Item	Grade
Data model, loading, proof of loading	48/50
Implementation strategies & updates	48/50
Total Grade	96/100

96/100
I am sorry to hear that you had to implement the project twice using two different systems. To add insult to injury, I think Redis is not a perfect fit for the data set because of the need of range queries. You will have to implement those using code.

Tom Lancaster
William Mass
Trina Rutz
CS588 Cloud and Cluster Databases
11/9/19

The Redis Tributers - Assignment 2

After substantial deliberation we have gone back to having two seperate implementations—and two separate reports. Due to some substantial issues with getting Cassandra to work through an interface other than CQL and the growing concern that we wouldn't have anything implemented by today to report on, three of us elected to implement a Redis database, on the assumption that we would have fewer implementation problems than we had with Cassandra.

Data set: We are still using the Freeway Data set, in hopes of comparing the Redis results with those of Cassandra.

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 indices you will create.

At its core, Redis—short for Remote Dictionary Server—is both a key-value store and a cache. The data is always read and modified from the computer's main memory, but stored on disk in a format that's only readable to load the data back into memory once the system restarts. While loaded into the memory, it functions as any similar key-value pairing, and stores data as follows (with one example provided for each of the tables in the Freeway Data Set):

freeway_detectors.csv

d1345 : (*highwayid*: 3, *milepost*: 14.32, *locationtext*: Sunnyside NB, *detectorclass*: 1, *lanenumber*: 1, *stationid*: 1045)

freeway_loopdata.csv

1345_09-15: (*volume*: 0, *speed*: 'null', *occupancy*: 0, *status*: 0, *dqflags*: 0)

freeway_stations.csv

s1045: (*highwayid*: 3, *milepost*: 14.32, *locationtext*: Sunnyside NB, *upstream*: 0, *downstream*: 1046, *stationclass*: 1, *numberlane*: 4, *latlon*: 45.43324,-122.565775, *length*: 0.94)

highways.csv

h3: (*shortdirection*: N, *direction*: NORTH, *highwayname*: I-205)

no discussion of partioning/sharding

The format for each of these rows is key:(field: value), and our data is indexed on the keys.

Present at least one variant you considered during the design phase.

We had a long discussion while working on Cassandra (where indexing and data structures are both extremely important to get right) on how best to organize the data, but most of the methods of organizing it for Cassandra weren't ideally suited for Redis due to the differing data models they incorporate into their data store. While Tom and Trina quickly came up with the method we ended up implementing a few other ideas were considered. Initially, ordering all the files by station id was considered, but that would have produced the same key for multiple files, which was deemed potentially confusing. Detector id was eventually chosen, as it was able to provide a better index on which to access the values.

The final keys used in the four tables are as follows:

- freeway_detectors.csv – 'd' + detector_id
- freeway_loopdata.csv – 'detector_id + '_' + date (mm-dd format; given data over a longer time period, yy-mm-dd format would have been adopted)
- freeway_stations.csv – 's' + stationid
- highways.csv – 'h' + highway_id

you will need mm-dd-hh-mm-ss format or all detector readings in the same day will have the same key! you will end up storing only one reading (each insert will erase the previous data with the same key)

Explain why you believe your final design is the preferred one.

In concatenating distinguishable letters with the detector, station, and highway Ids in the smaller tables, we can see at a glance what each key represents. For each file, we chose those values which made the file unique, which is why the freeway_loopdata table also incorporates the date along with the detector id.

Describe the process you will use to restructure the data to fit your model.

We didn't have to! Redis has a simple data structure store, so it was easy to make it work with the data model as presented by the files. This alone is such an improvement over Cassandra that it merits mention.

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

1) ***Count high speeds:*** **Find the number of speeds > 100 in the data set.**

```
count = 0
for value in redis_db.hmget(detector_id, 'speed'):
        if (value is null or value < 100):
                continue
        else (value > 100):
                count++
```

```
        return count
```

2. ***Volume*: Find the total volume for the station Foster NB for Sept 21, 2011.**

```
foster_nb_id = redis_db.get('Foster NB', stationid)
detect_id_list = redis_db.get(foster_nb_id, 'detectorid')
loop_list = [ ]
total_volume = 0

for detect_id in stationid_list:
        detect_id_date = detect_id + "_09_21"
        loop_list.append(detect_id_date)

for key in loop_list:
volume = redis_db.get(key, volume)
total_volume += volume;

return total_volume
```

You will need to iterate by 20-second intervals

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.**

```
foster_nb_id = redis_db.get('Foster NB', stationid)
detect_id_list = redis_db.get(foster_nb_id, 'detectorid')
length = redis_db.get(foster_nb_id, 'milepost')
date_list = [ ]
speed_total = 0
speed_count = 0
cars_total = 0
five_min_data = [ ]
for detect_id in stationid_list:
        for (i = 0, i < (24 * 60), i++) // Total minutes that gets converted to time
                time = minutes_to_hours(i)
                detect_id_date = detect_id + "_09-22" + time
                date_list.append(detect_id_date)

for key in date_list:
```

```
        loop for every 5 minute interval
speed = redis_db.get(key, 'speed')
                num_cars = redis_db.get(ket, 'occupancy')
        speed_total += speed
        cars_total += num_cars
        avg_speed = speed_total/cars_total

        travel_time_secs = (length / avg_speed) * 3600
        five_min_data.append(trave_time_secs)

return five_min_data
```

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.**

```
foster_nb_id = redis_db.get('Foster NB', stationid)
detect_id_list = redis_db.get(foster_nb_id, 'detectorid')
length = redis_db.get(foster_nb_id, 'milepost')
morning_list = [ ]
evening_list = [ ]
morning_times = [ ]
evening_times = [ ]
for detect_id in stationid_list:

while (i = 0, i < (24 * 60))
        if ((i ≥ 420 && i ≤ 540)
                time = minutes_to_seconds(i)
                detect_id_date = detect_id + "_09-22" + time
                morning_list.append(detect_id_date)
                i++
        else if  (i ≥ 960 && i ≤ 1080))
                time = minutes_to_seconds(i)
                detect_id_date = detect_id + "_09-22" + time
                evening_list.append(detect_id_date)
                i++
```

```
for key in morning_list:
        speed = redis_db.get(key, 'speed')
                num_cars = redis_db.get(ket, 'occupancy')
        speed_total += speed
        cars_total += num_cars
        avg_speed = speed_total/cars_total

        travel_time_secs = (length / avg_speed) * 3600
        morning_times.append(trave_time_secs)

for key in evening_list:
        speed = redis_db.get(key, 'speed')
                num_cars = redis_db.get(ket, 'occupancy')
        speed_total += speed
        cars_total += num_cars
        avg_speed = speed_total/cars_total

        travel_time_secs = (length / avg_speed) * 3600
        evening_times.append(trave_time_secs)

return morning_times, evening_times
```

this is not overall travel time?

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.**

```
station_id_keys = redis_db.keys('s*')
detect_id_list = [ ]
length = redis_db.get(foster_nb_id, 'milepost')
morning_list = [ ]
evening_list = [ ]
morning_times = [ ]
evening_times = [ ]
for key in station_id_keys:
det_id = redis_db.get(key, 'detectorid')
detect_id_list.append(det_id)
for detect_id in detect_id_list:
```

```
while (i = 0, i < (24 * 60))
if ((i ≥ 420 && i ≤ 540)
                time = minutes_to_seconds(i)
                detect_id_date = detect_id + "_09-22" + time
                morning_list.append(detect_id_date)
                i++
        else if  (i ≥ 960 && i ≤ 1080))
                time = minutes_to_seconds(i)
                detect_id_date = detect_id + "_09-22" + time
                evening_list.append(detect_id_date)
                i++

for key in morning_list:
        speed = redis_db.get(key, 'speed')
                num_cars = redis_db.get(ket, 'occupancy')
        speed_total += speed
        cars_total += num_cars
        avg_speed = speed_total/cars_total

        travel_time_secs = (length / avg_speed) * 60
        morning_times.append(trave_time_secs)

for key in evening_list:
        speed = redis_db.get(key, 'speed')
                num_cars = redis_db.get(ket, 'occupancy')
        speed_total += speed
        cars_total += num_cars
        avg_speed = speed_total/cars_total

        travel_time_secs = (length / avg_speed) * 60
        evening_times.append(trave_time_secs)

return morning_times, evening_times
```

you do not seem to compute travel time for each station separately. This is necessary, computing travel time by just averaging all speeds of all stations will generate an incorrect travel time.

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

```
johnson = 'Johnson Creek NB'
columbia = 'Columbia to I-205 NB'
stationid_keys = redis_db.keys('s*')
johnson_id = redis_db.get(johnson)
johnson_milepost = redis_db.get(johnson_id, ‘milepost’)
columbia_id = redis_db.get(columbia)
columbia_milepost = redis_db.(columbia_id, ‘milepost’)
northbound_rows = []
southbound_rows = []

#Divides station information by north and south bound
for key in stationid_keys:
if (redis_db.get(key, ‘highwayid’) == 3):
row = redis_db.hgetall(key)
northbound_rows.append(row)
else
row = redis_db.hgetall(key)
southbound_rows.append(row)

north = false
south = false
if johnson_milepost > columbia_milepost:
        north = true
else:
        south = true

#Algorithm to get route from Johnson to Columbia works in reverse--so find route from
#columbia to johnson, then reverse it

route = []
route.append([(columbia_id == row['stationid]) for row in n_or_sbound_rows])
```

```
if north:
        route = get_route(columbia_id, johnson_id, northbound_rows, route)
else:
        route = get_route(columbia_id, johnson_id, southbound_rows, route)

#prints location text for route from johnson to columbia
for row in reversed(route):
        print(row['locationtext')])
#algorithm to return route from johnson to columbia
def get_route(start_id, end_id, n_or_sbound_rows, route):
if start_id == end_id:
route.append([(end_id == row['stationid']) for row in n_or_sbound_rows])
return

for row in n_or_s_bound_rows:
                if row['downstream'] == start_id:
                route.append(row['locationtext'])
                get_route(row, end_id, n_or_sbound_rows)
                break
        return route
```

Prove that a few data items have been loaded into the system:

The *hgetall* command shows all the fields and values within a specific key in Redis. In this example, it is used to look up detectorid 'd1950' within the redis-cli.

```
127.0.0.1:6379> hgetall d1950
 1) "detectorid"
 2) "1950"
 3) "detectorclass"
 4) "1"
 5) "milepost"
 6) "21.12"
 7) "highwayid"
 8) "3"
 9) "stationid"
10) "1142"
11) "locationtext"
12) "I-205 NB at Glisan"
13) "lanenumber"
14) "2"
```

The *keys* command (which is included in Redis primarily as a debug command, and which isn't typically used to iterate over the data in tables) shows all the keys in the database. Each key has fields and values similar to those displayed in the photo above.

```
127.0.0.1:6379> keys *
 1) "d1950"
 2) "d1730"
 3) "d1370"
 4) "d1858"
 5) "d1953"
 6) "d1856"
 7) "d1857"
 8) "h3"
 9) "s1054"
10) "d1811"
11) "s1143"
12) "d1354"
13) "s1049"
14) "d1942"
15) "s1051"
16) "d1411"
17) "s1117"
18) "s1098"
19) "d1363"
20) "d1385"
21) "d1946"
22) "d1954"
23) "d1941"
24) "d1347"
25) "s1050"
26) "d1379"
27) "d1403"
28) "d1387"
29) "d1955"
30) "d1378"
31) "s1045"
32) "s1140"
33) "s1052"
34) "d1401"
35) "d1393"
36) "d1402"
37) "d1951"
38) "s1053"
39) "d1345"
40) "d1809"
```

Discuss how the design could handle updates to the data.

We can handle updates to our data by using our Python script to upload the new file of data into our database. Any new updates to our keys will overwrite the old data (assuming that the keys match). It should be noted that this isn't necessarily true for the loopdata file, as the Python script we have used to ingest the data uses a Redis command which will append to a key if it already exists, rather than replacing it outright. To edit that file, we would need to run a Redis delete command on the keys found in loopdata, prior to running a seperate command to load the data back in. This (and other reasons, including the nature of complex querying in Redis) make it seem like an ideal candidate for lua scripting.