Spatial_MongoDB

July 3, 2019

1 Spatial Queries with MongoDB

In this notebook, we will go over some of the basics in setting up and using spatial queries with a MongoDB database. In particular, we will use the `BrownDwarf` database we created in the first tutorial and accompanying notebook.

1.1 GeoJSON

MongoDB uses GeoJSON objects and 2dsphere indexes to store geometry information in its database. It's actually fairly straightforward to create an object of this type. You can create polygons, multipolygons, points, ranges (aka, linestrings), but unfortunately cannot create circles. If you wanted a circle you would have to code in an N-point polygon to approximate it, where N would be some moderately large number depending on how precise you want to be.

```
In [1]: # Example point for an object at lon/lat 42, -23
        geojson = { 'type': "Point", 'coordinates': [ 42., -23. ] }
        print(geojson)

        # Example simple polygon
        geojson = {
          'type': "Polygon",
          'coordinates': [ [ [ 0. , 0. ] , [ 3. , 6. ] , [ 6. , 1. ] , [ 0. , 0.  ] ] ]
        }
        print(geojson)
```

```
{'type': 'Point', 'coordinates': [42.0, -23.0]}
{'type': 'Polygon', 'coordinates': [[[0.0, 0.0], [3.0, 6.0], [6.0, 1.0], [0.0, 0.0]]]}
```

It's also possible to write up a function to convert a Virtual Observatory Space-Time Coordinate string to a geoJSON object. Here's a brief example that works with a simple polygon STC-S.

```
In [2]: def stcs_to_geojson(stcs):
            elems = stcs.split()
            if elems[0].upper() != 'POLYGON':
                print('Only works for POLYGON')
                return
```

```
            start = 1
            try:
                x = float(elems[start])
            except ValueError:
                start += 1

            points = []
            vertex = []
            # Loop over values, adding them to a points array
            for i, elem in enumerate(elems[start:]):
                if i%2 == 0 and i != 0:
                    points.append(vertex)
                    vertex = [float(elem)]
                else:
                    vertex.append(float(elem))

            # Add last vertex
            points.append(vertex)

            geojson = {
                'type': "Polygon",
                'coordinates': [ points ]
            }
            return geojson

        stcs = "POLYGON ICRS 0. 0. 3. 6. 6. 1. 0. 0."
        print(stcs)
        print(stcs_to_geojson(stcs))
```

```
POLYGON ICRS 0. 0. 3. 6. 6. 1. 0. 0.
{'type': 'Polygon', 'coordinates': [[[0.0, 0.0], [3.0, 6.0], [6.0, 1.0], [0.0, 0.0]]]}
```

For this tutorial, we won't be using Polygon or more complex shapes. However, if you were storing, say, FITS headers in your database, you could use the WCS information there to store actual footprints to search against.

1.2 Loading our Database

As in the HEALPix tutorial, we'll update our database to include coordinates in this notation and create an index to search for them. We'll use Points to store the geometries as these are point sources.

An important note: in geoJSON, the longitude is constrained to have valid values between -180 and 180 (latitude ranges from -90 and 90). So when we store Right Ascension, we'll want to make the appropriate modifications to store it as a valid longitude.

```
In [3]: import pymongo

        client = pymongo.MongoClient()  # default connection (ie, local)
```

```
        db = client['test']  # database
        dwarfs = db.dwarfs  # collection

        # Remove any existing coords.loc field
        dwarfs.update_many({}, {'$unset': {'coords.loc': ''}})

        # Loop over those without coords.healpix and set the value
        cursor = dwarfs.find({'coords.loc': {'$exists': False}})
        for doc in cursor:
            # RA must have values between -180 and 180
            geojson = { 'type': "Point", 'coordinates': [ doc['coords']['ra']-180., doc['coords'
            print(doc['source_id'], doc['coords']['ra'], doc['coords']['dec'], geojson)
            dwarfs.update_one({'_id': doc['_id']}, {'$set': {'coords.loc': geojson}})
```

```
11 181.889 -39.548 {'type': 'Point', 'coordinates': [1.88900000000001, -39.548]}
2 202.95387 -1.280556 {'type': 'Point', 'coordinates': [22.953869999999995, -1.280556]}
4 222.106791 10.533056 {'type': 'Point', 'coordinates': [42.10679099999999, 10.533056]}
7 219.868167 19.487472 {'type': 'Point', 'coordinates': [39.868167, 19.487472]}
14 342.472709 0.734611 {'type': 'Point', 'coordinates': [162.472709, 0.734611]}
15 332.05679 29.355972 {'type': 'Point', 'coordinates': [152.05678999999998, 29.355972]}
17 6.924875 5.061583 {'type': 'Point', 'coordinates': [-173.075125, 5.061583]}
19 327.068041 40.0665 {'type': 'Point', 'coordinates': [147.068041, 40.0665]}
20 165.54097 -34.509869 {'type': 'Point', 'coordinates': [-14.459030000000013, -34.509869]}
32 63.831417 -9.585167 {'type': 'Point', 'coordinates': [-116.168583, -9.585167]}
34 111.826001 17.167 {'type': 'Point', 'coordinates': [-68.173999, 17.167]}
36 72.753833 -34.0375 {'type': 'Point', 'coordinates': [-107.246167, -34.0375]}
53 228.753459 48.794889 {'type': 'Point', 'coordinates': [48.75345899999999, 48.794889]}
61 191.309 -44.485477 {'type': 'Point', 'coordinates': [11.308999999999997, -44.485477]}
63 53.537667 -49.893944 {'type': 'Point', 'coordinates': [-126.462333, -49.893944]}
80 238.24591 29.81342 {'type': 'Point', 'coordinates': [58.24591000000001, 29.81342]}
82 278.90792 32.998497 {'type': 'Point', 'coordinates': [98.90791999999999, 32.998497]}
83 236.94662 -24.397028 {'type': 'Point', 'coordinates': [56.946619999999996, -24.397028]}
86 9.067376 18.352889 {'type': 'Point', 'coordinates': [-170.932624, 18.352889]}
91 79.692333 -27.946028 {'type': 'Point', 'coordinates': [-100.307667, -27.946028]}
96 42.170846 -16.856022 {'type': 'Point', 'coordinates': [-137.82915400000002, -16.856022]}
98 40.297958 -3.449639 {'type': 'Point', 'coordinates': [-139.702042, -3.449639]}
```

And now we create an index using the `pymongo.GEOSPHERE` specification to create a 2dsphere index:

```
In [4]: if 'geosphere' not in dwarfs.index_information():
            dwarfs.create_index([('coords.loc', pymongo.GEOSPHERE)],
                                name='geosphere', background=True)
```

With an appropriate 2dsphere index created, you can make use of a variety of operators to search your database. These include `$geoWithin`, which returns objects that are entirely within a geoJSON geometry you specify; `$geoIntersects`, which returns anything that interscets your specified geometry (more useful for when your database contains things like polygons rather

than points); $near and $nearSphere, which returns an ordered list of targets within some $maxDistance in **meters**. This last part should illustrate what problem we'll have: these queries are based on the assumption that you are querying on the Earth's surface. We'll need to make some changes to use them for astronomical sources.

1.3 Astronomical Modifications

The MongoDB implementation of searches across geoJSON coordinates uses the WGS84 reference system. It's beyond the scope of this notebook to fully describe the World Geodetic System, but for what we are concerned, this models the Earth as an oblate spheroid so we need to take this into account when we construct any queries.

The way we'll get around this is to calculate the length of one degree, expressed in meters, as a function of latitude. This will allow us to scale our search radius to account for the curvature of the Earth. For clarity, here's the equation I used for computing the radius of curvature along the Meridian:

```
In [5]: import numpy as np

        def wgs_scale(lat):
            # Get scaling to convert degrees to meters at a given geodetic latitude (declination

            # Values from WGS 84
            a = 6378137.000000000000 # Semi-major axis of Earth
            b = 6356752.314140000000 # Semi-minor axis of Earth
            e = 0.081819190842600 # eccentricity
            angle = np.radians(1.0)

            # Compute radius of curvature along meridian (see https://en.wikipedia.org/wiki/Meri
            rm = a * (1 - np.power(e, 2)) / np.power( ( (1 - np.power(e, 2) * np.power( np.sin(n

            # Compute length of arc at this latitude (meters/degree)
            arc = rm * angle
            return arc
```

Let's use this to a search for targets with the $nearSphere operator.

```
In [6]: ra, dec = 220., 12.
        lon, lat = ra-180., dec
        radius = 10 # degrees
        radius *= wgs_scale(lat) # meters
        print(lon, lat, radius)

        # Perform the search
        cursor = dwarfs.find({'coords.loc': {
            '$nearSphere': {
                '$geometry': { 'type': 'Point',
                               'coordinates': [lon, lat]},
                '$maxDistance': radius
            } } })
```

```
        for doc in cursor:
            print(doc['source_id'], doc['coords']['ra'], doc['coords']['dec'], doc['coords']['lo
```

```
40.0 12.0 1106222.9012842258
4 222.106791 10.533056 {'type': 'Point', 'coordinates': [42.10679099999999, 10.533056]}
7 219.868167 19.487472 {'type': 'Point', 'coordinates': [39.868167, 19.487472]}
```

For simplicity, we can wrap this up in a function as follows:

```
In [7]: def cone_search(ra, dec, radius, collection=dwarfs, field='coords.loc'):

            scaling = wgs_scale(dec)
            meter_radius = radius * scaling
            lon, lat = ra-180., dec

            cursor = collection.find({'coords.loc': {
            '$nearSphere': {
                '$geometry': { 'type': 'Point',
                               'coordinates': [lon, lat]},
                '$maxDistance': meter_radius
            } } })

            return cursor
```

```
In [8]: # Use the function
        cursor = cone_search(220., 12., 10.)
        for doc in cursor:
            print(doc['source_id'], doc['coords']['ra'], doc['coords']['dec'], doc['coords']['lo
```

```
4 222.106791 10.533056 {'type': 'Point', 'coordinates': [42.10679099999999, 10.533056]}
7 219.868167 19.487472 {'type': 'Point', 'coordinates': [39.868167, 19.487472]}
```

If you want to mix and match queries, for example, a cone search just for L dwarfs, you can make use of the aggregation operator `$geoNear`. Here's an example:

```
In [9]: # For more details: https://docs.mongodb.com/manual/reference/operator/aggregation/geoNe
        cursor = dwarfs.aggregate( [
           {
              '$geoNear': {
                 'near': { 'type': "Point", 'coordinates': [ 40., 12. ] },
                 'spherical': True,
                 'query': { 'spectral_type': {'$regex' : 'L'}  },
                 'distanceField': "distance"
              }
           }
        ] )
        for doc in cursor:
            print(doc['source_id'], doc['spectral_type'], doc['coords']['ra'], doc['coords']['de
                  doc['distance']/wgs_scale(doc['coords']['dec']))  # convert calculated distanc
```

```
4 L3.5 222.106791 10.533056 2.550134503550854
7 L1.0 219.868167 19.487472 7.530547086210149
2 L6.0 202.95387 -1.280556 21.663684937415425
80 L0.0 238.24591 29.81342 24.683344761956167
53 L6.0 228.753459 48.794889 37.54508985231937
19 L6.0 327.068041 40.0665 95.16875104196578
15 L3.0 332.05679 29.355972 103.04662327506185
14 L1.0 342.472709 0.734611 122.31509257954042
86 L3.5 9.067376 18.352889 137.74573289868712
91 L0.0 79.692333 -27.946028 140.30110792007784
36 L0.5 72.753833 -34.0375 143.4554019787131
98 L0.0 40.297958 -3.449639 172.59272649495955
```

Note that the aggregation result added a `distance` column containing the distance to each result. This was output in meters, but I scaled it back to degrees using the same function `wgs_scale`. Had I supplied a maxDistance option or a limit option I could have reduced the number of outputs. In this case, I got every L dwarf in the database sorted by distance from my supplied coordinates. It should be fairly straight forward to udpate our cone search to also take in optional query parameters for a more robust search.

```
In [10]: def cone_search_advanced(ra, dec, radius, query={}, collection=dwarfs, field='coords.lo

             scaling = wgs_scale(dec)
             meter_radius = radius * scaling
             lon, lat = ra-180., dec

             cursor = dwarfs.aggregate( [
            {
               '$geoNear': {
                  'near': { 'type': "Point", 'coordinates': [ lon, lat ] },
                  'spherical': True,
                  'maxDistance': meter_radius,
                  'query': query,
                  'distanceField': distance_field
               }
            } ] )

             return cursor

In [11]: # All M-dwarfs within 45 degrees of 220, -32, sorted by distance
         cursor = cone_search_advanced(220., -23., radius=45, query={'spectral_type' : {'$regex'
         for doc in cursor:
             print(doc['source_id'], doc['spectral_type'], doc['coords']['ra'], doc['coords']['d
                   round(doc['distance']/wgs_scale(doc['coords']['dec']), 2))  # convert calcula
```

```
83 M9.0 236.94662 -24.397028 15.65
61 M9.0 191.309 -44.485477 31.86
```

```
11 M8.0 181.889 -39.548 36.27
```

Now, how close are these distance estimates? Let's use astropy functions to actually compute that.

```
In [12]: from astropy import units as u
         from astropy.coordinates import SkyCoord

         i_ra, i_dec = 220., 15.
         # Manual search to include all objects irrespective of distance or filters
         cursor = dwarfs.aggregate( [
            {
               '$geoNear': {
                  'near': { 'type': "Point", 'coordinates': [ i_ra - 180., i_dec ] },
                  'spherical': True,
                  'distanceField': "distance"
               }
            }] )

         print('Search location: {} {}'.format(i_ra, i_dec))
         print('ra\tdec\tmongo\tastropy\tdiff\tpercent')
         for doc in cursor:
             ra, dec = doc['coords']['ra'], doc['coords']['dec']
             m_dist = doc['distance']/wgs_scale(doc['coords']['dec'])

             c1 = SkyCoord(ra=ra*u.deg, dec=dec*u.deg, frame='icrs')
             c2 = SkyCoord(ra=i_ra*u.deg, dec=i_dec*u.deg, frame='icrs')
             sep = c1.separation(c2)
             diff = m_dist - sep.degree
             print('{:.3f} {:.3f}\t{:.2f}\t{:.2f}\t{:.3f}\t{:.2f}'.format(ra, dec, m_dist, sep.d
```

```
Search location: 220.0 15.0
ra        dec        mongo        astropy        diff        percent
219.868 19.487        4.51        4.49        0.025        0.56
222.107 10.533        4.95        4.92        0.031        0.64
238.246 29.813        22.48        22.39        0.095        0.42
202.954 -1.281        23.60        23.44        0.158        0.67
228.753 48.795        34.59        34.55        0.035        0.10
236.947 -24.397        42.94        42.73        0.214        0.50
278.908 32.998        56.20        55.99        0.209        0.37
191.309 -44.485        65.08        64.97        0.115        0.18
181.889 -39.548        65.26        65.09        0.172        0.26
165.541 -34.510        71.83        71.58        0.250        0.35
327.068 40.066        93.12        92.89        0.237        0.25
332.057 29.356        101.34        100.91        0.434        0.43
111.826 17.167        102.81        102.21        0.598        0.59
342.473 0.735        121.83        121.01        0.815        0.67
```

```
9.067 18.353        135.59        134.82        0.773        0.57
79.692 -27.946        141.70        141.07        0.637        0.45
6.925 5.062        142.52        141.57        0.942        0.67
53.538 -49.894        143.53        143.41        0.118        0.08
72.754 -34.038        145.41        144.89        0.517        0.36
63.831 -9.585        157.12        156.11        1.007        0.65
40.298 -3.450        169.57        168.45        1.128        0.67
42.171 -16.856        178.25        177.21        1.043        0.59
```

From the result above, we can see that the differences can be quite large, up to a degree, in terms of the computed distance. However, this is generally the case when the distances are fairly large. Within separations of 20 degrees or less, we see differences of order 0.1 degrees or less. At 5 degrees, the separation difference is of order 0.03 degrees, or 1.8 arcminutes. Remember, though, that we are **converting a distance in meters at single latitude** to an arc length. Ideally, we would be integrating over the surface of the sphere since the scaling between degrees to meters changes at each location. This difference would naturally become more prominent when the separation is over a degree. If your searches are typically small you should have no issues. However, you may still want to manually compute distances yourself from any cone search results.

1.4 Final Thoughts

In this notebook, we briefly went over the geoJSON notation and how to use geospatial queries with MongoDB. There is a lot of details we glossed over, so I encourage you to look over the documentation based on what you want to do. In contrast to the HEALPix tutorial, these searches are faster as they do not require us computing a large range of pixel values to query against. However, we do run into conversion issues as MongoDB uses a WGS84 reference system for its coordinate searches as opposed to a perfect sphere. It's not too difficult to work around these, but they're important to bear in mind if you require very precise results.