CrateDB Multi-Model Data Workshop

This workbook explores multi-model data queries with CrateDB, using data from the City of Chicago. You'll work with tables that contain data for:

- The 77 community areas that make up Chicago including their names, populations, and geospatial polygons describing each community's shape.
- 311 calls / reports from April 2024. 311 is a community issue reporting service: each report contains detail of the type of issue reported (for example graffiti or a broken streetlight), the status of the job, the location of the issue etc.
- Libraries located around the city: their locations, opening hours and other metadata.

We'll use maps to visualize the data, making this a fun, interactive experience.

Install Dependencies

First, install the required dependencies by executing the `pip install` command below.

```
! pip install -U ipyleaflet sqlalchemy-cratedb pandas
```

```
Requirement already satisfied: argon2-cffi in /usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1->widgetsnbextension~=
Requirement already satisfied: jupyter-core>=4.6.1 in /usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1->widgetsnbext
Requirement already satisfied: nbformat in /usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1->widgetsnbextension~=3.6
Requirement already satisfied: nbconvert>=5 in /usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1->widgetsnbextension~
Requirement already satisfied: nest-asyncio>=1.5 in /usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1->widgetsnbexten
Requirement already satisfied: Send2Trash>=1.8.0 in /usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1->widgetsnbexten
Requirement already satisfied: terminado>=0.8.3 in /usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1->widgetsnbextens
Requirement already satisfied: prometheus-client in /usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1->widgetsnbexten
Requirement already satisfied: nbclassic>=0.4.7 in /usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1->widgetsnbextens
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.10/dist-packages (from pexpect>4.3->ipython>=4.0.0->ipyw
Requirement already satisfied: wcwidth in /usr/local/lib/python3.10/dist-packages (from prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.
Requirement already satisfied: platformdirs>=2.5 in /usr/local/lib/python3.10/dist-packages (from jupyter-core>=4.6.1->notebook>=
Requirement already satisfied: notebook-shim>=0.2.3 in /usr/local/lib/python3.10/dist-packages (from nbclassic>=0.4.7->notebook>=
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packages (from nbconvert>=5->notebook>=4.4.1->wid
Requirement already satisfied: bleach!=5.0.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert>=5->notebook>=4.4.1->widg
Requirement already satisfied: defusedxml in /usr/local/lib/python3.10/dist-packages (from nbconvert>=5->notebook>=4.4.1->widgets
Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python3.10/dist-packages (from nbconvert>=5->notebook>=4.4.1
Requirement already satisfied: mistune<4,>=2.0.3 in /usr/local/lib/python3.10/dist-packages (from nbconvert>=5->notebook>=4.4.1->
Requirement already satisfied: nbclient>=0.5.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert>=5->notebook>=4.4.1->wi
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from nbconvert>=5->notebook>=4.4.1->widgetsn
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert>=5->notebook>=4.4.
Requirement already satisfied: tinycss2 in /usr/local/lib/python3.10/dist-packages (from nbconvert>=5->notebook>=4.4.1->widgetsnb
Requirement already satisfied: fastjsonschema>=2.15 in /usr/local/lib/python3.10/dist-packages (from nbformat->notebook>=4.4.1->w
Requirement already satisfied: jsonschema>=2.6 in /usr/local/lib/python3.10/dist-packages (from nbformat->notebook>=4.4.1->widget
Requirement already satisfied: argon2-cffi-bindings in /usr/local/lib/python3.10/dist-packages (from argon2-cffi->notebook>=4.4.1
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/dist-packages (from bleach!=5.0.0->nbconvert>=5->noteboo
Requirement already satisfied: attrs>=22.2.0 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6->nbformat->notebook
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=
Requirement already satisfied: referencing>=0.28.4 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6->nbformat->no
Requirement already satisfied: rpds-py>=0.7.1 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6->nbformat->noteboo
Requirement already satisfied: jupyter-server<3,>=1.8 in /usr/local/lib/python3.10/dist-packages (from notebook-shim>=0.2.3->nbcl
Requirement already satisfied: cffi>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from argon2-cffi-bindings->argon2-cffi->no
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.10/dist-packages (from beautifulsoup4->nbconvert>=5->noteb
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-packages (from cffi>=1.0.1->argon2-cffi-bindings->argo
Requirement already satisfied: anyio<4,>=3.1.0 in /usr/local/lib/python3.10/dist-packages (from jupyter-server<3,>=1.8->notebook-
Requirement already satisfied: websocket-client in /usr/local/lib/python3.10/dist-packages (from jupyter-server<3,>=1.8->notebook
Requirement already satisfied: idna>=2.8 in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0->jupyter-server<3,>=1.8
Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0->jupyter-server<3,>=
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0->jupyter-server<3,
Downloading sqlalchemy_cratedb-0.40.1-py3-none-any.whl (44 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 44.8/44.8 kB 2.7 MB/s eta 0:00:00
Downloading verlib2-0.2.0-py3-none-any.whl (8.9 kB)
Downloading pandas-2.2.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (13.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.1/13.1 MB 52.8 MB/s eta 0:00:00
Downloading crate-1.0.1-py3-none-any.whl (31 kB)
Downloading geojson-3.1.0-py3-none-any.whl (15 kB)
Downloading jedi-0.19.2-py2.py3-none-any.whl (1.6 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 43.3 MB/s eta 0:00:00
Installing collected packages: verlib2, jedi, geojson, pandas, crate, sqlalchemy-cratedb
  Attempting uninstall: pandas
    Found existing installation: pandas 2.2.2
    Uninstalling pandas-2.2.2:
      Successfully uninstalled pandas-2.2.2
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the
cudf-cu12 24.10.1 requires pandas<2.2.3dev0,>=2.0, but you have pandas 2.2.3 which is incompatible.
google-colab 1.0.0 requires pandas==2.2.2, but you have pandas 2.2.3 which is incompatible.
Successfully installed crate-1.0.1 geojson-3.1.0 jedi-0.19.2 pandas-2.2.3 sqlalchemy-cratedb-0.40.1 verlib2-0.2.0
```

Connect to CrateDB

Before going any further, you'll need to update the code below to include a connection string for your CrateDB cluster. If you prefer, you can set the environment variable `CRATEDB_CONNECTION_STRING` instead.

The code below assumes that you're using a managed CrateDB Cloud cluster. If you're running CrateDB locally (for example with Docker), use the "localhost" code block instead.

```
import os
import sqlalchemy as sa

# Define database address when using CrateDB Cloud.
# Please find these settings on your cluster overview page.
CONNECTION_STRING = os.environ.get(
    "CRATEDB_CONNECTION_STRING",
    "crate://<USERNAME>:<PASSWORD>@<HOST>/?ssl=true",
)

# # Define database address when using CrateDB on localhost.
# CONNECTION_STRING = os.environ.get(
#    "CRATEDB_CONNECTION_STRING",
#    "crate://crate@localhost/",
# )

# Connect to CrateDB using SQLAlchemy.
engine = sa.create_engine(
    CONNECTION_STRING,
    echo=sa.util.asbool(os.environ.get("DEBUG", "false")))
connection = engine.connect()
```

Create Tables

Next, we'll create three tables as follows:

- `community_areas` - to contain document data about the 77 community areas that make up the city of Chicago.
- `three_eleven_calls` - details about service requests placed with the Chicago 311 non-emergency issue reporting service.
- `libraries` - data about Chicago's public libraries, including their locations and opening times.

Run the code below to create them, taking a moment to understand the table schemas.

```
_ = connection.execute(sa.text(
"""
CREATE TABLE IF NOT EXISTS community_areas (
   areanumber INTEGER PRIMARY KEY,
   name TEXT,
   details OBJECT(DYNAMIC) AS (
       description TEXT INDEX USING fulltext,
       population BIGINT
   ),
   boundaries GEO_SHAPE INDEX USING geohash WITH (PRECISION='1m', DISTANCE_ERROR_PCT=0.025)
);
"""))

_  = connection.execute(sa.text(
"""
CREATE TABLE IF NOT EXISTS three_eleven_calls (
   srnumber TEXT,
   srtype TEXT,
   srshortcode TEXT,
   createddept TEXT,
   ownerdept TEXT,
   status TEXT,
   origin TEXT,
   createddate TIMESTAMP,
   lastmodifieddate TIMESTAMP,
   closeddate TIMESTAMP,
   week GENERATED ALWAYS AS date_trunc('week', createddate),
   isduplicate BOOLEAN,
   createdhour SMALLINT,
   createddayofweek SMALLINT,
   createdmonth SMALLINT,
   locationdetails OBJECT(DYNAMIC) AS (
       streetaddress TEXT,
       city TEXT,
       state TEXT,
       zipcode TEXT,
       streetnumber TEXT,
       streetdirection TEXT,
       streetname TEXT,
       streettype TEXT,
       communityarea SMALLINT,
       ward SMALLINT,
       policesector SMALLINT,
       policedistrict SMALLINT,
       policebeat SMALLINT,
       precinct SMALLINT,
       latitude DOUBLE PRECISION,
       longitude DOUBLE PRECISION,
       location GEO_POINT
   )
) PARTITIONED BY (week);
"""))

_ = connection.execute(sa.text(
"""
CREATE TABLE IF NOT EXISTS libraries (
   name TEXT,
   location OBJECT(DYNAMIC) AS (
       address TEXT,
       zipcode TEXT,
       communityarea INTEGER,
       position GEO_POINT
   ),
   hours ARRAY(TEXT),
   phone TEXT,
   website TEXT
);
"""))
```

Loading the Data

We'll load the data from files contained in the `cratedb-datasets` public GitHub repository. There's one file for each table:

- Data for the `community_areas` table is contained in a JSON file named `chicago_community_areas.json`.
- Data for the `three_eleven_calls` table is contained in a compressed JSON file named `311_records_apr_2024.json.gz`.
- Data for the `libraries` table is contained in a JSON dile named `chicago_libraries.json`.

The following code populates each table in turn, using `COPY FROM` statements.

```
def display_results(table_name, info):
    print(f"{table_name}: loaded {info['success_count']}, errors: {info['error_count']}")

    if info["error_count"] > 0:
        print(f"Errors: {info['errors']}")

# Load the community areas data file.
result = connection.execute(sa.text("""
    COPY community_areas
    FROM 'https://github.com/crate/cratedb-datasets/raw/main/academy/chicago-data/chicago_community_areas.json'
    RETURN SUMMARY;
    """))

display_results("community_areas", result.mappings().first())
```

```
community_areas: loaded 0, errors: 77
Errors: {'A document with the same primary key exists already': {'count': 77, 'line_numbers': [2, 8, 9, 19, 22, 33, 41, 42, 43, 44,
```

```
# Load the 311 calls data file.
result = connection.execute(sa.text("""
    COPY three_eleven_calls
    FROM 'https://github.com/crate/cratedb-datasets/raw/main/academy/chicago-data/311_records_apr_2024.json.gz'
    WITH (compression='gzip') RETURN SUMMARY;
    """))

display_results("three_eleven_calls", result.mappings().first())
```

```
three_eleven_calls: loaded 174092, errors: 0
```

```python
# Load the libraries data file.
result = connection.execute(sa.text("""
    COPY libraries
    FROM 'https://github.com/crate/cratedb-datasets/raw/main/academy/chicago-data/chicago_libraries.json'
    RETURN SUMMARY;
    """))

display_results("libraries", result.mappings().first())
```

```
libraries: loaded 81, errors: 0
```

Once the data's loaded, verify that the output shows 0 errors for each table. Next, we'll run a `REFRESH` command to make sure that the data's up to date before querying it. We'll also run `ANALYZE`, which collects statistics used by the query optimizer.

```python
_ = connection.execute(sa.text("REFRESH TABLE community_areas, three_eleven_calls, libraries"))
_ = connection.execute(sa.text("ANALYZE"))
```

Displaying Community Areas on a Map

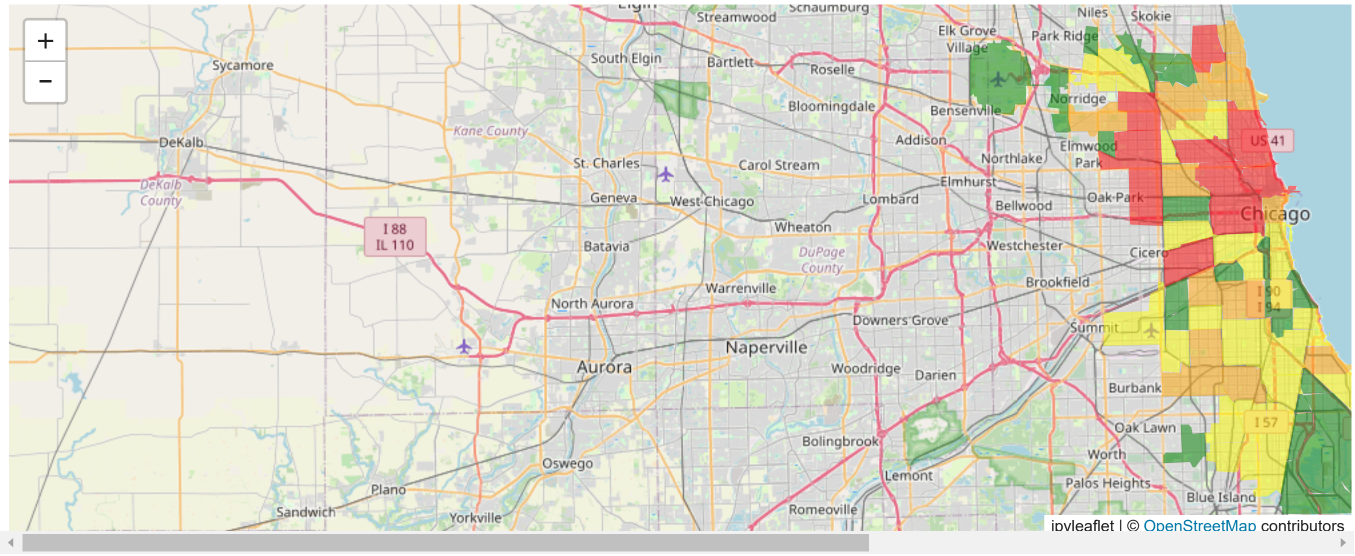
Let's begin to make sense of some of this data using a map. Chicago is divided into 77 community areas. We'll use these columns from the `community_areas` table:

- `name`: The name of the community area.
- `boundaries`: contains a GeoJSON MultiPolygon describing the community area's boundaries.
- `details`: an object, containing a `population` field, which holds the population for the community area.

The following code performs a simple `SELECT` query to get this information, adding it to a map and using the value of `details['population']` to colour code each community area. You'll see a map of Chicago with areas having the highest population in red and the lowest in green.

Use the map controls to move around and zoom in.

```python
import pandas as pd
import random
from ipyleaflet import Map, GeoJSON

center = (41.83068856472101, -87.74024963378908)
map = Map(center=center, zoom=10)

query = """
SELECT name, boundaries, details['population'] as population FROM community_areas
"""
df = pd.read_sql(query, CONNECTION_STRING)

def get_color_for_population(population):
    if population < 20000:
        return "green"
    elif population < 40000:
        return "yellow"
    elif population < 60000:
        return "orange"

    return "red"

for row in df.iterrows():
    community_area = GeoJSON(
        data=row[1]["boundaries"],
        style={
            "stroke": False,
            "fillColor": get_color_for_population(row[1]["population"]),
            "fillOpacity": 0.5
        }
    )

    map.add(community_area)

display(map)
```

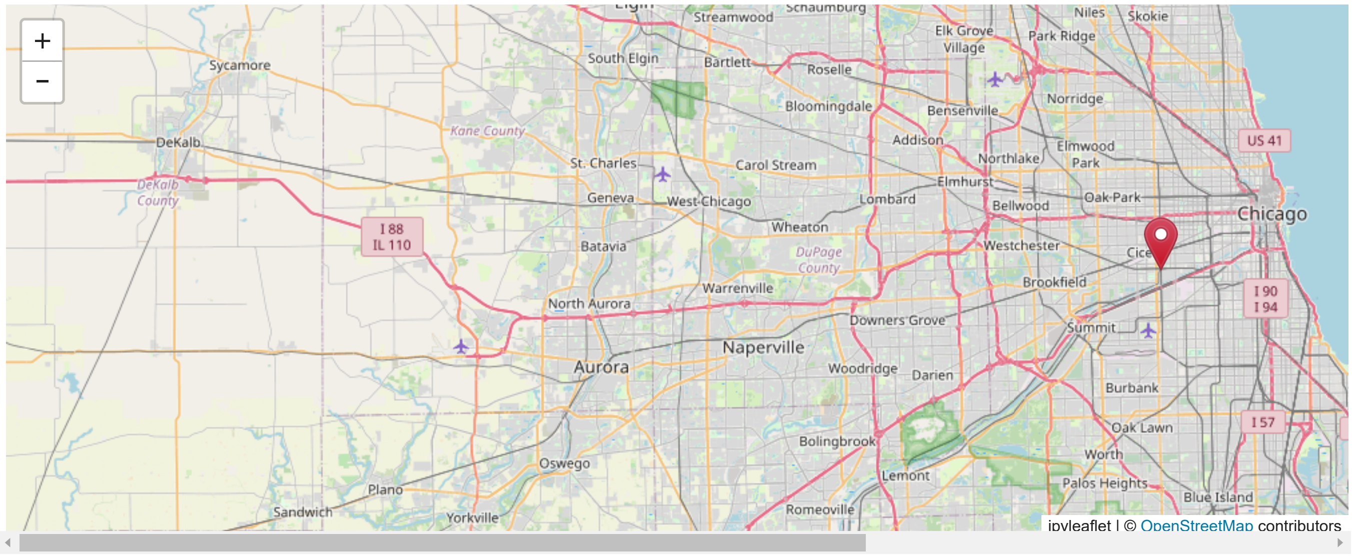


An Interactive Map / Finding Things by Distance

Next, we'll build a basic "store finder" interactive map. This approach could also be used to find nearby available cars in a ride hailing app, e-scooters with sufficient battery life to start a new ride nearby and so on.

The code below places a red marker on the map. Drag the red marker around Chicago. When you stop dragging, a `SELECT` query is executed, asking CrateDB to find the closest library to the pointer from data in the `libraries` table. We also retrieve the opening hours, stored as an array in CrateDB. The closest library is shown on the map as a blue marker - click this to see the opening hours and distance from the red marker.

```
from ipyleaflet import Icon, Marker
from ipywidgets import HTML

libraries_map = Map(center=center, zoom=10)
location_icon = Icon(icon_url='https://github.com/pointhi/leaflet-color-markers/raw/refs/heads/master/img/marker-icon-2x-red.png', ic
library_marker = None

def on_my_position_changed(pos):
    global library_marker

    my_lat = pos["new"][0]
    my_lon = pos["new"][1]
    query = f"""
    SELECT
        name,
        hours,
        location['position'] as location,
        trunc(distance('POINT({my_lon} {my_lat})', location['position']) / 1000, 2) AS distance
        FROM libraries ORDER BY distance ASC LIMIT 1;
    """

    df = pd.read_sql(query, CONNECTION_STRING)
    closest_library = df.values[0]
    library_lat = closest_library[2][1]
    library_lon = closest_library[2][0]
    library_distance = closest_library[3]

    if library_marker:
        libraries_map.remove(library_marker)

    library_marker = Marker(location = (library_lat, library_lon), draggable=False)
    library_details = HTML()

    library_opening_hours = [None] * 14
    library_opening_hours[::2] = ["<b>M</b>: ", "<br/><b>T</b>: ", "<br/><b>W:</b> ", "<br/><b>T:</b> ", "<br/><b>F:</b> ", "<br/><b
    library_opening_hours[1::2] = closest_library[1]
    library_details.value = f"<span style=\"color: #000000;\"><b>{closest_library[0]}</b><hr/>({library_distance}km)<br/>{''.join(lib
    library_marker.popup = library_details
    libraries_map.add(library_marker)


my_position = Marker(location=center, icon=location_icon, draggable=True)
my_position.observe(on_my_position_changed, "location")

libraries_map.add_control(my_position)
display(libraries_map)
```



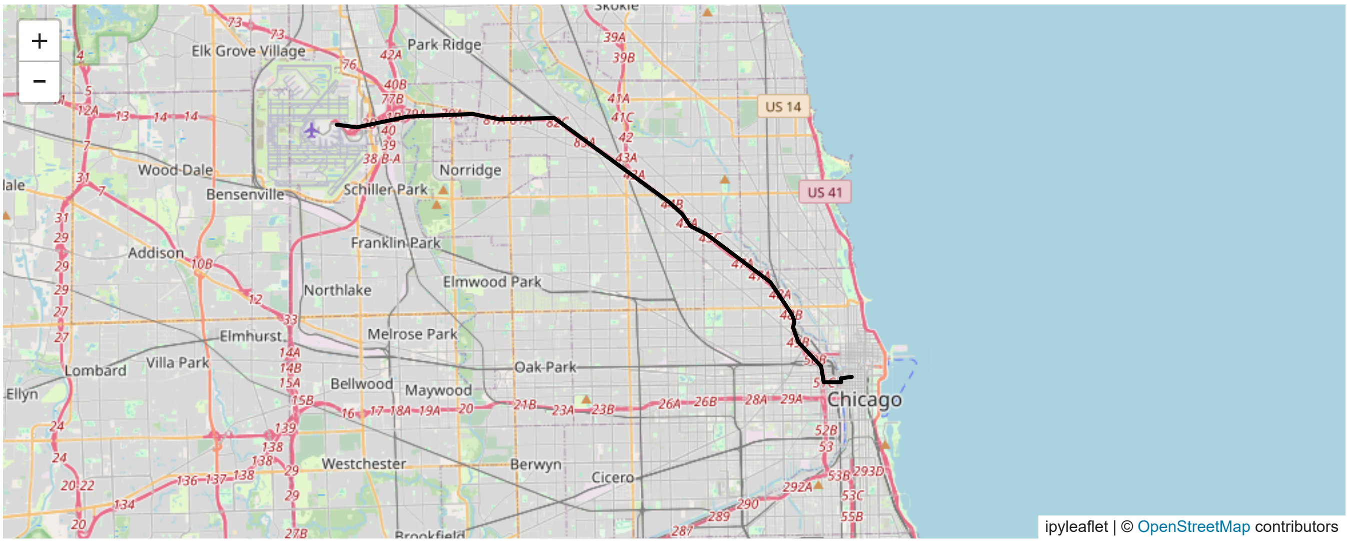
Finding Things Along the Way

Sometimes we want to look for data that's related to a specific area, or in the line of a path or a trip we're planning.

The code below contains GeoJSON for a line representing a trip from Chicago's Daley Center downtown to O'Hare Airport.

Run the code to see this path on the map. Remember you can use the map controls to zoom in and out and pan around.

```
trip_map = Map(center=[41.92424883732577, -87.72274017333986], zoom=11)

trip_geometry = {
    "coordinates": [
      [-87.63095706926296, 41.883920956255224],[-87.63093052767819, 41.88325569333841],
      [-87.63684297531508, 41.88322881741743],[-87.63682723619804, 41.88189296484862],
      [-87.64583001093926, 41.88176406531636],[-87.64556244595593, 41.8839084509878],
      [-87.64681360038576, 41.887978891258825],[-87.65712486706367, 41.89568681214507],
      [-87.65859173777416, 41.89703559399271],[-87.66010175174097, 41.90008630499267],
      [-87.6609646168648, 41.902847875583745],[-87.66061947081528, 41.90528823390659],
      [-87.66208634152613, 41.907856931373374],[-87.66786978418733, 41.915623971345894],
      [-87.67311334487873, 41.92011686521596],[-87.68725478998756, 41.927623866972624],
      [-87.69750427145591, 41.93394872585398],[-87.70600433948675, 41.93867612005508],
      [-87.71395364871834, 41.941703526516136],[-87.71855494590349, 41.94634069404421],
      [-87.72523341033431, 41.95064524453744],[-87.74318775119902, 41.960796298357224],
      [-87.75823682581736, 41.96896814729007],[-87.7659547090823, 41.97279282784706],
      [-87.7762448330368, 41.97829052959409],[-87.78428500170016, 41.98283266382293],
      [-87.81256731905091, 41.982340356805935],[-87.82639934099198, 41.98449314861523],
      [-87.85968209819193, 41.9836672569391],[-87.88581982097564, 41.9795526237948],
      [-87.89586899029486, 41.980297647123905]
    ],
  "type": "LineString"
}

trip_line = GeoJSON(
    data={
        "type": "Feature",
        "properties": {},
        "geometry": trip_geometry
    },
    style={
        "color": "#000000"
    })

trip_map.add(trip_line)
display(trip_map)
```



ipyleaflet | © OpenStreetMap contributors

We can use this path in database queries with CrateDB. The query below returns the name and GeoJSON representation of each of Chicago's community areas that our path passes through (intersects).

```
import json

query = f"""
  SELECT name, boundaries
  FROM community_areas
  WHERE intersects ('{json.dumps(trip_geometry)}'::object, boundaries)
"""

df = pd.read_sql(query, CONNECTION_STRING)

df
```

	name	boundaries
0	NEAR WEST SIDE	{'coordinates': [[[[-87.6375883858287, 41.8862...
1	PORTAGE PARK	{'coordinates': [[[[-87.75263506823083, 41.967...
2	LOOP	{'coordinates': [[[[-87.6094858028664, 41.8893...
3	WEST TOWN	{'coordinates': [[[[-87.65686079759237, 41.910...
4	OHARE	{'coordinates': [[[[-87.83658087874365, 41.986...
5	AVONDALE	{'coordinates': [[[[-87.6879867878517, 41.9361...
6	IRVING PARK	{'coordinates': [[[[-87.69474577254876, 41.961...
7	JEFFERSON PARK	{'coordinates': [[[[-87.75263506823083, 41.967...
8	NORWOOD PARK	{'coordinates': [[[[-87.78002228630051, 41.997...
9	LOGAN SQUARE	{'coordinates': [[[[-87.68284015972066, 41.932...

Next steps: Generate code with df | View recommended plots | New interactive sheet

Our trip passes through 10 different community areas... let's use another table in our dataset to add some additional context.

The `three_eleven_calls` table contains details of 311 work orders created in April 2024. When citizens want to report issues with city infrastructure, they call 311 or fill out an online form to create a report.

Each report has a request type (in the `srtype` column). We'll use that and the number of the community area that the issue was reported in (in the `locationdetails` object column) to count how many open 311 issues there are in each community area our trip passes through.

As we're driving, we'll only consider issues that might affect drivers: those relating to road signs, street lights or potholes in the road.

The code below runs a query using a Common Table Expression to return the name of each community area we'll pass through, how many relevant open issues there are in that area, and the boundaries of the area.

```
query=f"""
WITH IntersectingCommunities AS (
    SELECT areanumber, name, boundaries
    FROM community_areas
    WHERE intersects ('{json.dumps(trip_geometry)}'::object, boundaries)
)
```

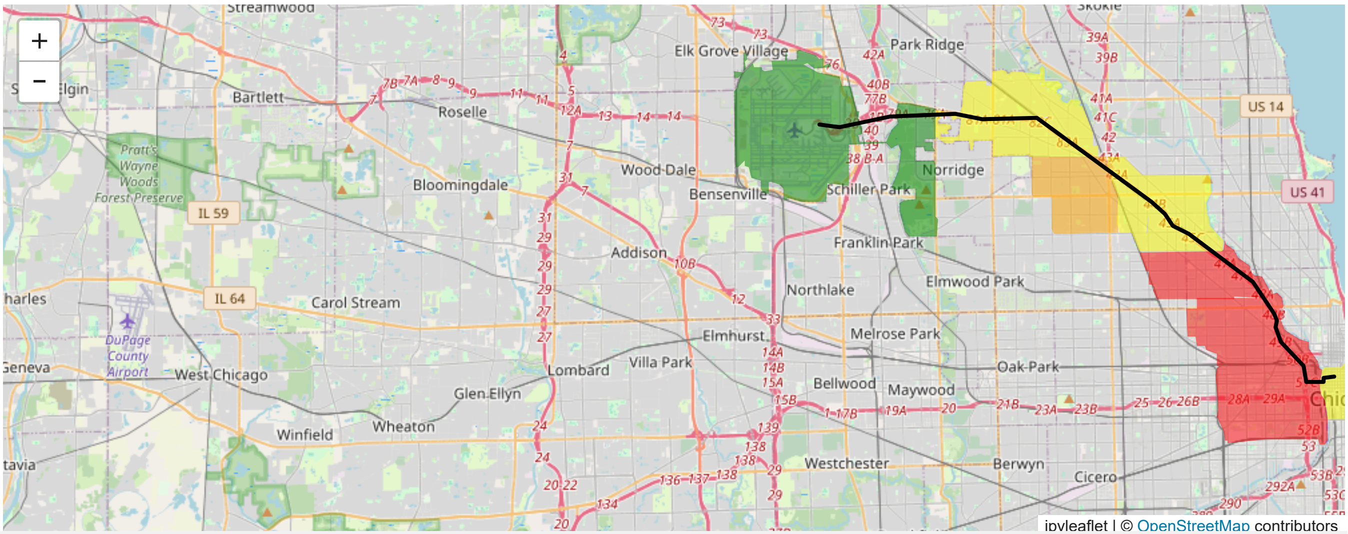
```
SELECT name,
       count(t.srtype) AS open_issues,
       boundaries
FROM IntersectingCommunities i, three_eleven_calls t
WHERE i.areanumber = t.locationdetails['communityarea']
      AND t.status = 'Open'
      AND (srtype LIKE 'Sign Repair%' OR srtype LIKE 'Street Light%' OR srtype LIKE 'Pothole%')
GROUP BY name, boundaries;
"""

df = pd.read_sql(query, CONNECTION_STRING)

df
```

	name	open_issues	boundaries
0	NORWOOD PARK	52	{'coordinates': [[[[-87.78002228630051, 41.997...
1	PORTAGE PARK	200	{'coordinates': [[[[-87.75263506823083, 41.967...
2	LOOP	72	{'coordinates': [[[[-87.6094858028664, 41.8893...
3	WEST TOWN	320	{'coordinates': [[[[-87.65686079759237, 41.910...
4	NEAR WEST SIDE	728	{'coordinates': [[[[-87.6375883858287, 41.8862...
5	IRVING PARK	140	{'coordinates': [[[[-87.69474577254876, 41.961...
6	JEFFERSON PARK	50	{'coordinates': [[[[-87.75263506823083, 41.967...
7	LOGAN SQUARE	390	{'coordinates': [[[[-87.68284015972066, 41.932...
8	OHARE	18	{'coordinates': [[[[-87.83658087874365, 41.986...
9	AVONDALE	106	{'coordinates': [[[[-87.6879867878517, 41.9361...

Next steps: Generate code with df View recommended plots New interactive sheet

This information is much more useful when displayed on a map. Let's show the boundaries of each community area we pass through on the map along with the line representing our journey, and colour code each community area such that red areas have the most issues, and green the least.

```
def get_color_for_issues(issue_count):
    if issue_count < 50:
        return "green"
    elif issue_count < 150:
        return "yellow"
    elif issue_count < 300:
        return "orange"

    return "red"


trip_with_issues_map = Map(center=[41.92424883732577, -87.72274017333986], zoom=11)

for row in df.iterrows():
    community_area = GeoJSON(
        data=row[1]["boundaries"],
        style={
            "stroke": False,
            "fillColor": get_color_for_issues(row[1]["open_issues"]),
            "fillOpacity": 0.5
        }
    )

    trip_with_issues_map.add(community_area)

trip_with_issues_map.add(trip_line)
display(trip_with_issues_map)
```



Library Opening Hours

The `libraries` table has a column named `hours`. This is an array of text values. Each entry in the array contains the library's opening hours for the day, or "CLOSED" if it isn't open that day.

The first entry in the array is for Monday, the last one for Sunday.

Run the query below to view some example data.

```
query = """
SELECT name, location['zipcode'] as zip, hours, phone FROM libraries LIMIT 5;
"""

df = pd.read_sql(query, CONNECTION_STRING)

df
```

```
            name    zip                                      hours           phone
0   Austin-Irving  60634  [10-5, 10-5, CLOSED, 10-5, 10-5, 12-4, CLOSED]  (312) 744-6222
1     Blackstone  60615     [CLOSED, 9-5, 9-5, 9-5, 9-5, 10-4, CLOSED]  (312) 747-0511
2  Budlong Woods  60659  [10-5, 10-5, CLOSED, 10-5, 10-5, 12-4, CLOSED]  (312) 742-9590
3      Edgebrook  60646          [11-5, 11-5, 11-5, 11-5, 11-5, 12-3, 12-2]  (312) 744-8313
4           Hall  60615     [CLOSED, 9-5, 9-5, 9-5, 9-5, 10-4, CLOSED]  (312) 747-2541
```

Next steps: Generate code with df | View recommended plots | New interactive sheet

We can use a slicing approach to selectively return data from the `hours` array. What if we're only interested in the weekend opening hours?

```
# Saturday and Sunday hours (array index 6 onwards...)
query = """
SELECT name, location['zipcode'] as zip, hours[6:] as weekend_hours, phone FROM libraries LIMIT 5;
"""

df = pd.read_sql(query, CONNECTION_STRING)

df
```

```
            name    zip  weekend_hours           phone
0   Austin-Irving  60634  [12-4, CLOSED]  (312) 744-6222
1     Blackstone  60615  [10-4, CLOSED]  (312) 747-0511
2  Budlong Woods  60659  [12-4, CLOSED]  (312) 742-9590
3      Edgebrook  60646      [12-3, 12-2]  (312) 744-8313
4           Hall  60615  [10-4, CLOSED]  (312) 747-2541
```

Next steps: Generate code with df | View recommended plots | New interactive sheet

We can find out which libraries are open on Monday (position 1 in the array) by checking they aren't "CLOSED" that day.

```
# Libraries that open on Monday (array index 1)
query = """
SELECT name, hours FROM libraries WHERE hours[1] != 'CLOSED' LIMIT 5;
"""

df = pd.read_sql(query, CONNECTION_STRING)

df
```

	name	hours
0	Austin-Irving	[10-5, 10-5, CLOSED, 10-5, 10-5, 12-4, CLOSED]
1	Budlong Woods	[10-5, 10-5, CLOSED, 10-5, 10-5, 12-4, CLOSED]
2	Edgebrook	[11-5, 11-5, 11-5, 11-5, 11-5, 12-3, 12-2]
3	Jeffery Manor	[9-5, 9-5, 9-5, 9-5, 9-5, 10-4, 11-2]
4	Kelly	[9-5, 9-5, 9-5, 9-5, 9-5, 10-4, 11-2]

Next steps: Generate code with df View recommended plots New interactive sheet

How can we find libraries that open every day? We can use the `array_position` function to find rows where the `hours` array doesn't contain an element "CLOSED"...

```
query = """
SELECT name, hours FROM libraries where array_position(hours, 'CLOSED') IS NULL LIMIT 5;
"""

df = pd.read_sql(query, CONNECTION_STRING)

df
```

	name	hours
0	Edgebrook	[11-5, 11-5, 11-5, 11-5, 11-5, 12-3, 12-2]
1	Jeffery Manor	[9-5, 9-5, 9-5, 9-5, 9-5, 10-4, 11-2]
2	Kelly	[9-5, 9-5, 9-5, 9-5, 9-5, 10-4, 11-2]
3	McKinley Park	[10-3, 10-3, 12-2, 10-3, 10-3, 11-3, 12-3]
4	Popular Library at Water Works	[9-5, 9-5, 9-5, 9-5, 9-5, 10-4, 11-2]

Next steps: Generate code with df View recommended plots New interactive sheet

Let's add a little more context to a query by combining data from the `libraries` and `community_areas` tables. Here, we want to find libraries closest to the Cermak-Chinatown "L" train stop that are open (not "CLOSED") on Monday.

We'll return the name of the library, the name of the community area that it's in, the distance from the "L" stop in km and Monday's opening hours.

```
query = """
SELECT
    l.name as library,
    c.name AS area,
    trunc(distance('POINT(-87.63036810347516 41.85389519931859)', location['position']) / 1000, 1) AS how_far,
    hours[1] AS monday_hours
FROM libraries l, community_areas c
WHERE hours[1] != 'CLOSED' AND c.areanumber = l.location['communityarea']
ORDER BY how_far ASC LIMIT 3"""

df = pd.read_sql(query, CONNECTION_STRING)

df
```

	library	area	how_far	monday_hours
0	Chinatown	DOUGLAS	0.1	10-3
1	Chinatown	DOUGLAS	0.1	10-3
2	Lozano	NEAR SOUTH SIDE	2.5	9-5

Next steps: Generate code with df | View recommended plots | New interactive sheet

Experimenting with JOINs

We'll end this workbook with a quick look at a couple of joins. The query below generates a report by community area of how many missed garbage collections were reported in a given week.

It does this by joining the `three_eleven_calls` and `community_areas` table.

```
# 311 calls for missed garbage collection for a given week by community area...

query = """
SELECT
    c.areanumber,
    c.name,
    count(t.srtype) AS num_complaints
FROM community_areas c
JOIN three_eleven_calls t ON
    c.areanumber = t.locationdetails['communityarea']
    AND t.srtype = 'Missed Garbage Pick-Up Complaint'
    AND t.week = 1713744000000
GROUP BY c.areanumber, c.name
ORDER BY c.areanumber ASC LIMIT 10;
"""

df = pd.read_sql(query, CONNECTION_STRING)

df
```

```
   areanumber               name  num_complaints
0           2        WEST RIDGE                8
1           4    LINCOLN SQUARE                4
2           6         LAKE VIEW                2
3           8   NEAR NORTH SIDE                2
4          10      NORWOOD PARK                2
5          11    JEFFERSON PARK                2
6          12       FOREST GLEN                2
7          14       ALBANY PARK               16
8          15      PORTAGE PARK                6
9          16       IRVING PARK                4
```

Next steps: Generate code with df | View recommended plots | New interactive sheet

What we see in the result above is a report that shows the area number, name and number of garbage pickup complaints.

Notice that only community areas with relevant complaints in the given week are shown. What if we want a report that includes all community areas? For that, we'll use a `LEFT JOIN`.

```
# 311 calls for missed garbage collection for a given week by community area...

query = """
SELECT
    c.areanumber,
    c.name,
    count(t.srtype) AS num_complaints
FROM community_areas c
LEFT JOIN three_eleven_calls t ON
    c.areanumber = t.locationdetails['communityarea']
    AND t.srtype = 'Missed Garbage Pick-Up Complaint'
    AND t.week = 1713744000000
GROUP BY c.areanumber, c.name
ORDER BY c.areanumber ASC LIMIT 10;
"""

df = pd.read_sql(query, CONNECTION_STRING)

df
```

```
   areanumber        name  num_complaints
0           1  ROGERS PARK               0
```