Mama Cass., formerly of the Redis Tributers
(Evan DePosit & Chad Tolleson)
CS 488/588 Cloud & Cluster Data Management - Fall 2019 Quarter
Project Part 2: Data Modeling and Application Design

Data Set: We selected Option 2: Freeway Data for our data set.

Option 2: Freeway Data (Relational Data)

This option focuses around freeway speed data and calculating travel times. The data set is relational.

Data Modeling & Application Design

In the second part of the project, teams design the storage/modeling of the provided data and designs how they will implement the questions provided for the selected data set. All for the system the team profiled in Task 1.

Meeting with Prof. Tufte happened at 9am on Tuesday November 12, 2019.

- Using the provided example data, explain how you store the data structures in your data store.
    - Show how the example data can be represented in the data model of your data store. Mention any indexes you will create.
      Cassandra is the preeminent column-family store type database. As such, we have the opportunity to modify the structure of the data from the provided csv files into a form that takes advantage of the column family and, more importantly, the super-column family data structures inherent to Cassandra. The key benefit of using column and super-column families is that the relevant data is assuredly stored In close physical proximity within the Cassandra system. We decided that of the four key csv files, we only need to utilize the contents of: freeway_stations.csv, freeway_detectors.csv, and freeway_loopdata.csv. We will henceforth refer to these three csv files as stations, detectors, and loopdata, respectively. We predict that most of the data querying needed to answer the four questions in part 3 will come from loopdata, with referential data points pulled from detectors. Since the loopdata is directly tied to the detectors using the detectorid, we decided to create a super-column family of the detectors and loopdata combined. In this super-column family, the detailed loopdata data points are contained within a column family that is itself a column within the detector data. This makes the detector / loopdata combined data structure a super-column family. I considered referring to this data structure as the brady_bunch, but will resist. Currently the only index

on the super-column family is the key, which is detectorid. We will likely add additional indexes as we solve the answers. The data contained within stations will be referenced much less frequently and is trivial compared to the detectors super-column family. So stations was imported as a regular column family, which is similar to a table in a RDBMS. The index for this column family is also the key, which is the stationed.

- Present at least one variant you considered during the design phase.
  The most obvious data structure variant we considered during the design phase is a direct representation of the csv files as column families in the Cassandra system. This means that the detectors and loopdata would be imported as regular column families (tables) without using a super-column family structure. In fact, as a backup, we did indeed import these tables in such a manner within a separate keyspace on our Cassandra system.

- Explain why you believe your final design is the preferred one
  In the spirit of using our NoSQL database, more specifically our column family store type database, in the manner in which it is most unique and arguable intended the super-column family structure is probably the correct approach.

- Describe the process you will use to restructure the data to fit your model.
  Cassandra is not a suitable platform for performing detailed and complex queries. As such, the main querying against Cassandra will consist of selecting and retrieving the appropriate data using both the primary indexes and the yet to be defined secondary indexes. The data will then get transformed and analyzed within Python (we are connecting to the Cassandra implementation using a Python package called "pycassa" which is based on the deprecated Cassandra Thrift interface).

- Outline implementation strategies in pseudo-code based on the capabilities of your data store and the indexes you plan to define for all of the provided questions.

  1. Count high speeds: Find the number of speeds > 100 in the data set.
     We can calculate this by creating a secondary index on the speed column in the detector & loopdata super-column family. Then we just pull the speed column for the rows where the speed is greater than 100. Then we find the count of rows in the pulled dataset. Here is an example of calculating the count by creating a secondary index on the super-column family using pyscassa:

```
count = 0
speed_expr = create_index_expression(‘speed’,100,GT)
clause = create_index_clause([speed_expr])
for key, row in detectors.get_indexed_slices(clause):
      count += 1
print(count)
```

2. Volume: Find the total volume for the station Foster NB for Sept 21, 2011. **First we will pull the station row for Foster NB from the station column family. Using the stationid from the station record (alternatively we can use the locationtext of “Foster NB”, which is also contained within the detector & loop data super-column family), we will pull the volume columns from the relevant rows from the detector & loop data super-column family by creating a secondary index on the stationid (or locationtext, as mentioned above) and then getting the row/columns using code similar to the following:**

```
total_volume = 0
stat_expr = create_index_expression(‘stationid’,1047)
clause = create_index_clause([stat_expr])
for key, row in detectors.get_indexed_slices(clause):
      total_volume += row[‘volume’]
print(total_volume)
```

3. Single-Day Station Travel Times: Find travel time for station Foster NB for 5-minute intervals for Sept 22, 2011. Report travel time in seconds.

```
length=0
stationIDList= list of all station ID
for stationID in stationIDList:
   station = station_col_fam.get(stationID)
   if station['locationtext'] == Foster NB:
       length= station['length']

detectorIDList= list of of all detector ID's

fosterDetectorSpeeds=[]
for detectorID in detectorIDList:
   detector= detector_col_fam.get(detectorID,
             columns=['locationtext', 'speed'])
   if detector['locationtext'] == 'Foster NB':
       fosterDetectorSpeeds.append(detector['speed'])

# average travel time = length / avg speed
avgSpeed=0
for speed in fosterDetectorSpeeds:
   avgSpeed = avgSpeed + speed
avgSpeed = avgSpeed / fosterDetectorSpeeds.length()

travelTime= length / avgSpeed
```

4. Peak Period Travel Times: Find the average travel time for 7-9AM and 4-6PM on September 22, 2011 for station Foster NB. Report travel time in seconds.
   First off, we will leverage the secondary index on stationid created in question 2, above. Using this index we will get rows from the detector & loopdata super-column family to retrieve rows for stationid 1047. Furthermore we will create secondary indexes for the time frames specified. Using the two secondary indexes we will get the rows needed to calculate the average travel times. Here is some pseudo-code for pulling the relevant records.

```
See Code Appendix #4
```

5. Peak Period Travel Times: Find the average travel time for 7-9AM and 4-6PM on September 22, 2011 for the I-205 NB freeway. Report travel time in minutes.
   Mirroring the solution to question 4, above, we will get the relevant records from Cassandra using secondary indexes and then perform the analysis within Python. The main exception is that we will not limit our "get" expression by stationid, rather we will pull the entire dataset for highwayid 3, which represents the NB freeway.

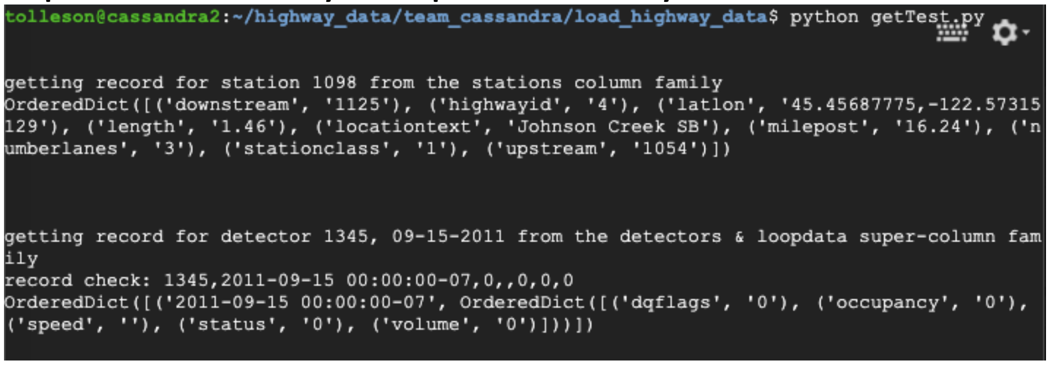
```
See Code Appendix #5
```

6. Route Finding: Find a route from Johnson Creek to Columbia Blvd on I-205 NB using the upstream and downstream fields.

```
stationIDList= list of all station ID's

stationRoute=[]
prevStation={}
for stationid in stationIDList:
   prevStation= station_col_fam.get(stationid)
   if station['locationtext'] == 'Johnson Creek NB':
       stationRoute.append(station['stationid'])
       break

for stationid stationIDList:
   station= statation_col_fam.get(stationid)

   if station['downstream'] == prevStation['stationid']:
       stationRoute.append(station['stationid'])

if station['downstream'] == prevStation['stationid'] and
            station['locationtext'] == 'columbia blvd':
       stationRoute.append(station['stationid'])
       break
```

- Include proof that you have loaded at least a few data items into your system.

The below screenshot is evidence that all three csv files are loaded into their respective column family and super-column family.



```
tolleson@cassandra2:~/highway_data/team_cassandra/load_highway_data$ python getTest.py


getting record for station 1098 from the stations column family
OrderedDict([('downstream', '1125'), ('highwayid', '4'), ('latlon', '45.45687775,-122.57315
129'), ('length', '1.46'), ('locationtext', 'Johnson Creek SB'), ('milepost', '16.24'), ('n
umberlanes', '3'), ('stationclass', '1'), ('upstream', '1054')])



getting record for detector 1345, 09-15-2011 from the detectors & loopdata super-column fam
ily
record check: 1345,2011-09-15 00:00:00-07,0,,0,0,0
OrderedDict([('2011-09-15 00:00:00-07', OrderedDict([('dqflags', '0'), ('occupancy', '0'),
('speed', ''), ('status', '0'), ('volume', '0')]))])
```

- Discuss how your design could handle updates to the data.
  One of the main strengths of the Cassandra system is the ability to quickly and easily ingest data. In fact, in a real implementation and usage of Cassandra with the highway/freeway data, Cassandra would likely be chosen as the system to initially receive and store the data from the detectors. The Cassandra system can handle multiple points of input and can coordinate the writing of data across the cluster. More specifically, the detectors would likely communicate with the computers at each station, and then the station computers would each connect to the Cassandra cluster to send the detector readings every 20 seconds. The Cassandra node that responds to the station computer first will be the write coordinator. The coordinating node knows which of the nodes in the cluster are the primary and secondary stores to which the station's data should be written. The coordinating node then sends the write commands to the primary and secondary storage nodes and waits to hear back from the writing node(s) (this depends on the consistency settings in the Cassandra system – we could make the coordinator wait for only one confirmation, wait for a quorum of confirmations, or wait for all nodes to confirm the writes). The availability of these updates then also depends on the number of reads responses required by the coordinating node before producing the data to the client.

Code Appendix #4

```
stat_expr = create_index_expression(‘stationid’,1047)
morn_start_expr = create_index_expression(‘starttime’, ‘2011-09-22 07:00:00-07’,GT)
morn_end_expr = create_index_expression(‘starttime’, ‘2011-09-12 09:00:20-07’, LT)
eve_start_expr = create_index_expression(‘starttime’, ‘2011-09-22 16:00:00-07’,GT)
eve_end_expr = create_index_expression(‘starttime’, ‘2011-09-12 18:00:20-07’, LT)
morn_clause = create_index_clause([stat_expr],[morn_start_expr],[morn_end_expr])
eve_clause = create_index_clause([stat_expr],[eve_start_expr],[eve_end_expr])

length = 0.0
length = station_col_fam.get(1047, columns=[‘length’]

morn_count = 0
morn_avg_speed = 0.0
morn_avg_travel_time = 0.0
for key, row in detectors.get_indexed_slices(morn_clause):
      morn_avg_speed += row[‘speed’]
      morn_count += 1
morn_avg_speed /= morn_count
morn_avg_travel_time = length / morn_avg_speed * 60 * 60
print(morn_avg_travel_time)

eve_count = 0
eve_avg_speed = 0.0
eve_avg_travel_time = 0.0
for key, row in detectors.get_indexed_slices(eve_clause):
      eve_avg_speed += row[‘speed’]
      eve_count += 1
eve_avg_speed /= eve_count
eve_avg_travel_time = length / eve_avg_speed * 60 * 60
print(eve_avg_travel_time)
```

Code Appendix #5

```
hwy_expr = create_index_expression(‘highwayid’,3)
morn_start_expr = create_index_expression(‘starttime’, ‘2011-09-22 07:00:00-07’,GT)
morn_end_expr = create_index_expression(‘starttime’, ‘2011-09-12 09:00:20-07’, LT)
eve_start_expr = create_index_expression(‘starttime’, ‘2011-09-22 16:00:00-07’,GT)
eve_end_expr = create_index_expression(‘starttime’, ‘2011-09-12 18:00:20-07’, LT)
morn_clause = create_index_clause([hwy_expr],[morn_start_expr],[morn_end_expr])
eve_clause = create_index_clause([hwy_expr],[eve_start_expr],[eve_end_expr])

length = 0.0
stationIDList= list of all station ID's with highway == 3
for stationid in stationIDList:
      length += station_col_fam.get(stationid, columns=[‘length’])

morn_count = 0
morn_avg_speed = 0.0
morn_avg_travel_time = 0.0
for key, row in detectors.get_indexed_slices(morn_clause):
      morn_avg_speed += row[‘speed’]
      morn_count += 1
morn_avg_speed /= morn_count
morn_avg_travel_time = length / morn_avg_speed * 60
print(morn_avg_travel_time)

eve_count = 0
eve_avg_speed = 0.0
eve_avg_travel_time = 0.0
for key, row in detectors.get_indexed_slices(eve_clause):
      eve_avg_speed += row[‘speed’]
      eve_count += 1
eve_avg_speed /= eve_count
eve_avg_travel_time = length / eve_avg_speed * 60
print(eve_avg_travel_time)
```