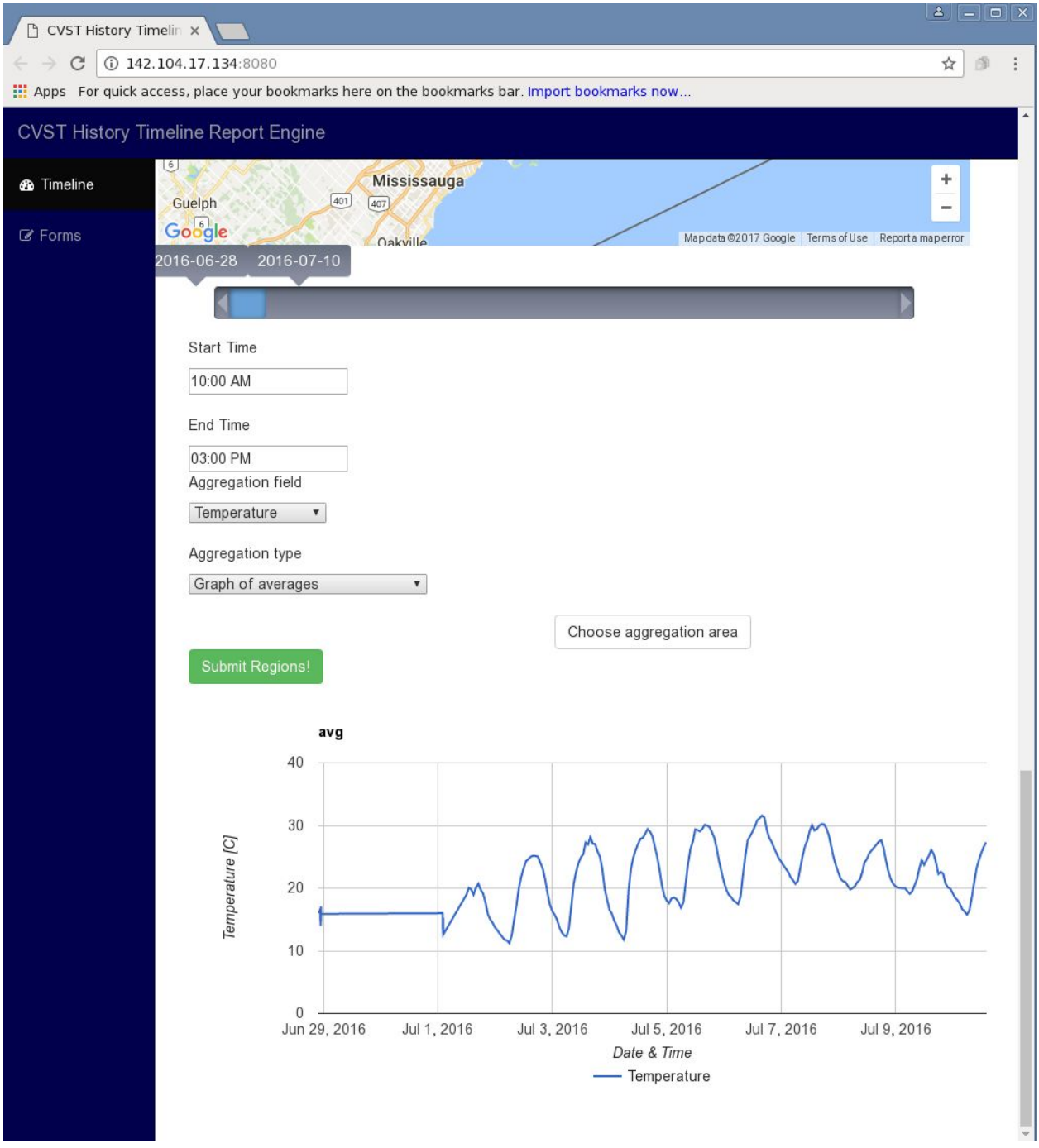


Figure E.4 - Front page timeline scrolled down, displaying the average temperature graphed in a line chart, as a result of the inputs submitted in Figure E.3.

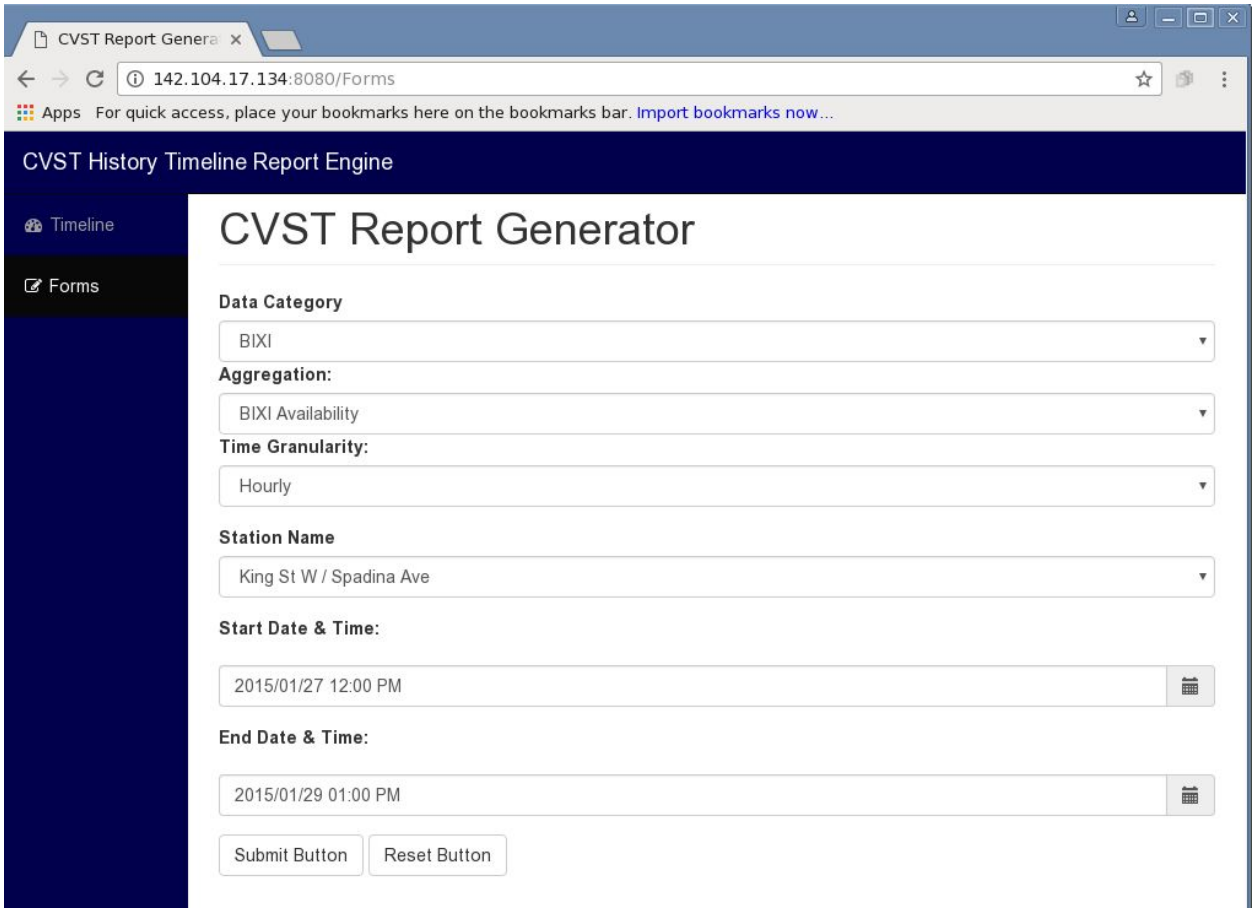


Figure E.4 - Report Generator page displaying the input fields user can select to specify the type of analytics.

Appendix F: Overview of Elasticsearch Components

The Elasticsearch database stores the data from the spark engine. It also takes in queries that are formatted by the server and returns query results.

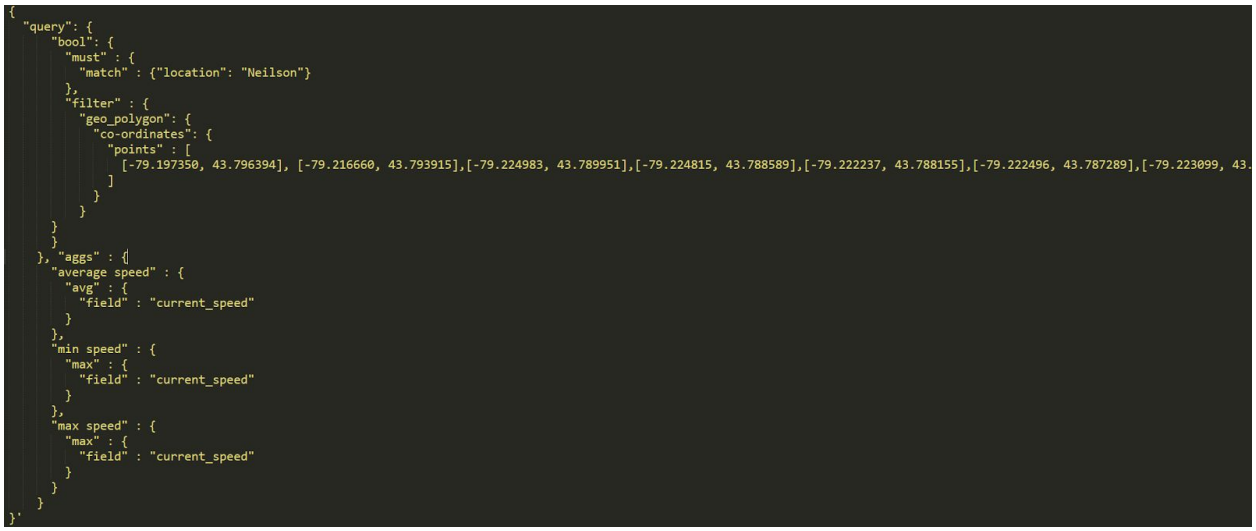


Figure F.1 - Example of a query string into Elasticsearch for filtering by space. Points represent a polygon of latitude/longitude coordinates

Figure F.2 - Example of a query for time aggregation. It groups data into intervals of 1 hour and aggregates it.

Figure F.3 - Query output for the query string in F.1 The results represents all sensors located inside the polygon specified by the “points” array and are located on Neilson Rd.

Appendix G: Overview of Spark Components

The backend Spark component consists of two main functionalities. First, a streaming job collects the data given by CVST's pub-sub. Afterwards, hourly jobs are run on this data as it is streamed in to populate Elasticsearch. A loading feature is also included to populate Elasticsearch with data from before the streaming system was set up.

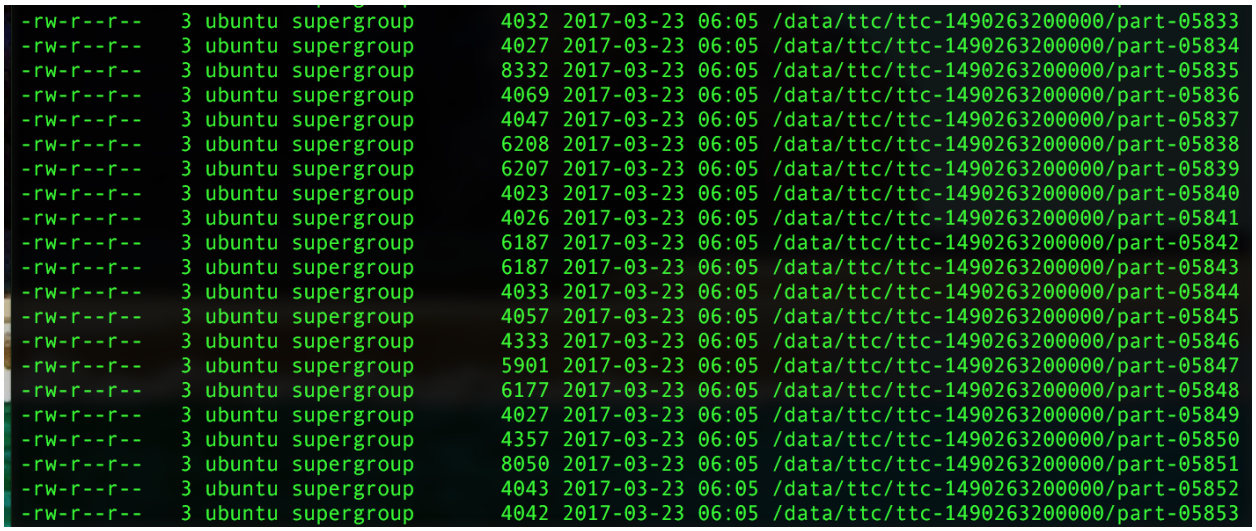
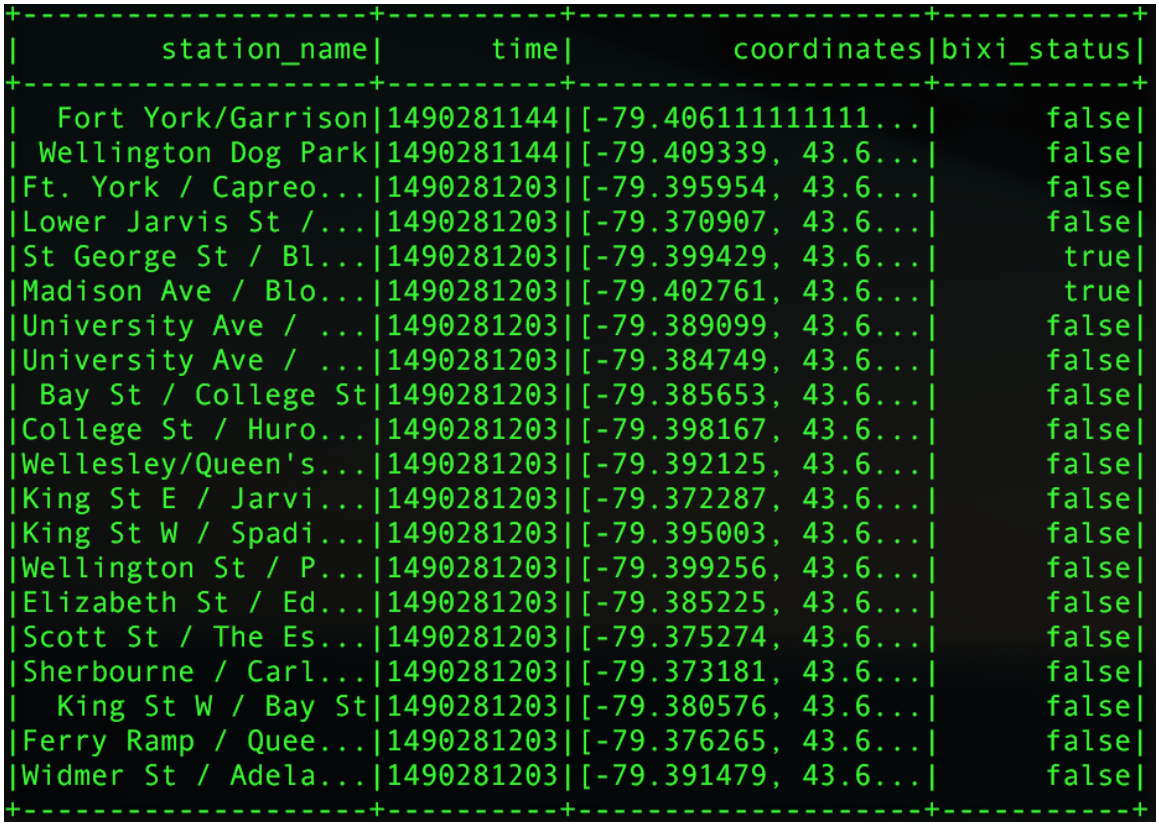


Figure G.1 - This is an example of data stored in HDFS. Each file contains part of the the TTC data retrieved for timestamp 1490263200 or March 23, 2017 between 9:00 am and 9:59 am.



station_name	time	coordinates	bixi_status
Fort York/Garrison	1490281144	[-79.406111111111...	false
Wellington Dog Park	1490281144	[-79.409339, 43.6...	false
Ft. York / Capreo...	1490281203	[-79.395954, 43.6...	false
Lower Jarvis St /...	1490281203	[-79.370907, 43.6...	false
St George St / Bl...	1490281203	[-79.399429, 43.6...	true
Madison Ave / Blo...	1490281203	[-79.402761, 43.6...	true
University Ave / ...	1490281203	[-79.389099, 43.6...	false
University Ave / ...	1490281203	[-79.384749, 43.6...	false
Bay St / College St	1490281203	[-79.385653, 43.6...	false
College St / Huro...	1490281203	[-79.398167, 43.6...	false
Wellesley/Queen's...	1490281203	[-79.392125, 43.6...	false
King St E / Jarvi...	1490281203	[-79.372287, 43.6...	false
King St W / Spadi...	1490281203	[-79.395003, 43.6...	false
Wellington St / P...	1490281203	[-79.399256, 43.6...	false
Elizabeth St / Ed...	1490281203	[-79.385225, 43.6...	false
Scott St / The Es...	1490281203	[-79.375274, 43.6...	false
Sherbourne / Carl...	1490281203	[-79.373181, 43.6...	false
King St W / Bay St	1490281203	[-79.380576, 43.6...	false
Ferry Ramp / Quee...	1490281203	[-79.376265, 43.6...	false
Widmer St / Adela...	1490281203	[-79.391479, 43.6...	false

Figure G.2 - This is an example of a snippet of Bixi data analytics, specifically the downtime which returns whether or not there are bikes available at a given station. Rather than printed to console, this is normal saved to Elasticsearch.

Appendix H: Overview of Server Components

The following images show code snippets on the server code show the general layout of the server. The main server class sets up the websocket and logging as well as a connect to the Elasticsearch Database. Different data type requests are handled inside different Handlers, which parse in input and export the appropriate queries.

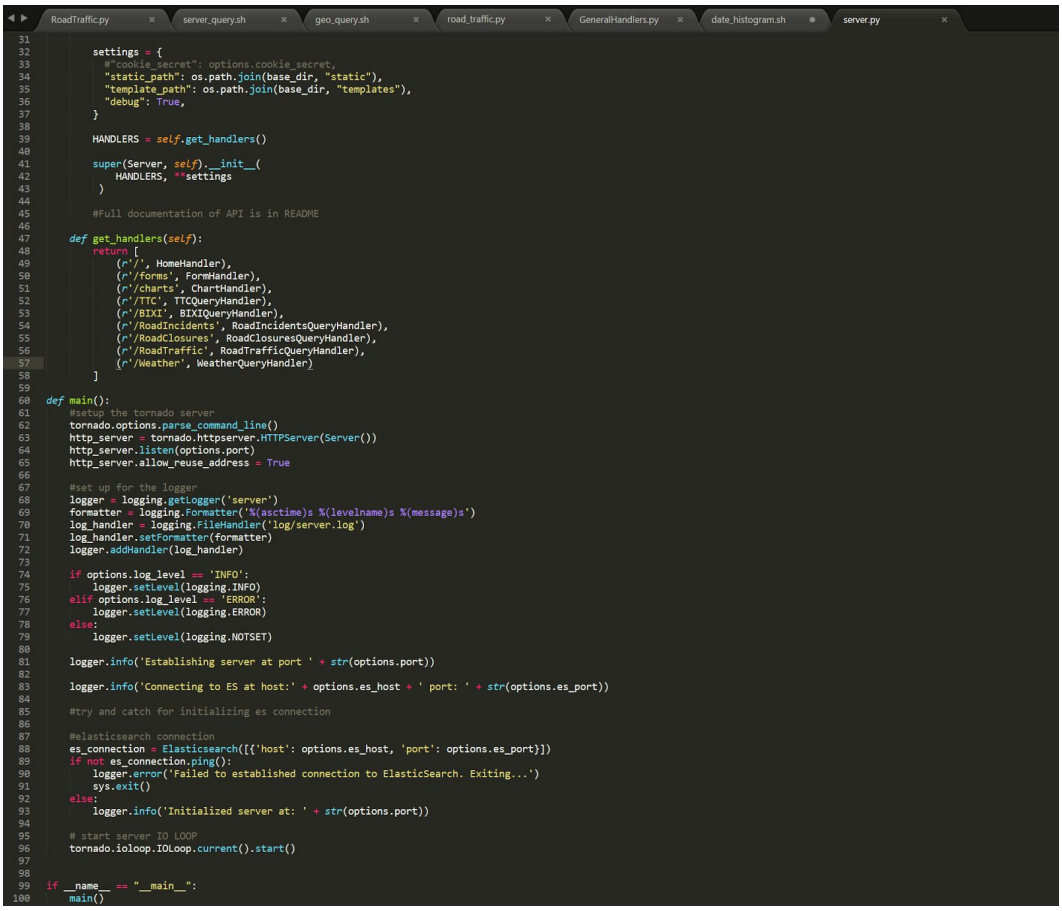


Figure H.1 - Code snippet of the main server class. The “Handlers” perform various actions to parse user input, format queries and parse Elasticsearch Response

```
class RoadTrafficQueryHandler(tornado.web.RequestHandler):

    #@gen.coroutine
    def post(self):
        logger = logging.getLogger('server')

        logger.info("RoadTraffic POST: " + self.request.remote_ip)

        #try catch for start and end time
        try:
            stringStartDate = self.get_argument('startDate')
            stringEndDate = self.get_argument('endDate')
            timeInterval = self.get_argument('period', None)
        except Exception as e:
            self.set_status(400)
            self.finish("<html><title>400: Missing URL Arguments</title</html>")
            return


        #try catch for checking if start and end times are valid
        try:
            startTime = time.mktime(datetime.datetime.strptime(stringStartDate, "%Y/%m/%d %I:%M %p").timetuple())
            endTime = time.mktime(datetime.datetime.strptime(stringEndDate, "%Y/%m/%d %I:%M %p").timetuple())

            startTime = datetime.datetime.fromtimestamp(int(stringStartDate))
            endTime = datetime.datetime.fromtimestamp(int(stringEndDate))

            if startTime > endTime:
                raise ValueError('End time less than start time')
        except Exception as e:
            self.set_status(400)
            self.finish("<html><title>400: Invalid start/end time arguments</title</html>")
            return

        #check body arguments
        try:
            body_args = tornado.escape.json_decode(self.request.body)
        except Exception as e:
            self.set_status(400)
            self.finish("<html><title>400: Missing body arguments</title</html>")
            return

        sensorIncludes = None
        sensorExcludes = None
        geoPoints = None

        try:
            sensorIncludes = body_args['station_includes']
        except KeyError as e:
            pass

        try:
            sensorExcludes = body_args['station_excludes']
        except KeyError as e:
            pass

        try:
            geoPoints = body_args['geo_points']
        except KeyError as e:
            pass

        print("Includes:" + str(sensorIncludes) + "\nExcludes: " + str(sensorExcludes) + "\nGeo_points" + str(geoPoints))

        try:
            queryResults = self.queryES(startTime, endTime, sensorIncludes, sensorExcludes, geoPoints, timeInterval)
        except Exception as e:
            self.set_status(500)
            self.finish("<html><title>500: Unable to connect to elastic search</title></html")
        #   print(str(e))
            return
```

Figure H.2 - Code snippet of the handler for road traffic sensors. The above code does error checking as well as parsing for user inputs

```
    self.write(queryResults)

@gen.coroutine
def queryES(self, startTime, endTime, includes, excludes, geoLocations, interval):

    es = Elasticsearch([{'host': options.es_host, 'port' : options.es_port}])
    if not es.ping():
        raise ValueError("Connection failed")
        print("unable to connect to")

    #query_string for the query
    query_string = Search(using=es, index="road_traffic")

    #Individual aggregation/filter fields
    date_range = Q({"range" : {"timestamp" : {"gte": startTime, "lte" : endTime, "format" : "epoch_second"}}})

    geo_point_query = Q({"geo_polygon" : {"co-ordinates" : {"points" : geoLocations}}})

    #filter based on geo location if parameters are provided
    if(geoLocations != None):
        combined_query = Q('bool', must=[date_range, geo_point_query])
        query_string.query(combined_query)
    else:
        query_string.query(date_range)

    #Histogram aggs if interval is provided
    if(interval != None):
        interval_aggs = A({"date_histogram": {"field": "timestamp", "interval" : interval, "format" : "dd-MM-YYYY HH-mm-ss"}})
        query_string.aggs.bucket("road_traffic", interval_aggs)



    #aggregations for max/min/avg

    query_string.metric('max_curr_speed', 'max', field='current_speed')
    query_string.metric('avg_curr_speed', 'avg', field='current_speed')
    query_string.metric('min_curr_speed', 'min', field='current_speed')

    query_string.metric('max_delta_speed', 'max', field='delta')
    query_string.metric('avg_delta_speed', 'avg', field='delta')
    query_string.metric('min_delta_speed', 'min', field='delta')

    #print(query_string.to_dict())


    response = query_string.execute()

    return str(response)
```

Figure H.3 - Code snippet of the handler for road traffic sensors. The above code creates the query strings which are sent to Elasticsearch