Contents

1 Planteamiento del problema

2 Datos y estructura de los datos suministrados

Se nos suministran dos bases de datos correspondientes a dos ciudades brasileñas distintas, **Salvador de Bahía** y **Río de Janeiro**. En cada de una de ellas encontramos posiciones de distintos sujetos estudiados identificados a través de un código. Cada base de datos contiene una tabla llamada *posicionesgps* en la que encontramos un registro por cada posición tomada por cada sujeto entre los días 2015-02-17 08:00:05 y 2015-03-04 08:18:05.

La estructura de los registros es la siguiente:

Parámetros	
Id	Identificador numérico de la posición (clave primaria)
IdServidor	Identificador numérico del servidor que realiza la inserción (PK)
Recurso	Nombre del recurso (tetra:1234567)
Latitud	Real que representa la latitud GPS
Longitud	Real que representa la longitud GPS
Velocidad	Entero que representa la velocidad instantánea
Orientación	Entero que representa la orientación respecto al norte en grados
Cobertura	Booleano que indica si hay cobertura
Error	Booleano que nos indica si ha habido algún error en la toma de la posición

En base de datos, el tipo de datos guardado es:

```
mysql> explain posicionesgps;
```

Field	Type	Null	Key	Default
id	bigint(10)	NO	PRI	0
idServidor	int(10) unsigned	NO	PRI	0
recurso	varchar(100)	YES	MUL	NULL
latitud	double	YES		NULL
longitud	double	YES		NULL
velocidad	tinyint(10) unsigned	YES		NULL
orientacion	smallint(10) unsigned	YES		NULL
cobertura	tinyint(10) unsigned	YES		NULL
error	tinyint(10) unsigned	YES		NULL
antigua	tinyint(10) unsigned	YES		0
fecha_timestamp	timestamp	NO	MUL	CURRENT_TIMESTAMP
automático	tiniyint(10) unsigned	NO	MUL	0

Para este estudio se ha trabajado sólo con los siguientes datos,

1. Id
2. Recurso
3. Latitud
4. Longitud
5. Velocidad
6. Fecha

2.1 Análisis de los datos

2.2 Espacio en disco

Con la cantidad de posiciones suministradas, cuánto ocupa cada posición en disco, para hacernos una idea de cuántas posiciones sería posible acumular en función de la frecuencia de éstas sobre un espacio en disco finito.

En nuestra base de datos llamada **Río de Janeiro** contamos con **6928467** posiciones y en **Salvador de Bahía** contamos con **4599974** posiciones.

El tamaño en disco de nuestras bases de datos es,

```
mysql> SELECT table_schema as 'Database',
            table_name AS 'Table',
            round(((data_length + index_length) / 1024 / 1024), 2)
            FROM information_schema.TABLES
            ORDER BY (data_length + index_length) DESC;
```

Database	Table	Size in MB	Size in KB
rio	posicionesgps	1205.64	120564000
bahia	posicionesgps	961.42	96142000

Lo cual nos da una idea de cuánto puede ocupar una toma de posición en disco.

El total de posiciones almacenadas en río es de 6928467 luego podemos estimar el tamaño de una posición en,

$$\frac{120564000}{6928467} = 17.4012519653KB$$

El total de posiciones almacenadas en bahía es de 4599974, luego

$$\frac{96142000}{4599974} = 20.9005529162KB$$

Podemos aproximar el tamaño de una posición por unos 19 KB.

Supongamos que una consola tiene unos 1GB de almacenamiento. Podemos almacenar unas 52631 posiciones en estos 30GB.

Los datos han sido recogidos entre las fechas 2015-02-17 08:00:05 y 2015-03-04 08:18:05, lo que hace una diferencia de 360 horas.

Tenemos 5014 distintos tipos de sujetos a estudiar en la base de datos de río:

```
mysql> USE rio;
mysql> SELECT COUNT(distinct(recurso))
            FROM posicionesgps;
```

count(distinct(recurso))
5014

Lo que nos da una frecuencia de toma de :

$$\frac{6928467}{5014 \cdot 360} = 3.83$$

posiciones a la hora.

Si aumentáramos esta frecuencia a una posición cada 30 segundos, conseguri˚iamos una frecuencia de 120 posiciones a la hora, luego un único sujeto, en una jornada laboral de 8 horas, ocuparía en espacio de 19.2 MB.

2.3 Implementación de los datos en clases de Python

La estructura de los datos es implementable en diversos lenguajes, pero se elige Python por su simplicidad y ya que es el lenguaje científico más usado hoy en día.

Se define la clase `Posición` de la siguiente manera,

```
class Position:
    def __init__(self, id, resource, lat
                 , lon, speed, track, date):
        self.id = id
        self.resource = resource
        self.lat = lat
        self.lon = lon
        self.speed = speed
        self.track = track
        self.date = date
```

A partir de esta clase definiremos una serie de métodos propios a ésta que nos permitirán saber si un punto está en un vecindario asociado a la posición. Vamos a utilizar la noción de distancia euclídea como concepto en el que apoyarnos.

```
        def distance_eu(self, q):
                return math.sqrt((self.lat - q.lat)**2
                        + (self.lon - q.lon)**2)
```

3 Nociones de vecindario

Con el fin de realizar los algoritmos de consolidación, hemos realizado un estudio acerca de distintos tipos de vecindarios a utilizar para los algoritmos de consolidación propios y los algoritmos de *clustering* utilizados que usaremos más adelante.

3.1 Vecindario simple

Utilizando la distancia euclídea, definimos un vecindario como aquel conjunto de puntos que se encuentran a una distancia euclídea menor que ϵ con respecto su centro p_0, es decir,

$$d_E(p_0, p) = \sqrt{lat_p - lat_{p_0})^2 + (long_p - long_{p_0})^2} < \epsilon$$

donde p es un punto con latitud lat_p y longitud $long_p$.
Su implementación en Python es la siguiente,

```
def is_in_neighborhoodByEUSimple(self, q, eps):
        return self.distance_eu(q) < eps
```

3.2 Vecindario involucrando el módulo de la velocidad

En el momento que se toma la posición p_0, aparte de la latidud y su longitud, se toma la velocidad instantánea del sujeto. Podemos considerar en este caso que, dado que nuestro sujeto se encuentra a mayor velocidad, puntos más alejados de lo que consideraríamos en el primer caso (fuera de nuestro vecindario simple), podrían estar dentro de nuestro nuevo radio, que dependería de la velocidad instantánea. Así, definimos nuestro nuevo vecindario:

$$d_E(p_0, p) = \sqrt{lat_p - lat_{p_0})^2 + (long_p - long_{p_0})^2} < \epsilon \cdot vel_{p_0}$$

donde vel_{p_0} es la velocidad instantánea de nuestro punto centro.
Su implementación en Python es la siguiente,

```
def is_in_neighborhoodT0Reachable(self, q, eps):
        return self.distance_eu(q) < eps * self.speed
```

3.3 Vecindad t0-alcanzable

Si fijamos un intervalo de tiempo t_0, podemos definir una vecindad t_0-alcanzable como aquellos puntos que nuestro sujeto puede alcanzar en un tiempo t_0.

Un sujeto que se desplace a velocidad reducida, tendrá una vecindad t_0-alcanzable más reducido que otro que se desplace a una velo$vel_{p_0} \cdot t_0$.

$$d_E(p_0, p) = \sqrt{lat_p - lat_{p_0})^2 + (long_p - long_{p_0})^2} < vel_{p_0} \cdot t_0$$

Éste es un caso concreto del vecindario involucrando la velocidad.
Su implementación en Python es la siguiente,

```
        def is_in_neighborhoodT0Reachable(self, q, t0):
                return self.distance_eu(q) < t0 * self.speed
```

3.4 Vecindario involucrando el tiempo

Las posiciones de nuestros sujetos vienen muestreadas además con el instante en el que fueron tomadas. Podemos considerar que el tiempo entre tomas también es una distancia y definir un vecindario. Definimos esta distancia temporal como la resta de ambos instantes, y el vecindario como:

$$d_T(p_0, p) = time_p - time_{p_0} < \delta$$

```
        def is_neighboorhoudByTime(self, q, lapse):
                foo = time.mktime(self.date.timetuple())
                bar = time.mktime(q.date.timetuple())
                return abs(foo - bar) < lapse
```

4 Algoritmos de consolidación simples

Utilizando las nociones de vecindario definidas en la sección anterior, nos planteamos la idea de definir unos algoritmos de consolidación simples con el fin de mantener la base de datos en un tamaño más o menos estable.

Una primera aproximación sería una creación de un trigger o un pequeño programa en el momento de inserción en base de datos que comparara la última posición recibida para ese sujeto con la nueva a insertar. Se compararía la distancia entre éstas con una distancia euclídea simple, y si ésta estuviera bajo el límite permitido (es decir, muy próxima), se obviaría.

Una segunda aproximación será definir una tarea programada `cron` (ya que nuestros dispositivos están basados en una distribución de Linux) que cada cierto tiempo ejecutara una consolidación sobre estos.

Estas consolidaciones menos avanzadas se realizarán sobre posiciones antiguas, es decir, según el tamaño de la base de datos y el nivel crítico al que puede llegar a estar, mandaremos un cierto número de posiciones a realizar la consolidación.

Utilizando los tres tipos de vecindarios que hemos definido, definimos el siguiente método que realizará la consolidación del tipo que le indiquemos,

```
"Consolidation By distance. Usage: consolidationByDistance(listOfPositionsTo
def ConsolidationByDistance(listPositions, typeOfDistance, eps, t0):
        print "Starting consolidation by: {0}".format(str(typeOfDistance))
        i = 0
        result = []
        while i < len(listPositions) - 1:
                # Neighborhood: Distance EU simple
                if typeOfDistance == 0:
                        if not listPositions[i].is_in_neighborhoodByEUSimple
                                result.append(listPositions[i])
                        else:
                                print "[{0}]: {1} removed!".format(str(datet
                # Neighborhood: Distance EU relative to speed
                elif typeOfDistance == 1:
                        if not listPositions[i].is_in_neighborhoodByEURelati
                                result.append(listPositions[i])
                        else:
```

```
                    print "[{0}]: {1} removed!".format(str(date
            # Neighborhood t0 reachable
            elif typeOfDistance == 2:
                    if not listPositions[i].is_in_neighborhoodT0Reachab
                            result.append(listPositions[i])
                    else:
                            print "[{0}]: {1} removed!".format(str(date
            else:
                    raise ValueError('That distance does not exist')
            i=i+1

    #Por defecto anadiremos la ultima posicion, ya que no tiene siguien
    result.append(listPositions[len(listPositions) - 1])

    return result
```

5 K-means

6 DJ-Cluser

7 Canocopy

8 Conclusiones