Tool for Querying the National Household Travel Survey Data

by

Akash Rathore

B.E., Ujjain Engg. College, India, 2015

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2018

Approved by:

Major Professor
Dr. Doina Caragea

Copyright

Abstract

The goal of the project is to create a database for storing the National Household Travel Survey (NHTS) data, and a user interface to query the database. Currently, the survey data is stored in excel files in the CSV format, which makes it hard to perform complex analyses over the data. Analyses of interest to transportation community include comparisons of the trips made by urban household to those made by rural household, finding the average trip time spent based on ethnicity, the total travel time of a particular household, the preferred vehicle by a specific household, average time spent per shopping trip, etc. The tool designed for the purpose of querying the NHTS database is a Python-based Web application. Django is used as the Web framework for this project and PostgreSQL is used for the back-end purpose. The user interface consists of various drop-down lists, text-boxes, buttons and other user interface components that facilitate querying the database and presenting the results in formats that allow easy interpretation. FusionCharts Django-Wrapper and FusionCharts Jquery-Plugin are used to visualize the data in the chart form. A Codebook of the NHTS dataset is also linked for the reference purpose at any point for the user. The tool built in the project allows the user to get a deeper understanding of the data, not only by plotting the data in the form of line charts, bar charts, two column graph, but also by providing the results of the queries in the CSV format for further analysis.

Table of Contents

List of Figures

Acknowledgments

I would like to express my very great appreciation to Dr. Doina Caragea for her valuable and constructive suggestions during the planning and development of this Project. Her willingness to give her time so generously has been very much appreciated. A special thanks to my committee members Dr. Cornelia Caragea and Dr. Torben Amtoft for guiding me. I would also like to thank Dr. Greg Newmark to work with the NHTS data and giving me this opportunity.

Dedication

I would like to dedicate my work to Dr. Doina Caragea and the Department of the Computer Science and Engineering at Kansas State University.

Preface

The report contains the idea and description of my Master's project at Kansas State University, 2018. This document will help you to understand the concepts and uses of the tool that is developed in the project. I am thankful to Dr. Doina Caragea for providing me this opportunity.

Chapter 1

Introduction

The goal of the project is to develop a tool to query the National Household Travel Survey (NHTS). NHTS which provides information to assist transportation planners and policy makers who need comprehensive data on travel and transportation patterns in the United States. The data collected by the NHTS can have an ample number of uses such as exploring traffic safety, congestion, the environment, energy consumption, demographic trends, bicycle and pedestrian studies, and transit planning. The tool that is designed to achieve the task of querying the data is built using Django 2.0.2 with Python 3.6.3. Django is a Python-based Web-development framework, which is well-optimized in terms of performance. The database used with the Django is PostgreSQL 9.6, and was chosen because of its flexibility with the number of columns.

1.1 What the NHTS Data Includes

The 2009 NHTS survey updates information gathered in the 2001 NHTS survey and in prior Nationwide Personal Transportation Surveys (NPTS) conducted in 1969, 1977, 1983, 1990, and 1995. The NHTS dataset (2009) consist of the following items along with a large number of other details:

- household data on the relationship of household members, education level, income, housing characteristics, and other demographic information;

- information on each household vehicle, including year, make, model and estimates of annual miles traveled;
- data about drivers, including information on travel as part of work;
- data about one-way trips taken during a designated 24-hour period (the household's travel day) including the time the trip began and ended, length of the trip, the composition of the travel party, mode of transportation, purpose of the trip, and the specific vehicle used (if a household vehicle);
- information to describe characteristics of the geographic area in which the sample household and workplace of sample persons are located;
- data on telecommuting;
- public perceptions of the transportation system;
- data on Internet usage; and
- the typical number of transit, walk and bike trips made over a period longer than the 24-hour travel day.

1.2 What the NHTS Data Does Not Include

The dataset consists of a lot of details about the travels and vehicles of various households. But, there are things which are not included in the dataset in order to respect the privacy of each household. Some of the details which the datasets do not have are:

- information regarding the costs of travel;
- information about specific travel routes or types of roads used in a trip;
- information that would identify the exact household or workplace location; and the traveler's reason for selecting a specific mode of travel over another mode.

1.3 NHTS Data Uses

The primary use of the NHTS data is to have a better understanding of travel behavior (Newmark and Plaut, 2005). The major idea behind the data collection is to help Department of Transportation officials to assess program initiatives, review programs and policies, study current mobility issues and plan for the future. There are many groups that use NHTS data for their researches in the transportation research community, including academics, consultants, and government. The majority of these groups use the data in order to examine:

- travel behavior at the individual and household level;
- the characteristics of travel, such as trip chaining, use of the various modes for traveling, amount and purpose of travel by time of day and day of the week, vehicle occupancy, and a host of other attributes;
- the relationship between demographics and travel; and
- the public's perceptions of the transportation system.

According to (Newmark, 2014), the NHTS data can also be used to:

- quantify travel behavior,
- analyze changes in travel characteristics like the means of transportation or number of vehicles owned over time,
- relate travel behavior to the geographical location of the traveler, and
- study the relationship of geographical location and travel over time.

Groups and people who are not from the transportation background use the NHTS data to connect the role of transportation with other aspects of our lives. Medical researchers use the data to determine crash exposure rates of drivers and passengers, including the elderly, who have heightened morbidity and mortality rates. The data is also used by Safety specialists to study the accident risk of school-age children, particularly when they are traveling on their own by walking or biking. Social service agencies also use the data in order to need to know more about how low-income households currently meet their travel needs.

The goal of the NHTS tools is to make it easy for the research transportation community to answer queries that would facilitate effective usage of the NHTS data.

Chapter 2

Data Organization

As mentioned in the previous chapter, the data collected in the 2009 by the National Household Travel Survey (NHTS) provides information to assist transportation planners and policy makers who need comprehensive data on travel and transportation patterns in the United States.

2.1 Data Collected

The NHTS/NPTS serves as the inventory of the nation's daily travel. The data includes the information about the daily trips made in a duration of 24-hours and includes:

- purpose of the trip (work, shopping, etc.);
- means of transportation used (car, bus, subway, walk, etc.);
- travel time of the trip;
- exact time of each trip i.e. start and end time;
- day of the week when the trip took place; and
- if the trip is done using a private vehicle then:
  - number of people in the vehicle, i.e., vehicle occupancy;

  - driver characteristics (age, sex, worker status, education level, etc.); and
  - vehicle specifications (make, model, model year, amount of miles driven in a year).

According to the NHTS website these data are collected for:

- all trips,
- all modes,
- all purposes,
- all trip lengths, and
- areas of the country, urban and rural.

2.2 Organization of the Data Files

The data collected by the NHTS is stored and available in two formats i.e. SAS Windows binary and CSV. The CSV files are chosen to analyze the data for this project. The data is stored in four CSV files where each file stores data related to each household and focuses on some specific aspects. Below is the brief description of the four files and the data available in each one of these:

- Hhv2Pub: This file contains information about a house like number of members, number of vehicles, the total income of the household, state, urban/rural and have 43 different columns giving more information about each household. It contains houseid which acts as a primary key to distinguish each household.

- Perv2Pub: This file contains information about each member of a household. Each member has a personid which when used with houseid can help to identify each row uniquely. It provides information like if a specific person from a household can drive or not if the person was born in the USA, persons level of education, how many jobs the person has, race of the person etc. This file has 117 columns providing more specific details about the travel behavior of each person.

- Vehv2Pub: This file provides data related to each vehicle available in every household. A vehicle has a VEHID, that along with houseid is used as a primary key. It contains relevant information about every vehicle present in each household like its make, odometer reading, vehicle age, vehicle model etc. In total, this file contains 61 column providing further details about the vehicle and household that owns it.

- Dayv2Pub: This file has all the information related to a particular trip made by a person from a household. It contains data like duration of a trip, the vehicle used, the person driving the vehicle on that trip, the number of people on that trip, the purpose of the trips (i.e. work, shopping etc). This file is the largest as it has all the trips made by almost every member of each and every person of that household. It has 112 columns which provide information about the trips made.

The ER-diagram of the database created based on the NHTS data, which will be used with the tool is given in Figure 2.1:

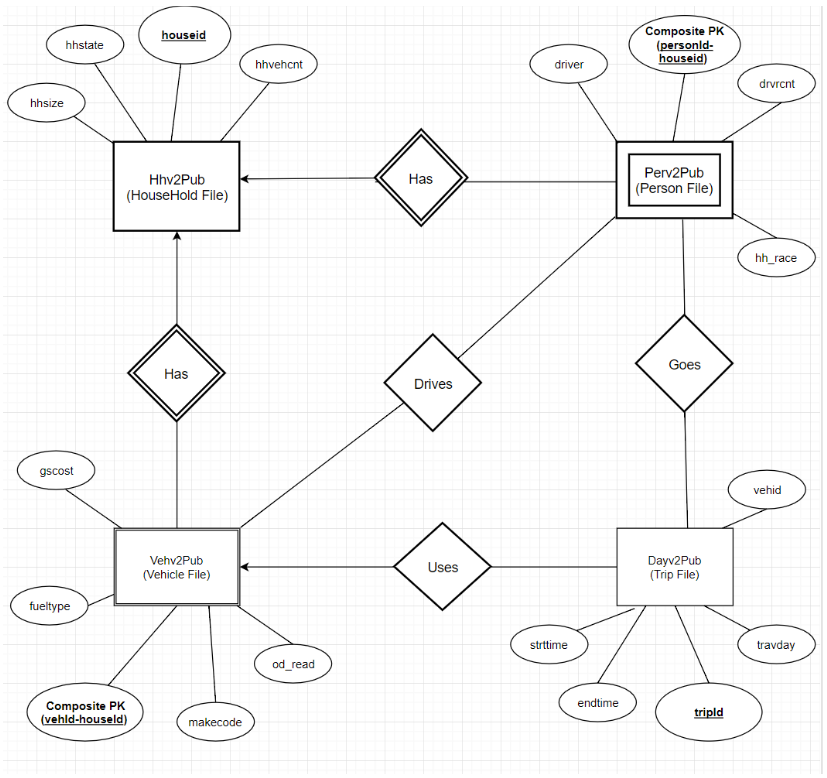


Figure 2.1: *ER-Diagram*

It has the important attributes of each entity as the number of attributes are too many only important ones are listed in the above diagram. The one-way arrow in the diagram represents the one-to-many relationship. So, a two-way arrow represents a many-to-many relationship.

Chapter 3

Examples of Queries

In this chapter, I will show sample PostgreSQL queries and the corresponding Django version of each query along with the explanation. In the tool, we have an option of typing a raw PostgreSQL query and the window looks like Figure 3.1:

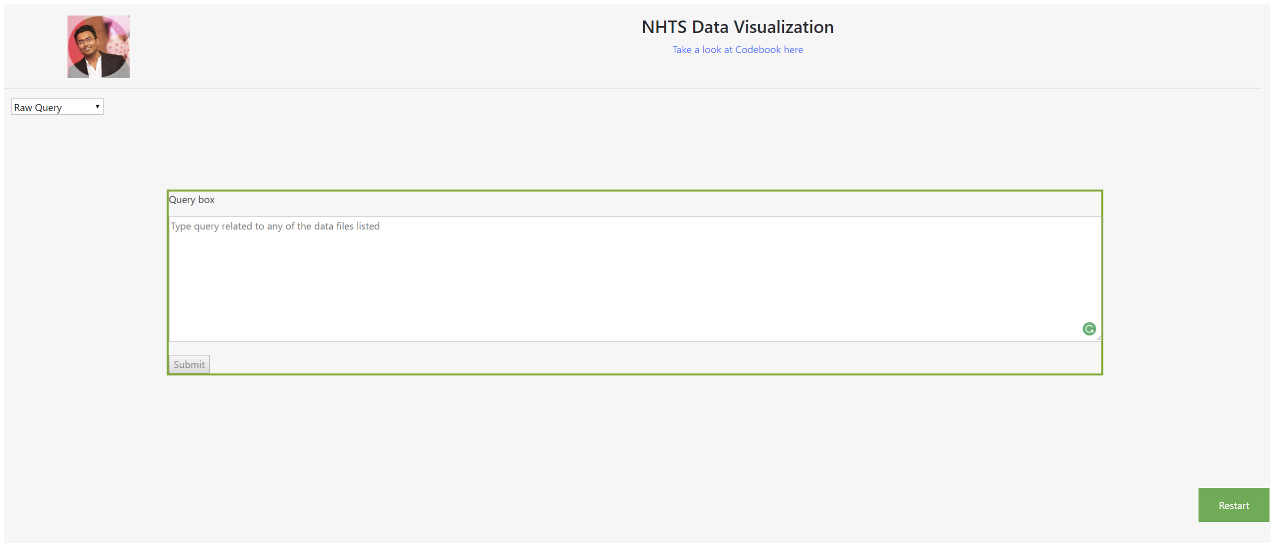

NHTS Data Visualization
Take a look at Codebook here
Raw Query
Query box
Type query related to any of the data files listed
Submit
Restart

Figure 3.1: *Raw PostgreSQL option*

User can type the query in the query box and the tool will display the result. Also, as in Django data is controlled using the models and the views so, it is important to convert a raw query to its Django version in order to access the data. Here are some examples with increasing complexity of the queries with an explanation.

i) To get all the houseid from the hhv2pub (Household) table.

PostgreSQL-

```
Select houseid from hhv2pub order by houseid;
```

In Django, a model returns a Queryset for every query written in the Django Views file. The above query can be performed in Django by calling the Hhv2Pub class from the Django model like:

Django-

```
queryset = Hhv2Pub.objects.values(houseid).order_by(houseid)
```

The above statement will return a Queryset that will contain all the houseids present in the hhv2pub table in ascending order.

ii) A query to get all the data of the vehv2pub (Vehicle) table

PostgreSQL-

```
Select * from vehv2pub;
```

Django-

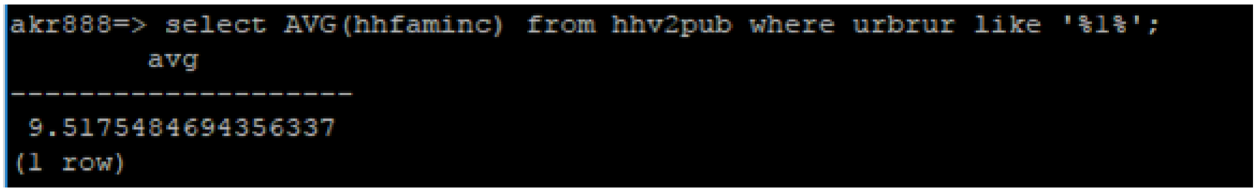
```
Queryset= Vehv2Pub.objects.all()
```

Here, objects.all() returns a Queryset consist of all the information of the rows.

iii) A query to get the average of income of all the urban housholds from the hhv2pub (HouseHold) table will look like:

PostgreSQL-

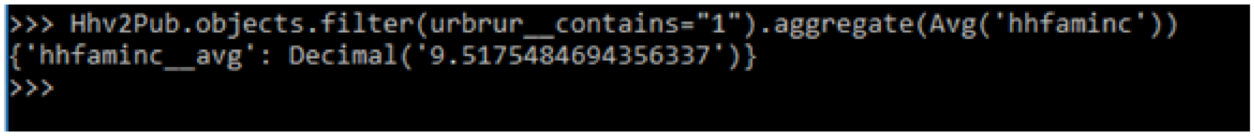
```
select AVG(hhfaminc) from hhv2pub where urbrur like ’%1%’;
```



```
akr888=> select AVG(hhfaminc) from hhv2pub where urbrur like '%1%';
        avg
--------------------
 9.5175484694356337
(1 row)
```

Django-

```
queryset=Hhv2Pub.objects.filter(urbrur__contains="1").aggregate(Avg(hhfaminc))
```



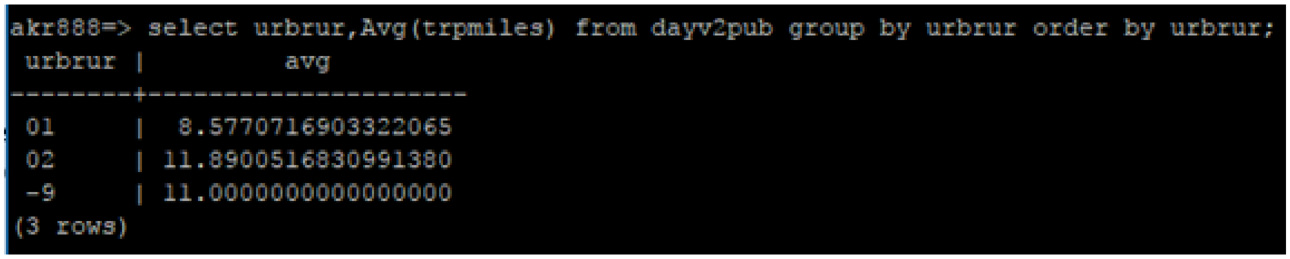
```
>>> Hhv2Pub.objects.filter(urbrur__contains="1").aggregate(Avg('hhfaminc'))
{'hhfaminc__avg': Decimal('9.5175484694356337')}
>>>
```

As it can be seen the 'where' clause of PostgreSQL is implemented using the filter in Django.

iv) To get the average miles of the trip made by urban and rural areas in ascending order to compare, the query will be:

PostgreSQL-

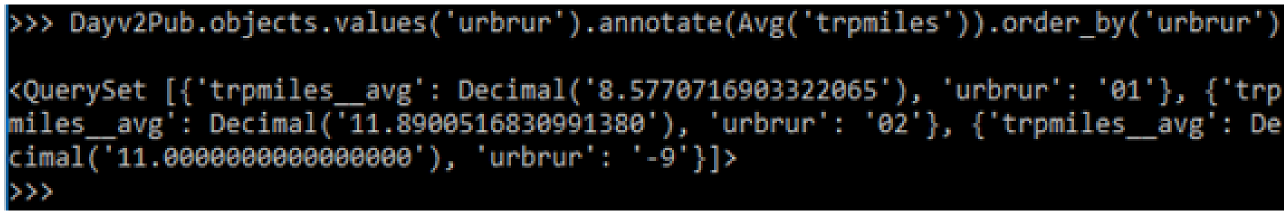
```
select urbrur, Avg(trpmiles) from dayv2pub group by urbrur order by urbrur;
```



```
akr888=> select urbrur,Avg(trpmiles) from dayv2pub group by urbrur order by urbrur;
 urbrur |         avg
--------+---------------------
 01     |  8.5770716903322065
 02     | 11.8900516830991380
 -9     | 11.0000000000000000
(3 rows)
```

Django-

```
Dayv2Pub.objects.values(urbrur).annotate(Avg(trpmiles)).order_by(urbrur)
```



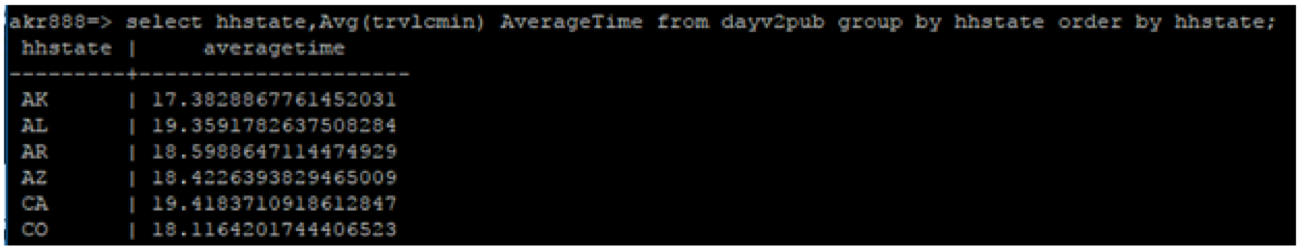
```
>>> Dayv2Pub.objects.values('urbrur').annotate(Avg('trpmiles')).order_by('urbrur')

<QuerySet [{'trpmiles__avg': Decimal('8.5770716903322065'), 'urbrur': '01'}, {'trp
miles__avg': Decimal('11.8900516830991380'), 'urbrur': '02'}, {'trpmiles__avg': De
cimal('11.0000000000000000'), 'urbrur': '-9'}]>
>>>
```

Here, 'annotate' is used in Django to get the average of each row grouped based on the urban and rural.

v) To get average time spent on trips by the household of each state queries will be written as:

PostgreSQL-

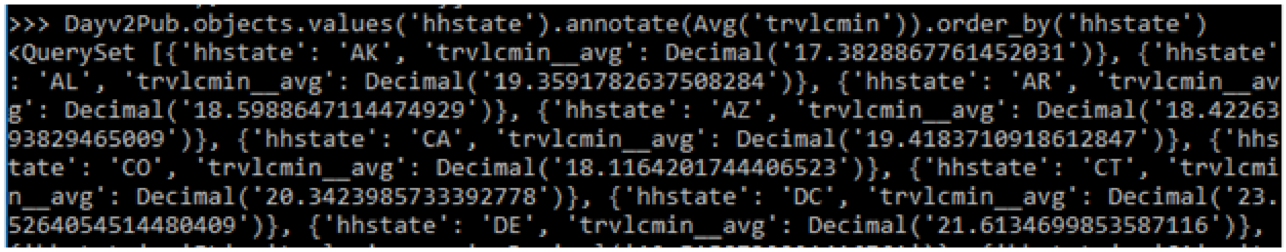
```
select hhstate,Avg(trvlcmin) AverageTime from dayv2pub group by hhstate order by
hhstate;
```



```
akr888=> select hhstate,Avg(trvlcmin) AverageTime from dayv2pub group by hhstate order by hhstate;
 hhstate |     averagetime
---------+---------------------
 AK      | 17.3828867761452031
 AL      | 19.3591782637508284
 AR      | 18.5988647114474929
 AZ      | 18.4226393829465009
 CA      | 19.4183710918612847
 CO      | 18.1164201744406523
```

Django-

```
Dayv2Pub.objects.values('hhstate').annotate(Avg('trvlcmin')).order_by('hhstate')
```

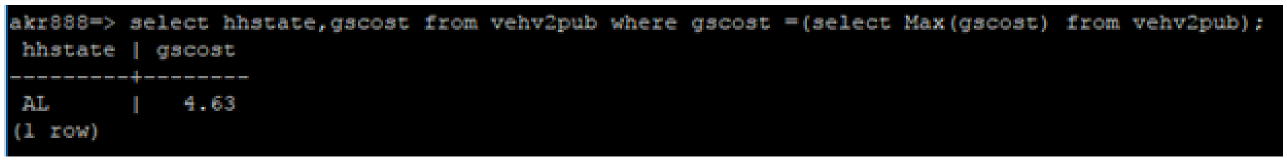


```
>>> Dayv2Pub.objects.values('hhstate').annotate(Avg('trvlcmin')).order_by('hhstate')
<QuerySet [{'hhstate': 'AK', 'trvlcmin__avg': Decimal('17.3828867761452031')}, {'hhstate'
: 'AL', 'trvlcmin__avg': Decimal('19.3591782637508284')}, {'hhstate': 'AR', 'trvlcmin__av
g': Decimal('18.5988647114474929')}, {'hhstate': 'AZ', 'trvlcmin__avg': Decimal('18.42263
93829465009')}, {'hhstate': 'CA', 'trvlcmin__avg': Decimal('19.4183710918612847')}, {'hhs
tate': 'CO', 'trvlcmin__avg': Decimal('18.1164201744406523')}, {'hhstate': 'CT', 'trvlcmi
n__avg': Decimal('20.3423985733392778')}, {'hhstate': 'DC', 'trvlcmin__avg': Decimal('23.
5264054514480409')}, {'hhstate': 'DE', 'trvlcmin__avg': Decimal('21.6134699853587116')},
```

vi) Query to find which state has the maximum gas cost from the vehv2pub(Vehicle) table:

PostgreSQL-

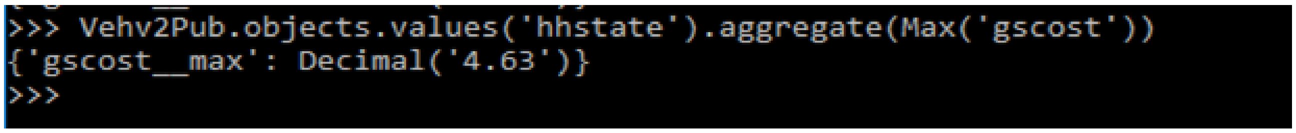
```
select hhstate,gscost from vehv2pub where gscost =(select Max(gscost) from vehv2pub);
```



```
akr888=> select hhstate,gscost from vehv2pub where gscost =(select Max(gscost) from vehv2pub);
 hhstate | gscost
---------+--------
 AL      |   4.63
(1 row)
```

Django-

```
Vehv2Pub.objects.values('hhstate').aggregate(Max('gscost'))
```

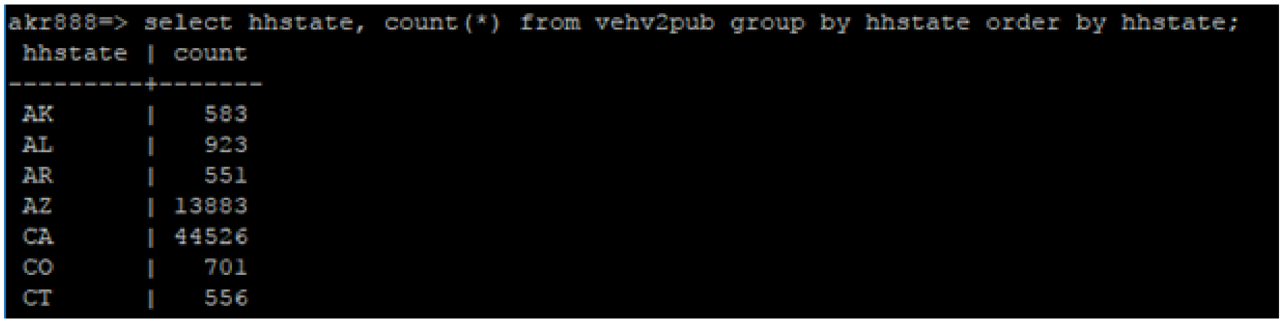


```
>>> Vehv2Pub.objects.values('hhstate').aggregate(Max('gscost'))
{'gscost__max': Decimal('4.63')}
>>>
```

vii) Query to get the total count of vehicles in each state lexically sorted according to the state name:

PostgreSQL-

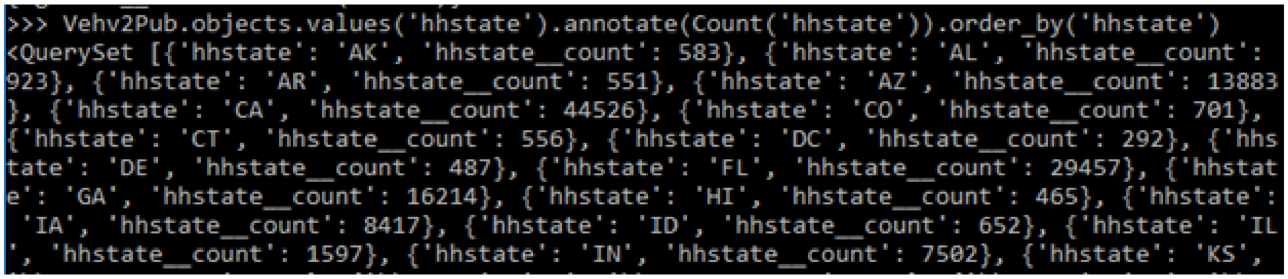
```
select hhstate, count(*) from vehv2pub group by hhstate order by hhstate;
```



```
akr888=> select hhstate, count(*) from vehv2pub group by hhstate order by hhstate;
 hhstate | count
---------+-------
 AK      |   583
 AL      |   923
 AR      |   551
 AZ      | 13883
 CA      | 44526
 CO      |   701
 CT      |   556
```

Django-

```
Vehv2Pub.objects.values('hhstate').annotate(Count('hhstate')).order_by('hhstate')
```

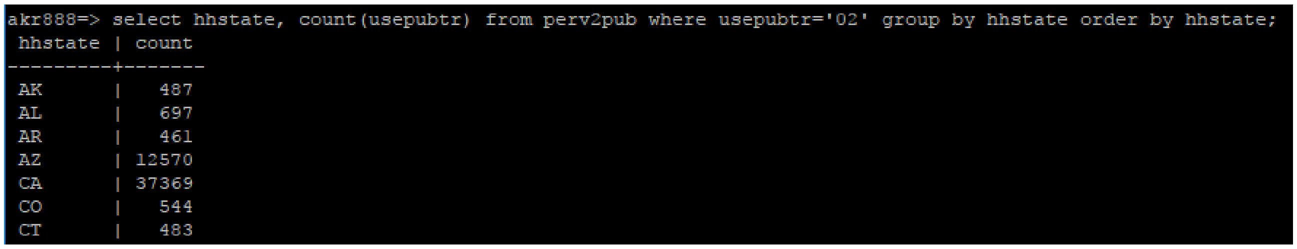


```
>>> Vehv2Pub.objects.values('hhstate').annotate(Count('hhstate')).order_by('hhstate')
<QuerySet [{'hhstate': 'AK', 'hhstate__count': 583}, {'hhstate': 'AL', 'hhstate__count':
923}, {'hhstate': 'AR', 'hhstate__count': 551}, {'hhstate': 'AZ', 'hhstate__count': 13883
}, {'hhstate': 'CA', 'hhstate__count': 44526}, {'hhstate': 'CO', 'hhstate__count': 701},
{'hhstate': 'CT', 'hhstate__count': 556}, {'hhstate': 'DC', 'hhstate__count': 292}, {'hhs
tate': 'DE', 'hhstate__count': 487}, {'hhstate': 'FL', 'hhstate__count': 29457}, {'hhstat
e': 'GA', 'hhstate__count': 16214}, {'hhstate': 'HI', 'hhstate__count': 465}, {'hhstate':
 'IA', 'hhstate__count': 8417}, {'hhstate': 'ID', 'hhstate__count': 652}, {'hhstate': 'IL
', 'hhstate__count': 1597}, {'hhstate': 'IN', 'hhstate__count': 7502}, {'hhstate': 'KS',
```

viii) Query to get the number of people not using public transport for their commute in each state sorted lexically according to state from perv2pub(Person) table:

PostgreSQL-

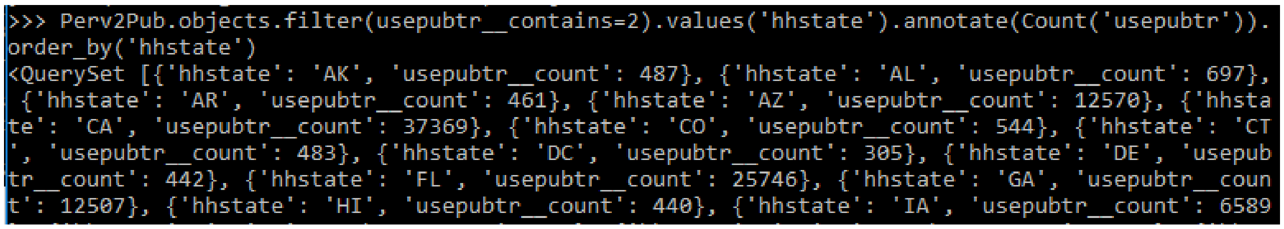
```
select hhstate, count(usepubtr) from perv2pub where usepubtr='02' group by hhstate
order by hhstate;
```



```
akr888=> select hhstate, count(usepubtr) from perv2pub where usepubtr='02' group by hhstate order by hhstate;
 hhstate | count
---------+-------
 AK      |   487
 AL      |   697
 AR      |   461
 AZ      | 12570
 CA      | 37369
 CO      |   544
 CT      |   483
```

Django-

```
Perv2Pub.objects.filter(usepubtr__contains=2).values('hhstate').annotate(Count
('usepubtr')).order_by('hhstate')
```

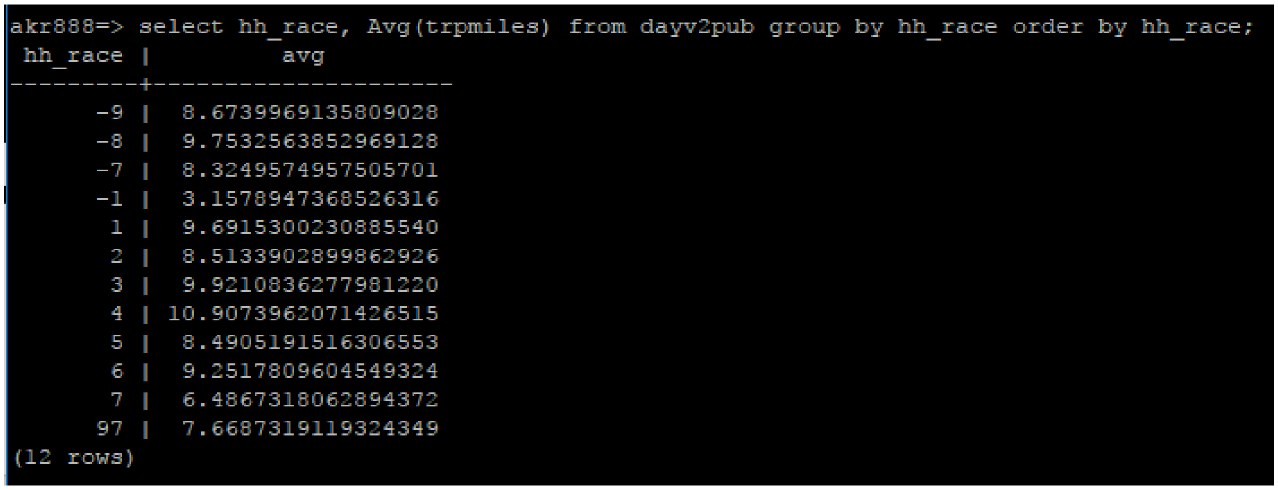


```
>>> Perv2Pub.objects.filter(usepubtr__contains=2).values('hhstate').annotate(Count('usepubtr')).
order_by('hhstate')
<QuerySet [{'hhstate': 'AK', 'usepubtr__count': 487}, {'hhstate': 'AL', 'usepubtr__count': 697},
 {'hhstate': 'AR', 'usepubtr__count': 461}, {'hhstate': 'AZ', 'usepubtr__count': 12570}, {'hhsta
te': 'CA', 'usepubtr__count': 37369}, {'hhstate': 'CO', 'usepubtr__count': 544}, {'hhstate': 'CT
', 'usepubtr__count': 483}, {'hhstate': 'DC', 'usepubtr__count': 305}, {'hhstate': 'DE', 'usepub
tr__count': 442}, {'hhstate': 'FL', 'usepubtr__count': 25746}, {'hhstate': 'GA', 'usepubtr__coun
t': 12507}, {'hhstate': 'HI', 'usepubtr__count': 440}, {'hhstate': 'IA', 'usepubtr__count': 6589
```

ix) Query to retrieve the average of the trips in miles based on the ethnicity of the people on the trip from perv2pub(Person) table:

PostgreSQL-

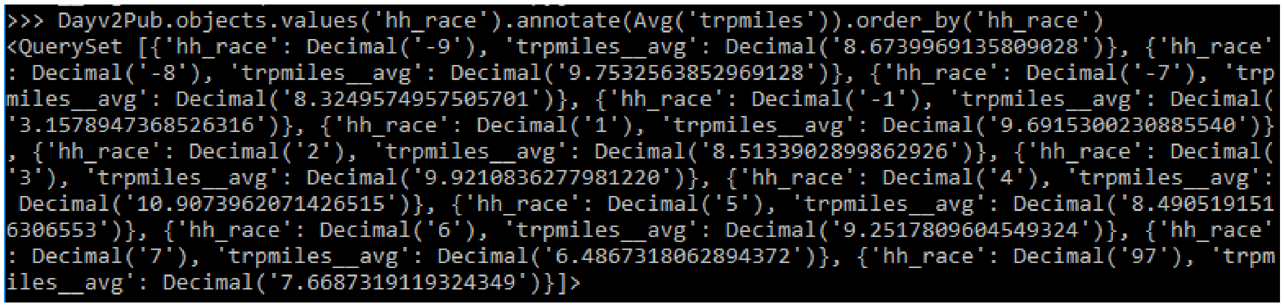
```
select hh_race, Avg(trpmiles) from dayv2pub group by hh_race order by hh_race;
```



```
akr888=> select hh_race, Avg(trpmiles) from dayv2pub group by hh_race order by hh_race;
 hh_race |         avg
---------+---------------------
      -9 |  8.6739969135809028
      -8 |  9.7532563852969128
      -7 |  8.3249574957505701
      -1 |  3.1578947368526316
       1 |  9.6915300230885540
       2 |  8.5133902899862926
       3 |  9.9210836277981220
       4 | 10.9073962071426515
       5 |  8.4905191516306553
       6 |  9.2517809604549324
       7 |  6.4867318062894372
      97 |  7.6687319119324349
(12 rows)
```

Django-

```
Dayv2Pub.objects.values('hh_race').annotate(Avg('trpmiles')).order_by('hh_race')
```



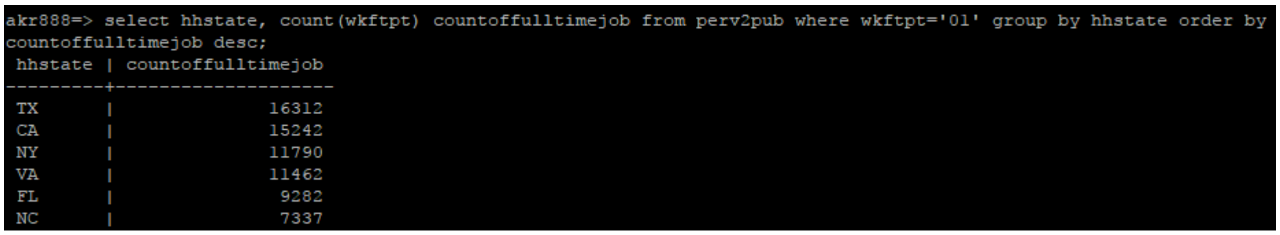
```
>>> Dayv2Pub.objects.values('hh_race').annotate(Avg('trpmiles')).order_by('hh_race')
<QuerySet [{'hh_race': Decimal('-9'), 'trpmiles__avg': Decimal('8.6739969135809028')}, {'hh_race'
: Decimal('-8'), 'trpmiles__avg': Decimal('9.7532563852969128')}, {'hh_race': Decimal('-7'), 'trp
miles__avg': Decimal('8.3249574957505701')}, {'hh_race': Decimal('-1'), 'trpmiles__avg': Decimal(
'3.1578947368526316')}, {'hh_race': Decimal('1'), 'trpmiles__avg': Decimal('9.6915300230885540')}
, {'hh_race': Decimal('2'), 'trpmiles__avg': Decimal('8.5133902899862926')}, {'hh_race': Decimal(
'3'), 'trpmiles__avg': Decimal('9.9210836277981220')}, {'hh_race': Decimal('4'), 'trpmiles__avg':
 Decimal('10.9073962071426515')}, {'hh_race': Decimal('5'), 'trpmiles__avg': Decimal('8.490519151
6306553')}, {'hh_race': Decimal('6'), 'trpmiles__avg': Decimal('9.2517809604549324')}, {'hh_race'
: Decimal('7'), 'trpmiles__avg': Decimal('6.4867318062894372')}, {'hh_race': Decimal('97'), 'trpm
iles__avg': Decimal('7.6687319119324349')}]>
```

x) Query to get the total number of people having full-time jobs group by each state in a descending order of number of people with full-time jobs:

PostgreSQL-

```
select hhstate, count(wkftpt) countoffulltimejob from perv2pub where wkftpt='01'
```

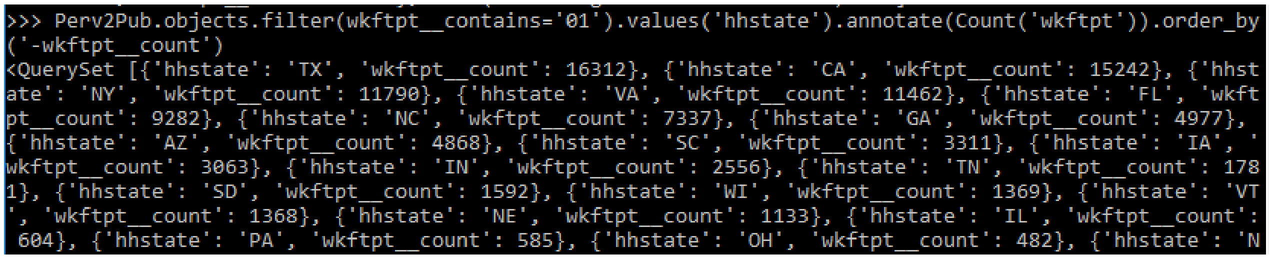
```
group by hhstate order by countoffulltimejob desc;
```



```
akr888=> select hhstate, count(wkftpt) countoffulltimejob from perv2pub where wkftpt='01' group by hhstate order by
countoffulltimejob desc;
 hhstate | countoffulltimejob
---------+--------------------
 TX      |              16312
 CA      |              15242
 NY      |              11790
 VA      |              11462
 FL      |               9282
 NC      |               7337
```

Django-

```
Perv2Pub.objects.filter(wkftpt__contains='01').values('hhstate').annotate(Count
('wkftpt')).order_by('-wkftpt__count')
```



```
>>> Perv2Pub.objects.filter(wkftpt__contains='01').values('hhstate').annotate(Count('wkftpt')).order_by
('-wkftpt__count')
<QuerySet [{'hhstate': 'TX', 'wkftpt__count': 16312}, {'hhstate': 'CA', 'wkftpt__count': 15242}, {'hhst
ate': 'NY', 'wkftpt__count': 11790}, {'hhstate': 'VA', 'wkftpt__count': 11462}, {'hhstate': 'FL', 'wkft
pt__count': 9282}, {'hhstate': 'NC', 'wkftpt__count': 7337}, {'hhstate': 'GA', 'wkftpt__count': 4977},
{'hhstate': 'AZ', 'wkftpt__count': 4868}, {'hhstate': 'SC', 'wkftpt__count': 3311}, {'hhstate': 'IA', '
wkftpt__count': 3063}, {'hhstate': 'IN', 'wkftpt__count': 2556}, {'hhstate': 'TN', 'wkftpt__count': 178
1}, {'hhstate': 'SD', 'wkftpt__count': 1592}, {'hhstate': 'WI', 'wkftpt__count': 1369}, {'hhstate': 'VT
', 'wkftpt__count': 1368}, {'hhstate': 'NE', 'wkftpt__count': 1133}, {'hhstate': 'IL', 'wkftpt__count':
 604}, {'hhstate': 'PA', 'wkftpt__count': 585}, {'hhstate': 'OH', 'wkftpt__count': 482}, {'hhstate': 'N
```

Above examples are meant to illustrate the kind of queries that can be used to retrieve data in both PostgreSQL and Django ORM (Object Relation Model) from the NHTS database.

Chapter 4

User Interface

4.1 Tool Workflow

This section is to explain the project flow and the way the tool can be used to get the desired outcome. The browser window at the starting of the project will look as Figure 4.1:

A link to the NHTS codebook is also given in the header of the project and available on every page just in case if the user has to check codes for any column stored in the database table. This link can be seen in the Figure 4.1

Here the user has two options. One is to write a raw PostgreSQL query and the other is to select the Aggregated option from the drop-down list given on the very first page.

4.2 Raw Queries

If the user selects the raw query option then the text-box at the right bottom of the window will be enabled and now the user can type the PostgreSQL query. The moment user will start typing the query the submit button will be enabled just below the text box. After typing the query the user can submit it and the result of the query will be presented to the user in the form of a table. A sample query along with its output is shown in the Figure 4.2

In the text-box shown in Figure 4.2, the user typed a query-

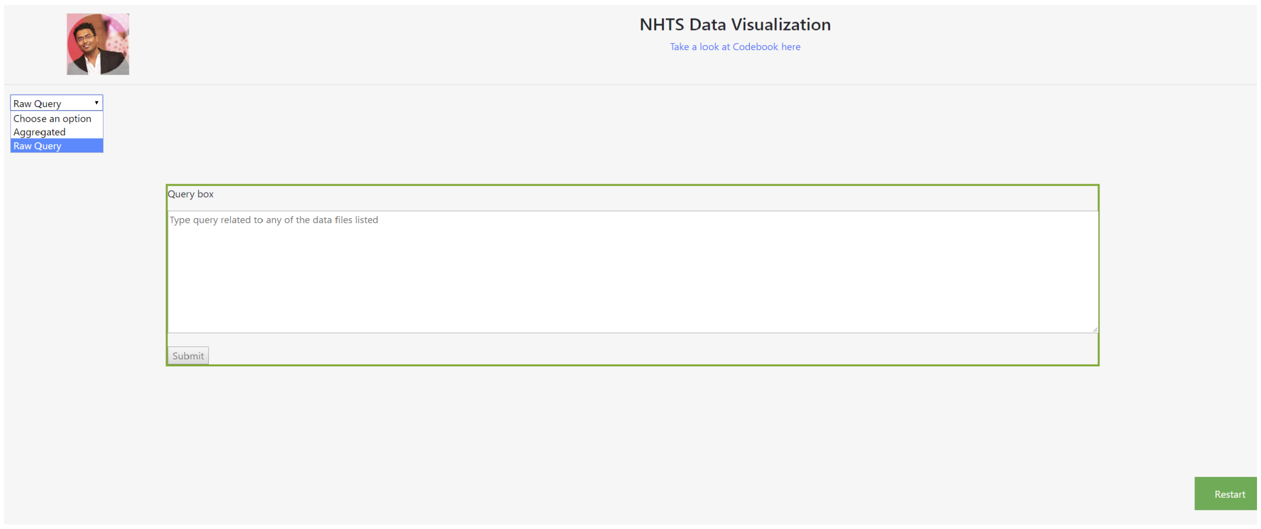
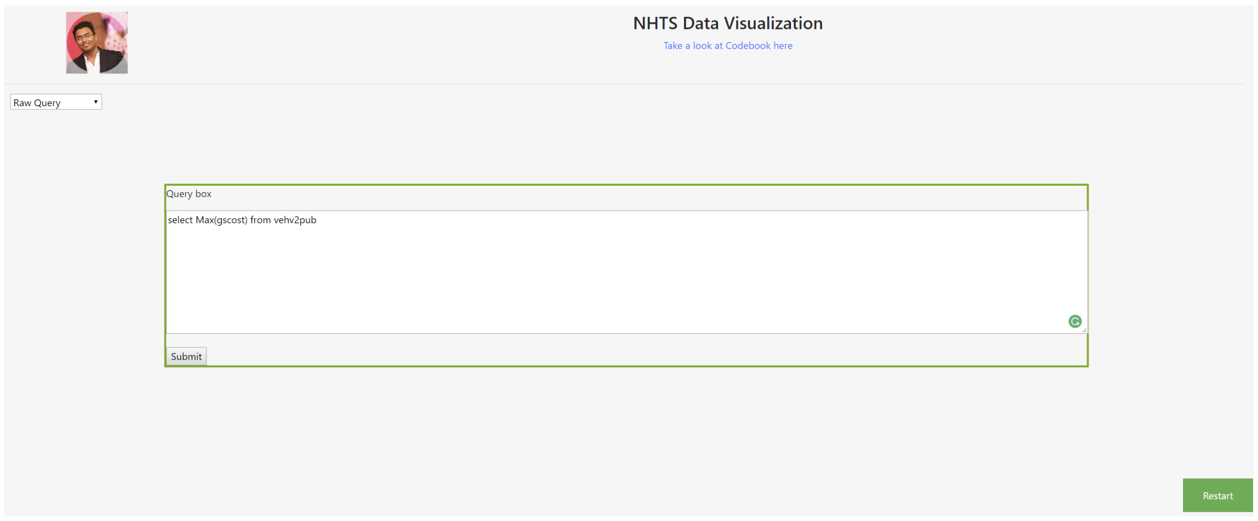


Figure 4.1: *Home Page of the tool*



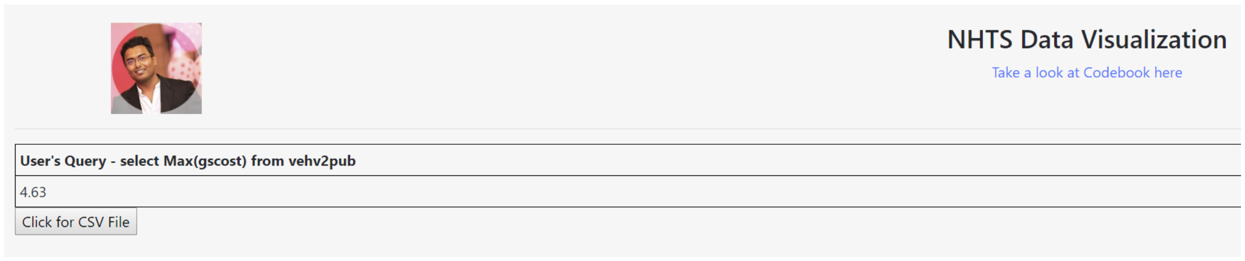
```
select Max(gscost) from vehv2pub
```

Figure 4.2: *Raw Query Option*

```
select Max(gscost) from vehv2pub
```

A query to get the maximum gas cost from the vehicle table. The result of the above query will look as Figure 4.3

At this point, the user can click the restart button to write another query or to select the Aggregated option from the drop-down list.



User's Query - select Max(gscost) from vehv2pub

4.63

Figure 4.3: *Result of Raw Query*

4.3 Aggregated Queries

Now, if the user selects Aggregated from the drop-down list then the user has multiple options to get the data from the database along with the corresponding graph options to analyze the retrieved data clearly.

Lets take a look at the Aggregated option. Here, the user has three choices Urban-Rural, Ethnicity, and State. If the user selects Urban-Rural from the drop-down list, the user will have another drop-down list with lots of options to see the Aggregated results.

The drop-down list of the available options at this point will look as shown in Figure 4.4:

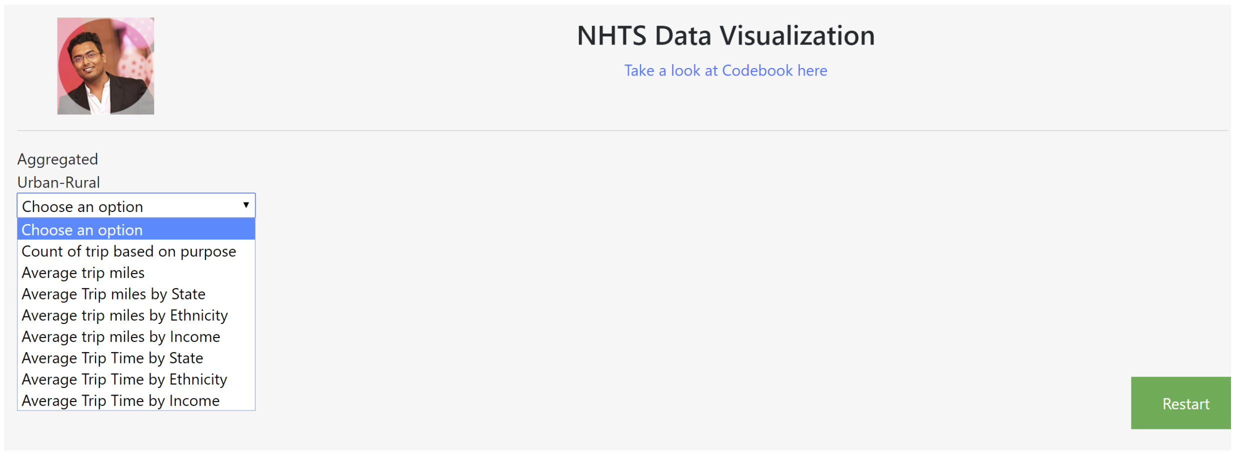
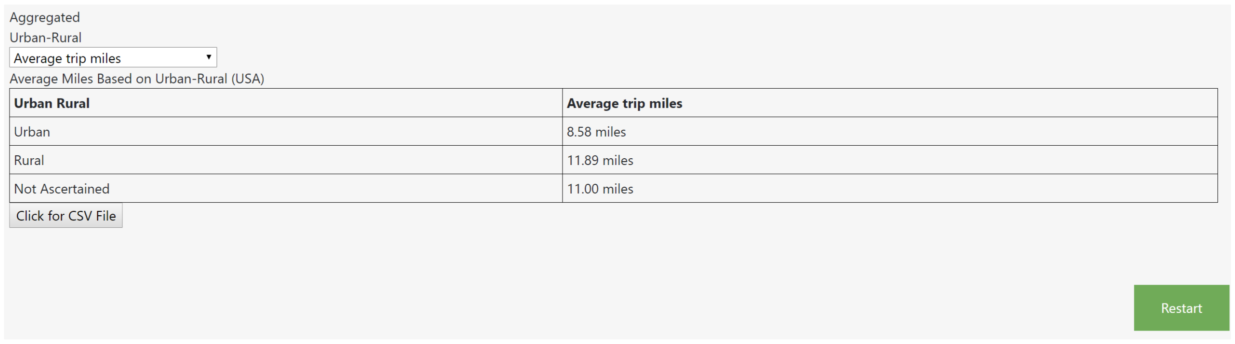

NHTS Data Visualization
Take a look at Codebook here
Aggregated
Urban-Rural
Choose an option
Choose an option
Count of trip based on purpose
Average trip miles
Average Trip miles by State
Average trip miles by Ethnicity
Average trip miles by Income
Average Trip Time by State
Average Trip Time by Ethnicity
Average Trip Time by Income
Restart

Figure 4.4: *Options In Urban-Rural type Queries*

From the above-shown list if the user selects 'Average trip miles' option then the user will be prompted to a table containing the data of average trip miles traveled in the entire USA by the people categorized as either Urban, Rural or Not ascertained. The result at this point will look like Figure 4.5



Aggregated

Urban-Rural

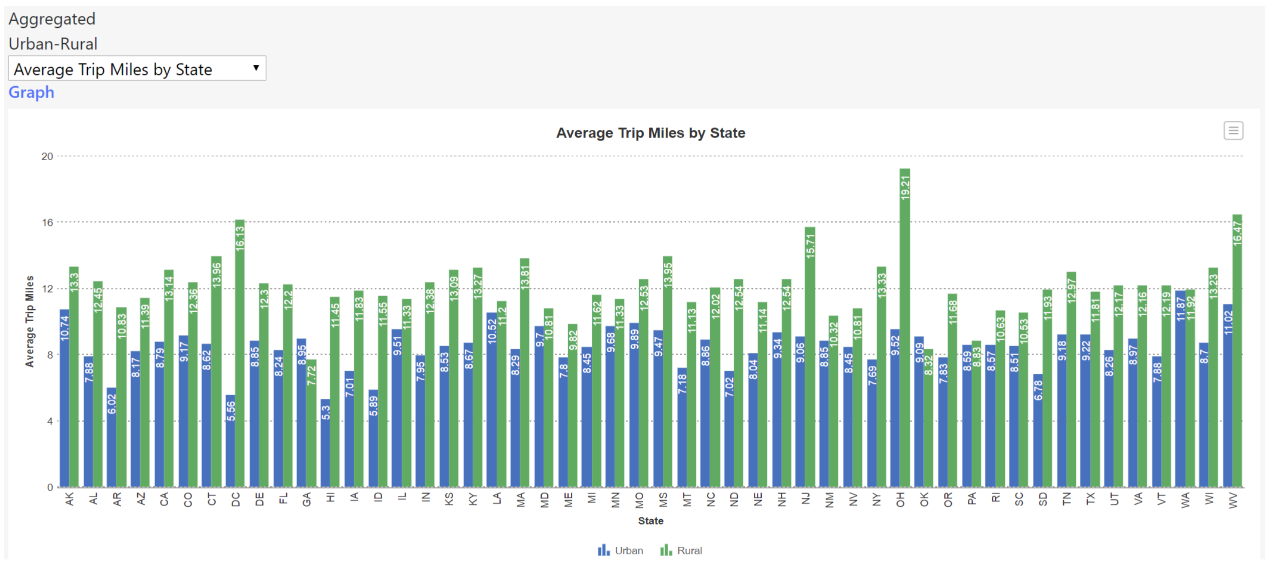
Average trip miles

Average Miles Based on Urban-Rural (USA)

Urban Rural	Average trip miles
Urban	8.58 miles
Rural	11.89 miles
Not Ascertained	11.00 miles

Figure 4.5: *Output of Average Trip miles*

At this point, the user still has the drop-down list under the Urban-Rural option available to select other options. So, if the user selects another option like Average Trip Miles by State then the page will automatically submit and pull the new results which will have a collapsible two column bar graph option representing the average trip miles of people in each state categorized as urban and rural which will look like Figure 4.6



Aggregated

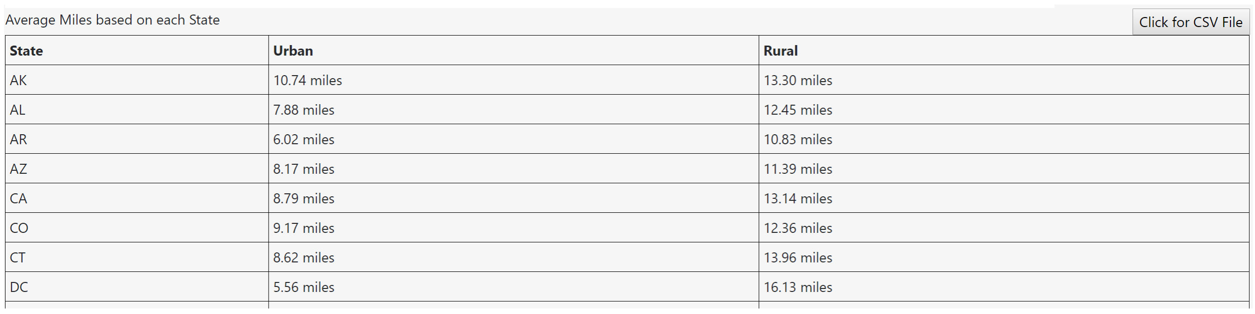
Urban-Rural

Average Trip Miles by State

Figure 4.6: *Graph of Average Trip Miles by State*

In the same result shown above, a table is also available just below the graph with a button to download the table data in CSV format (shown below) and the similar kind of option is available throughout the project just above every table. The table output of the same result is given in the Figure 4.7

The Restart button is given on every page in the project. The user can click this button



Average Miles based on each State

Click for CSV File

State	Urban	Rural
AK	10.74 miles	13.30 miles
AL	7.88 miles	12.45 miles
AR	6.02 miles	10.83 miles
AZ	8.17 miles	11.39 miles
CA	8.79 miles	13.14 miles
CO	9.17 miles	12.36 miles
CT	8.62 miles	13.96 miles
DC	5.56 miles	16.13 miles

Figure 4.7: *Table of Average Trip Miles by State*

and can reach the home page from every page of the project.

Now, lets say user selected the State option from the drop-down list under the Aggregated option. Then, the user will have different options available. The drop-down list under the State is given in the Figure 4.8

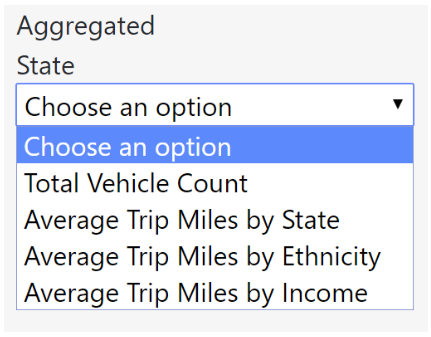
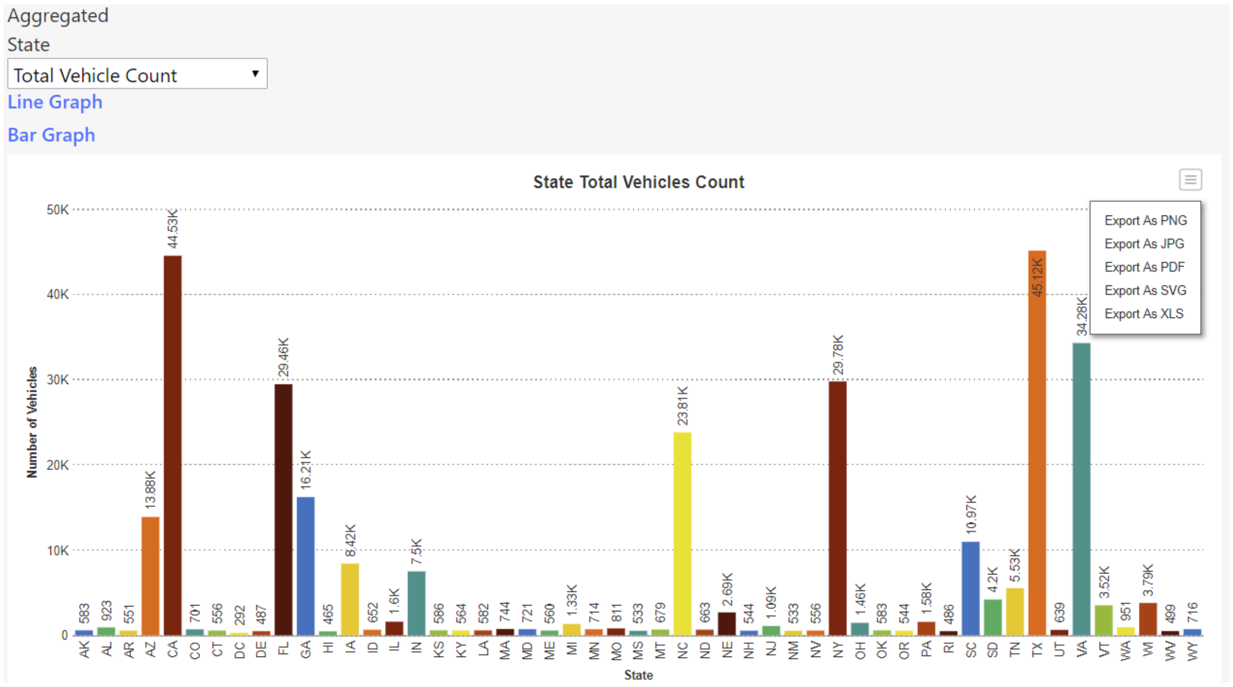

Aggregated
State
Choose an option
Choose an option
Total Vehicle Count
Average Trip Miles by State
Average Trip Miles by Ethnicity
Average Trip Miles by Income

Figure 4.8: *Drop-down Under the State*

Here if the user selects the first option i.e. Total Vehicle Count then just by selecting the form will submit and will take the user to the result page where there will be a table containing all the State codes with a number of vehicles in each state and both Line and Bar chart option available under a collapsible panel.

If the user clicks the Line graph, a line graph will be shown just below the Line graph link. Here, the user has options to download both the line and bar graph in various formats like PNG, PDF, XLS etc. The bar graph with its all download options is shown in the Figure 4.9

One more important thing that is done here from the user's perspective is that all the codes of income, race, and urban-rural are converted into their actual meaning while display-


Aggregated

State

Total Vehicle Count

Line Graph

Bar Graph

Figure 4.9: *Bar Graph of State Vehicle Count with Download options*

ing. What this means is that in the database ethnicity is not stored as a string instead it is stored as a numeric value representing a particular race and same is the case for trip purpose and income. Initially, when the data was retrieved user will have to look at the codebook in order to get a more vivid sense of data but now in the tool, I have created different python dictionaries to store codes as keys and the corresponding meaning of that code as a value. This dictionary is passed at the front end to display more meaningful data to the user. An example of the income code converted to the corresponding income range is shown in the Figure 4.10

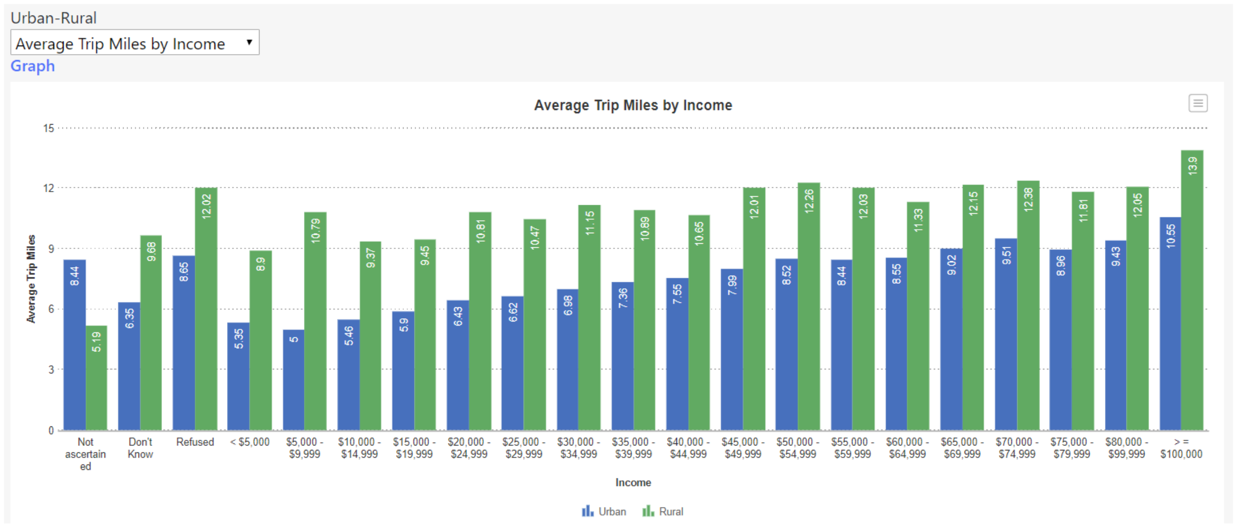


Figure 4.10: *Income Range in the x-axis using Python Dictionary*

Chapter 5

Technical Details of the Tool

5.1 Software Requirements of the Tool

Following are the basic system requirements to run the tool:

RAM - 2 GB or above.

Operating System - Any operating system compatible to run Django 2.0.2, PostgreSQL 9.4 and Python 3.4 or above.

Processor - Multi-core processor above 2.0gz(preferred) but can work with Single core 1.0gz or above.

Browser - Internet Explorer(preferred) but will work with other browsers as well.

Storage - 4 GB or above.

5.2 Installation Guide

The is an installation guide for Windows users, for other operating systems like OSX, Linux the below-listed software can be found online and can be installed in a similar way. Here is the list of all the required software you need to install in order to run the tool along with the links to help you install each of these:

Python Version 3.4

You need to download Python Version 3.4 or above which can be installed by using the following link: (van Rossum, 2018)

https://www.python.org/downloads/

Then you have to install Python which can be done easily by using the following link: https://www.ics.uci.edu/~pattis/common/handouts/pythoneclipsejava/python.html

Pip

Pip is package management system used to install and manage software packages written in Python. This comes with the 3.4 or higher version of Python. Just to run it on the Command line, make sure you add the path to Python34 in your systems environment variable which can be done using the following link:

https://dev.to/el_joft/installing-pip-on-windows

Django 2.0.2

Django Can be installed on windows using the following command at the command prompt:

```
- pip install Django
```

For the help below is the link with all the instruction regarding the installation of the Django version 2.0 required for running the tool: (DjangoSoftware, 2018)

https://docs.djangoproject.com/en/2.0/howto/windows/

To make Django work with our PostgreSQL database we need to install a package name psycopg2 2.7.3 or above which can be installed using pip just like we installed Django

```
- pip install psycopg2
```

PostgreSQL 9.6 or above

PostgreSQL 9.6 or above can be easily downloaded and installed by using the following link: (PostgreSQL, 2018)

https://www.enterprisedb.com/downloads/postgres-postgresql-downloads#windows

FusionCharts-suite-XT and Fusioncharts-jquery-plugin

Download the fusionCharts-suite-xt django wrapper and fusioncharts jquery plugin from the link attached below: (FusionCharts, 2018b)

https://www.fusioncharts.com/jquery-charts/

Place the FusionCharts library inside the "static/FusionCharts" folder in your project. An installation guide for the Fusioncharts wrappers is also available on GitHub and can be cloned using the below link: (FusionCharts, 2018a)

https://github.com/fusioncharts/django-wrapper

5.3 How to Run the Tool

These are the steps one need to follow in order to run the entire project after the proper installation of the above-mentioned tools.

i) The first step to run the project is to set up the data in the database. As PostgreSQL is used in the project, one needs to put the data in the database on a Postgres Database.

ii) One can create a database in PostgreSQL by simply typing the following command:

```
- create database nhts;
```

iii) After this, we need to create the tables to store the data in the database. These tables can be simply created by running the queries given in the createtable.txt file.

iv) At this point, tables are created in a database named nhts and we have to fill the tables with the data stored in the CSV files which can be done by the following command:

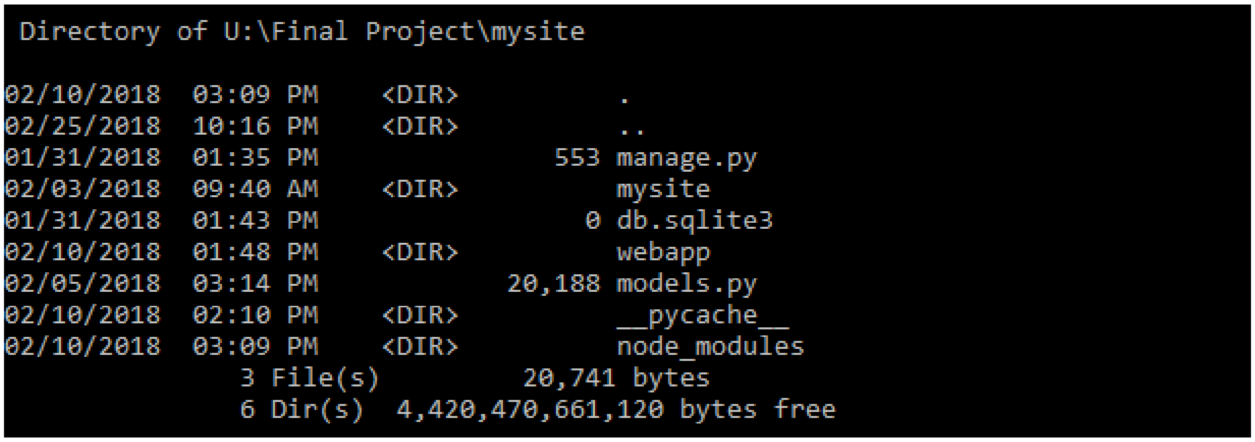
```
- \COPY hhv2pub FROM 'U:\Final Project\HHV2PUB.csv' DELIMITER ',' CSV ;
```

Where hhv2pub is the table name and the bold value is the path of the CSV file provided by the NHTS for this project. The same command can be repeated for the rest of the tables and this will populate the database.

v) Now, place the entire folder of the source code in a folder and then navigate to that folder in the command prompt. At this point, you should be inside the folder my site. Here, if you type command:

```
- dir
```

You should be able to see the following files at the command prompt as shown in Figure 5.1



```
 Directory of U:\Final Project\mysite

02/10/2018  03:09 PM    <DIR>          .
02/25/2018  10:16 PM    <DIR>          ..
01/31/2018  01:35 PM               553 manage.py
02/03/2018  09:40 AM    <DIR>          mysite
01/31/2018  01:43 PM                 0 db.sqlite3
02/10/2018  01:48 PM    <DIR>          webapp
02/05/2018  03:14 PM            20,188 models.py
02/10/2018  02:10 PM    <DIR>          __pycache__
02/10/2018  03:09 PM    <DIR>          node_modules
               3 File(s)         20,741 bytes
               6 Dir(s)  4,420,470,661,120 bytes free
```

Figure 5.1: *Screenshot of the Folder mysite*

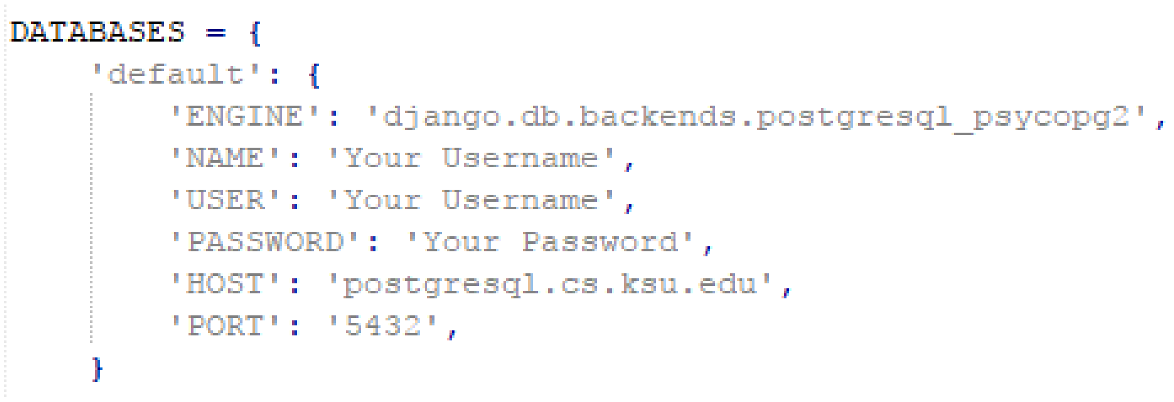
vi) As the database setup is done, next step is to connect the database with the Djangos built-in server.

vii) This can be done by putting your database credentials in a file called settings.py in the project under the folder /mysite/mysite/settings.py which will look like Figure 5.2 and the details can be filled likewise.

viii) After putting your credentials in the file /mysite/mysite/settings.py, run the following command in the command prompt, make sure you are currently in the folder of mysite which contains the file manage.py, to check if the database connection has been established

```
- python manage.py inspectdb
```

ix) Now, we need to make the models in the Django to access the data from the front end of the application which can be done simply by typing the following commands:



```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'Your Username',
        'USER': 'Your Username',
        'PASSWORD': 'Your Password',
        'HOST': 'postgresql.cs.ksu.edu',
        'PORT': '5432',
    }
```

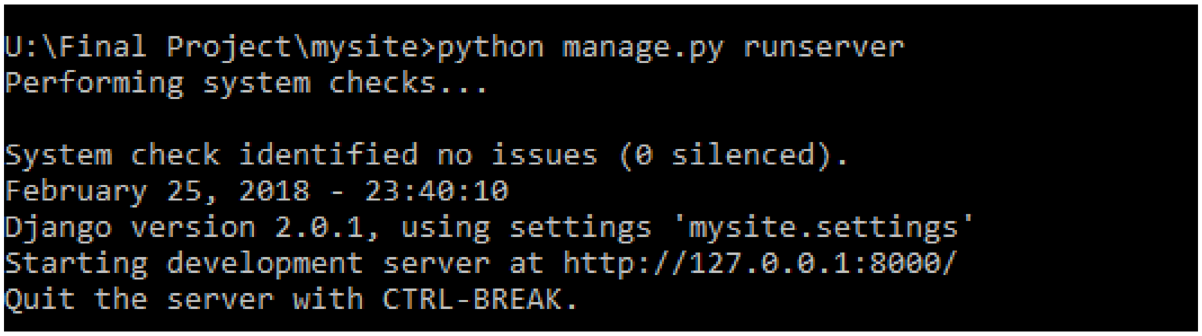
Figure 5.2: *Screenshot of the settings.py File*

```
- python manage.py migrate
- python manage.py makemigrations
```

x) At this point, you can start the tool by typing the following command:

```
- python manage.py runserver
```

The above command will start the Django server and will look something like Figure 5.3:



```
U:\Final Project\mysite>python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
February 25, 2018 - 23:40:10
Django version 2.0.1, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Figure 5.3: *Screenshot of the Server Running in the Command Prompt*

xi) Now, if you open your browser and type

```
- 127.0.0.1:8000/
```

It will start the tool in your browser.

Chapter 6

Implementation Details

In any Django-based application, the database connection is done by using the models.py file under the project folder. Once the connection to the database is established using models.py 5.2, one can start writing code to present information to users.

In Django, web pages and other content are delivered by views. Each view is represented by a simple Python function (or method, in the case of class-based views). Django will choose a view by examining the URL that is requested. More details can be found at (DjangoTutorial, 2018).

The first thing we need to do is to determine what information/resources we want to be able to display in our pages, and then define appropriate URLs for returning those resources. Then, we need to create the URL mapper, views, and templates to display those pages. Figure 6.1 shows the main flow of data and things that need to be implemented when handling an HTTP request/response. As we've already created the model, the main things that we will need to create are:

- URL mappers to forward the supported URLs (and any information encoded in the URLs) to the appropriate view functions.
- View functions to get the requested data from the models, create an HTML page displaying the data, and return it to the user to view in the browser.

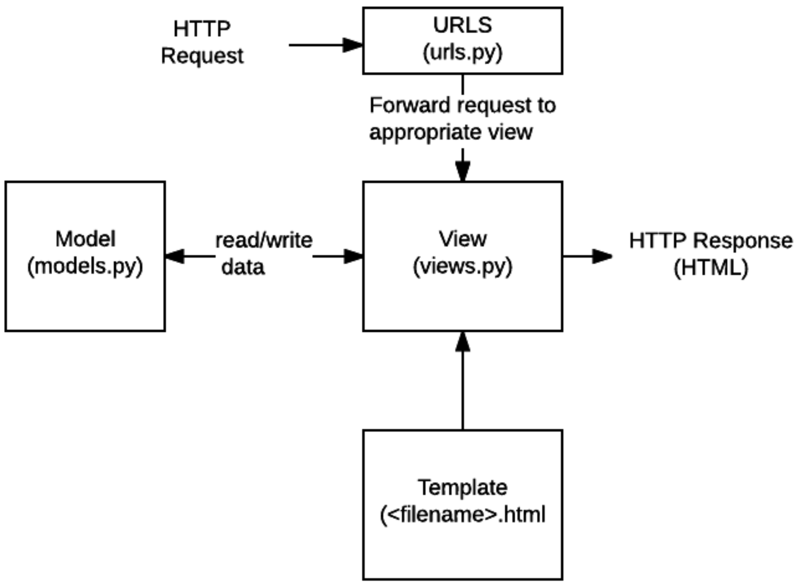


Figure 6.1: *Django Request Flow*

- Templates to be used by the views to render the data.

Let's take an example to illustrate the entire implementation of one of the functions from our tool. Let's say we want to query the data about the average time spent on a trip in each state, and render the data into an HTML file. This can be done with the following steps:

- First, we need to create the URL pattern to uniquely identify the requested page, which can be done by using regex in urls.py file in our application under our project directory as:

  ```
  urlpatterns = [ url(r'^nonquery/state/stateaction', views.stateaction,
  name='stateaction')]
  ```

  By adding the above URL in our urls.py file, we make sure that whenever the URL beginning with 'nonquery/state/stateaction' will be requested, from the running Django server, a Python function named 'stateaction' will be called from our views.py file, as it is linked in this urls.py file.

- At this point, we need to define the stateaction function in the views.py file. This stateaction function will query the database using the models.py, will put the data into a list and will pass this list of data to an HTML page to render the data and represent in the desired way. The code for this function will look like:

```
def stateaction(request):
    if request.method=='POST':
        selected = request.POST.get('state', None)
        if selected=="avgtriptimebystate":
            try:
                urbrur1 = Dayv2Pub.objects.values('hhstate').
                  annotate(trvlcmin__avg=Round(Avg('trvlcmin'))).order_by('hhstate'
                result=[]
                for p in urbrur1:
                    result.append(p)
                return render(request, 'webapp/avgtriptimebystate.html',
                  {"result":result})
            except:
                return HttpResponse("No data available")
        else:
            return render(request, 'webapp/state.html')
        return HttpResponse("Reached POST request State")
```

Now, lets break each line of the above code to understand the use of every single line in the stateaction function. In the project, every request to the server is sent using POST method and that is the reason an *if* condition, is used to check the request==POST for security reasons. After this, we have the below line which basically means that we are extracting the value of the selected option of the drop-downlist name 'state' which is present on the page from which the request has been sent:

```
        selected = request.POST.get('state', None)
```

The next step is to compare the selected option with the appropriate option in order to proceed further accordingly and hence we need another *if* condition:

```
            if selected=="avgtriptimebystate":
```

At this point, we have checked that the requested data by the user represents the average time spent on a trip in each state.

The next step is to fetch the requested data out of the database using the Django model. The data requested is stored in the Dayv2Pub table in our database and therefore the query is written using the Dayv2Pub class from our model in the try block in our sample code like shown below:

```
              try:
                    urbrur1 = Dayv2Pub.objects.values('hhstate').
                              annotate(trvlcmin__avg=Round(Avg('trvlcmin'))).
                              order_by('hhstate')
```

Here, *annotate* is a built-in function and can be used to calculate the average, max, min, etc. The 'trvlcminavg' is an alias for the Average of the 'trvlcmin' field present in the class Dayv2Pub representing the calculated travel time of a trip. The Round here is a class with a function defined in the views.py to round the result of the 'Avg' up to two decimal places like:

```
class Round(Func):
    function = 'ROUND'
    template='%(function)s(%(expressions)s, 2)'
```

The above Django query will return a list of objects each consisting of a dictionary with two pairs:

– hhstate (column name) as key and state name as the value

  - trvlcminavg as key and rounded decimal average trip time as the value

The loop after this query will fetch each object and will append it to a list created with the name result in the mentioned code as:

```
result=[]
for p in urbrur1:
    result.append(p)
```

This list will be sent to render to an HTML page that will be used to represent the data to the user by using:

```
return render(request, 'webapp/avgtriptimebystate.html', {"result":result})
```

- After fetching the required data from the database and sending it to the appropriate HTML page, the next step is to create the HTML file to process the list sent by the function in the views.py. In Django, Python code can be embedded into HTML pages using specific syntax. An example is given below:

```
<table style="width:100%" id="dataTable" name="dataTable">
<tr>
<th>State</th>
<th>Average Trip Time(in min)</th>
</tr>
{% for j in result %}
<tr>
<td>{{j.hhstate}}</td>
<td>{{j.trpmiles__avg}} miles</td>
</tr>
{% endfor %}
</table>
```

The above code can be placed with other HTML code in order to generate the table with States in one column and average trip time in the second. A loop is used here, which will read each of the list entries is like:

```
{% for j in result %}
<tr>
<td>{{j.hhstate}}</td>
<td>{{j.trpmiles__avg}} miles</td>
</tr>
{% endfor %}
```

The tr, td are the HTML tags used to create the table and the loop syntax is very Django specific, to process the Python list in an HTML page.

In sum, the above steps are being followed in order to create each and every page requested by the user and can be followed as an example to present more data to the user.

Chapter 7

Future Expansions and Conclusions

7.1 Future Expansions

There are many different aspects, tests, and functionalities remaining here which can be developed in the future. Future work concerns building a more dynamic mechanism to query the database and building a tool to display the data in even more possible ways. This tool can become a guideline for anyone who wants to develop more functions for querying the NHTS datasets. A better understanding of the trips and tours made by each individual household can also be achieved the future. Check-boxes for differentiating the data based on each state, race or income group can be implemented to get a deeper understanding of the data.

There is an umpteen number of attributes available for each table, attributes which can be presented in the front end for the user to analyze the datasets on various aspects, which are not used in this tool because of the lack of time. A couple of such attributes are VEHCOUNT (Vehicle count in each household), MAKECODE (Vehicle make code), ODREAD (odometer reading), VEHAGE (vehicle age) etc which can be used to analyze things like most preferred vehicle brand, make whose vehicles are most used by the people from different states, etc. There is also room for improvement in the design and flow of the project when there will be a lot more options available than what is currently existing. Future work can also be done in the direction of applying some algorithm to get a bigger picture out of the entire travel survey data, as the dataset is really large and consists of lots of important attributes.

7.2 Conclusions

The tool that is developed during this project successfully retrieves the data from the database, and plots the desired graphs to visualize the data. The tool is tested and is working according to the requirements of the user. Various data visualization options are given to the user and also the data is presented in the appropriate tables along with the downloading option available for each table. In sum, a basic tool is developed for the end user during this project which can be the basis for the future programmers to develop more constructive functionalities for the end user.

Bibliography

DjangoSoftware (2018). Django 2.0. https://www.djangoproject.com/download/.

DjangoTutorial (2018). Django-Tutorial. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Home_page.

FusionCharts (2018a). FusionCharts-Django Wrapper. https://www.fusioncharts.com/django-charts/.

FusionCharts (2018b). FusionCharts-Jquery-Plugin. https://www.fusioncharts.com/jquery-charts/.

Newmark, G. L. (2014). Conducting visitor travel surveys: A transit agency perspective. *Journal of Public Transportation*, 17:136–156.

Newmark, G. L. and Plaut, P. O. (2005). Shopping trip-chaining behavior at malls in a transitional economy. *Transportation Research Record*, 1939:174–183.

PostgreSQL (2018). PostgreSQL 9.6. https://www.postgresql.org/download/windows/.

van Rossum, G. (2018). Python 3.4. https://www.python.org/downloads/.