SA_naive_init

May 26, 2025

1 Covered the following in 4 hours (Win10+WSL2), zero prior knowledge of maps

[1]:
```
import geopandas as gpd, sqlalchemy as sa

PG_DSN = "postgresql+psycopg://postgres:geo@localhost:5432/osm"
engine = sa.create_engine(PG_DSN)
```

[2]:
```
# quick smoke-test, should give (1,)
with engine.connect() as conn:
    print(conn.execute(sa.text("SELECT 1")).fetchone())
```

```
(1,)
```

[3]:
```
roads = gpd.read_postgis(
    """
    SELECT way AS geom, highway
    FROM planet_osm_line
    WHERE highway IS NOT NULL
    """,
    engine,
    geom_col="geom"
)
roads.head()
```

[3]:

	geom	highway
0	LINESTRING (3593834.467 4178048.947, 3593822.7…	track
1	LINESTRING (3603831.847 4169794.22, 3603840.50…	track
2	LINESTRING (3603835.733 4169822.583, 3603825.7…	track
3	LINESTRING (3603653.859 4169787.693, 3603665.5…	footway
4	LINESTRING (3603606.225 4169766.659, 3603610.4…	footway

[4]:
```
import h3, geopandas as gpd
from shapely.geometry import box, Polygon

# Cyprus bounding box as Shapely polygon
```

```
# Builds an axis-aligned rectangle (Polygon) whose lower-left vertex is␣
 ↪(min_lon, min_lat) = (32.25, 34.5) and upper-right vertex is (max_lon,␣
 ↪max_lat) = (34.7, 35.7).
bbox_poly = box(32.25, 34.5, 34.7, 35.7)

# Res 8  0.6 km² hexes (~450 m across)
RES = 8
# This is the core spatial discretisation: one H3 index per hex covering the␣
 ↪island.
# Those IDs are what we'll later join against roads, POIs, etc.
# 1) Converts the Shapely polygon to a GeoJSON dict
# 2) Asks H3 to return every cell ID whose center lies inside the polygon at␣
 ↪resolution RES
# 3) Turns the generator into a Python list of 64-bit H3 indexes
hex_ids = list(h3.polyfill_geojson(bbox_poly.__geo_interface__, RES))

# GeoPandas needs real geometries, not just IDs, to plot, spatial-join, or␣
 ↪write to files.
# List-comprehension that, for every H3 ID:
# • returns its corner coordinates as a [(lat, lon), …] list
# • wraps those corners in a Shapely Polygon
# The result is a Python list of Shapely hex outlines.
geoms = [Polygon(h3.h3_to_geo_boundary(h, geo_json=True)) for h in hex_ids]

# Builds a GeoPandas GeoDataFrame with two columns:
# • the integer/str cell IDs
# • their hex outlines
# Setting crs='EPSG:4326' tags the coordinates as WGS-84 lat/lon.
# The result is a first-class spatial table to plot, save it to GeoPackage/
 ↪Parquet, join it against road segments, aggregate metrics
#
# CRS = Coordinate Reference System - a formal definition of how numbers in␣
 ↪your geometry columns map to positions on the globe
# EPSG: European Petroleum Survey Group, https://en.wikipedia.org/wiki/
 ↪EPSG_Geodetic_Parameter_Dataset#Common_EPSG_codes
# WGS: World Geodetic System, https://en.wikipedia.org/wiki/
 ↪World_Geodetic_System#WGS_84
# EPSG:4326 is the EPSG code for the WGS-84 geographic CRS:
# • axes are latitude, longitude (in that order when stored as tuples)
# • units are decimal degrees
# • it's the default system used by GPS, OSM (OpenStreetMap), most web APIs,␣
 ↪and MapLibre/Leaflet tiles
hex_gdf = gpd.GeoDataFrame({'h3': hex_ids, 'geometry': geoms}, crs="EPSG:4326")
hex_gdf.head()
```

[4]:

	h3	geometry
0	883f6b210dfffff	POLYGON ((32.41637 35.48444, 32.41364 35.47986…
1	883f6b2e41fffff	POLYGON ((32.55191 35.44619, 32.54918 35.44161…
2	882da4609bfffff	POLYGON ((33.96336 35.41412, 33.96058 35.40959…
3	882da625b9fffff	POLYGON ((34.59282 34.81763, 34.59004 34.8131,…
4	882da685d1fffff	POLYGON ((33.65528 34.80702, 33.65253 34.80246…

[5]:

```
# osm2pgsql imports OSM geometries in EPSG 3857 (Web-Mercator, metres).
# This lets GeoPandas receive WGS-84 geometries that match our hex grid:
# ST_Transform(way,4326) for geometry and ST_Length(way) / 1000.0 for km

SQL_ROADS = """
SELECT
    ST_Transform(way, 4326)               AS geom,      -- geometry in lat/lon
    highway,
    name,
    ST_Length(way) / 1000.0              AS length_km   -- metres → km (3857)
FROM planet_osm_line
WHERE highway IS NOT NULL
      AND highway NOT IN ('footway', 'path', 'cycleway');
"""
roads = gpd.read_postgis(SQL_ROADS, con=engine, geom_col="geom", crs="EPSG:
 ↪4326")
# This should tell that we fetched  153 k road segments with 4 columns (geom,␣
 ↪highway, name, length_km)
print(roads.shape)
# Sanity check
roads.head()
```

```
(153479, 4)
```

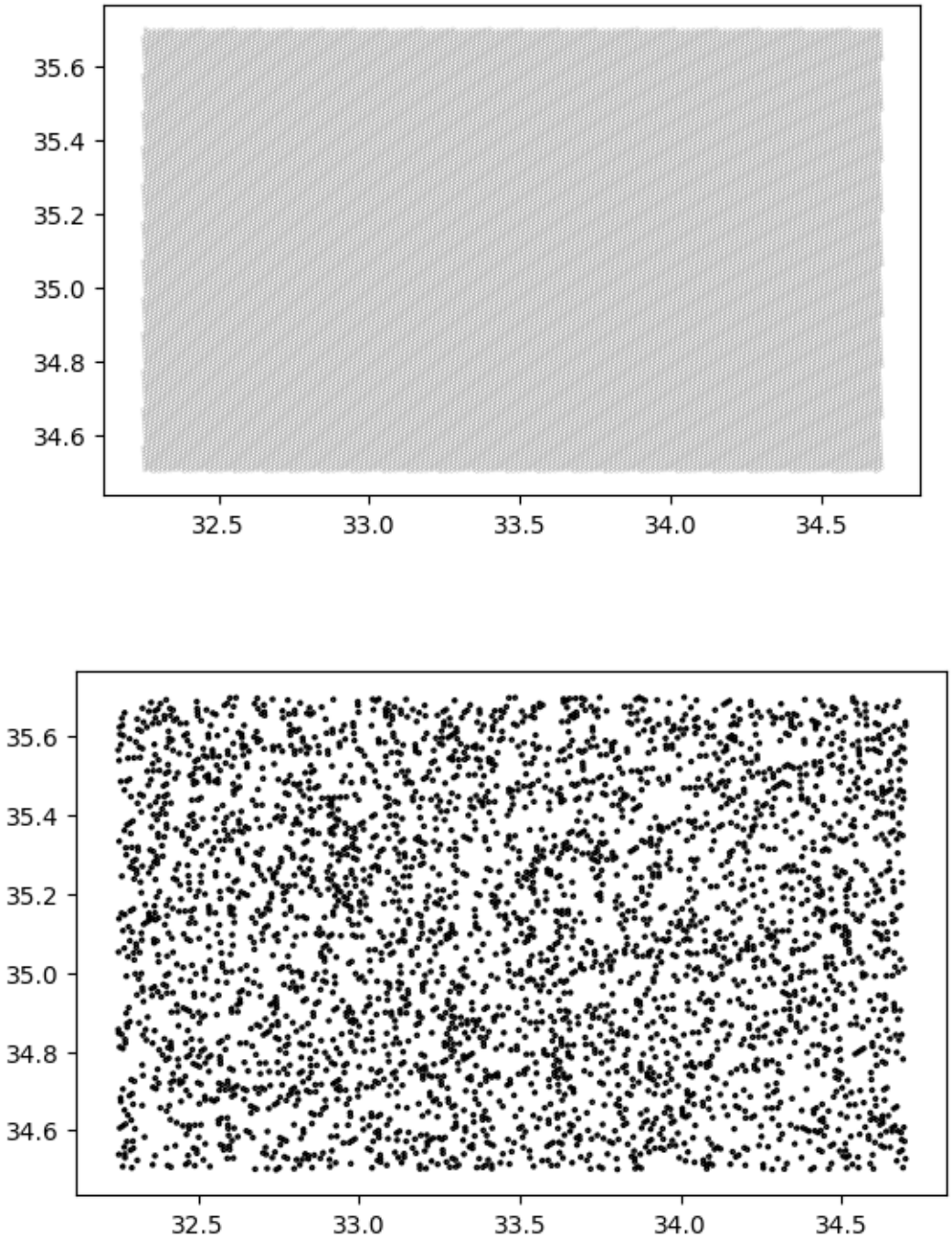
[5]:

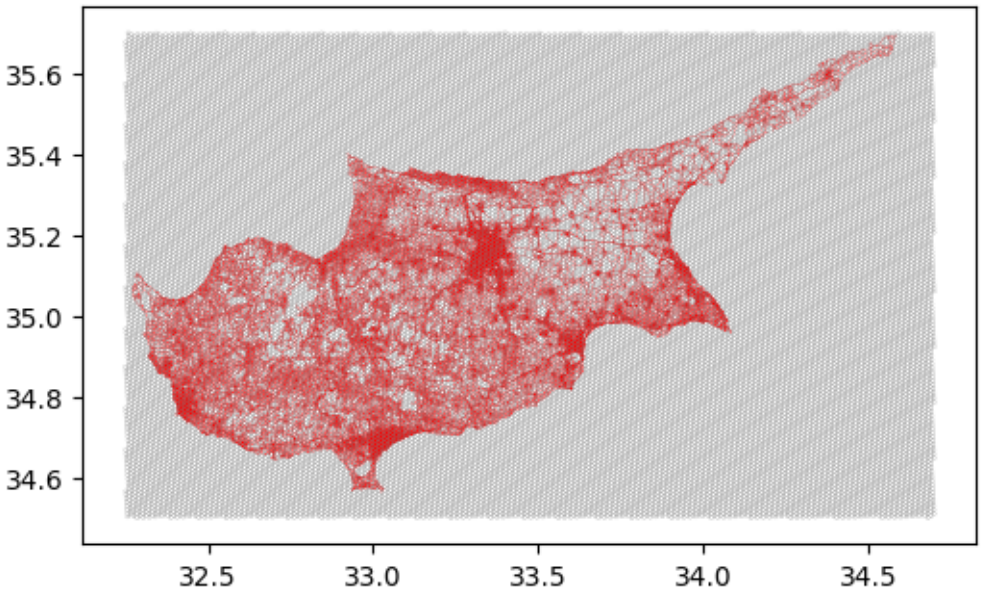
	geom	highway	name	length_km
0	LINESTRING (32.28396 35.10419, 32.28386 35.104…	track	None	0.081982
1	LINESTRING (32.37377 35.0435, 32.37385 35.0434…	track	None	0.239232
2	LINESTRING (32.37381 35.04371, 32.37372 35.043…	track	None	0.113109
3	LINESTRING (32.37097 35.04309, 32.371 35.04312…	service	None	0.060353
4	LINESTRING (32.37029 35.043, 32.3703 35.0431, …	service	None	0.066558

[6]:

```
# Show only the outlines (no fill, thinner lines)
hex_gdf.boundary.plot(linewidth=0.15, edgecolor='grey', figsize=(6,6))

# Random 10 % sample
hex_gdf.sample(frac=0.1).plot(edgecolor='black', facecolor='none')

# Plot roads under the grid
ax = roads.plot(color='red', linewidth=0.2, figsize=(6,6))
hex_gdf.boundary.plot(ax=ax, linewidth=0.15, edgecolor='grey')
```

[6]: <Axes: >



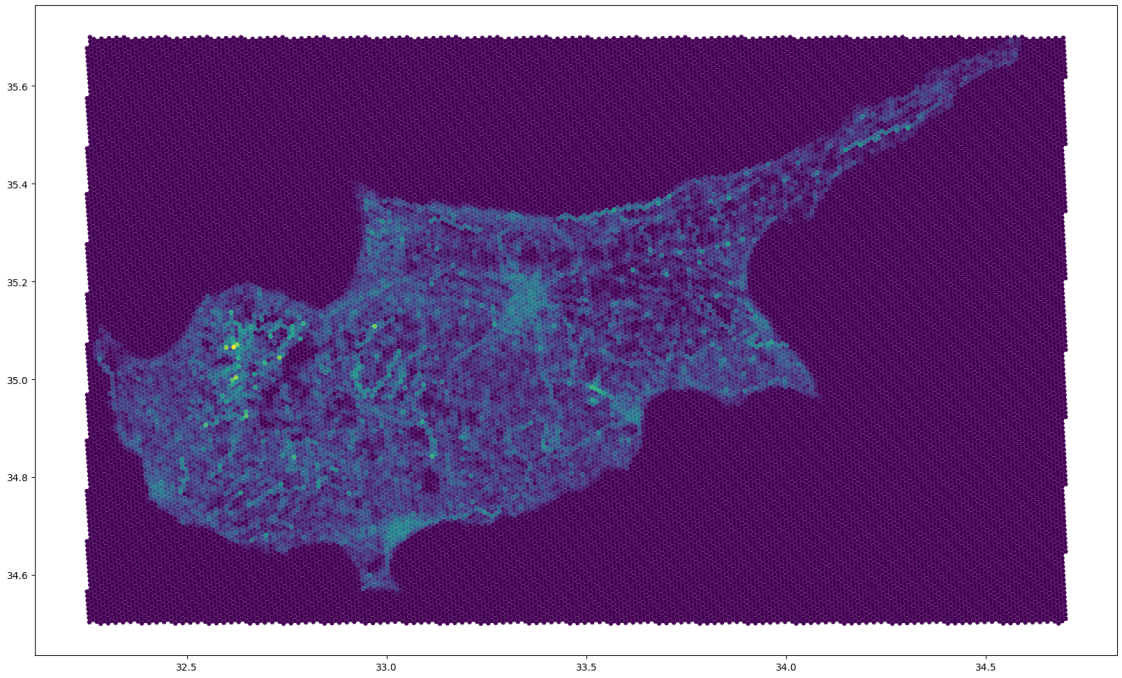


[7]:
```
# Converts geometric overlap into a simple attribute key (h3) you can aggregate␣
 ↪on.
# • gpd.sjoin = spatial join
# • Takes every LINESTRING in roads and finds the hex(es) in hex_gdf whose␣
 ↪polygons intersect it
# • how="inner" keeps only matches; road segments that fall outside the hex␣
 ↪grid are dropped
# • The join copies the hex's h3 value onto each matching road row
# Result: joined still looks like a road table, but now has an extra h3 column␣
 ↪telling you which cell each segment belongs to
joined = gpd.sjoin(roads, hex_gdf, how="inner", predicate="intersects")

# Computes the length_km metric: the total kilometres of OSM road centre-lines␣
 ↪whose geometry falls inside each H3 cell at resolution 8.
# • groupby('h3') bundles all rows that share the same cell ID
# • .length_km.sum() totals the pre-computed kilometre length in each bundle
# • reset_index() turns the grouped index back into an ordinary column so the␣
 ↪result is a two-column DataFrame
coverage = (joined.groupby('h3') # one bin per cell
            .length_km.sum() # add up km of all roads inside
            .reset_index())

# Produces a final hex_cov GeoDataFrame:
# every H3 cell polygon plus a numeric coverage value,
# ready for choropleth plotting, export to Parquet, or loading into Superset.
# • hex_gdf has all hexes-even those with no roads
```

```
# • .merge(..., how='left') attaches the length_km totals to the matching hex␣
 ↪by key h3
# • Hexes that had no matching roads come out with NaN; .fillna({'length_km':␣
 ↪0}) turns those into 0 km
hex_cov = (hex_gdf
    .merge(coverage, on='h3', how='left') # attach the metric
           .fillna({'length_km': 0})
          )
```

[8]:
```
hex_cov.plot(column='length_km', cmap='viridis',
             linewidth=0, figsize=(20,20), legend=False)
```

[8]: <Axes: >



[9]:
```
# hex_cov.to_parquet("cyprus_res8_roadkm.parquet", index=False)
```

2 Tentative plan for further study

2.1 Week 1

Day	Objective	Key commands / checkpoints
1	**Advanced OSM import** – flex-config, tag filters, diff tables	*osm2pgsql –append* with `--flex-config=cyprus.lua`; monitor `pg_stat_statements`

Day	Objective	Key commands / checkpoints
2	**PostGIS tuning drill** – try SP-GIST vs GIST on `planet_osm_line`	`CREATE INDEX ... USING SPGIST (way);` and compare `EXPLAIN ANALYZE`
3	**Vector tiles** – Tippecanoe → PMTiles	`tippecanoe -zg -o cyprus.pmtiles roads.geojson` ; `pmtiles index cyprus.pmtiles`
4	**Tile serving & render** – go-pmtiles + MapLibre	`go-pmtiles serve cyprus.pmtiles`; simple HTML using MapLibre GL JS 3.x
5	**Land/sea mask + styling**	Download Natural Earth coastlines; `ogr2ogr` into PostGIS; add second vector layer; adjust style JSON

Deliverables:

- running local map stack with custom tiles
- MBTiles vs PMTiles comparison
- disk/RAM numbers observed

2.2 Week 2

Day	Objective	Key commands / checkpoints
6	**Minutely diff replication**	`osmium apply-update cyprus.pbf state.txt -o diff.osc.gz` → flex-config `--append`
7	**OSRM backend & latency bench**	`docker run osrm/osrm-backend osrm-extract` → `/route` & `/match`; benchmark with Locust
8	**docker-compose stack + healthchecks**	define PostGIS, OSRM, go-pmtiles; `healthcheck:` for each service
9	**Terraform GCP sandbox**	modules for Cloud SQL (PostGIS 15), GCS bucket, Cloud Run serving `pmtiles`; run `terraform plan` → `apply`
10	**Superset dashboard + design doc**	`docker-compose up superset`; build H3 coverage & OSRM latency charts; 1-page design trade-off (vector-vs-raster, diff cadence, cost)

Deliverable:

- docker-compose.yml + terraform directory
- Superset dashboard screenshot
- 1-page PDF "Cyprus mini-pipeline: scaling plan & costs"