Diplomatura en BIG DATA

Bases de datos de grafos con manejo de datos espaciales

Un análisis comparativo

Federico Martínez

19/02/2015

Diplomatura en Big Data

Introducción

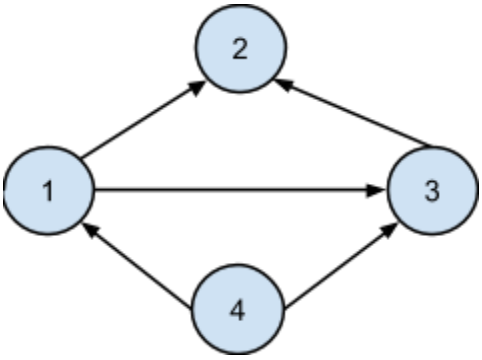
Las bases de datos NoSQL han adquirido una gran popularidad en el último tiempo. Entre los distintos tipos que existen, las bases de datos de grafos se destacan por su capacidad de almacenar elementos de características diferentes, no estructuradas, y, fundamentalmente, las relaciones que existen entre ellos.

En la actualidad existen varias bases de datos de grafos, tanto open source, como closed source. Son varias las que introducen la posibilidad de manejar además datos espaciales o georreferenciados. Entre ellas podemos citar: Neo4j, ArangoDB, OrientDB, Titan; las cuales permiten en distinta medida almacenar y consultar datos espaciales.

En este trabajo nos centraremos en estudiar y comparar las posibilidades que ofrecen dos de ellas: Neo4j y ArangoDB.

Bases de datos de grafos

Las bases de datos de grafos se destacan dentro del mundo NoSQL por estar basadas en teoría de grafos. Un grafo es una estructura matemática compuesta por un conjunto *V* de nodos y un conjunto de arcos o relaciones entre estos nodos *E*.

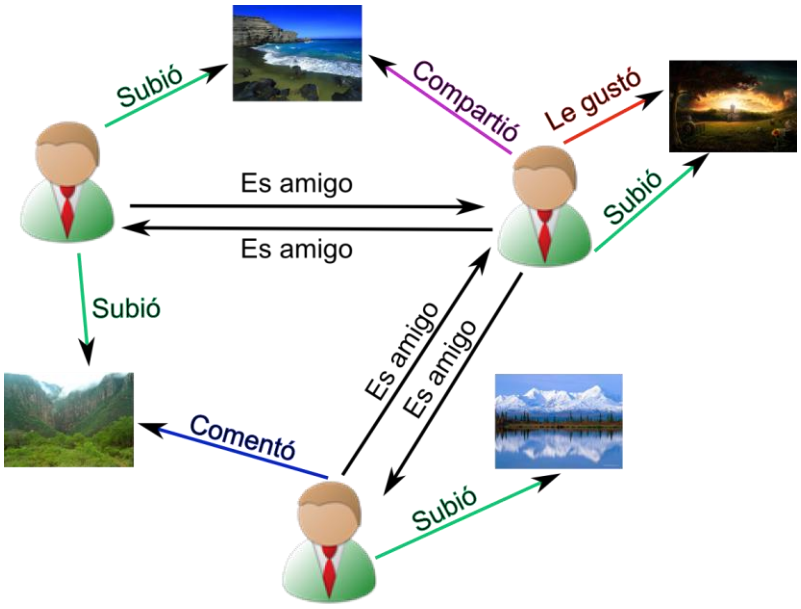


Grafo de cuatro nodos y cinco ejes entre ellos.

Los nodos y los ejes pueden contener propiedades variables y de distintos tipos.

Los grafos son una herramienta muy flexible para el modelado y análisis de problemas y cuentan, además, con una base teórica y algorítmica muy amplia, que las bases de datos de grafos pueden explotar.

Las bases de datos de grafos son muy útiles cuando el problema que se quiere resolver tiene distintos tipos de entidades relacionadas entre sí. Un ejemplo típico es una red social. En ella los usuarios son nodos en un grafo y la “amistad” entre dos usuarios es un eje que los relaciona. Estos nodos pueden tener propiedades como el nombre, edad, género, etc. Las fotos subidas por los usuarios pueden ser también un nodo, con otro conjunto de propiedades: url, fecha de publicación, etc. En este caso, pueden existir relaciones entre usuarios y fotos con distinta semántica: le gusta la imagen, comparte la imagen, está etiquetado en la imagen, etc. Cada una de estas relaciones puede modelarse con un eje.



Las bases de datos de grafos están optimizadas para un tipo de consulta denominada "traversal". Estas consultas se caracterizan porque recorren el grafo en base a las relaciones de los nodos. Siguiendo con el ejemplo anterior, podemos pensar que una consulta típica es: ¿Cuáles son los amigos de mis amigos?

Neo4j

Neo4j es una base de datos de grafos desarrollada por Neo Technologies. Desarrollada en Java, es, en la actualidad, la base de datos de grafos más popular. Su lenguaje de consulta es *Cypher* y permite mediante pattern matching realizar fácilmente consultas que requieren recorrer el grafo. Además de su interfaz nativa en Java, presenta una interfaz REST. Puede ser utilizada también utilizando el framework Blueprints para trabajar con bases de datos de grafos.

```
MATCH (yo:Person)-[:Amigo]->(amigo:Person)-[:Amigo]->(amigo2:Person)
WHERE yo.nombre = 'Federico Martinez' and amigo2.nombre <> yo.nombre
RETURN amigo2.nombre
```

Consulta en Cypher para obtener los amigos de mis amigos

Neo4j Spatial

El manejo de datos espaciales en Neo4j se realiza mediante una librería *plug in* llamada Neo4j Spatial.
Neo4j Spatial utiliza *layers* para organizar los datos geométricos, almacenando un *árbol r* que permite indexar, es decir realizar búsquedas de forma eficiente. Es importante notar que el árbol se almacena como nodos y ejes dentro del árbol.
Los tipos de layer que provee esta librería son variados, teniendo por ejemplo un layer específico para trabajar con archivos *Open Street Map.*
Las operaciones de consulta soportadas son muy variadas, siendo estas las siguientes:

- Contain
- Cover
- Covered By
- Cross
- Disjoint
- Intersect
- Intersect Window
- Overlap
- Touch
- Within
- Within Distance

Una descripción detallada de todas estas operaciones puede encontrarse en *JTS Topology Suite*
Estas operaciones están disponibles únicamente utilizando la API nativa en Java. La API rest presenta un subconjunto más acotado de operaciones:

- findClosestGeometries
- findGeometriesInBBox
- findGeometriesWithinDistance

Mientras que utilizando Cypher, el repertorio es también limitado:

- withinDistance
- bbox (geometrías dentro de un rectángulo)

ArangoDB

ArangoDB es una base de datos multi modelo y de código abierto. Anteriormente conocida como Avocado DB, Arango permite manejar datos de tipo Clave / Valor (Key Value store), documentos y grafos. Estos modelos pueden mezclarse, pudiendo tener un grafo coexistiendo con documentos.

En Arango, para definir un grafo, se deben definir colecciones, por lo menos una, que van a contener a los nodos y luego otras colecciones que van a actuar como ejes entre nodos de esas colecciones. La interfaz de Arango es REST y permite ejecutar código JavaScript en el servidor, de manera similar a procedimientos almacenados (stored procedures).
El lenguaje de consulta de Arango es el Arango Query Language (AQL). AQL es un lenguaje declarativo, es decir describe el resultado y no como debe obtenerse y está pensado para facilitar el trabajo con colecciones de documentos que presentan relaciones entre sí.

```
LET doble_amigos = (
   FOR p IN PATHS(users, es_amigo, "outbound",{minLength:2, maxLength: 2})
   FILTER p.source.name == "Federico Martinez"
   RETURN p.destination
)
RETURN doble_amigos
```

Consulta en AQL para obtener los amigos de mis amigos

Geo Functions

El manejo de datos geográficos ya viene incluido en Arango. Para poder utilizar las funciones que provee la base de datos, es necesario tener un índice geométrico sobre la colección con la que se quiere trabajar, de lo contrario las consultas fallan.
Los índices que provee ArangoDB se basan en curvas de Hilbert y permiten indexar documentos (o nodos, son lo mismo para Arango) que contenga un par de atributos (o un atributo de tipo array con 2 elementos) con la latitud y la longitud.
ArangoDB provee un conjunto más reducido de funciones geométricas, las mismas son:

- NEAR
- WITHIN
- WITHIN_RECTANGLE
- IS_IN_POLYGON

Dataset experimental

Para analizar las capacidades de manejo de datos geográficos que presentan ambas bases de datos decidimos utilizar un dataset proveniente de openflights. El mismo presenta información de aeropuertos, los cuales tienen, entre otros atributos, latitud y longitud. Además hay aerolíneas y rutas. Las rutas van de un aeropuerto a otro y son gestionadas por una aerolínea.
El dataset consta de:

- **6977 aeropuertos con los siguientes datos:**

Airport ID	Identificador Único de OpenFlights para ese aeropuerto
Name	Nombre del aeropuerto
City	Ciudad principal a la que el aeropuerto brinda servicio
Country	País o territorio donde se encuentra el aeropuerto
IATA/FAA	Código FAA de 3 letras para los aeropuertos de Estados Unidos, y código IATA de 3 letras para los demás aeropuertos. Null si no tiene código asignado.
ICAO	Código ICAO de 4 letras. Null si no tiene código asignado.
Latitude	Grados decimales, hasta 6 dígitos significativos. Negativo para el hemisferio sur, positivo para el norte.
Longitude	Grados decimales, hasta 6 dígitos significativos. Negativo para el oeste, positivo para el este.
Altitude	Altura de la ubicación del aeropuerto, en pies.
Timezone	Zona horaria desde UTC
DST	Horario de verano (Daylight savings time). Es uno de los siguientes E (Europa), A (US/Canada), S (America del Sur), O (Australia), Z (Nueva Zelanda), N (Ninguna) or U (Desconocido).
Tz database time zone	Zona horaria en el formato "tz" (Olson)

Registros de ejemplo:

507, "Heathrow", "London", "United Kingdom", "LHR", "EGLL", 51.4775, -0.461389, 83, 0, "E", "Europe/London"
26, "Kugaaruk", "Pelly Bay", "Canada", "YBB", "CYBB", 68.534444, -89.808056, 56, -7, "A", "America/Edmonton"

Este conjunto de datos pesa 830 KB

- **5888 aerolíneas con los siguientes datos:**

Airline ID	Identificador único de OpenFlights para esta aerolínea
Name	Nombre de la aerolínea
Alias	Alias de la aerolínea
IATA	Código IATA de 2 letras, si está disponible
ICAO	Código ICAO de 3 letras, si está disponible
Callsign	Callsign de la aerolínea
Country	País o territorio donde está basada la aerolíneas
Active	"Y" Si la aerolínea está activa, "N" si no.

Registros de ejemplo:

324,"All Nippon Airways","ANA All Nippon Airways","NH","ANA","ALL NIPPON","Japan","Y"
412,"Aerolíneas Argentinas",\N,"AR","ARG","ARGENTINA","Argentina","Y"

Este conjunto de datos pesa 380 KB

- **59036 rutas entre 3209 aeropuertos, con 531 aerolíneas, con los siguientes datos:**

Airline	Código IATA de 2 letras o ICAO de 3 letras para la aerolínea
Airline ID	Identificador único de OpenFlights para esta aerolínea
Source airport	Código IATA de 3 letras o ICAO de 4 para el aeropuerto de origen
Source airport ID	Identificador único de OpenFlights para este aeropuerto
Destination airport	Código IATA de 3 letras o ICAO de 4 para el aeropuerto de destino
Destination airport ID	Identificador único de OpenFlights para este aeropuerto
Codeshare	"Y" Si esta ruta es en verdad operada por otro carrier, vacío si no
Stops	Número de paradas en este vuelo, 0 indica que el vuelo es director
Equipment	Códigos de 3 letras para el tipo de avino generalmente usado en este

	vuelo

Registros de ejemplo:

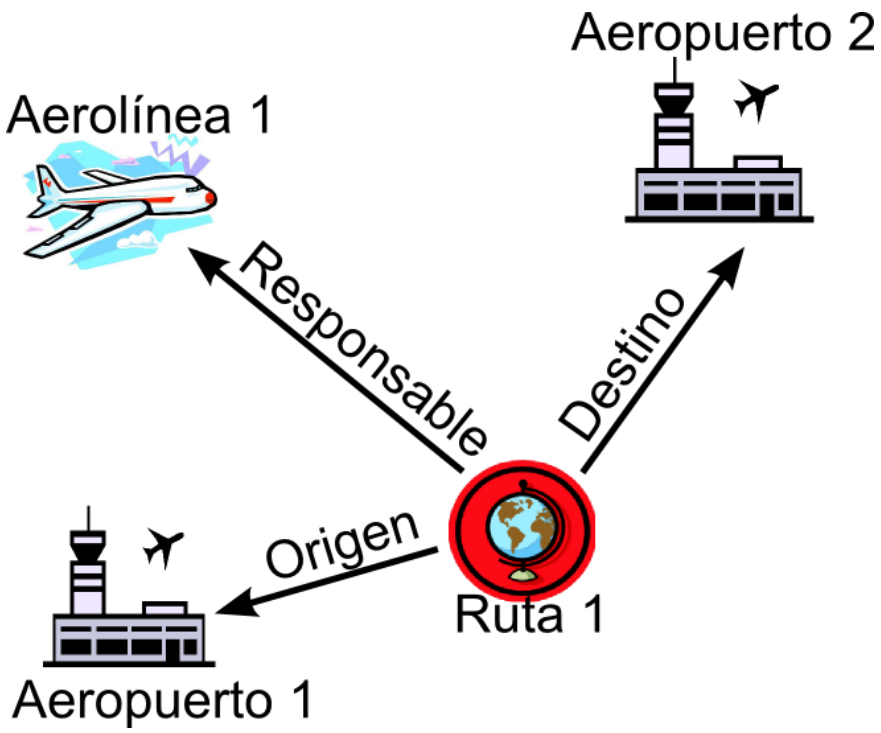
BA,1355,SIN,3316,LHR,507,,0,744 777
BA,1355,SIN,3316,MEL,3339,Y,0,744
TOM,5013,ACE,1055,BFS,465,,0,320

Este conjunto de datos pesa 2.26 MB

Modelo de datos

En base a los datos presentes en el dataset, decimos utilizar el siguiente modelo:

- Cada aeropuerto es un nodo, al igual que las aerolíneas y las rutas.
- Un eje entre una ruta y una aerolínea indica que esta es responsable de la ruta.
- Las rutas pueden relacionarse con los aeropuertos mediante dos tipos distintos de ejes: Origen y Destino, los cuales indican, respectivamente, si el aeropuerto es donde la ruta comienza o termina.



Carga de datos

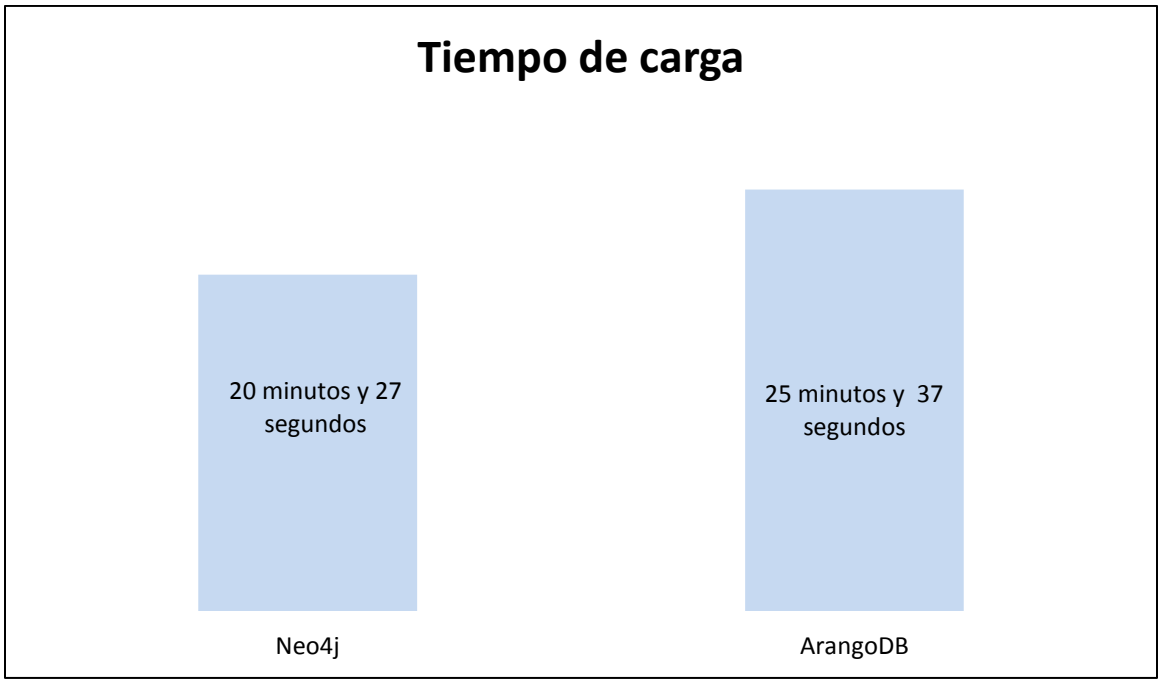
El dataset obtenido no requirió prácticamente de saneamiento. El principal problema encontrado fue que algunas rutas carecían de la información acerca de sus orígenes o destinos, por lo que decidimos descartarlas.
La implementación de la carga de los datos la realizamos utilizando el lenguaje de programación Python, que posee librerías para interactuar con ambas bases de datos mediante su interfaz REST.
A nivel código, creamos una clase llamada *DataLoader* que recibe los archivos *csv* a leer y una estrategia para interactuar con la base de datos. Utilizamos Py2Neo, para interactuar con Neo4j más algunas llamadas REST de manera directa. Para Arango utilizamos py-arango

La carga de datos y los experimentos siguientes se hicieron en una máquina virtual, virtualizada con Virtual Box. La máquina virtual dispone de 3 GB de RAM y 2 cores. La máquina física es un Core i5 4690 de 3.5 GHz con 16 GB de RAM.

El código fuente Python se encuentra disponible en el apéndice 1 y además puede encontrarse en: https://gist.github.com/federicoemartinez/5aafba331a319f821d94

Performance de la carga de datos

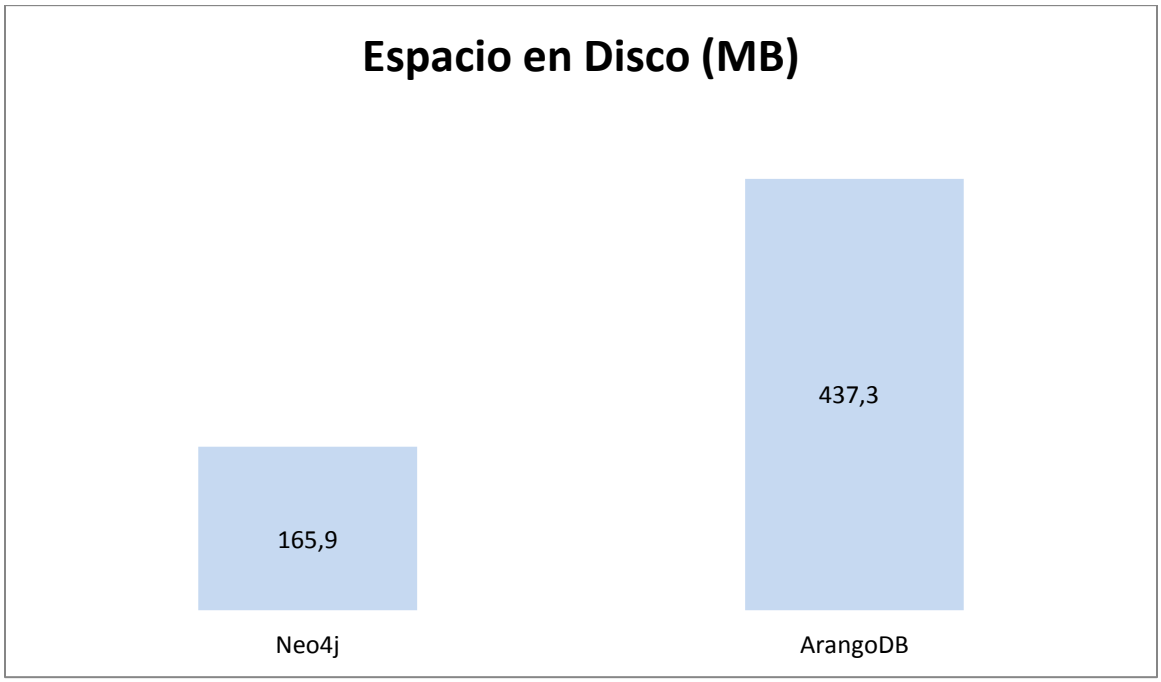
El siguiente gráfico ilustra los tiempos de carga en cada uno de los casos:



Neo4j	1227.451 (20 minutos 27 segundos)

ArangoDB	1537.899 (25 minutos 37 segundos)
Diferencia	310.448 (5 minutos 10 segundos)

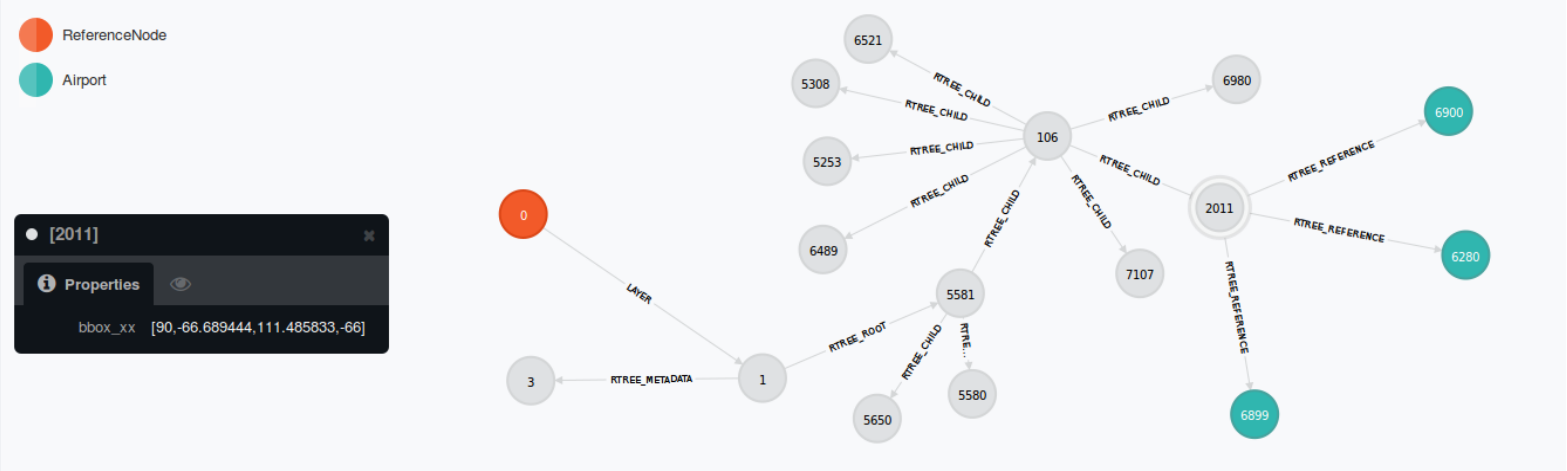
El siguiente gráfico ilustra el espacio en disco consumido por cada una de las bases de datos



Neo4j	165.9 MB
ArangoDB	437.3 MB
Diferencia	271.4 MB

Índice Rtree en Neo4j

Al finalizar la carga, en el caso de Neo4j, se puede observar el rtree que sirve de índice en la misma interfaz de la herramienta:



- El nodo de color rojo es la raíz con que se relacionaran todas las raíces de los rtrees de las distintas layers.
- Los nodos grises son los nodos del rtree (puede observarse que el nodo gris seleccionado tiene un atributo indicando el box que abarca)
- Los nodos verdes son aeropuertos, indexados por este rtree.

Experimentación

Consultas

A fin de comparar las características de manejo espacial que presentan ambas bases de datos decidimos realizar diversas consultas sobre nuestro dataset, a fin de analizar facilidad de escritura y desempeño.

Las siguientes son las consultas que realizamos:

1. Aeropuerto a menos de 100 kilómetros de buenos aires (del congreso)

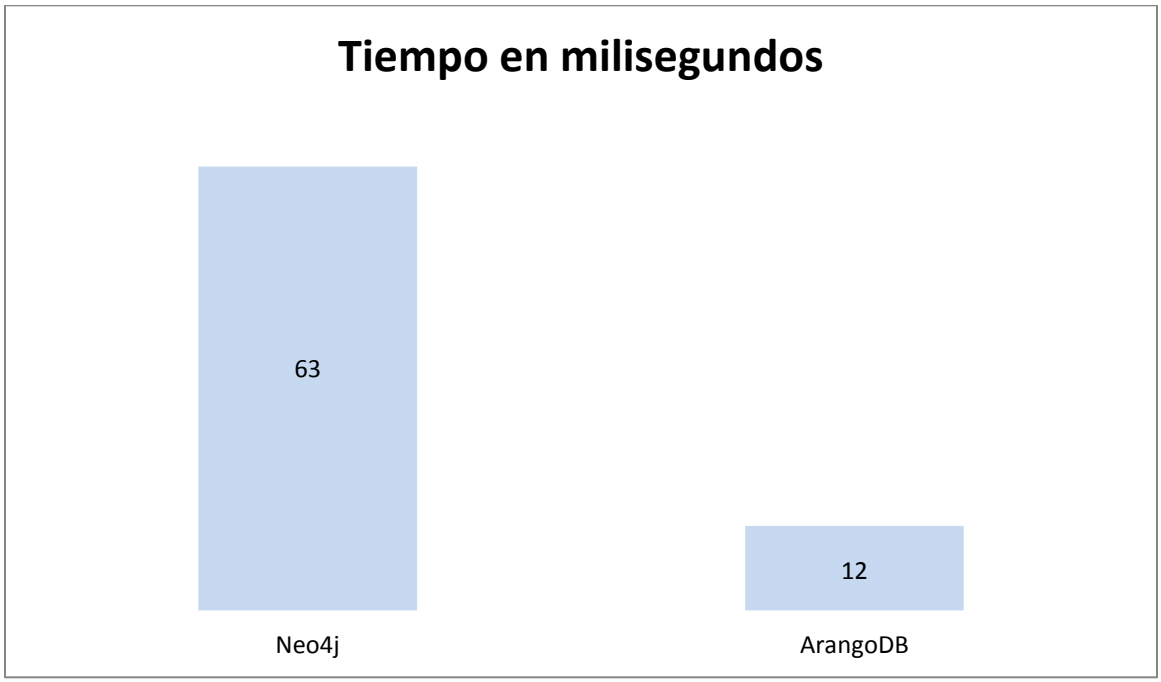
Resolución en Cypher

```
START n = node:geom("withinDistance:[-34.6082595,-58.3841733,100.0]")
      RETURN n.name
```

Resolución en AQL

```
FOR d in WITHIN(airports,-34.6082595,-58.3841733,100000)
      RETURN d.name
```

Comparación de performance



2. Aeropuertos a 100 kilómetros de buenos aires que permiten viajar a barajas

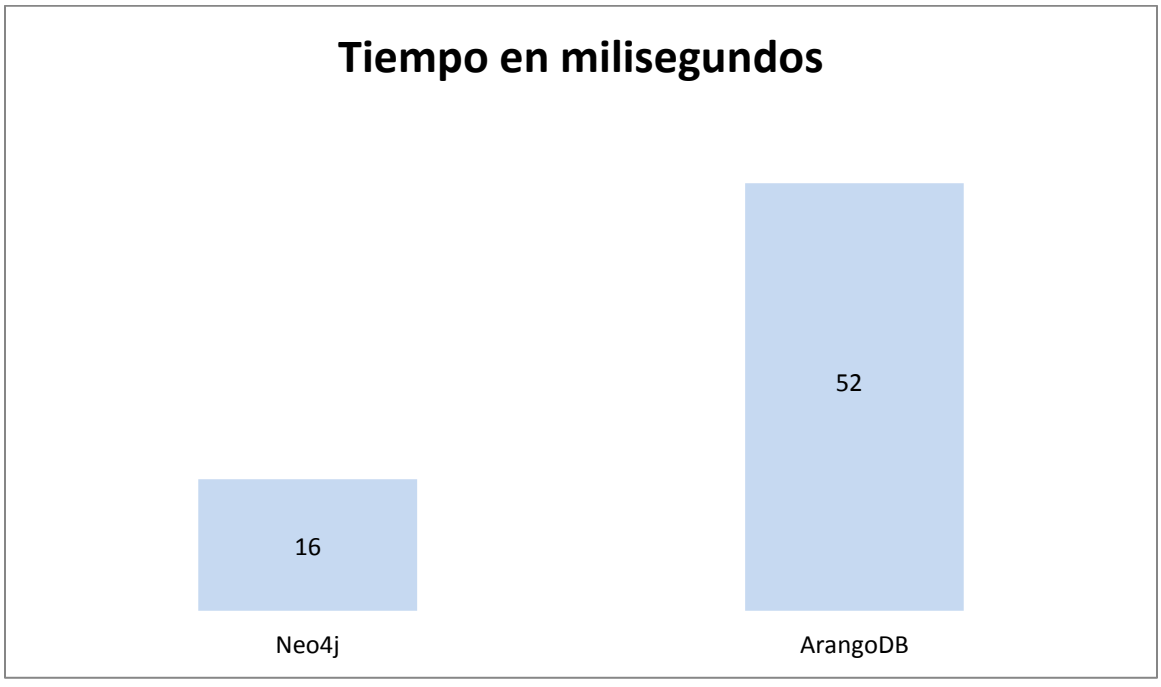
Resolución en Cypher

```
START n = node:geom("withinDistance:[-34.6082595,-58.3841733,100.0]")
WITH n match (n) -[:source_airport]->(r) <-[:destination_airport]- (m:Airport)
WHERE m.name = "Barajas" return distinct n.name
```

Resolución en AQL

```
FOR d in WITHIN(airports,-34.6082595,-58.3841733,100000)
    FOR a_route IN NEIGHBORS(routes, source_airport, d, "inbound")
        FOR destination in NEIGHBORS(airports, destination_airport,
a_route["vertex"], "outbound")
          FILTER destination["vertex"].name == "Barajas"
          COLLECT airport_name = d.name
          RETURN airport_name
```

Comparación de performance



3. Itinerarios que parten de un aeropuerto a menos de 400 kilómetros de buenos aires, tienen un salto y terminan en barajas

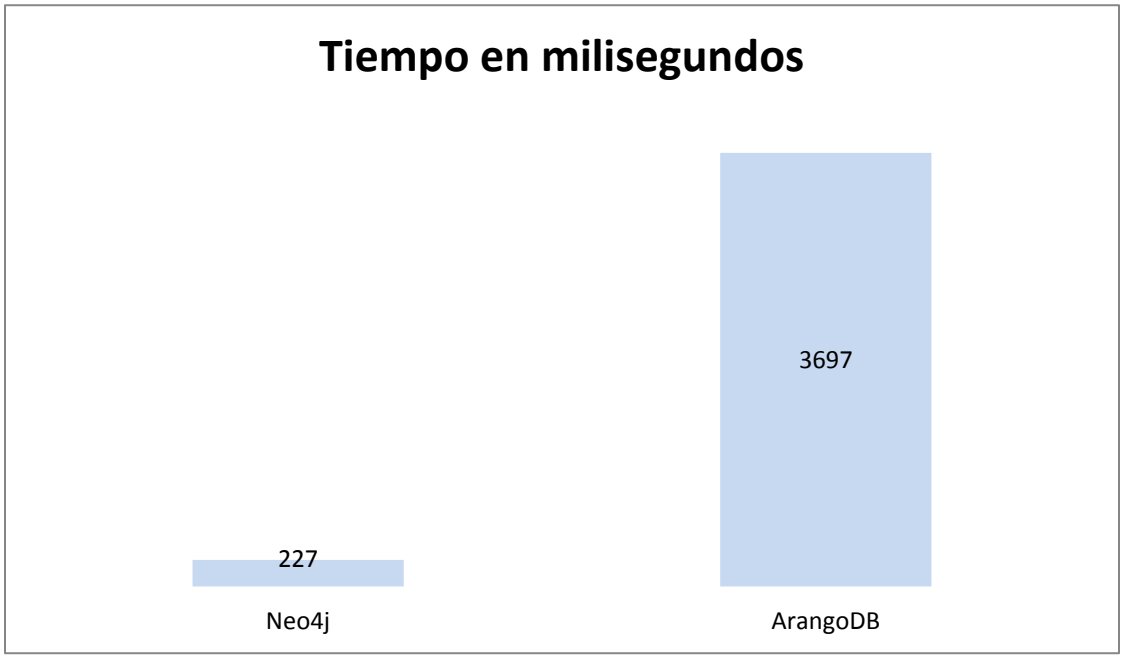
Resolución en Cypher

```
START n = node:geom("withinDistance:[-34.6082595,-58.3841733,400.0]")
WITH n MATCH p = (n:Airport) -[:source_airport]->(:Route)<-[:destination_airport]-
(hop:Airport)-[:source_airport]->(:Route)<-[destination_airport]-(m:Airport)
WHERE m.name = "Barajas"
RETURN DISTINCT n.name, hop.name, m.name
```

Resolución en AQL

```
FOR d IN WITHIN(airports,-34.6082595,-58.3841733,400000)
  FOR a_route IN NEIGHBORS(routes, source_airport, d, "inbound")
    FOR a_hop_airport IN NEIGHBORS(airports, destination_airport, a_route["vertex"],
"outbound")
      FOR a_hop_route IN NEIGHBORS(routes, source_airport, a_hop_airport["vertex"],
"inbound")
        FOR destination in NEIGHBORS(airports, destination_airport,
a_hop_route["vertex"], "outbound")
          FILTER destination["vertex"].name == "Barajas"
          COLLECT a_path = [d.name, a_hop_airport["vertex"].name,
destination["vertex"].name]
          RETURN a_path
```

Comparación de performance



4. Itinerarios que parten de un aeropuerto a menos de 400 km de Buenos Aires, tienen un salto y terminan a menos de 400 km de Chipre

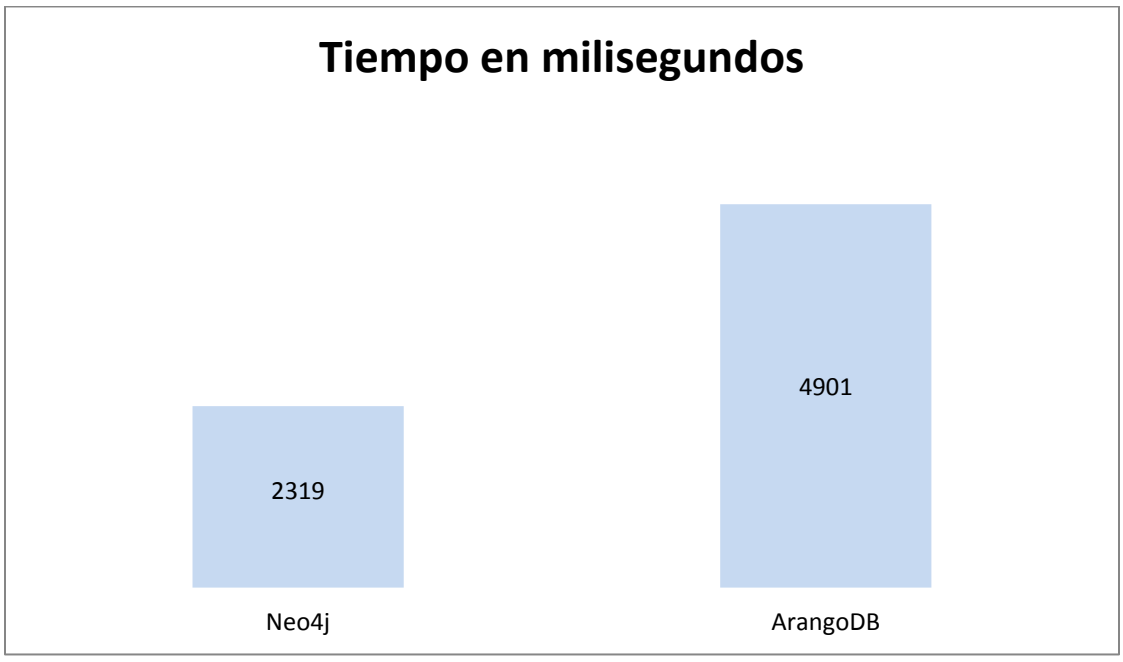
Resolución en Cypher

```
START n = node:geom("withinDistance:[-34.6082595,-58.3841733,400.0]"),
      k = node:geom("withinDistance:[35.509723,24.065552,400.0]")
WITH n,k MATCH (n) -[:source_airport]->(:Route)<-[:destination_airport]-
(hop:Airport)-[:source_airport]->(:Route)<-[destination_airport]-(k)
RETURN DISTINCT n.name, hop.name, k.name
```

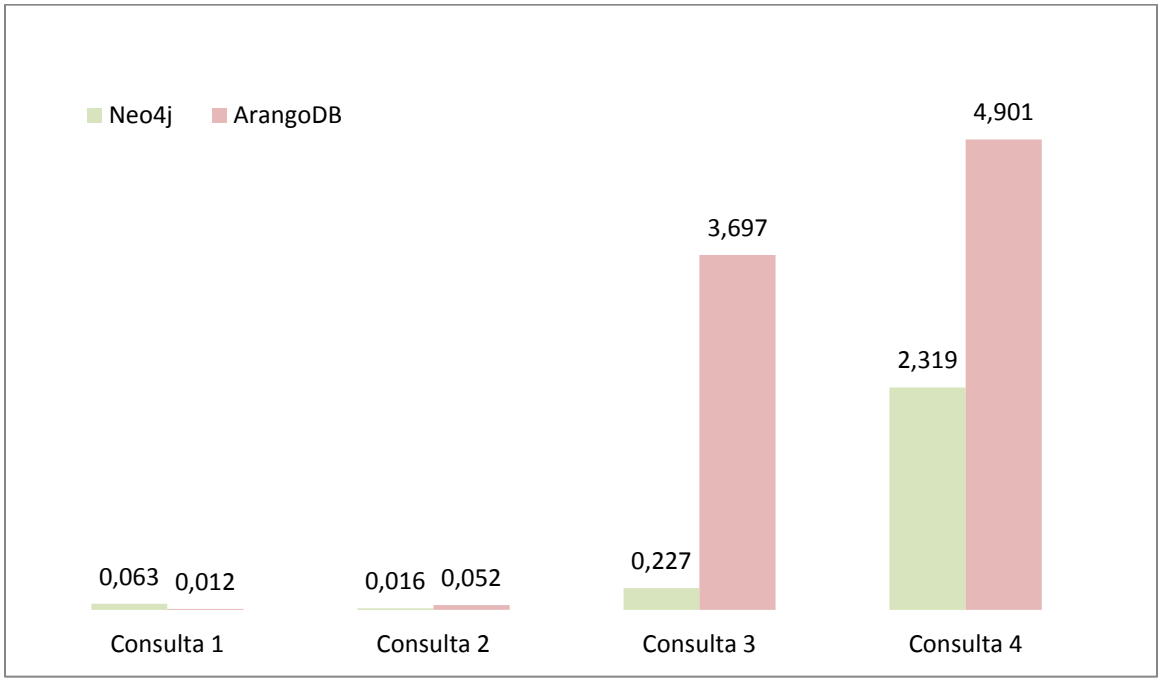
Resolución en AQL

```
LET destinos = (FOR a IN WITHIN(airports,35.509723,24.065552,400000) RETURN a.name)
FOR d IN WITHIN(airports,-34.6082595,-58.3841733,400000)
  FOR a_route IN NEIGHBORS(routes, source_airport, d, "inbound")
    FOR a_hop_airport IN NEIGHBORS(airports, destination_airport, a_route["vertex"],
"outbound")
       FOR a_hop_route IN NEIGHBORS(routes, source_airport, a_hop_airport["vertex"],
"inbound")
             FOR destination IN NEIGHBORS(airports, destination_airport,
a_hop_route["vertex"], "outbound")
               FILTER destination["vertex"].name IN destinos
               COLLECT a_path = [d.name, a_hop_airport["vertex"].name,
destination["vertex"].name]
               RETURN a_path
```

Comparación de performance



Comparación global de performance



	Consulta 1	Consulta 2	Consulta 3	Consulta 4
Neo4j	0,063	0,016	0,227	2,319
ArangoDB	0,012	0,052	3,697	4,901
ArangoDB/Neo4j	0,1904762	3,25	16,28634361	2,113410953

Carga de mapas

ArangoDB solo soporta indexar colecciones de puntos, pero Neo4j es capaz de indexar distintos tipos comunes de geometrías: Point, LineString, Polygon, etc. En particular es capaz de cargar datos a partir de archivos de OpenStreetMap o Shapefiles.

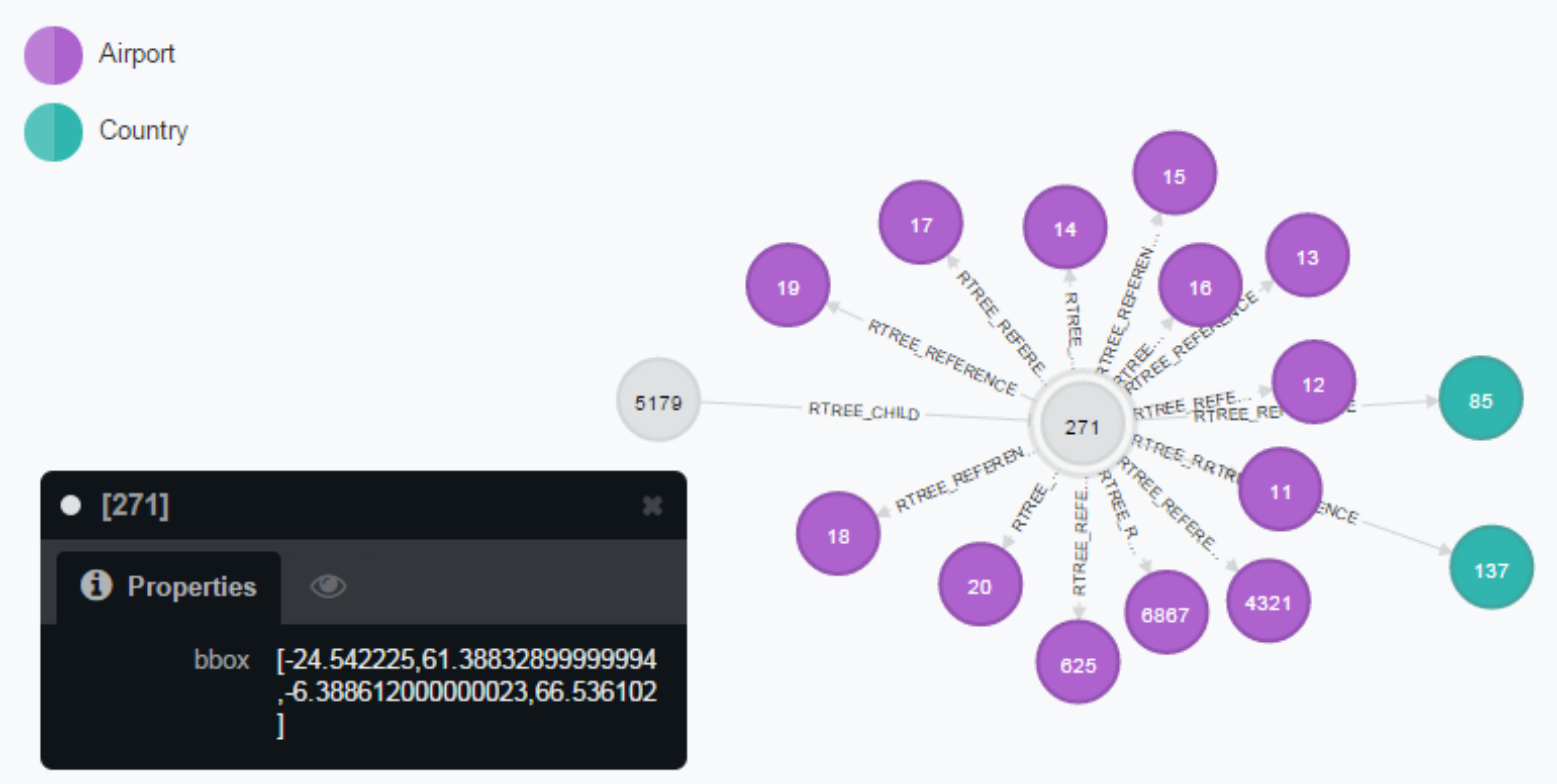
Decidimos explorar estas características que nos permitirían realizar consultas más complejas a las realizadas en la sección anterior.

Carga de shapefiles

Con la intención de experimentar las posibilidades de trabajar con este tipo de archivos, cargamos un archivo *Shapefile* con los contornos de todos los países del mundo. Para cargar el *shapefile* usamos la API Java que provee Neo4j Spatial. La carga es muy sencilla y genera un layer donde los nodos contienen un atributo denominado “wkb" que es la representación de la geometría del país en *“Well known Binary format”*, un formato estándar para la representación de geometrías.

El código para cargar el *shapefile* se encuentra disponible en el apéndice 2 *y además* se encuentra en: https://gist.github.com/federicoemartinez/e61e4902b369bcb5e24e

El tipo de layer que se genera es incompatible con el layer que utilizamos en la etapa anterior para cargar los aeropuertos. Por eso, lo que hicimos fue convertir el archivo csv con los datos del aeropuerto en un archivo *shapefile*. Para hacer esto utilizamos el servicio web gratuito que puede encontrarse en gisconvert. Una vez obtenido el archivo *shapefile*, se procedió a la carga utilizando el mismo mecanismo que usamos para cargar los países. El resultado es un layer que tiene nodos de tipo aeropuerto y país.



Una vez cargados los aeropuertos, cargamos las aerolíneas y las rutas de la misma manera que antes, ya que estas no tienen atributos geográficos.

Con este nuevo layer y utilizando Java para acceder a la base de datos es posible construir queries que usen todos los métodos de la *jts*. Por ejemplo podemos responder la siguiente pregunta:

- ¿Cuáles son los destinos posibles partiendo de aeropuertos en Argentina?

Es importante notar que la misma no se puede responder con los datos anteriores, ya que no hay forma de decir si un aeropuerto está en argentina o no: podíamos decir que la distancia al congreso sea menor a una cierta cantidad de kilómetros, pero eso no nos permitía cubrir todos los aeropuertos de argentina salvo que agrandemos el radio a considerar lo suficiente, pero en ese caso estaríamos considerando aeropuertos que no pertenecen a nuestro país.

El código para resolver esa consulta fuente se encuentra disponible en el apéndice 3 y también se encuentra en: https://gist.github.com/federicoemartinez/4ae36bdd018c9e9ec0ef

Carga de archivos OSM

Neo4j permite cargar archivos de Open Street Map. Decidimos experimentar la carga de estos datos.

El código que permite cargar un archivo OSM se encuentra en el apéndice 4 y además está disponible en https://gist.github.com/federicoemartinez/a0080894aba009c4a089

Conclusiones

Como conclusión del trabajo podemos afirmar que si bien ambas bases de datos proveen soporte para el manejo de datos geográficos y ambas funcionan bien para responder consultas sencillas, Neo4j logra imponerse como una mejor alternativa cuando las consultas requieren recorrer el grafo o funcionalidades más complejas como importar mapas.
Si bien Arango logra resolver correctamente las consultas sencillas (por ejemplo la consulta 1 que no requería hacer traversals es resuelta más rápidamente por Arango), no está a la altura del soporte que brinda Neo4j. Las principales falencias que encontramos en ArangoDB son:

1) Su lenguaje de consulta no parece estar pensado para recorrer grafos. Parece haber sido diseñado para trabajar con documentos y luego se le agregaron casi de manera ad hoc nociones de grafos. Esto genera que escribir queries que combinen información espacial y recorrer el grafo sea ineficiente además de complejo.
2) Si bien es muy bueno que la base brinde soporte geográfico *built in*, el mismo es muy básico. Por esa razón es imposible escribir consultas que requieren de capacidades más complejas.

Trabajo futuro

En este trabajo nos limitamos a analizar solo dos bases de datos, pero en el mercado existen otras alternativas, algunas muy interesantes como Titan, que permite procesar en paralelo de manera distribuida; y que brindan soporte espacial.
Una extensión inmediata a este trabajo es incluir estas bases de datos a la comparación y analizar su desempeño y las posibilidades que ofrecen.

Apéndices

Apéndice 1: Código Python para la carga de datos

```
1.  import csv
2.  import uuid
3.
4.  # Cosas para neo4j
5.  from py2neo import Graph, Node, Relationship
6.  import json
7.  import requests
8.
9.  # Cosas para arango
10. from arango import Arango
11.
12.
13. class DataLoader(object):
14.
15.     def load_data(self, airport_file, airline_file, route_file, graph_loader):
16.         airlines = csv.reader(airline_file, delimiter=",")
17.         for airline_id, name, alias, iata, icao, callsign, country, active in airlines:

18.             graph_loader.create_airline(
19.                 airline_id, name, alias, iata, icao, callsign, country, active)
20.         airports = csv.reader(airport_file, delimiter=",")
21.         for airport_id, name, city, country, iata, icao, lat, long, altitude, timezone,
     dst, tz in airports:
22.             graph_loader.create_airport(airport_id, name, city, country, iata, icao, fl
    oat(
23.                 lat), float(long), altitude, timezone, dst, tz)
24.         routes = csv.reader(route_file, delimiter=",")
25.         for airline, airline_id, source_airport, source_airport_id, destination_airport
    , destination_airport_id, codeshare, stops, equipement in routes:
26.             if source_airport_id == "\\N" or airline_id == "\\N" or destination_airport
    _id == "\\N":
27.                 continue
28.             graph_loader.create_route(
29.                 airline_id, source_airport_id, destination_airport_id, codeshare, stops
    , equipement)
30.
31.
32. class Neo4JGraphLoader(object):
33.
34.     def __init__(self, url="http://localhost:7474/db/data/"):
35.         self._url = url
36.         self._graph = Graph(self._url)
37.         self._headers = {'content-type': 'application/json'}
38.         url = self._url + "index/node/"
39.         # Parametros necesarios para crear un indice geometrico
40.         payload = {
41.             "name": "geom",
42.             "config": {
43.                 "provider": "spatial",
44.                 "geometry_type": "point",
45.                 "lat": "lat",
46.                 "lon": "long"
```

```
            }
        }
        r = requests.post(url, data=json.dumps(payload), headers=self._headers)
        self._airports = {}
        self._airlines = {}

    def create_airport(self, airport_id, name, city, country, iata, icao, lat, long, altitude, timezone, dst, tz):
        n = Node("Airport", airport_id=airport_id, name=name, city=city, country=country, iata=iata,
                 icao=icao, lat=lat, long=long, altitude=altitude, timezone=timezone, dst=dst, tz=tz)

        n, = self._graph.create(n)
        self._airports[airport_id] = n

        # Lo agregamos al indice
        url = self._url + "index/node/geom"
        payload = {'value': 'dummy', 'key': 'dummy', 'uri': str(n.uri)}
        r = requests.post(url, data=json.dumps(payload), headers=self._headers)

        # Al layer
        url = self._url + "ext/SpatialPlugin/graphdb/addNodeToLayer"
        payload = {'layer': 'geom', 'node': str(n.uri)}
        r = requests.post(url, data=json.dumps(payload), headers=self._headers)
        return n

    def create_airline(self, airline_id, name, alias, iata, icao, callsign, country, active):
        n = Node("Airline", airline_id=airline_id, name=name, alias=alias,
                 iata=iata, icao=icao, callsign=callsign, country=country, active=active)
        n, = self._graph.create(n)
        self._airlines[airline_id] = n
        return n

    def create_route(self, airline_id, source_airport_id, destination_airport_id, codeshare, stops, equipement):
        n = Node("Route", route_id=str(uuid.uuid4()),
                 codeshare=codeshare, stops=stops, equipement=equipement)
        r1 = Relationship(
            self._airports[source_airport_id], "source_airport", n)
        self._graph.create(r1)
        r1 = Relationship(
            self._airports[destination_airport_id], "destination_airport", n)
        self._graph.create(r1)
        r1 = Relationship(self._airlines[airline_id], "airline_responsible", n)
        self._graph.create(r1)


class ArangoGraphLoader(object):

    def __init__(self, host="localhost", port=8529):
        self._arango = Arango(host=host, port=port)
        if "openflights" not in self._arango.databases["user"]:
            db = self._arango.add_database("openflights")
            db.add_collection("airports")
            db.collection("airports").add_geo_index(fields=["long", "lat"])
            db.add_collection("airlines")
                db.add_collection("routes")
                graph = db.add_graph("openflights_graph")
```

```
                    graph.add_vertex_collection("airports")
                    graph.add_vertex_collection("airlines")
                    graph.add_vertex_collection("routes")
                    db.add_collection("source_airport", is_edge=True)
                    db.add_collection("destination_airport", is_edge=True)
                    db.add_collection("airline_responsible", is_edge=True)
                    graph.add_edge_definition(
                        edge_collection="source_airport",
                        from_vertex_collections=["routes"],
                        to_vertex_collections=["airports"])
                    graph.add_edge_definition(
                        edge_collection="destination_airport",
                        from_vertex_collections=["routes"],
                        to_vertex_collections=["airports"])
                    graph.add_edge_definition(
                        edge_collection="airline_responsible",
                        from_vertex_collections=["routes"],
                        to_vertex_collections=["airlines"])
                else:
                    db = self._arango.db("openflights")
                    graph = db.graph("openflights_graph")
                self._db = db
                self._graph = graph

            def create_airport(self, airport_id, name, city, country, iata, icao, lat, l
    ong, altitude, timezone, dst, tz):
                return self._graph.add_vertex("airports", {"_key": airport_id,
                                                           "lat": lat,
                                                           "long": long,
                                                           "name": name,
                                                           "city": city,
                                                           "country": country,
                                                           "iata": iata,
                                                           "icao": icao,
                                                           "altitude": altitude,
                                                           "timezone": timezone,
                                                           "dst": dst,
                                                           "tz": tz})

            def create_airline(self, airline_id, name, alias, iata, icao, callsign, coun
    try, active):
                return self._graph.add_vertex("airlines", {"_key": airline_id,
                                                           "name": name,
                                                           "alias": alias,
                                                           "iata": iata,
                                                           "icao": icao,
                                                           "callsign": callsign,
                                                           "country": country,
                                                           "active": active})

            def create_route(self, airline_id, source_airport_id, destination_airport_id
    , codeshare, stops, equipement):
                ruuid = str(uuid.uuid4())
                r = self._graph.add_vertex("routes", {
                                                   "_key": ruuid, "codeshare": codeshare, "stops
    ": stops, "equipement": equipement})
                self._graph.add_edge(
                    "source_airport",
                    {
                        "_from": "routes/" + ruuid,
                        "_to": "airports/" + source_airport_id,
```

```
159.                     })
160.                 self._graph.add_edge(
161.                     "destination_airport",
162.                     {
163.                         "_from": "routes/" + ruuid,
164.                         "_to": "airports/" + destination_airport_id,
165.                     })
166.                 self._graph.add_edge(
167.                     "airline_responsible",
168.                     {
169.                         "_from": "routes/" + ruuid,
170.                         "_to": "airlines/" + airline_id,
171.                     })
```

Apéndice 2: Carga de datos desde un *shapefile*

```
1.  package com.mycompany.neo4j_test;
2.  import java.io.FileNotFoundException;
3.  import java.io.IOException;
4.  import java.util.logging.Level;
5.  import java.util.logging.Logger;
6.  import org.neo4j.gis.spatial.ShapefileImporter;
7.  import org.neo4j.graphdb.GraphDatabaseService;
8.  import org.neo4j.graphdb.factory.GraphDatabaseFactory;
9.
10. public class Queries {
11.     public static void main(String[] args) {
12.
13.         GraphDatabaseService database = new GraphDatabaseFactory().newEmbeddedDatabase(
    DB_PATH);
14.         try {
15.             ShapefileImporter importer = new ShapefileImporter(database);
16.             importer.importFile(SHAPE_FILE, "world");
17.         } catch (FileNotFoundException ex) {
18.             Logger.getLogger(Queries.class.getName()).log(Level.SEVERE, null, ex);
19.         } catch (IOException ex) {
20.             Logger.getLogger(Queries.class.getName()).log(Level.SEVERE, null, ex);
21.         } finally {
22.             database.shutdown();
23.         }
24.     }
25. }
```

Apéndice 3: Consulta espacial sobre un layer cargado desde un *shapefile*

```
1.  package com.mycompany.neo4j_test;
2.  import java.io.FileNotFoundException;
3.  import java.io.IOException;
4.  import java.util.Map;
5.  import java.util.logging.Level;
6.  import java.util.logging.Logger;
7.  import org.neo4j.cypher.javacompat.ExecutionEngine;
8.  import org.neo4j.cypher.javacompat.ExecutionResult;
9.  import org.neo4j.gis.spatial.Layer;
10. import org.neo4j.gis.spatial.ShapefileImporter;
11. import org.neo4j.gis.spatial.SpatialDatabaseService;
12. import org.neo4j.gis.spatial.pipes.GeoPipeFlow;
13. import org.neo4j.gis.spatial.pipes.GeoPipeline;
14. import org.neo4j.graphdb.GraphDatabaseService;
15. import org.neo4j.graphdb.ResourceIterator;
16. import org.neo4j.graphdb.Transaction;
17. import org.neo4j.graphdb.factory.GraphDatabaseFactory;
18. /**
19.  *
20.  * @author Admin
21.  */
22. public class Queries {
23.     public static void main(String[] args) {
24.         GraphDatabaseService database = new GraphDatabaseFactory().newEmbeddedDatabase(
    DATABASE_PATH);
25.         try(Transaction tx = database.beginTx()){
26.
27.         // Obtenemos primero la geometria de argentina
28.         SpatialDatabaseService spatialService = new SpatialDatabaseService(database );

29.         Layer layer = spatialService.getLayer( "world" );
30.         GeoPipeline pipeline = GeoPipeline.start( layer )
31.         .copyDatabaseRecordProperties( "NAME" )
32.         .propertyFilter( "NAME", "Argentina" );
33.
34.
35.         GeoPipeFlow flow = pipeline.next();
36.
37.         // Buscamos los aeropuertos contenidos en Argentina
38.         pipeline = GeoPipeline.start( layer )
39.         .coveredByFilter(flow.getGeometry());
40.
41.         // Creamos una query en cypher que parte de los aeropuertos
42.         // encontrados y obtiene los aeropuertos de destino
43.         String query = "START airport=node(";
44.         while(pipeline.hasNext()){
45.             flow = pipeline.next();
46.             query += flow.getId();
47.             if(pipeline.hasNext()){
48.                 query +=", ";
49.             }
50.         }
```

```
51.         query += ") match (airport:Airport)-[source_airport]->(r:Route)<-
    [destination_airport]-
    (m:Airport) return distinct airport.name as desde, m.name as hasta";
52.
53.         ExecutionEngine engine;
54.         engine = new ExecutionEngine( database );
55.         ExecutionResult result = engine.execute( query);
56.
57.         // mostramos los resultados por pantalla
58.         ResourceIterator<Map<String, Object>> a;
59.         a = result.iterator();
60.         while(a.hasNext()){
61.             Map<String, Object> elem = a.next();
62.             System.out.println(elem.get("desde").toString() + " --
    > " + elem.get("hasta").toString());
63.         }
64.
65.
66.
67.         tx.success();
68.         }
69.         finally{
70.             database.shutdown();
71.         }
72.
73.     }
74. }
```

Apéndice 4: Carga de datos desde un archivo Open Street Map

```
1.  package com.mycompany.neo4j_test;
2.  import java.io.FileNotFoundException;
3.  import java.io.IOException;
4.  import java.util.HashMap;
5.  import java.util.Map;
6.  import java.util.logging.Level;
7.  import java.util.logging.Logger;
8.  import javax.xml.stream.XMLStreamException;
9.  import org.neo4j.gis.spatial.osm.OSMImporter;
10. import org.neo4j.graphdb.GraphDatabaseService;
11. import org.neo4j.graphdb.factory.GraphDatabaseFactory;
12. import org.neo4j.unsafe.batchinsert.BatchInserter;
13. import org.neo4j.unsafe.batchinsert.BatchInserters;
14. /**
15.  *
16.  * @author Admin
17.  */
18. public class Queries {
19.     public static void main(String[] args) {
20.         GraphDatabaseService db = null;
21.         BatchInserter batchInserter = null;
22.         boolean shutdown = true;
23.         try{
24.             OSMImporter importer = new OSMImporter("onestreet");
25.             Map<String, String> config = new HashMap<String, String>();
26.             config.put("neostore.nodestore.db.mapped_memory", "90M" );
27.             config.put("dump_configuration", "true");
28.             config.put("use_memory_mapped_buffers", "true");
29.             batchInserter = BatchInserters.inserter(DB_PATH, config);
30.             shutdown = false;
31.             importer.importFile(batchInserter, OSM_FILE_PATH, false);
32.             batchInserter.shutdown();
33.             shutdown = true;
34.             db = new GraphDatabaseFactory().newEmbeddedDatabase(DB_PATH);
35.             importer.reIndex(db, 10000);
36.         }
37.         catch (IOException ex) {
38.             System.out.println(ex);
39.             Logger.getLogger(Queries.class.getName()).log(Level.SEVERE, null, ex);
40.         } catch (XMLStreamException ex) {
41.             System.out.println(ex);
42.             Logger.getLogger(Queries.class.getName()).log(Level.SEVERE, null, ex);
43.         }finally{
44.             if(db != null){
45.                 db.shutdown();
46.             }
47.             if(! shutdown){
48.                 batchInserter.shutdown();
49.             }
50.         }
51.
52.     }
53. }
```