Bases de Datos No Estructuradas

UNAM— Semestre 2022-2

Integrantes:
Andrés Urbano Guillermo Gerardo
Avitúa Varela Fernando
Santa Rita Vizuet Fernando

§ Bases de Datos de Documentos con MongoDB §

Introducción

MongoDB es un sistema de base de datos NoSQL, orientado a documentos y de código abierto. En lugar de guardar los datos en tablas, tal y como se hace en las bases de datos relacionales, MongoDB guarda estructuras de datos BSON (una especificación similar a JSON) con un esquema dinámico, haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida.

Todo el código del proyecto se puede encontrar en el siguiente proyecto de Deepnote

MongoDB tiene las siguientes características:

- *Consultas ad hoc:* MongoDB soporta la búsqueda por campos, consultas de rangos y expresiones regulares. Las consultas pueden devolver un campo específico del documento pero también puede ser una función definida por el usuario para su mejor ocupación.
- *Indexación:* Cualquier campo en un documento de MongoDB puede ser indexado, al igual que es posible hacer índices secundarios. El concepto de índices en MongoDB es similar al empleado en base de datos relacionales.
- *Replicación:* MongoDB soporta el tipo de replicación primario-secundario. Cada grupo de primario y sus secundarios se denomina replica. El primario puede ejecutar comandos de lectura y escritura. Los secundarios replican los datos del primario y sólo se pueden usar para lectura o para copia de seguridad, pero no se pueden realizar escrituras. Los secundarios tienen la habilidad de poder elegir un nuevo primario en caso de que el primario actual deje de responder.
- *Balanceo de carga:* MongoDB puede escalar de forma horizontal usando el concepto de shard. El desarrollador elige una clave de sharding, la cual determina cómo serán distribuidos los datos de una colección. Los datos son divididos en rangos (basado en la clave de sharding) y distribuidos a través de múltiples shard. Cada shard puede ser una réplica set. MongoDB tiene la capacidad de ejecutarse en múltiple servidores, balanceando la carga y/o replicando los datos para poder mantener el sistema funcionando en caso de que exista un fallo de hardware. La configuración automática es fácil de implementar bajo MongoDB y se pueden agregar nuevas servidores a MongoDB con el sistema de base de datos funcionando.
- *Almacenamiento de archivos:* MongoDB puede ser utilizado como un sistema de archivos, aprovechando la capacidad de MongoDB para el balanceo de carga y la replicación de datos en múltiples servidores. Esta funcionalidad, llamada GridFS e incluida en la distribución oficial, implementa sobre los drivers, no sobre el servidor, una serie de funciones y métodos para manipular archivos y contenido. En un sistema con múltiple servidores, los archivos pueden ser distribuidos y replicados entre los mismos de forma transparente, creando así un sistema eficiente tolerante de fallos y con balanceo de carga.

Objetivo:

Implementar un sistema de pases en bicicleta que cumpla los siguientes requisitos:

1. Un usuario se pueda dar de alta junto con información de ubicación (coordenadas), debe usar al menos un lugar (casa por ejemplo) pero con opción de agregar más (trabajo, escuela, etc.)
2. El usuario debe poder buscar las estaciones de bicicleta que le queden más cerca a sus lugares.
3. También debe poder planear viajes, dado un tiempo que quiere viajar, el sistema debe recomendar viajes usando como salida sus estaciones mas cercanas (o estaciones específicas seleccionadas por el usuario) y los destinos que le tomen mas o menos ese tiempo.
4. El usuario debe poder dar la opción de que su viaje sea redondo (mismo punto de partida y salida). En este caso, solo se debe tomar en cuenta los datos que pasan por otras estaciones, a menos que el tiempo sea muy corto.

Implementación de la base de datos

Previo al guardado de los datos en la base de datos de Mongo, hacemos un preprocesamiento de estos mediante la librería Pandas, implementada en Python. El procesamiento sigue los siguientes pasos:

1. Concatenar todos los conjuntos de datos, es decir, crear un sólo conjunto de datos de los viajes en bicicleta de todos los años.
2. Obtener los datos no repetidos de las columnas de estaciones de bicicleta.
3. Construir le primer conjunto de datos con los nombres de estación, id de estación, latitud y longitud.
4. Emplear una función de agregación agrupando sobre las estaciones de origen y destino para obtener el promedio de viaje de una estación a otra, dando lugar al segundo conjunto de datos.

Una vista previa de ambos conjuntos de datos es la siguiente:

```
In [107]: df_estacion.head()
```

	id_estacion	nombre_estacion	lat	long
0	294	Washington Square E	40.730494	-73.995721
1	285	Broadway & E 14 St	40.734546	-73.990741
2	247	Perry St & Bleecker St	40.735354	-74.004831
3	357	E 11 St & Broadway	40.732618	-73.991580
4	401	Allen St & Rivington St	40.720196	-73.989978

```
In [108]: df_viajes.head()
```

	start_id	end_id	duracion_prom
0	72	72	1428.977867
1	72	79	1117.452555
2	72	82	1516.500000
3	72	83	923.157895
4	72	116	824.120000

Figure 1: Vista previa de los conjuntos de datos limpios antes de ser cargados a MongoDB.

Se presentan las instrucciones para el agregado de los datos limpios de las bicicletas:

- Inicializamos Mongo.

```
myclient = pymongo.MongoClient('mongodb://localhost:27017/')
mydb = myclient['mydatabase1']
```

- Creamos la colección de la base de datos de estaciones.

```
try:
  mydb.create_collection('Estaciones')
  estaciones = mydb.get_collection('Estaciones')
except:
  print('Ya se había creado la colección')
```

- Insertamos los datos en la colección creada. Creamos el índice geográfico. Esto nos permite hacer consultas con palabras clave como near o maxdistance que probarán ser muy útiles para resolver los problemas en cuestión.

```
mydb.Estaciones.create_index([("loc", GEO2D)])
dicc_estaciones = []

for row in df_estacion.iterrows():
    idd = int(row[1]["id_estacion"])
    lat, long = row[1]["lat"], row[1]["long"]
    nombre = row[1]['nombre_estacion']

    dicc = {
        "loc":[lat, long],
        "estacion_id": idd,
        "estacion_nombre":nombre
        }

    dicc_estaciones.append(dicc)

estaciones.insert_many(dicc_estaciones)
```

Ahora cargamos el conjunto de datos del tiempo de viaje promedio entre estaciones:

- Creamos la colección de la base de datos de estaciones.

```
try:
  mydb.create_collection('Tiempos')
  tiempos = mydb.get_collection('Tiempos')
except:
  print('Ya se había creado la colección')
```

- Insertamos los datos en la colección creada.

```
dicc_tiempos = []

for row in df_viajes.iterrows():
    inicio = int(row[1]['start_id'])
    fin = int(row[1]['end_id'])
    promedio = row[1]['duracion_prom']

    dicc = {
```

```
        "inicio":inicio,
        "fin": fin,
        "promedio":promedio
        }

    dicc_tiempos.append(dicc)

tiempos.insert_many(dicc_tiempos)
```

Aplicación

En esta sección se implementan los requerimientos establecidos en la tarea.

1) Usuarios

La creación de usuarios se implementó usando clases. A continuación se presenta el código del constructor, así como los métodos para agregar una dirección y agregar la entrada a la base de datos. Existe más funcionalidad para la clase, pero se mencionarán los métodos específicos en las siguientes secciones

```
class Usuario(object):
  def __init__(self,diccionario, propiedades={}):
    '''
    Constructor
    El diccionario tiene que tener al menos la llave nombre, lo demás es opcional
    '''
    if 'nombre' not in diccionario.keys():
      raise ValueError('El nombre es necesario')
    self.diccionario=diccionario

  def agrega_direccion(self,nombre,coordenadas,properties):
    '''
    nombre: nombre de la ubicación (casa, trabajo, etc)
    coordenadas: la coordenada de la ubicación
    propiedades: diccionario de propiedades del objeto {'lugar':'IIMAS'}
    '''
    geoJSON={
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [coordenadas[0], coordenadas[1]]
      },
      "properties": properties
    }

    if not self.diccionario.get('ubicaciones',False):
      self.diccionario['ubicaciones']={}

    self.diccionario['ubicaciones'][nombre]=geoJSON
    return self.diccionario

  def agrega_BD(self, coleccion):
    '''
    Método par agregar la instancia de usuario a la base de datos
    '''
```

```
        self.coleccion=coleccion
        return coleccion.insert_one(self.diccionario).inserted_id
```

A continuación se muestra el ejemplo de la creación de un usuario así como de su inserción en la base.

```
# Se crea la instancia
u1=Usuario({'nombre':'Fernando Avitúa'})
# Se asignan dos direcciones diferentes
u1.agrega_direccion('casa',[40.7192258,-73.9885016],{'lugar':'Paloma azul'})
u1.agrega_direccion('trabajo',[10,5],{'lugar':'IIMAS'})

# Se agrega a la colección de BD
usuarios=mydb.get_collection('usuarios')
u1.agrega_BD(usuarios)

# Se realiza el query para obtener el resultado
usuarios.find_one({'nombre':'Fernando Avitúa'})
```

Que tiene como resultado

```
{
    "_id": ObjectId ("62733b50525714cc52e421ab"),
    "nombre": "Fernando Avitúa",
    "ubicaciones": {
        "casa": {
            "type": "Feature",
            "geometry": {"type": "Point", "coordinates": [40.7192258, -73.9885016]},
            "properties": {"lugar": "Paloma azul"},
        },
        "trabajo": {
            "type": "Feature",
            "geometry": {"type": "Point", "coordinates": [10, 5]},
            "properties": {"lugar": "IIMAS"},
        },
    },
}
```

2) Estaciones más cercanas

Esta pregunta se implementó como método de cada usuario con el código que se muestra a continuación

```
  def busca_estacion_cercana(self,direccion='casa',dist_max=100,num_limite=5):
    lat,long=self.diccionario['ubicaciones'][direccion]['geometry']['coordinates']
    query = {"loc": SON([("$near", [lat,long] ), ("$maxDistance", dist_max)])}

    return list(estaciones.find(query).limit(num_limite))
```

Vemos las palabras clave $\$near$ y $\$maxDistance$ que hacen las consultas sumamente sencillas. Tenemos posibilidad de buscar estaciones que no superen una distancia corta o larga (usando el argumento $dist\_max$) y devolver el número de resultados (usando el argumento num_limite) resultados que nos sean útiles

Ejecutando este método con el usuario creado previamente, obtenemos

```
u1.busca_estacion_cercana("casa", num_limite=2)

>>
[
    {
        "_id": ObjectId("62733b4a525714cc52e42065"),
        "loc": [40.72019576, -73.98997825],
        "estacion_id": 401,
        "estacion_nombre": "Allen St & Rivington St",
    },
    {
        "_id": ObjectId("62733b4a525714cc52e4213a"),
        "loc": [40.7172274, -73.98802084],
        "estacion_id": 311,
        "estacion_nombre": "Norfolk St & Broome St",
    },
]
```

3) Planificación de viajes

Se desarrolla la siguiente función que tiene por objetivo trazar una ruta por las estaciones de bicicleta. El objetivo es recorrer la mayor cantidad de estaciones en el menor tiempo posible dado.

La idea detrás de este algoritmo versa en tomar el conjunto de datos de viajes promedio cargados en la colección *Tiempos*, pues este provee medidas buenas tiempo que podría tomar cualquier persona. La función debería tomar los siguientes argumentos:

- Nombre de usuario
- Estación de partida (coordenadas en concreto o alguno de los lugares preferido guardado en la base de datos de usuarios).
- Tiempo de viaje.

La función encarga de implementar la idea anterior es la siguiente.

```
def viaje(usuario, partida, tiempo):
    """
    Función encargada de crear una ruta para algún usuario.

    Parámetros:
        usuario: Persona en la base de datos que quiere un viaje.
        partida: Coordenadas de la forma [lat, long] o sitio preferido
                 guardado en la base de datos de usuario.
        tiempo:  Tiempo aproximado en segundos que le usuario desea pasar
                 en el viaje.

    Salida:
        Tiempo tomado entre estaciones.
        Gráfica de la ruta que seguirá el usuario.
    """
    ruta=None
    # verifica que exista el usuario y tenga más de 2 elementos
    try:
        atributos = mydb['usuarios'].find({'nombre':usuario})[0]
    except:
        return f'El usuario no existe'
```

```
22
23      # coordenadas del punto de partida
24      if type(partida) == list:
25          if len(partida) == 2:
26              coordenadas = partida
27          else:
28              return f'Introduce correctamente el punto de partida'
29
30      # el punto de partida es un lugar guardado por el usuario
31      elif type(partida) == str:
32          try:
33              coordenadas = atributos['ubicaciones'][partida]['geometry']['coordinates']
34          except:
35              return f'Este usuario no tiene el lugar guardado'
36
37      # buscamos la estación más cercana al punto de partida
38      try:
39          posibles = mydb.Estaciones.find({"loc": SON([("$near", coordenadas),
40                                                       ("$maxDistance", 0.01)])})
41          closest = posibles[0]
42      except:
43          return f'Las estaciones están muy lejos de tí'
44
45      # establecemos parámetros del punto de partida
46      tiempo_restante = tiempo
47      coord_actual = closest['loc']
48      est_nombre = closest['estacion_nombre']
49      id_actual = closest['estacion_id']
50
51      # diccionario donde se guardan las estaciones
52      estaciones_resultado = {id_actual:[coord_actual, est_nombre]}
53
54
55      if ruta == None:
56
57          # el ciclo while se encarga de seguir agregando estaciones al viaje
58          while tiempo_restante > 0:
59              new_closest = mydb.Estaciones.find({"loc": SON([("$near", coord_actual),
60                                                              ("$maxDistance", 0.5)])})
61
62              # busca el punto más cercano a la estación anterior sin repetir
63              for i in new_closest:
64                  if i['estacion_id'] in estaciones_resultado.keys():
65                      continue
66                  else:
67                      new_closest = i
68                      break
69
70              coord_actual = new_closest['loc']
71              new_closest_id = new_closest['estacion_id']
72              new_est_nombre = new_closest['estacion_nombre']
73              tiempo_tomado = mydb.Tiempos.find({'inicio':id_actual,
74                                                 'fin':new_closest_id})[0]['promedio']
75              print(f'Toma {np.round(tiempo_tomado,1)} segundos ir de "{estaciones_re...)
76              id_actual = new_closest_id
77              estaciones_resultado[id_actual] = [coord_actual, new_est_nombre]
78              tiempo_restante -= float(tiempo_tomado)
79
80      # coordenadas de las estaciones para graficar
```

```
    trayectoria_x = []
    trayectoria_y = []
    for key in estaciones_resultado:
        trayectoria_x.append(estaciones_resultado[key][0][0])
        trayectoria_y.append(estaciones_resultado[key][0][1])

    # gráfica interactiva con Plotly
    fig = px.scatter(df_estacion, x='lat', y='long')
    fig.add_trace(go.Scatter(x=trayectoria_x, y=trayectoria_y))
    fig.update_layout(template='plotly_dark', width=600)
    fig.write_image(f"{usuario}.pdf")
    fig.show()
```

La función anterior funciona sólo si el usuario está en la base de datos.

Supongamos el usuario *Fernando Santa Rita* solicita un viaje desde la posición $[40.689, -74]$ y con una duración de $10'000$ segundos (casi tres horas). El resutlado es el siguiente:

Toma 1552.5 segundos ir de "Atlantic Ave Furman St" a "Henry St Atlantic Ave"
Toma 412.6 segundos ir de "Henry St Atlantic Ave" a "Clinton St Joralemon St"
Toma 473.6 segundos ir de "Clinton St Joralemon St" a "Montague St Clinton St"
Toma 586.5 segundos ir de "Montague St Clinton St" a "Clinton St Tillary St"
Toma 1047.4 segundos ir de "Clinton St Tillary St" a "Cadman Plaza E Tillary St"
Toma 649.0 segundos ir de "Cadman Plaza E Tillary St" a "Jay St Tech Pl"
Toma 1052.0 segundos ir de "Jay St Tech Pl" a "Lawrence St Willoughby St"
Toma 483.1 segundos ir de "Lawrence St Willoughby St" a "Duffield St Willoughby St"
Toma 358.4 segundos ir de "Duffield St Willoughby St" a "Johnson St Gold St"
Toma 482.4 segundos ir de "Johnson St Gold St" a "Concord St Bridge St"
Toma 264.6 segundos ir de "Concord St Bridge St" a "Sands St Gold St"
Toma 2459.7 segundos ir de "Sands St Gold St" a "Front St Gold St"
Toma 197.9 segundos ir de "Front St Gold St" a "York St Jay St"

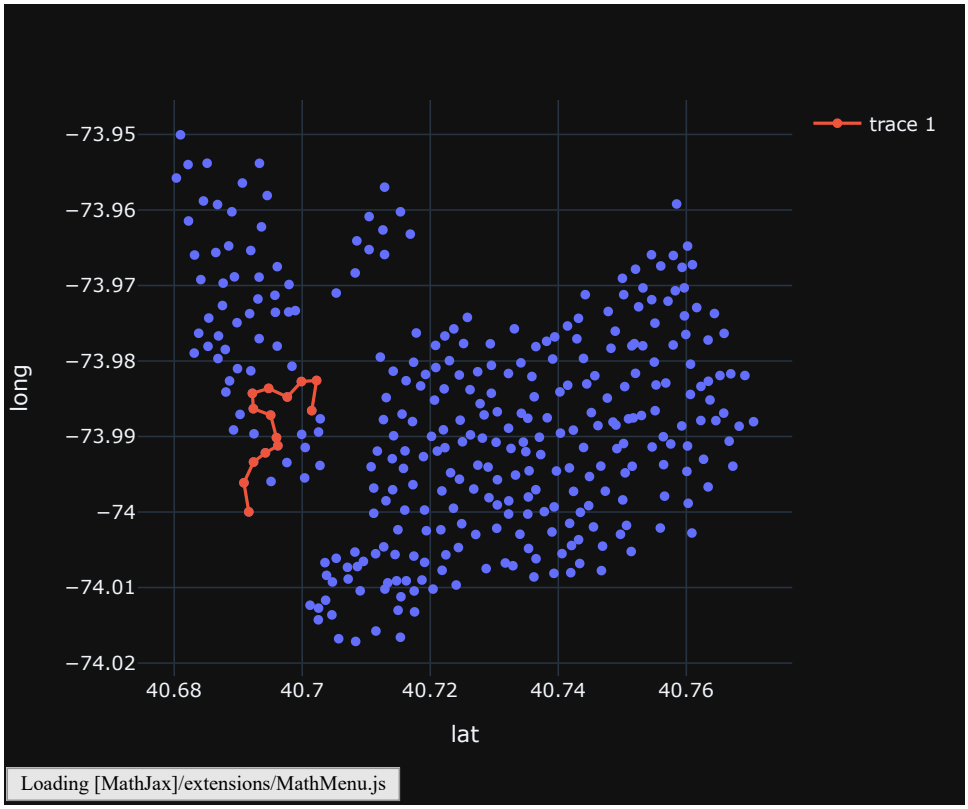

Figure 2: Viaje empleando la función `viaje` con usuario *Fernando Santa Rita* desde el punto $[40.689, -74]$ con una duración de aproximadamente 3 horas

El usuario dio una posición en específico en el mapa, pero también se puede hacer referencia a uno de los lugares guardados por el usuario. Supongamos el usuario *Guillermo Urbano* y desea hacer un viaje con sus amigos con una duración de 10′000 segundos. El resultado es el siguiente:

Toma 785.1 segundos ir de "E 55 St Lexington Ave" a "E 53 St Lexington Ave"
Toma 1582.0 segundos ir de "E 53 St Lexington Ave" a "E 51 St Lexington Ave"
Toma 1665.2 segundos ir de "E 51 St Lexington Ave" a "E 48 St 3 Ave"
Toma 1313.2 segundos ir de "E 48 St 3 Ave" a "E 47 St 2 Ave"
Toma 503.9 segundos ir de "E 47 St 2 Ave" a "E 45 St 3 Ave"
Toma 276.5 segundos ir de "E 45 St 3 Ave" a "E 43 St 2 Ave"
Toma 2122.8 segundos ir de "E 43 St 2 Ave" a "1 Ave E 44 St"
Toma 427.3 segundos ir de "1 Ave E 44 St" a "E 47 St 1 Ave"
Toma 258.5 segundos ir de "E 47 St 1 Ave" a "E 51 St 1 Ave"
Toma 1299.3 segundos ir de "E 51 St 1 Ave" a "E 52 St 2 Ave"

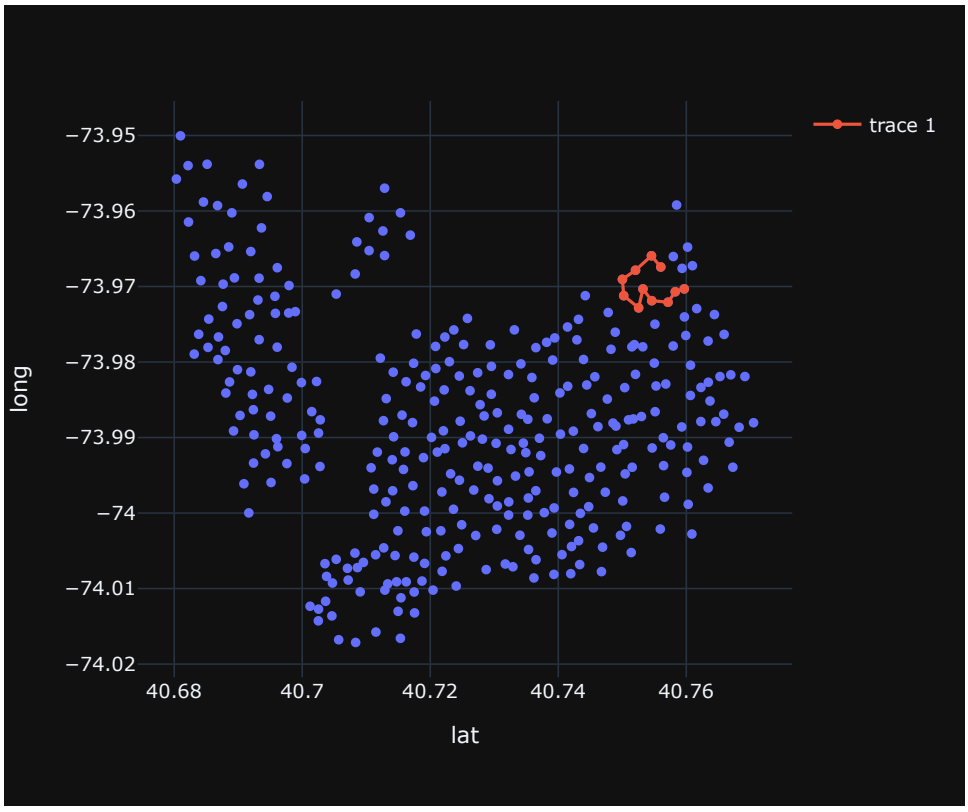

Figure 3: Viaje empleando la función `viaje` con usuario *Guillermo Urbano* desde su lugar guardado *escuela*, con una duración de aproximadamente 3 horas.

4) Viaje redondo

Esta pregunta de igual manera se implementó como método de usuario. El viaje redondo se interpretó como la ida y el regreso a una sola estación. Es decir, dada una ubicación, el método busca un número de estaciones cercanas (usando el método definido en la sección 2) y a partir de ellas busca estaciones tales que su tiempo total de recorrido de ida y de regreso sea menor o igual al tiempo establecido.

A continuación se muestra el código usado para esta parte

```
def encontrar_paseo(self,ubicacion,tiempo_limite=1000,resultados_limite=5
,dist_max=100,num_limite=5):

    # Lista de num_limite de estaciones cercanas
     lista_de_id_1=[dicc['estacion_id'] for dicc in
     self.busca_estacion_cercana(ubicacion,dist_max,num_limite)]
```

```
7
8       # Listas donde guardaremos las columnas de el df
9       id_intermedio=[] # El id al que se viaja en el paseo
10      id_inicial=[] # El id de la estaciona donde se comienza el viaje
11      duracion_total=[] # Duración total del paseo redondo
12
13      print(lista_de_id_1)
14      for id_1 in lista_de_id_1:
15
16        # Encontramos todos los viajes que están a menos del
17        # tiempo de distancia de las estaciones cercanas
18        primer_destino=viajes.find({'start_id':id_1
19        ,'duracion_prom':{'$lt':tiempo_limite}})
20
21        # Calculamos el tiempo y los destinos que están a menos del
22        # tiempo limite tomando en cuenta el viaje redondo
23
24        for destino in primer_destino:
25
26          id_2=destino['end_id']
27
28          duracion_anterior=destino['duracion_prom']
29
30          limite_actual=tiempo_limite-duracion_anterior
31
32          segundo_destino=viajes.find_one({'start_id':id_2,'end_id':id_1
33          ,'duracion_prom':{'$lt':limite_actual}})
34
35          if segundo_destino:
36
37            duracion_actual=segundo_destino['duracion_prom']
38
39            if id_2==id_1:
40              duracion_actual=duracion_anterior
41
42            duracion_total.append(duracion_anterior+duracion_actual)
43            id_intermedio.append(id_2)
44            id_inicial.append(id_1)
45
46
47      df_sort=pd.DataFrame({'id_inicial':id_inicial,'id_intermedio':id_intermedio
48      ,'duracion_redondo':duracion_total}).sort_values('duracion_redondo')
49      .iloc[:resultados_limite]
50
51      return df_sort
```

Se regresa un DataFrame de pandas para visualizar más fácil los resultados. Ejecutando esta función desde el usuario $u1$ para la ubicación de *casa* se obtiene el siguiente DataFrame

id_inicial	id_intermedio	duracion_redondo
311	502.0	554.067117
311	356.0	569.068049
410	317.0	631.294992
410	150.0	639.230876
311	340.0	646.567980

Figure 4: Resultado de ejecutar la función *encontrar_paseo*

Es decir, los paseos que menos duran saliendo de las 5 estaciones más cercanas al punto de *casa* salen de las estaciones de la columna *id_inicial*, llegan a las de la columna *id_intermedio* y ese paseo redondo tiene una duración de *duracion_redondo*.

A continuación mostramos una gráfica de los paseos escogidos a partir de la ubicación original

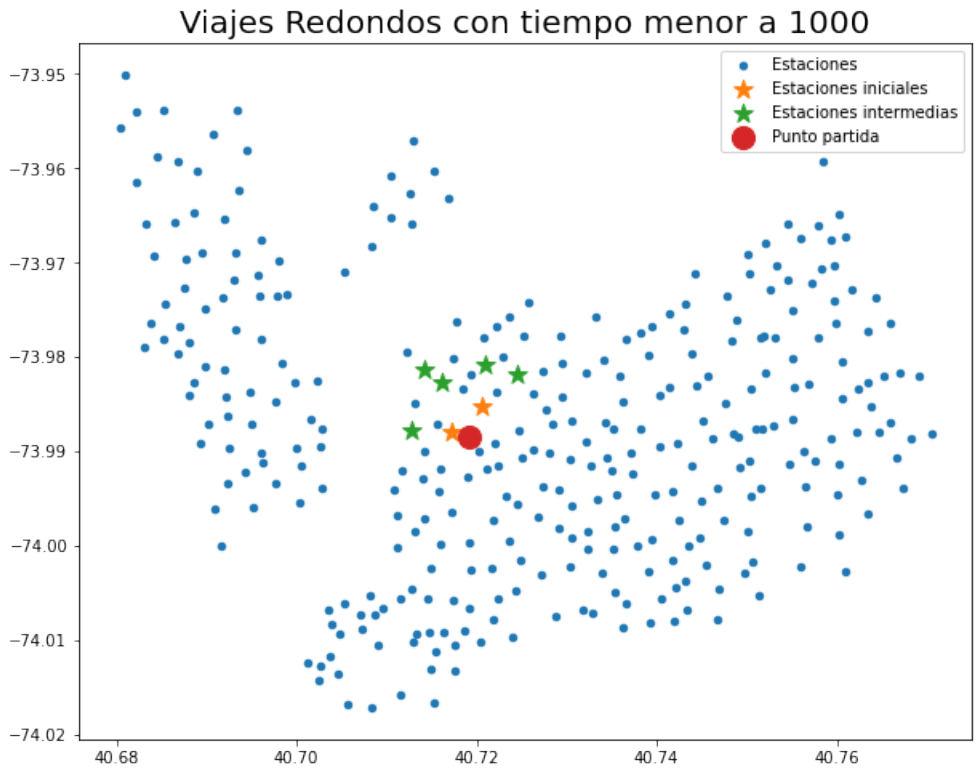

Figure 5: Los puntos de las estaciones y los involucrados en el cálculo de los paseos de la Figura 4

Conclusiones

Las bases de datos de documento son muy flexibles, permiten almacenar información variable de las mismas entidades del mundo real. En bases de datos relacionales y en las de familia de columnas no se puede realizar esto de manera tan libre de restricciones.

Un ejemplo muy bueno son los puntos geoespaciales. Cada punto tiene su valor de latitud y longitud, pero los demás atributos como nombre del lugar, concurrencia promedio y otras banderas que se pueden usar para ayudar al análisis geoespacial, sólo tienen sentido en la estructura más flexible del formato de JSON.

Como todos los nuevos paradigmas de bases de datos no estructuradas tiene una curva de aprendizaje algo elevada, pero aprendiendo la notación, se puede trabajar de manera muy fluida y eficaz. El problema inicial de escoger la estructura de almacenamiento de los datos se vuelve muy importante, como vimos en este ejemplo