Spatial Cypher Cheat Sheet - Intro To Geospatial Cypher Functions With Neo4j

Intro To Cypher & The Property Graph Data Model

The variable ***p*** is bound to the path connecting SFO and RSW

Nodes are defined within parentheses `()` properties within curly braces `{}`

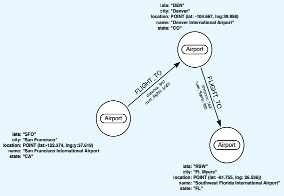
The variable length path operator * defines paths of arbitrary length. We specify a maximum length of 2 relationships.

```
MATCH p=(sfo:Airport {iata: "SFO"})-[b:FLIGHT_TO*2]->(rsw:Airport {iata: "RSW"}) RETURN *
```

Search the graph for the following pattern.

This graph pattern represents a node with the label Airport that has a property iata with the value “SFO”, connected through one or more outgoing relationships of type FLIGHT_TO to a node with the label Airport and iata value “RSW” . We bind variables sfo, f, and rsw to refer to each part of the graph pattern later in the query if needed.

Return any variables that we’ve matched on in previous parts of the query


iata: "DEN"
city: "Denver"
location: POINT (lat: -104.667, lng:39.858)
name: "Denver International Airport"
state: "CO"
Airport
FLIGHT_TO
distance: 967
num_flights: 5065
FLIGHT_TO
distance: 1607
num_flights: 385
Airport
iata: "SFO"
city: "San Francisco"
location: POINT (lat:-122.374, lng:y:37.619)
name: "San Francisco International Airport
state: "CA"
Airport
iata: "RSW"
city: "Ft. Myers"
location: POINT (lat: -81.755, lng: 26.536))
name: "Southwest Florida International Airport"
state: "FL"

Neo4j is a database management system (DBMS) that uses the **property graph data model** which is composed of nodes, relationships, and properties to model, store, and query data as a graph. Nodes can have one or more labels, relationships have a single type and direction. Key-value pair properties can be stored on nodes and relationships.

The **Cypher query language** is used to query data and interact with Neo4j. Cypher is a declarative query language that uses ASCII-art like syntax to define graph patterns that form the basis of most query operations. Nodes are defined with parenthesis, relationships with square brackets, and can be combined to create complex graph patterns. Common Cypher commands are **`MATCH`** (find where the graph pattern exists), **`CREATE`** (add data to the database using the specified graph pattern), and **`RETURN`** (return a subset of the data as a result of a traversal through the graph.

Spatial Point Type

Neo4j supports 2D or 3D geographic (WGS84) or cartesian coordinate reference system (CRS) points

```
RETURN point( {latitude:37.62245, longitude:-122.383989} )
```

Creating a point by specifying latitude/longitude. WGS84 is assumed when using lat/lon.

Point data can be stored as properties on nodes or relationships. Here we create an Airport node and set its location as a point.

```
CREATE (a:Airport)
SET a.iata = "SFO",
a.location = point( {latitude:37.62245, longitude:-122.383989})
RETURN a
```

```
CREATE POINT INDEX airportIndex
FOR (a:Airport) ON (a.location)
```

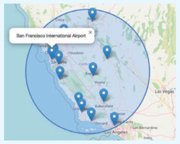
Database indexes are used to speed up search performance. Here we create a database index on the location property for Airport nodes. This will help us find airports faster when searching for airports by location (radius distance or bounding box spatial search).

Spatial Cypher Functions

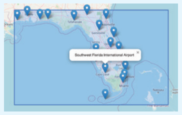
Radius Distance

To find nodes close to other nodes in the graph we can use the `point.distance()` function to perform a radius distance search

```
MATCH (a:Airport)
WHERE point.distance(
  a.location,
  point({latitude:37.55948, longitude:-122.32544})) < 20000
RETURN a
```



Within Bounding Box



```
MATCH (a:Airport)
WHERE point.withinBBox(
  a.location,
  point({longitude:-122.325447, latitude: 37.55948 }),
  point({longitude:-122.314675 , latitude: 37.563596}))
RETURN a
```

To search for nodes within a bounding box we can use the `point.withinBBox()` function.

Geocoding

```
CALL apoc.spatial.geocode('SFO Airport') YIELD location
---------------------------------------------------------------
{
  "description": "San Francisco International Airport, 780,
South Airport Boulevard, South San Francisco, San Mateo County,
CAL Fire Northern Region, California, 94128, United States",
  "longitude": -122.38398938548363,
  "latitude": 37.622451999999996,
}
```

To geocode a location description into latitude, longitude location we can use the `apoc.spatial.geocode()` procedure. By default this procedure uses the Nominatim geocoding API but can be configured to use other geocoding services, such as Google Cloud.

Data Import

We can use Cypher to import data into Neo4j from formats such as CSV and JSON, including GeoJSON.

CSV

Using the LOAD CSV Cypher command to create an airport routing graph.

1 - Create a constraint on the field that identies uniquesness, in this case Airport IATA code. This ensures we won’t create duplicate airports but also creates a database index to improve performance of our data import steps below.

```
CREATE CONSTRAINT FOR (a:Airport) REQUIRE a.iata IS UNIQUE;
```

2 - Create Airport nodes, storing their location, name, IATA code, etc as node properties.

```
LOAD CSV WITH HEADERS
FROM "https://cdn.neo4jlabs.com/data/flights/airports.csv"
AS row
MERGE (a:Airport {iata: row.IATA_CODE})
ON CREATE SET a.city = row.CITY,
              a.name = row.AIRPORT,
              a.state = row.STATE,
              a.country = row.country,
              a.location =
                point({ latitude: toFloat(row.LATITUDE),
                        longitude: toFloat(row.LONGITUDE)
                });
```

3 - Create FLIGHT_TO relationships connecting airports with a connecting flight. Increment the num_flights counter variable to keep track of the number of flights between airports per year.

```
LOAD CSV WITH HEADERS
FROM "https://cdn.neo4jlabs.com/data/flights/flights.csv" AS row
CALL {
  WITH row
  MATCH (origin:Airport {iata: row.ORIGIN_AIRPORT})
  MATCH (dest:Airport {iata: row.DESTINATION_AIRPORT})
  MERGE (origin)-[f:FLIGHT_TO]->(dest)
    ON CREATE SET
      f.num_flights = 0, f.distance = toInteger(row.DISTANCE)
    ON MATCH SET
      f.num_flights = f.num_flights + 1
} IN TRANSACTIONS OF 100000 ROWS;
```

GeoJSON

We can also store arrays of Points to represent complex geometries like lines and polygons, for example to represent land parcels.

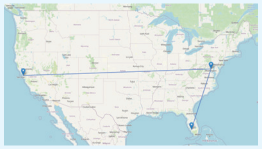
```
CALL apoc.load.json('https://cdn.neo4jlabs.com/data/landgraph/parcels.geojson')
YIELD value
UNWIND value.features AS feature
CREATE (p:Parcel) SET
  p.coordinates = [coord IN feature.geometry.coordinates[0] | point({latitude: coord[1], longitude: coord[0]})]
  p += feature.properties;
```

Routing With Path Finding Algorithms

Shortest Path

The shortestPath function performs a **binary breadth-first search** to find the shortest path between nodes in the graph.

```
MATCH p = shortestPath(
    (:Airport {iata: "SFO"})-[:FLIGHT_TO*..10]->(:Airport {iata: "RSW"})
) RETURN p
```

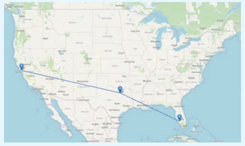


Shortest Weighted Path

Often we want to consider the shortest weighted path taking into account distance, time or some other cost stored as relationship properties. Dijkstra and A* are two algorithms that take relationship (or edge) weights into account when calculating the shortest path.

Dijkstra’s Algorithm

```
MATCH (origin:Airport {iata: "SFO"})
MATCH (dest:Airport {iata: "RSW"})
CALL
  apoc.algo.dijkstra(
    origin,
    dest,
    "FLIGHT_TO",
    "distance"
  )
YIELD path, weight
UNWIND nodes(path) AS n
RETURN {
  airport: n.iata,
  lat: n.location.latitude,
  lng: n.location.longitude
} AS route
```

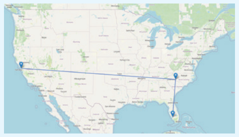


Dijkstra’s algorithm is similar to a breadth-first search, but takes into account relationship properties (distance) and prioritizes exploring low-cost routes first using a priority queue.

A Algorithm*

```
MATCH (origin:Airport {iata: "SFO"})
MATCH (dest:Airport {iata: "RSW"})
CALL
  apoc.algo.aStarConfig(
    origin,
    dest,
    "FLIGHT_TO",
    {
     pointPropName: "location",
     weight: "distance"
    }
  )
YIELD weight, path
RETURN weight, path
```

The A* algorithm adds a heuristic function to choose which paths to explore. In our case the heuristic is the distance to the final destination.



Spatial Cypher Cheat Sheet - Using Neo4j With Python

The Neo4j Python Driver

Creating A GeoDataFrame From Data Stored In Neo4j

In this section we'll use the Neo4j Python Driver to **create a GeoDataFrame** of our flight data. We'll also compute weighted degree centrality so we can plot airport size relative to their "importance" in the US airline network. The **Neo4j Python Driver** can be installed with:

```
pip install neo4j
```

```
import neo4j

NEO4J_URI           = "neo4j://localhost:7689"
NEO4J_USER          = "neo4j"
NEO4J_PASSWORD      = "letmeinnow"
NEO4J_DATABASE      = "neo4j"
```

Connection credentials for our Neo4j database

```
driver = neo4j.GraphDatabase.driver(
  NEO4J_URI, auth=(NEO4J_USER, NEO4J_PASSWORD)
)
```

Create a connection to the database

Define a Cypher query to fetch data from Neo4j

```
AIRPORT_QUERY = """
  MATCH (origin:Airport)-[f:FLIGHT_TO]->(dest:Airport)
  CALL {
   WITH origin
   MATCH (origin)-[f:FLIGHT_TO]-()
   RETURN sum(f.num_flights) AS origin_centrality
  }
  CALL {
   WITH dest
   MATCH (dest)-[f:FLIGHT_TO]-()
   RETURN sum(f.num_flights) AS dest_centrality
  }
  RETURN {
    origin_wkt:
      "POINT (" + origin.location.longitude + " " + origin.location.latitude + ")",
    origin_iata: origin.iata,
    origin_city: origin.city,
    origin_centrality: origin_centrality,
    dest_centrality: dest_centrality,
    dest_wkt:
      "POINT (" + dest.location.longitude + " " + dest.location.latitude + ")",
    dest_iata: dest.iata,
    dest_city: dest.city,
    distance: f.distance,
    num_flights: f.num_flights,
    geometry:
      "LINESTRING (" + origin.location.longitude + " " + origin.location.latitude + ","
      + dest.location.longitude + " " + dest.location.latitude + ")"
    }
```

Weighted degree centrality is a measure of a node's importance in the network and is the sum of all relationship weights connected to a given node.

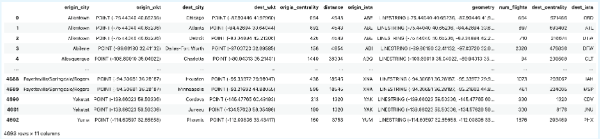
We return the geometry of our origin and destination airports, and the flight route as Well Known Text (WKT). POINT for the airports and LINESTRING for the flight route. We'll parse this WKT when defining the geometry in our GeoDataFrame.

```
def get_airport(tx):
    results = tx.run(AIRPORT_QUERY)
    df = results.to_df(expand=True)
    df.columns =
     ['origin_city','origin_wkt', 'dest_city', 'dest_wkt', 'origin_centrality', 'distance', 'origin_iata',
      'geometry','num_flights', 'dest_centrality', 'dest_iata']
    df['geometry'] = geopandas.GeoSeries.from_wkt(df['geometry'])
    df['origin_wkt'] = geopandas.GeoSeries.from_wkt(df['origin_wkt'])
    df['dest_wkt'] = geopandas.GeoSeries.from_wkt(df['dest_wkt'])
    gdf = geopandas.GeoDataFrame(df, geometry='geometry')
    return gdf
```

The Neo4j Python Driver has a .to_df() function which will convert a Neo4j result set to a pandas DataFrame

Here we parse the WKT columns into GeoSeries and convert our pandas DataFrame into a GeoPandas GeoDataFrame

```
with driver.session(database=NEO4J_DATABASE) as session:
    airport_df = session.execute_read(get_airport)
```

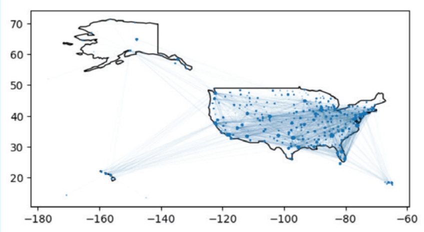


	origin_city	origin_wkt	dest_city	dest_wkt	origin_centrality	distance	origin_iata	geometry	num_flights	dest_centrality	dest_iata
0	Allentown	POINT (-75.44040 40.65236)	Chicago	POINT (-87.90446 41.97960)	654	4543	ABE	LINESTRING (-75.44040 40.65236, -87.90446 41.9...	654	571466	ORD
1	Allentown	POINT (-75.44040 40.65236)	Atlanta	POINT (-84.42694 33.64044)	692	4543	ABE	LINESTRING (-75.44040 40.65236, -84.42694 33.6...	997	693402	ATL
2	Allentown	POINT (-75.44040 40.65236)	Detroit	POINT (-83.34884 42.21206)	426	4543	ABE	LINESTRING (-75.44040 40.65236, -83.34884 42.2...	710	216674	DTW
3	Abilene	POINT (-99.68190 32.41132)	Dallas-Fort Worth	POINT (-97.03720 32.89595)	156	4654	ABI	LINESTRING (-99.68190 32.41132, -97.03720 32.8...	2020	476838	DFW
4	Albuquerque	POINT (-106.60919 35.04022)	Charlotte	POINT (-80.94313 35.21401)	1449	38334	ABQ	LINESTRING (-106.60919 35.04022, -80.94313 35....	94	200508	CLT
...	...	...	...	...	...	...	...	...	...	...	...
4688	Fayetteville/Springdale/Rogers	POINT (-94.30681 36.28187)	Houston	POINT (-95.33972 29.98047)	438	18545	XNA	LINESTRING (-94.30681 36.28187, -95.33972 29.9...	1073	293067	IAH
4689	Fayetteville/Springdale/Rogers	POINT (-94.30681 36.28187)	Minneapolis	POINT (-93.21692 44.88055)	596	18545	XNA	LINESTRING (-94.30681 36.28187, -93.21692 44.8...	461	224005	MSP
4690	Yakutat	POINT (-139.66023 59.50336)	Cordova	POINT (-145.47765 60.49183)	213	1320	YAK	LINESTRING (-139.66023 59.50336, -145.47765 60...	330	1320	CDV
4691	Yakutat	POINT (-139.66023 59.50336)	Juneau	POINT (-134.57628 58.35496)	199	1320	YAK	LINESTRING (-139.66023 59.50336, -134.57628 58...	330	8178	JNU
4692	Yuma	POINT (-114.60597 32.65658)	Phoenix	POINT (-112.00806 33.43417)	160	3753	YUM	LINESTRING (-114.60597 32.65658, -112.00806 33...	1376	293469	PHX

4693 rows × 11 columns

We now have a GeoDataFrame where each row is a flight route between two airports. We can plot the airport and routes, using the centrality metric to size airport nodes: more important airports should be larger.

```
world = geopandas.read_file(
 geopandas.datasets.get_path('naturalearth_lowres')
)

ax = world[world.continent == 'North America']
 .plot(color='white', edgecolor='black')

flights_gdf = flights_gdf.set_geometry("origin_wkt")
flights_gdf.plot(ax=ax, markersize='origin_centrality')

flights_gdf = flights_gdf.set_geometry("geometry")
flights_gdf.plot(ax=ax, markersize=0.1, linewidth=0.01)
```



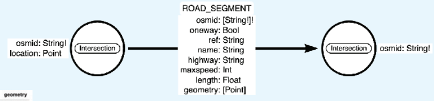
Working With OpenStreetMap Data

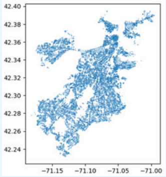
Loading A Road Network WIth OSMNx

In this section we will import data from **OpenStreetMap** into Neo4j using the **OSMNx Python package**. Below is the property graph data model we will use to model the road network of Boston.

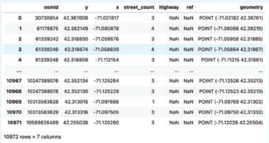
```
pip install osmnx
```

```
import osmnx as ox

G = ox.graph_from_place("Boston, MA, USA", network_type="drive")
fig, ax = ox.plot_graph(G)

gdf_nodes, gdf_relationships = ox.graph_to_gdfs(G)
gdf_nodes.reset_index(inplace=True)
gdf_relationships.reset_index(inplace=True)
```





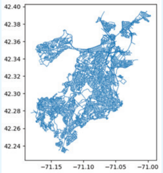
gdf_nodes



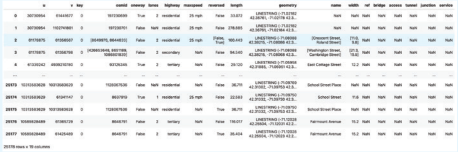
	osmid	y	x	street_count	highway	ref	geometry
0	30730954	42.367608	-71.021817	3	NaN	NaN	POINT (-71.02182 42.36761)
1	61178875	42.382149	-71.080878	4	NaN	NaN	POINT (-71.08088 42.38215)
2	61339242	42.318850	-71.059576	3	NaN	NaN	POINT (-71.05958 42.31885)
3	61339246	42.318674	-71.058839	4	NaN	NaN	POINT (-71.05884 42.31867)
4	61339248	42.318808	-71.112154	3	NaN	NaN	POINT (-71.11215 42.31881)
...	...	...	...	...	...	...	...
10967	10247389076	42.352134	-71.125264	3	NaN	NaN	POINT (-71.12526 42.35213)
10968	10247389078	42.352130	-71.125229	3	NaN	NaN	POINT (-71.12523 42.35213)
10969	10313583628	42.313015	-71.097688	1	NaN	NaN	POINT (-71.09769 42.31302)
10970	10313583629	42.313316	-71.097505	3	NaN	NaN	POINT (-71.09750 42.31332)
10971	10589828489	42.255038	-71.120280	3	NaN	NaN	POINT (-71.12028 42.25504)

10972 rows × 7 columns

Here is our nodes GeoDataFrame. Each row represents an intersection in the Boston road network.



gdf_relationships



	u	v	key	osmid	oneway	lanes	highway	maxspeed	reversed	length	geometry	name	width	ref	bridge	access	tunnel	junction	service
0	30730954	61441677	0	197230699	True	2	residential	25 mph	False	33.072	LINESTRING (-71.02182 42.36761, -71.02178 42.3...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	30730954	1102741801	0	197230701	False	NaN	residential	25 mph	False	278.885	LINESTRING (-71.02182 42.36761, -71.02184 42.3...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	61178875	61356567	0	[8649976, 8644633]	False	2	residential	25 mph	[False, True]	160.443	LINESTRING (-71.08088 42.38215, -71.08086 42.3...	[Crescent Street, Roland Street]	[11.0, 9.8]	NaN	NaN	NaN	NaN	NaN	NaN
3	61178875	61356798	0	[426653648, 8651189, 1086801839]	False	2	secondary	NaN	False	94.540	LINESTRING (-71.08088 42.38215, -71.08068 42.3...	[Washington Street, Cambridge Street]	[21.3, 19.5]	NaN	NaN	NaN	NaN	NaN	NaN
4	61339242	4939210780	0	93125345	True	2	tertiary	NaN	False	29.120	LINESTRING (-71.05958 42.31885, -71.05951 42.3...	East Cottage Street	12.2	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
25173	10313583628	10313583629	0	1128067536	False	NaN	residential	NaN	False	36.711	LINESTRING (-71.09769 42.31302, -71.09753 42.3...	School Street Place	NaN	NaN	NaN	NaN	NaN	NaN	NaN
25174	10313583629	61341147	0	8637919	True	1	residential	25 mph	False	22.583	LINESTRING (-71.09750 42.31332, -71.09730 42.3...	School Street	11.6	NaN	NaN	NaN	NaN	NaN	NaN
25175	10313583629	10313583628	0	1128067536	False	NaN	residential	NaN	True	36.711	LINESTRING (-71.09750 42.31332, -71.09753 42.3...	School Street Place	NaN	NaN	NaN	NaN	NaN	NaN	NaN
25176	10589828489	61365729	0	8646791	False	2	tertiary	NaN	False	116.017	LINESTRING (-71.12028 42.25504, -71.12031 42.2...	Fairmount Avenue	15.2	NaN	NaN	NaN	NaN	NaN	NaN
25177	10589828489	61425489	0	8646791	False	2	tertiary	NaN	True	35.404	LINESTRING (-71.12028 42.25504, -71.12023 42.2...	Fairmount Avenue	15.2	NaN	NaN	NaN	NaN	NaN	NaN

25178 rows × 19 columns

Here is our relationships GeoDataFrame. Each row represents a road segment connecting two intersections.

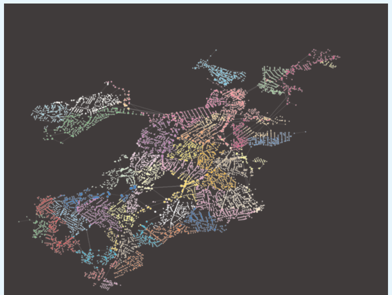
```
node_query = '''
    UNWIND $rows AS row
    WITH row WHERE row.osmid IS NOT NULL
    MERGE (i:Intersection {osmid: row.osmid})
        SET i.location =
         point({latitude: row.y, longitude: row.x }),
            i.ref = row.ref,
            i.highway = row.highway,
            i.street_count = toInteger(row.street_count)
    RETURN COUNT(*) as total
    '''
```

Define a Cypher query to add intersection nodes from the nodes GeoDataFrame

Adding road segments from the relationships GeoDataFrame connecting intersection nodes

```
rels_query = '''
    UNWIND $rows AS road
    MATCH (u:Intersection {osmid: road.u})
    MATCH (v:Intersection {osmid: road.v})
    MERGE (u)-[r:ROAD_SEGMENT {osmid: road.osmid}]->(v)
        SET r.oneway = road.oneway,
            r.lanes = road.lanes,
            r.ref = road.ref,
            r.name = road.name,
            r.highway = road.highway,
            r.max_speed = road.maxspeed,
            r.length = toFloat(road.length)
    RETURN COUNT(*) AS total
    '''

def insert_data(tx, query, rows, batch_size=1000):
    total = 0
    batch = 0

    while batch * batch_size < len(rows):
        results = tx.run(query, parameters = {
          'rows':
            rows[batch * batch_size: (batch + 1) * batch_size]
            .to_dict('records')
        }).data()
        print(results)
        total += results[0]['total']
        batch += 1

with driver.session() as session:
    session.execute_write(insert_data, node_query, gdf_nodes.drop(columns=['geometry']))
    session.execute_write(insert_data, rels_query, gdf_relationships.drop(columns=['geometry']))
```

Because our GeoDataFrames can be very large we break them up into batches to avoid sending too much data to the database at once.



Visualizing the Boston road network using Neo4j Bloom, styled using betweenness centrality and Louvain for neighborhood detection