TRABAJO PRÁCTICO

Bases de datos NoSQL - Grafos

21 de noviembre del 2018

Grupo 2

Integrantes:

Integrante	Legajo
Bellini, Juan	52056
Goffan, Martin	55431
Horvat, Eric	55564
Lobo, Daniel	51171

Índice

Parte 1: Modelado en PG

1.1) Crear ambas tablas en PG

```
CREATE TABLE public.categories (
	venueid TEXT,
	venuecategory TEXT,
	latitude DOUBLE PRECISION,
	longitude DOUBLE PRECISION,
	cattype TEXT );

CREATE TABLE public.trajectories (
	userid INTEGER,
	venueid TEXT,
	utctimestamp TIMESTAMP WITHOUT TIME ZONE,
	tpos BIGINT );
```

1.2) Poblar categories con la información provista en el CSV

Poblar categories con la información provista en el CSV. Sin embargo, en dicho archivo la información de latitude y longitude provista en el CSV no es la correspondiente al Venue sino a la trayectoria. Completar latitude y longitude en esta tabla tomando el promedio de las latitudes y longitudes de los usuarios que visitaron el mismo Venue.

1.2.1) Crear tabla auxiliar a partir de csv

```
CREATE TABLE aux (
	userid INTEGER,
	latitude DOUBLE PRECISION,
	longitude DOUBLE PRECISION,
	utctimestamp TIMESTAMP,
	venueid TEXT,
	venuecategory TEXT,
	cattype TEXT );
```

1.2.2) Poblar tabla auxiliar desde csv

```
\copy aux FROM './tpgrafo.csv' DELIMITER '|' CSV HEADER
```

1.2.3) Poblar categories desde tabla auxiliar

```
INSERT INTO
   PUBLIC.categories ( venueid, venuecategory, latitude, longitude, cattype )
   SELECT
      venueid,
      venuecategory,
```

```
    Avg(latitude),
    Avg(longitude),
    cattype
  FROM
    aux
  GROUP BY
    venueid,
    venuecategory,
    cattype;
```

1.3) Desarrollar un algoritmo

Para la generación sintética de trayectorias desarrollar un algoritmo (línea de comandos está OK) que pueble la tabla trajectories con trayectorias "realistas" que utilicen los venueid provistos. Se deberá parametrizar:

a. *la cantidad de trayectorias que se quieren generar (o sea la cantidad de userids)*
b. *el intervalo de tiempo total de la simulación (ej: desde 05/10/2010 08:30:20 hasta 06/10/2010 22:00:00)*
c. *la cantidad mínima y máxima de visitas a venueids que debe generarse en u mismo día para cada trayectoria (ej: minimo 0 visitas y máxima 5 visitas en un mismo día por trayectoria).*
d. *El nombre del archivo de salida CSV con esta información. Es decir, las columnas generadas serán: userid, venueid, utctimestamp, tpos: valor por trayectoria que comienza en 1 y se genera en forma consecutiva (sin huecos).*

Con respecto al algoritmo que se ocupa de la generación sintética de trayectorias a desarrollar, utilizamos un script en node.js el cual se encuentra en el path:

`/data/data_generation/generate.js`

1.4) Desarrollar un algoritmo Java de Pruning

Para mejorar la trayectoria sintética anterior y que resulte más realista, desarrollar un algoritmo Java de Pruning que sobre un archivo de trayectorias generado con el formato explicado en el paso anterior permita "borrar" aquellas visitas a venueid imposibles de que hubieran ocurrido por la distancia que tienen. Este programa debe recibir como parámetros:

e. *el nombre del archivo con el formato explicado en el paso anterior*
f. *una velocidad expresada en km /h (ej: 100.4 km/h)*
g. *El nombre de un un archivo de salida (mismo formato). Si existieran 2 venueid visitados consecutivos donde las distancias no permiten que esa visita sea posible, esos venueids se saltearán y las posiciones se renombrarán para que sigan siendo consecutivas en el archivo de salida.*

Ej: si una trayectoria contiene v1, v2, v3, v4, v1, v5 la distancia entre v2 y v3 supera la posibilidad de visita con esa velocidad y la distancia de entre v2 y v4 tampoco puede haber sido visitada a esa velocidad, según los tiempos y distancias existentes, se devolverá v1, v2, v1, v5. Las posiciones de la trayectorias irán del 1 al 4, en lugar de ir del 1 al 6.

El algoritmo Java de Pruning fue resuelto en el punto anterior.

1.5) Generar 4 tablas

2. *Utilizando el algoritmo propuesto, generar 4 tablas con las siguientes características:*
    a. *Trajectories_ss: 1000 usuarios, 100 venues promedio por trayectoria*
    b. *Trajectories_sl: 1000 usuarios, 200 venues promedio por trayectoria*
    c. *Trajectories_ls: 10000 usuarios, 100 venues promedio por trayectoria*
    d. *Trajectories_ll: 10000 usuarios, 200 venues promedio por trayectoria*

Para generar estos archivos, se tiene en cuenta el valor esperado de visitas por día, y se lo multiplica por la cantidad de días necesarios para que el valor esperado total sea el indicado en cada caso. Se obtienen las tablas cambiando el *config.json* y ejecutando:

```
node generate.js config.json
```

Para importar estos archivos, ejecutar:

```
CREATE TABLE public.trajectories_xx (
        userid INTEGER,
        venueid TEXT,
        utctimestamp TIMESTAMP WITHOUT TIME ZONE,
        tpos BIGINT );
```

Luego setear formato de fechas:

```
set datestyle to European; -- El formato de las fechas es European
```

Insertar los datos desde el .csv:

```
\copy trajectories_xx FROM './trajectories_xx.csv' DELIMITER ';' CSV HEADER
```

Finalmente crear índices, PKs y FKs:

```
ALTER TABLE categories ADD PRIMARY KEY(venueid);
CREATE INDEX xx_user_ix ON trajectories_xx(userid);
ALTER TABLE trajectories_xx ADD CONSTRAINT xx_venue FOREIGN KEY(venueid) REFERENCES categories;
```

Como se puede observar, en SQL usamos índices. Para cada una de las tablas “trajectories_xx”, se aplica un índice en *userid.* Además se aplicó una foreign key a categories y en esa tabla se agregó una primary key en el campo *venueid*.

Parte 2: Modelado en JanusGraph

Producir un código (script) para crear los vértices y relaciones con los mismos datos (reales y sintéticos) que se generaron en la parte PG. Aquí quedarán 8 archivos (si eligen Graphframes) o 4 grafos en Cassandra si eligen JanusGraph con la info correspondiente. Proveer los archivos que generan esto y también los archivos binarios generados.

En la clase App.java se encuentran los métodos:

1. private static void createSchema(final JanusGraph graph)
    a. Crea un esquema dado el grafo pasado como parámetro.
2. private static void loadProvidedData(final JanusGraph graph, final String dataFilePath, final boolean loadStops)
    a. Carga la data proveída (esto incluye venues, subcategories y categories)
3. private static void loadSyntheticData(final JanusGraph graph, final String dataFilePath)
    a. Cada la data sintética en el grafo dado.

Parte 3. Consultas y Comparación

Consulta 1

“Patrón Home-Station-Airport’: Encontrar trayectorias que van desde un venue con Category ‘Home’, a uno de tipo ‘Station’, y de ahí a uno de tipo ‘Airport’, en forma consecutiva, es decir sin paradas intermedias.

Mostrar en la salida el userid, y la lista de posiciones desde donde se verifica el patrón.

Consulta 2

Para cada trayectoria encontrar los caminos de cualquier longitud que van desde un venue de Category “Home”, a uno con Categoría “Airport”, en el mismo día.

Consulta 3

“Listar las subtrayectorias que comienzan y finalizan en el mismo Venue en el mismo día.”

Consulta 4

“Para cada usuario, encontrar la/s subtrayectoria/s de mayor longitud (cada subtrayectoria corresponde a un día). Deberá retornarse cada usuario, y día/s correspondiente/s a dichas sub trayectorias, junto con stop inicial y final”.

SQL

Las queries ejecutadas para obtener los resultados de cada consultas se encuentran en la carpeta ./sql/queries, enumerandolas según el enunciado escrito anteriormente. Los resultados promedios, de 5 corridas, son:

1.

	xs	ss	sl	ls	ll
Tiempo	77ms	0.95s	1.72s	8.8s	18.08s

2.

	xs	ss	sl	ls	ll
Tiempo	46ms	3.36s	6.26s	35.27s	77.2s

3.

	xs	ss	sl	ls	ll
Tiempo	47ms	4.52s	5.5s	29.22s	61.58s

4.

	xs	ss	sl	ls	ll
Tiempo	46ms	4.61s	5.74s	29.1s	64.34s

También se encararon las consultas con productos escalares de *trayectories* basados en userid.
En nuestras queries son las del formato "*_*alt.sql*".
Los resultados promedios, de 5 corridas, son:

1.

	xs	ss	sl	ls	ll
Tiempo	37ms	67ms	0.11s	0.27s	0.31s

2.

	xs	ss	sl	ls	ll
Tiempo	44ms	78ms	85ms	0.15s	0.41s

3.

	xs	ss	sl	ls	ll
Tiempo	36ms	0.14s	0.765s	3.49s	7.02s

4.

	xs	ss	sl	ls	ll
Tiempo	52ms	1.176s	1.68s	7.58s	15.71s

NoSQL

1.

	ss	sl	ls	ll
Tiempo	16.03s	-	-	-

2.

	ss	sl	ls	ll
Tiempo	1min 52s	-	-	-

3.

	ss	sl	ls	ll
Tiempo	45min	-	-	-

4.

	ss	sl	ls	ll
Tiempo	X	X	X	X