Exercise 4: Cartography

Here we learn to use Altair for rendering maps. We will also get some practice with color scales.

For a primer on cartography in Altair, take a look at this resource from our colleagues at the University of Washington.

Chicago ward map

We're going to start by downloading some geoJSON data for wards in the City of Chicago. You can find the data here.

We load the data into python and render it using Altair.

In [2]:
```
# PROMPT: import geojson, pandas (to use later), and altair
%pip install geojson
%pip install pandas
%pip install altair
import geojson
import pandas as pd
import altair as alt

import json # just for viewing purposes
```

```
Requirement already satisfied: geojson in /Users/chrislowzx/data227/data-viz/exercise/.venv/lib/python3.12/site-packages (
3.2.0)

[notice] A new release of pip is available: 24.3.1 -> 25.2
[notice] To update, run: pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
Collecting pandas
  Downloading pandas-2.3.3-cp312-cp312-macosx_11_0_arm64.whl.metadata (91 kB)
Collecting numpy>=1.26.0 (from pandas)
  Downloading numpy-2.3.4-cp312-cp312-macosx_14_0_arm64.whl.metadata (62 kB)
Requirement already satisfied: python-dateutil>=2.8.2 in /Users/chrislowzx/data227/data-viz/exercise/.venv/lib/python3.12/
site-packages (from pandas) (2.9.0.post0)
Collecting pytz>=2020.1 (from pandas)
  Using cached pytz-2025.2-py2.py3-none-any.whl.metadata (22 kB)
Collecting tzdata>=2022.7 (from pandas)
  Using cached tzdata-2025.2-py2.py3-none-any.whl.metadata (1.4 kB)
Requirement already satisfied: six>=1.5 in /Users/chrislowzx/data227/data-viz/exercise/.venv/lib/python3.12/site-packages
(from python-dateutil>=2.8.2->pandas) (1.17.0)
Downloading pandas-2.3.3-cp312-cp312-macosx_11_0_arm64.whl (10.7 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10.7/10.7 MB 10.5 MB/s eta 0:00:00 0:00:01
Downloading numpy-2.3.4-cp312-cp312-macosx_14_0_arm64.whl (5.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.1/5.1 MB 11.0 MB/s eta 0:00:00a 0:00:01
Using cached pytz-2025.2-py2.py3-none-any.whl (509 kB)
Using cached tzdata-2025.2-py2.py3-none-any.whl (347 kB)
Installing collected packages: pytz, tzdata, numpy, pandas
Successfully installed numpy-2.3.4 pandas-2.3.3 pytz-2025.2 tzdata-2025.2

[notice] A new release of pip is available: 24.3.1 -> 25.2
```

```
[notice] To update, run: pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
Collecting altair
  Using cached altair-5.5.0-py3-none-any.whl.metadata (11 kB)
Collecting jinja2 (from altair)
  Using cached jinja2-3.1.6-py3-none-any.whl.metadata (2.9 kB)
Collecting jsonschema>=3.0 (from altair)
  Downloading jsonschema-4.25.1-py3-none-any.whl.metadata (7.6 kB)
Collecting narwhals>=1.14.2 (from altair)
  Downloading narwhals-2.9.0-py3-none-any.whl.metadata (11 kB)
Requirement already satisfied: packaging in /Users/chrislowzx/data227/data-viz/exercise/.venv/lib/python3.12/site-packages
(from altair) (25.0)
Collecting typing-extensions>=4.10.0 (from altair)
  Using cached typing_extensions-4.15.0-py3-none-any.whl.metadata (3.3 kB)
Collecting attrs>=22.2.0 (from jsonschema>=3.0->altair)
  Downloading attrs-25.4.0-py3-none-any.whl.metadata (10 kB)
Collecting jsonschema-specifications>=2023.03.6 (from jsonschema>=3.0->altair)
  Downloading jsonschema_specifications-2025.9.1-py3-none-any.whl.metadata (2.9 kB)
Collecting referencing>=0.28.4 (from jsonschema>=3.0->altair)
  Downloading referencing-0.37.0-py3-none-any.whl.metadata (2.8 kB)
Collecting rpds-py>=0.7.1 (from jsonschema>=3.0->altair)
  Downloading rpds_py-0.28.0-cp312-cp312-macosx_11_0_arm64.whl.metadata (4.1 kB)
Collecting MarkupSafe>=2.0 (from jinja2->altair)
  Downloading markupsafe-3.0.3-cp312-cp312-macosx_11_0_arm64.whl.metadata (2.7 kB)
Using cached altair-5.5.0-py3-none-any.whl (731 kB)
Downloading jsonschema-4.25.1-py3-none-any.whl (90 kB)
Downloading narwhals-2.9.0-py3-none-any.whl (422 kB)
Using cached typing_extensions-4.15.0-py3-none-any.whl (44 kB)
Using cached jinja2-3.1.6-py3-none-any.whl (134 kB)
Downloading attrs-25.4.0-py3-none-any.whl (67 kB)
Downloading jsonschema_specifications-2025.9.1-py3-none-any.whl (18 kB)
Downloading markupsafe-3.0.3-cp312-cp312-macosx_11_0_arm64.whl (12 kB)
Downloading referencing-0.37.0-py3-none-any.whl (26 kB)
Downloading rpds_py-0.28.0-cp312-cp312-macosx_11_0_arm64.whl (348 kB)
Installing collected packages: typing-extensions, rpds-py, narwhals, MarkupSafe, attrs, referencing, jinja2, jsonschema-sp
ecifications, jsonschema, altair
Successfully installed MarkupSafe-3.0.3 altair-5.5.0 attrs-25.4.0 jinja2-3.1.6 jsonschema-4.25.1 jsonschema-specifications
-2025.9.1 narwhals-2.9.0 referencing-0.37.0 rpds-py-0.28.0 typing-extensions-4.15.0

[notice] A new release of pip is available: 24.3.1 -> 25.2
[notice] To update, run: pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

A Quick look at the JSON object

Side Note

A geojson object loaded with geojson.load() needs to be passed *correctly* to Altair. If the GeoJSON object is not formatted correctly you could use alt.Data to parse it and create an alt.Data object.

We will load the data into a DataFrame using Pandas, but before we do that here is a quick look at the JSON object.

In [3]:

```python
# PROMPT: load geojson data from https://data.cityofchicago.org/Facilities-Geographic-Boundaries/Boundaries-Wards-2015-202
# HINT: go to provided URL > click Export > geoJSON

# Load the GeoJSON file
with open("../exercise_data/chicago-ward-boundaries.geojson") as f:
    chi_map = geojson.load(f)

chi_map["features"][0]["geometry"]["coordinates"] = [[
    [-87.696235,41.857555], [-87.696252,41.857378], [-87.695807,41.857386], '...'
]] # Just for illustration purposes

chi_map_formatted_str = json.dumps(chi_map, indent=2)
print(chi_map_formatted_str[:800],'\n\t\t...')

print('\nAll the info on ward 12:')
display( chi_map["features"][0] )
print('\nThe "properties" associated with ward 12:')
display( chi_map["features"][0]["properties"] )
print('\nThe corresponding ward number:')
chi_map["features"][0]["properties"]["ward"]
```


```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "MultiPolygon",
        "coordinates": [
          [
            [
              -87.696235,
              41.857555
            ],
            [
              -87.696252,
              41.857378
            ],
            [
              -87.695807,
              41.857386
            ],
            "..."
          ]
        ]
      },
      "properties": {
        "shape_area": "116096507.849",
        "shape_leng": "93073.3408379",
        "ward": "12"
      }
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "MultiPolygon",
        "coordinates": [
          [
            [
              [
                -87.662889,
                41.798838

                ...
```


```
All the info on ward 12:
{"geometry": {"coordinates": [[[-87.696235, 41.857555], [-87.696252, 41.857378], [-87.695807, 41.857386], "..."]], "type":
"MultiPolygon"}, "properties": {"shape_area": "116096507.849", "shape_leng": "93073.3408379", "ward": "12"}, "type": "Feat
ure"}
```

```
The "properties" associated with ward 12:
{'shape_area': '116096507.849', 'shape_leng': '93073.3408379', 'ward': '12'}
The corresponding ward number:
```

Out[3]: '12'

A basic map of Chicago wards

Read the geojson data into a DataFrame

It is *convenient* to have every row correspond to a feature. This is acheived by:

- Having a `type` column where the value for each row is 'Feature' (see above).
- Having a `geometry` column with the ward's geometry (see above).
- Having a `ward` column with a unique identifier for each ward (for later use).

In [5]:

```
chicago_wards_df = pd.read_json('../exercise_data/chicago-ward-boundaries.geojson')

display(chicago_wards_df.head(1))
print('\nKeys in features:')
display(chicago_wards_df.loc[0].features.keys())
print("\nKeys in features['properties']:")
display(chicago_wards_df.loc[0].features['properties'].keys())

print('\nEach row should have a type=Feature, a geometry, and a ward identifier:')
chicago_wards_df['type'] = chicago_wards_df.features.apply(lambda x: x['type']) # Required!
chicago_wards_df['geometry'] = chicago_wards_df.features.apply(lambda x: x['geometry'])
chicago_wards_df['ward'] = chicago_wards_df.features.apply(lambda x: x['properties']['ward'])
display(chicago_wards_df.head())
```

	type	features
0	FeatureCollection	{'type': 'Feature', 'properties': {'shape_area...

```
Keys in features:
dict_keys(['type', 'properties', 'geometry'])
Keys in features['properties']:
dict_keys(['shape_area', 'shape_leng', 'ward'])
Each row should have a type=Feature, a geometry, and a ward identifier:
```

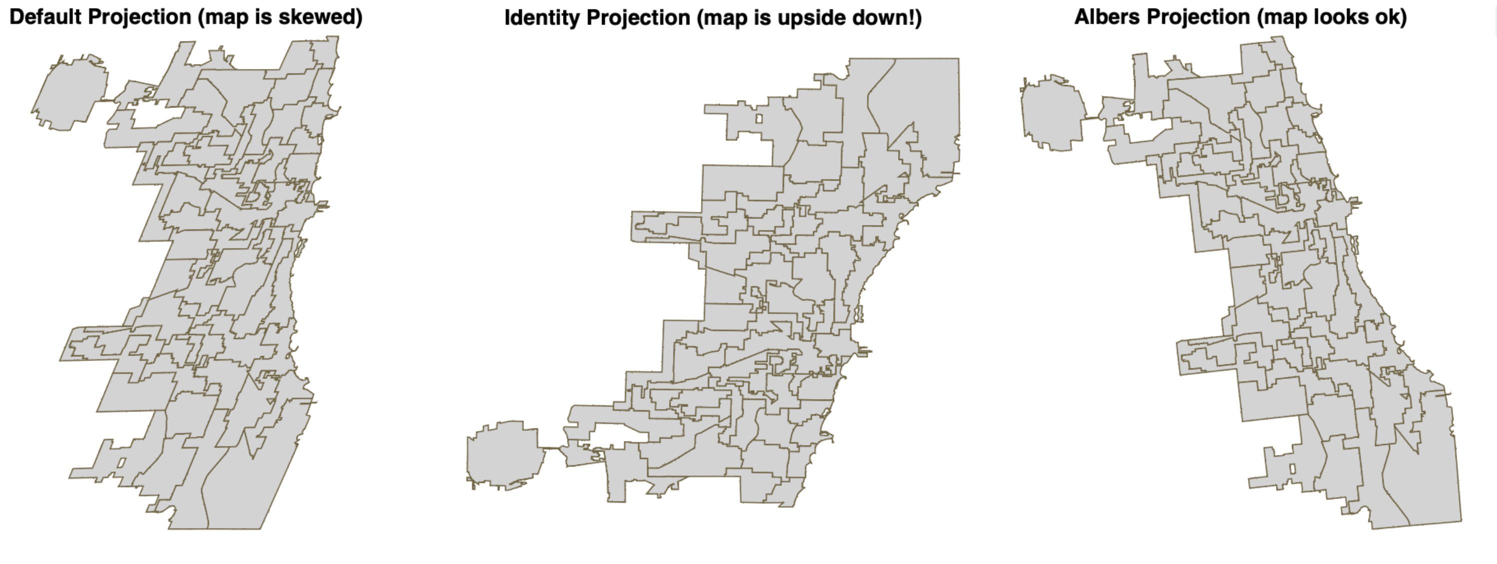
	type	features	geometry	ward
0	Feature	{'type': 'Feature', 'properties': {'shape_area...	{'type': 'MultiPolygon', 'coordinates': [[[[-8...	12
1	Feature	{'type': 'Feature', 'properties': {'shape_area...	{'type': 'MultiPolygon', 'coordinates': [[[[-8...	16
2	Feature	{'type': 'Feature', 'properties': {'shape_area...	{'type': 'MultiPolygon', 'coordinates': [[[[-8...	15
3	Feature	{'type': 'Feature', 'properties': {'shape_area...	{'type': 'MultiPolygon', 'coordinates': [[[[-8...	20
4	Feature	{'type': 'Feature', 'properties': {'shape_area...	{'type': 'MultiPolygon', 'coordinates': [[[[-8...	49

Test some projections

In [6]:

```
# Test chart
test_default_projection_type = alt.Chart(
    chicago_wards_df,
    title="Default Projection (map is skewed)"
).mark_geoshape(
    fill='#d3d3d3', # '#2a1d0c',
    stroke='#706545',  # Optional: Outline color
    strokeWidth=0.75    # Optional: Outline width
)

test_identity_projection_type = alt.Chart(
    chicago_wards_df,
    title="Identity Projection (map is upside down!)"
).mark_geoshape(
    fill='#d3d3d3', # '#2a1d0c',
    stroke='#706545',  # Optional: Outline color
    strokeWidth=0.75    # Optional: Outline width
).project(
    type='identity'
)

test_albers_projection_type = alt.Chart(
    chicago_wards_df,
    title="Albers Projection (map looks ok)"
).mark_geoshape(
    fill='#d3d3d3', # '#2a1d0c',
    stroke='#706545',  # Optional: Outline color
    strokeWidth=0.75    # Optional: Outline width
).project(
    type='albers'
)

test_default_projection_type | test_identity_projection_type |  test_albers_projection_type
```

Out[6]:



Note

The `project(type='...')` in the Altair code specifies the type of projection to use when rendering geographic data in a chart.

The `identity` projection means that no transformation or projection of the geographic coordinates should be applied because the geographic data is already in the format that can be rendered as-is.

Without choosing `identity` Altair would attempt to apply a default geographic projection to convert the geographic coordinates to the chart's coordinate system.

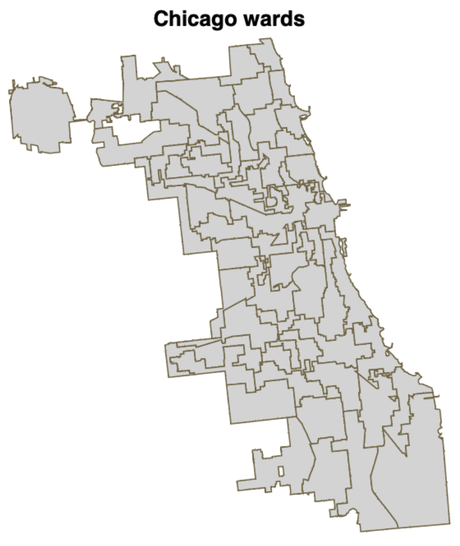
The `albers` projection is an equal-area conic projection (a U.S.-centric configuration of `conicEqualArea`).

`albersUsa` U.S.-centric composite with projections for the lower 48 states, Hawaii, and Alaska.

Here is a map with a tooltip that indicates the ward number

In [7]:
```
alt.Chart(
    chicago_wards_df,
    title="Chicago wards").mark_geoshape(
    fill='#d3d3d3', # '#2a1d0c',
    stroke='#706545',
    strokeWidth=0.75
).encode(
    tooltip=[
        alt.Tooltip('ward:N', title='Ward')
    ]
).properties(
    width=400
).project(
    type='albers'
)
```

Out[7]:



PROMPT: *describe the map you made. Speculate on the choice of map projection (if you had to decide on one from scratch) and why it might be chosen.*

[Your answer here]: The map outlines the boundaries of all 50 wards in the city of Chicago. When plotted with the default projection, the map looks stretched and slightly distorted. Using the identity projection, the map flips upside down because no transformation was applied to the coordinate system. The Albers projection, however, displays the map correctly oriented and proportioned, making it the most suitable among the three.

If I were choosing a projection from scratch, I would use the Albers equal-area conic projection. It seems to minimize distortion over mid-latitudes and preserve area well relative to other methods. This makes it ideal for accurately displaying ward boundaries at the city level.

Fill the map in with data

Now we have a simple map of Chicago's wards, but we want to fill it in with data. For this, we will need another data source. We will pull in data from a Tableau Public example about home sales in Chicago, and we will visualize different attributes of the data using color and point encodings.

First, lets set of the fill color of our map based on which type of home had the highest count of sales in each ward in 2020.

In [8]:
```
# PROMPT: load the data about home sales per Chicago ward
# HINT:  go to provided URL > Download Data > Data > Show Fields to filter > Download
# HINT: read the data using the argument `encoding="UTF-16"` to avoid an error

home_sales = pd.read_csv("../exercise_data/chicago-home-sales.csv", sep='\t', encoding="UTF-16")
home_sales
```

Out[8]:

	2020Sales count	2020Sales median	Chicago Ward	Property Type
0	98.0	$455,375	45	Multi-Family (2-6 unit)
1	440.0	$354,950	45	Single-Family
2	133.0	$164,500	45	Condo
3	24.0	$231,500	18	Multi-Family (2-6 unit)
4	421.0	$203,000	18	Single-Family
...	...	...	...	...
145	162.0	$435,000	11	Single-Family
146	98.0	$290,000	11	Condo
147	68.0	$120,000	10	Multi-Family (2-6 unit)
148	237.0	$135,000	10	Single-Family
149	3.0	$22,000	10	Condo

150 rows × 4 columns

Find the property type that was sold the most in each ward

`DataFrameGroupBy.idxmax()` return index of first occurrence of maximum over requested axis.

Once we have teh index of the maximal sales (for each ward) we can grap the property type from the `home_sales` DataFrame.

In [9]:
```
most_common_sales = home_sales.groupby(['Chicago Ward']).idxmax()['2020Sales count'].reset_index()
most_common_sales['2020Sales count'] = most_common_sales['2020Sales count'].apply(
                                            lambda x: home_sales.loc[x, 'Property Type']
                                        )
most_common_sales.rename(columns={'2020Sales count': 'Property Type'}, inplace=True)
print('most_common_sales:')
display(most_common_sales.head(10))
```

```
most_common_sales:
```

	Chicago Ward	Property Type
0	1	Condo
1	2	Condo
2	3	Condo

3	4	Condo
4	5	Condo
5	6	Single-Family
6	7	Single-Family
7	8	Single-Family
8	9	Single-Family
9	10	Single-Family

There is another way to do it:

`pandas.core.groupby.DataFrameGroupBy.transform` returns a DataFrame having the same indexes as the original object filled with the transformed values.

The following code illustrates how that works.

In [10]:
```
print('groupby and max - 50 wards make 50 groups:')
display(home_sales.groupby(['Chicago Ward'])['2020Sales count'].max())

print('\n\n groupby and transform("max") - 50x3=150 entries/indices are kept,')
print('\t\t\t\teach with the max of its group:')
series_with_maxs = home_sales.groupby(['Chicago Ward'])['2020Sales count'].transform("max")
display(series_with_maxs)

print('\n\nBoolean series with True where max values appear (for each group):')
home_sales['2020Sales count']==series_with_maxs
```

```
groupby and max - 50 wards make 50 groups:
```

```
Chicago Ward
1       554.0
2       930.0
3       263.0
4       332.0
5       284.0
6       244.0
7       225.0
8       391.0
9       367.0
10      237.0
11      162.0
12       78.0
13      392.0
14      180.0
15       97.0
16      134.0
17      256.0
18      421.0
19      593.0
20      191.0
21      432.0
22       92.0
23      318.0
24      207.0
25      174.0
26      143.0
27      263.0
28      177.0
29      237.0
30      183.0
31      136.0
32      600.0
33      243.0
34      484.0
35      150.0
36      263.0
37      164.0
38      507.0
39      352.0
40      274.0
41      583.0
42     1049.0
43      731.0
44      836.0
45      440.0
46      689.0
```

```
47     410.0
48     566.0
49     384.0
50     199.0
Name: 2020Sales count, dtype: float64

 groupby and transform("max") - 50x3=150 entries/indices are kept,
                               each with the max of its group:
0      440.0
1      440.0
2      440.0
3      421.0
4      421.0
       ...
145    162.0
146    162.0
147    237.0
148    237.0
149    237.0
Name: 2020Sales count, Length: 150, dtype: float64

Boolean series with True where max values appear (for each group):
```

Out[10]:
```
0      False
1       True
2      False
3      False
4       True
       ...
145     True
146    False
147    False
148     True
149    False
Name: 2020Sales count, Length: 150, dtype: bool
```

In [11]:
```
# PROMPT: use altair to make a choropleth map of which type of home
#         has the highest count of sales per ward
#
# HINT: this stackoverflow page might help you with a required data transformation
#       https://stackoverflow.com/questions/15705630/get-the-rows-which-have-the-max-value-in-groups-using-groupby
#       See also the side note above

series_with_maxs = home_sales.groupby(['Chicago Ward'])['2020Sales count'].transform("max")
idx =  series_with_maxs==home_sales['2020Sales count']
most_common_sales2 = home_sales[idx]
most_common_sales2.head()
```

Out[11]:

	2020Sales count	2020Sales median	Chicago Ward	Property Type
1	440.0	$354,950	45	Single-Family
4	421.0	$203,000	18	Single-Family
7	150.0	$447,000	35	Single-Family
10	180.0	$229,000	14	Single-Family
12	191.0	$193,000	20	Multi-Family (2-6 unit)

Now we need to merge the popular sales data with the wards data

This can be done my using `merge` and passing Altair the merged DataFrame.

Alternatively, this can be done from within the chart:

The Altair equivalent of `merge`: `alt.LookupData` and `alt.Chart.transform_lookup`

`alt.LookupData`

Used to prepare a secondary dataset to be joined (merged) into a chart's primary dataset *based on a common key* field.

This is similar to performing a *left join* using `pandas.merge`, i.e., to enrich the primary data with additional fields, e.g., for coloring, sizing, or tooltips.

Arguments

- data: the additional DataFrame (or other supported structure) you want to join.
- key: the field in the additional data that matches the field in the primary dataset.
- fields: a list of fields from the additional data to be included in your primary dataset. These fields become available for encoding (like color, tooltip, etc.) in the chart.

`alt.Chart.transform_lookup`

Used to perform the merge (aka "the lookup transformation").

How It Works

- The primary dataset is already being used in the chart.
- Lookup Data prepares the secondary dataset, the key column name, and the names of the columns with the additional information.
- Altair matches the rows in the primary data with rows in the lookup data based on the specified common key.
- The specified fields from the lookup data are inserted into the primary dataset wherever there's a match.

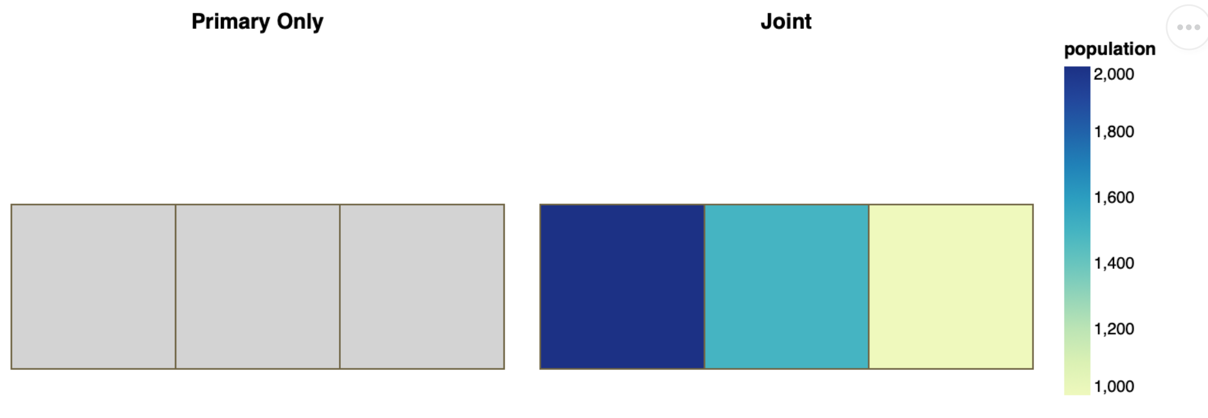
Simple Example (code below)

- The primary GeoJSON data describe three square regions.
- The secondary DataFrame contains the population numbers for each region.
- The joint chart will use the population data to color the regions.
- Regions in the primary dataset are associated with regions in the secondary dataset through the common key `region_id`.

In [13]:
```
# Read the GeoJSON file
primary_data = pd.read_json("../exercise_data/SimpleGeojsonExample.geojson")
primary_data['type'] = primary_data.features.apply(lambda x: x['type']) # Required!
primary_data['geometry'] = primary_data.features.apply(lambda x: x['geometry'])
primary_data['region_id'] = primary_data.features.apply(lambda x: x['properties']['region_id'])

# Primary chart (no population data)
primary_chart = alt.Chart(
    primary_data, title='Primary Only'
).mark_geoshape(
        fill='#d3d3d3',
```

```
        stroke='#706545'
).encode(
).project(
    type='identity'
)

# Lookup (secondary) data: DataFrame with additional population information
additional_data = pd.DataFrame({
    'region_id': [1, 2, 3],
    'population': [1000, 1500, 2000]
})

# Chart with a lookup transformation
joint_chart = alt.Chart(
    primary_data, title='Joint'
).mark_geoshape(
    stroke='#706545'
).transform_lookup(
    lookup='region_id',  # Key in the GeoJSON primary dataset
    from_=alt.LookupData(additional_data,
                         key='region_id',
                         fields=['population'])  # Lookup in the DataFrame
).encode(
    color='population:Q',
    tooltip=[alt.Tooltip('population:Q',title="Population:")]
).project(
    type='identity'
)

(primary_chart | joint_chart)
```

Out[13]:



Lookup Data for property sales in Chicago Wards:

In [14]:
```
alt.LookupData(
            data=most_common_sales, key='Chicago Ward', fields=['Property Type']
        )
```

Out[14]:
```
LookupData({
  data:     Chicago Ward           Property Type
  0            1                   Condo
  1            2                   Condo
  2            3                   Condo
  3            4                   Condo
  4            5                   Condo
  5            6           Single-Family
  6            7           Single-Family
  7            8           Single-Family
  8            9           Single-Family
  9           10           Single-Family
  10          11           Single-Family
  11          12  Multi-Family (2-6 unit)
  12          13           Single-Family
  13          14           Single-Family
  14          15  Multi-Family (2-6 unit)
  15          16           Single-Family
  16          17           Single-Family
  17          18           Single-Family
  18          19           Single-Family
  19          20  Multi-Family (2-6 unit)
  20          21           Single-Family
  21          22           Single-Family
  22          23           Single-Family
  23          24  Multi-Family (2-6 unit)
  24          25                   Condo
  25          26                   Condo
  26          27                   Condo
  27          28  Multi-Family (2-6 unit)
  28          29           Single-Family
  29          30           Single-Family
  30          31           Single-Family
  31          32                   Condo
  32          33                   Condo
  33          34           Single-Family
  34          35           Single-Family
  35          36           Single-Family
  36          37           Single-Family
  37          38           Single-Family
  38          39           Single-Family
  39          40                   Condo
  40          41           Single-Family
  41          42                   Condo
  42          43                   Condo
  43          44                   Condo
  44          45           Single-Family
  45          46                   Condo
  46          47                   Condo
  47          48                   Condo
  48          49                   Condo
  49          50                   Condo,
  fields: ['Property Type'],
  key: 'Chicago Ward'
})
```

Note

The data used for the GeoJSON and the data in the home_sales DataFrame **must match**.

Specifically, the ward IDs in the GeoJSON file have to **match exactly** with the Chicago Ward field in the home_sales DataFrame.

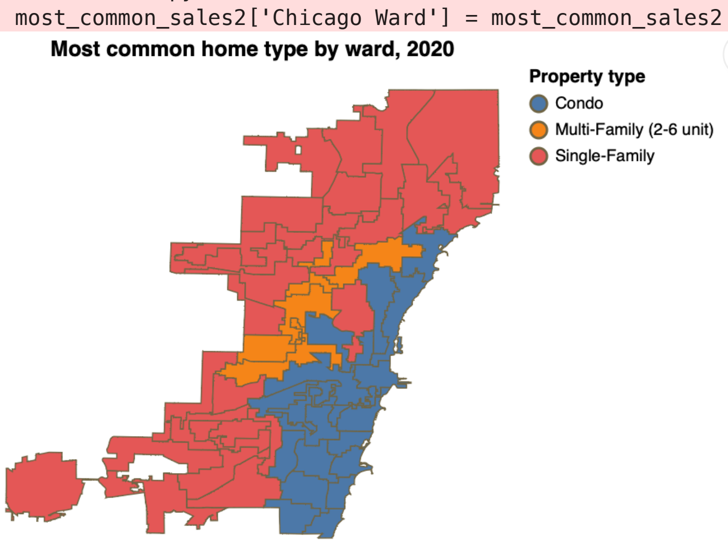
(The fact that the column in the DataFrame is called 'Chicago Ward' and not 'ward' is ok - we specify the column name in the `key` argument.)

In [ ]:
```
# PROMT: Create a chjoropleth of Chicago wards,
```

```
#        where the color each ward corresponds
#        to the type of property most commonly
#        sold there.

# [Your Answer Here]: Join together chicago_wards_df and most_common_sales on ward number,

chicago_wards_df['ward'] = chicago_wards_df['ward'].astype(int)
most_common_sales2['Chicago Ward'] = most_common_sales2['Chicago Ward'].astype(int)

# Choropleth: color each ward by its most common 2020 property type
choropleth = (
    alt.Chart(chicago_wards_df, title='Most common home type by ward, 2020')
    .mark_geoshape(stroke='#706545', strokeWidth=0.75)
    .transform_lookup(
        lookup='ward',
        from_=alt.LookupData(
            most_common_sales2[['Chicago Ward', 'Property Type']],
            key='Chicago Ward',
            fields=['Property Type']
        )
    )
    .encode(
        color=alt.Color('Property Type:N', title='Property type'),
        tooltip=[
            alt.Tooltip('ward:O', title='Ward'),
            alt.Tooltip('Property Type:N', title='Most common')
        ]
    )
    .project(type='identity')
)

choropleth
```

```
/var/folders/vp/npcvxcc52xqfjgmswcdds6yc0000gn/T/ipykernel_79873/3388840751.py:10: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-vi
ew-versus-a-copy
  most_common_sales2['Chicago Ward'] = most_common_sales2['Chicago Ward'].astype(int)
```

Out[ ]:



PROMPT: describe the map above, especially how you chose your color scale or speculate on how the default colors where chosen, if you went with the default

[Your Answer Here]: The map shows which type of property had the highest number of sales in each of Chicago's 50 wards in 2020. Each ward is colored according to the top-selling property type: condos, single-family homes, or multi-family buildings. I used the default Altair color scale, which automatically assigns distinct colors for categorical data (blue for condo, red for single-family, orange for multi-family).

The colors are chosen to make the categories easy to tell apart without having to customize the palette. Since there are only three categories, the default colors work well here. They're clear, balanced, and make the map easy to read. This way, you can quickly see patterns, like how condos are concentrated near the lakefront and single-family homes dominate the outer wards.

Now, we want to see the median home price in each ward for each type of property

In [16]:

```
# PROMPT: use altair to make three choropleth maps of median home prices per ward, one for each home type
# HINT: the median home sales values need to be converted to a numeric data type
# HINT: you might want to separate the home sales data into different dataframes for each map

def dollars_to_float(s):
    if isinstance(s, str):

        s = s.replace("$", "").replace(",", "")
    return float(s)

home_sales["Median Sale Price"] = home_sales["2020Sales median"].apply(dollars_to_float)
```

In [17]:

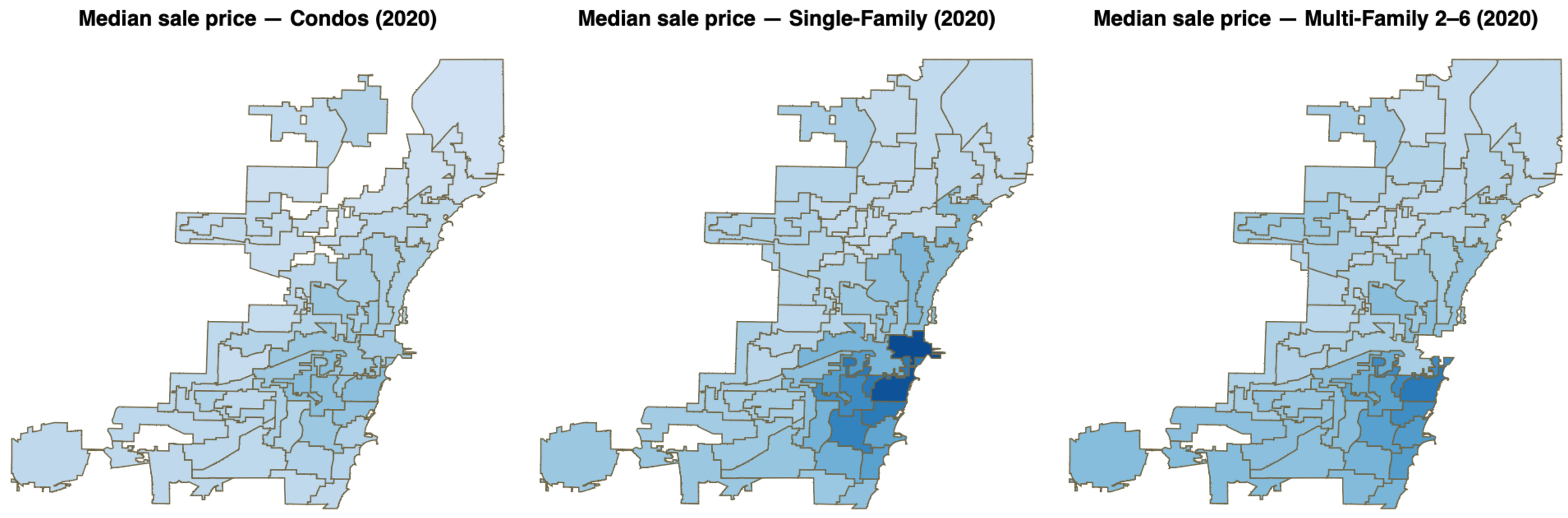
```
# [Separate the DataFrames]
condos_df = home_sales[home_sales['Property Type'] == 'Condo'][['Chicago Ward','Median Sale Price']]
sf_df     = home_sales[home_sales['Property Type'] == 'Single-Family'][['Chicago Ward','Median Sale Price']]
mf_df     = home_sales[home_sales['Property Type'] == 'Multi-Family (2-6 unit)'][['Chicago Ward','Median Sale Price']]

chicago_wards_df['ward'] = chicago_wards_df['ward'].astype(int)
for df in (condos_df, sf_df, mf_df):
    df['Chicago Ward'] = df['Chicago Ward'].astype(int)
```

In [18]:

```
# [Create the charts]
def price_map(df, title, scheme):
    return (
        alt.Chart(chicago_wards_df, title=title)
        .mark_geoshape(stroke='#706545', strokeWidth=0.75)
        .transform_lookup(
            lookup='ward',
            from_=alt.LookupData(df, key='Chicago Ward', fields=['Median Sale Price'])
        )
        .encode(
            color=alt.Color('Median Sale Price:Q', title='Median price', scale=alt.Scale(scheme=scheme)),
            tooltip=[
                alt.Tooltip('ward:O', title='Ward'),
                alt.Tooltip('Median Sale Price:Q', title='Median price', format='$,')
            ]
        )
        .project(type='identity')
    )

condo_map = price_map(condos_df, 'Median sale price — Condos (2020)', 'blues')
sf_map    = price_map(sf_df,     'Median sale price — Single-Family (2020)', 'reds')
mf_map    = price_map(mf_df,     'Median sale price — Multi-Family 2-6 (2020)', 'oranges')
```

```
condo_map | sf_map | mf_map
```

Out[18]:



PROMPT: Describe the map you made, especially how you chose your color scale or speculate on how the default colors where chosen, if you went with the default. Did this map require a different color scale than above? Why or why not?

[Your Answer Here]: The three choropleth maps show the median home sale prices for condos, single-family homes, and multi-family units across Chicago's wards in 2020. Each map uses a continuous color scale, where darker shades indicate higher median prices. I used Altair's default quantitative color scale, which automatically assigns a gradient that is easy to read and emphasizes variation in price across wards.

I didn't need a different color scale here compared to the earlier categorical map because this time the data are **numeric**, not categorical. A gradient is more appropriate for showing differences in magnitude. Using the same color scheme across the three maps also makes it easier to compare relative price levels between property types and wards.

Show the *ratio* of single family home sales (counts) to condo sales in each ward.

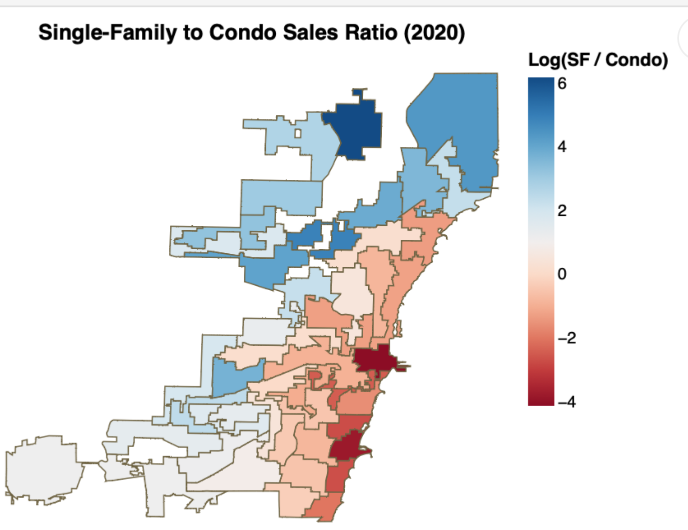
In [19]:
```
# PROMPT: use altair to make a choropleth map of the ratio of single family/condo sales counts
# HINT: you may need to reshape the data on home sales using the pandas pivot method
import math

home_sales_wide = home_sales.pivot(index="Chicago Ward", columns="Property Type", values="2020Sales count").reset_index()
home_sales_wide["Sale Ratio"] = home_sales_wide["Single-Family"] / home_sales_wide["Condo"]
home_sales_wide["Log Sale Ratio"] = home_sales_wide["Sale Ratio"].apply(math.log)
home_sales_wide.head()
```

Out[19]:

Property Type	Chicago Ward	Condo	Multi-Family (2-6 unit)	Single-Family	Sale Ratio	Log Sale Ratio
0	1	554.0	108.0	195.0	0.351986	-1.044165
1	2	930.0	52.0	81.0	0.087097	-2.440735
2	3	263.0	70.0	112.0	0.425856	-0.853655
3	4	332.0	21.0	95.0	0.286145	-1.251258
4	5	284.0	57.0	75.0	0.264085	-1.331486

In [21]:
```
# Create the chart here
# Choropleth of single-family to condo sale ratio
ratio_map = (
    alt.Chart(chicago_wards_df, title='Single-Family to Condo Sales Ratio (2020)')
    .mark_geoshape(stroke='#706545', strokeWidth=0.75)
    .transform_lookup(
        lookup='ward',
        from_=alt.LookupData(
            home_sales_wide,
            key='Chicago Ward',
            fields=['Sale Ratio', 'Log Sale Ratio']
        )
    )
    .encode(
        color=alt.Color(
            'Log Sale Ratio:Q',
            title='Log(SF / Condo)',
            scale=alt.Scale(scheme='redblue')
        ),
        tooltip=[
            alt.Tooltip('ward:O', title='Ward'),
            alt.Tooltip('Sale Ratio:Q', title='Sale Ratio', format='.2f'),
            alt.Tooltip('Log Sale Ratio:Q', title='Log Ratio', format='.2f')
        ]
    )
    .project(type='identity')
)

ratio_map
```

Out[21]:



PROMPT: Describe the map you made, especially how the color scale was chosen (whether manually or by default). Did this map require a different color scale than above? Why or why not?

[Your Answer Here]: The map shows the log of the ratio between single-family and condo home sales in each ward.

I used a diverging red-blue color scale to emphasize which type dominates in each area. Red indicates wards with more condo sales (negative value), blue shows wards with more single-family sales (positive Log(SF / Condo)), and white represents a roughly even split.

A diverging scale works better here than the previous sequential scale because the data center around a meaningful midpoint (a ratio of 1, or log ratio of 0). Unlike the earlier maps, which showed price or counts, this map captures relative differences between two categories, so a diverging scale makes the pattern clearer.

Try another!

Now, it's your turn to find a data source to map. We suggest finding data representing a place you are familiar with, maybe your hometown.

PROMPT: Describe your dataset, and provide a link to any data sources you used

[Describe Your Data Here] The dataset I used contains the resident population of Singapore by planning area from the 2019 Census. Each row represents one planning area and includes the total population as well as a breakdown by age and gender.

The corresponding geospatial data comes from the 2019 URA Master Plan Planning Area Boundary GeoJSON file, which provides polygon boundaries for each planning area in Singapore.

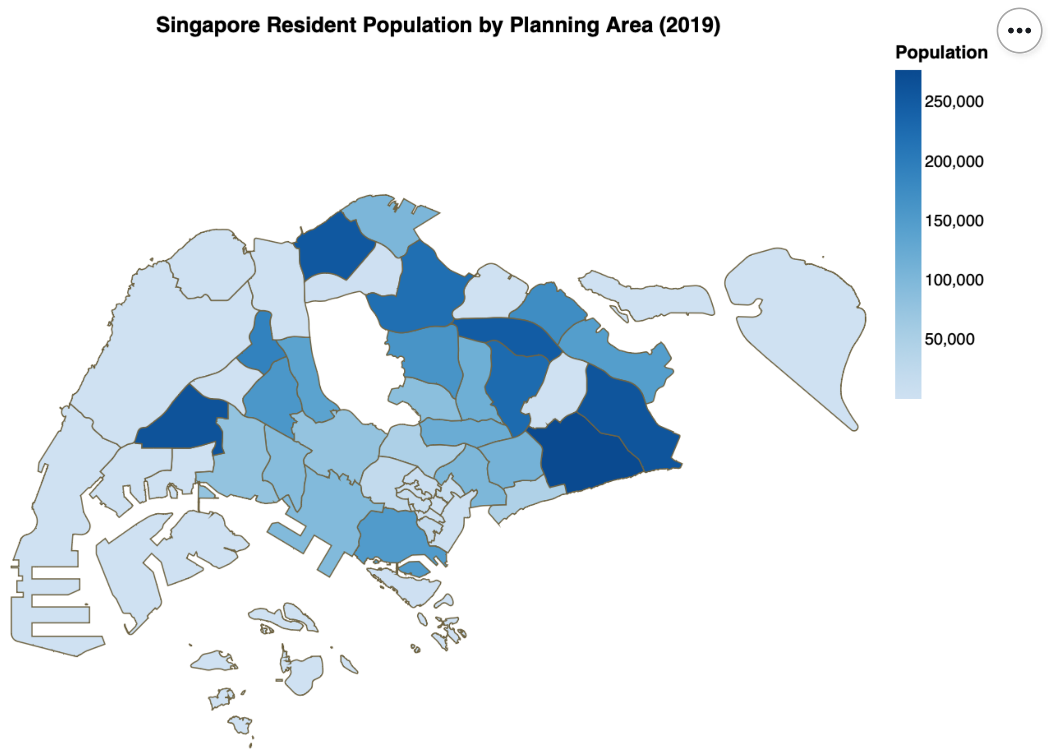
Population data: Singapore Department of Statistics (SingStat). Link: https://data.gov.sg/datasets/d_d95ae740c0f8961a0b10435836660ce0/view?utm

GeoJSON boundary data: Urban Redevelopment Authority (URA). Link: https://data.gov.sg/datasets/d_4765db0e87b9c86336792efe8a1f7a66/view?utm

This combination allows us to visualize population distribution geographically across Singapore.

In [ ]:
```python
import json, re, pandas as pd, altair as alt


with open('../exercise_data/MasterPlan2019PlanningAreaBoundaryNoSea.geojson') as f:
    sg_geojson = json.load(f)
pop_pa = pd.read_csv('../exercise_data/ResidentPopulationbyPlanningAreaSubzoneofResidenceAgeGroupandSexCensusofPopulation2

# Extract planning-area totals from CSV
pop_total = pop_pa[pop_pa['Number'].str.endswith(' - Total')].copy()
pop_total['PLN_AREA_N'] = pop_total['Number'].str.replace(' - Total', '', regex=False)
pop_total['Total_Total'] = pd.to_numeric(pop_total['Total_Total'], errors='coerce')
pop_total = pop_total[['PLN_AREA_N', 'Total_Total']]

# Get GeoJSON "Description"
def extract_pln_area(desc):
    m = re.search(r"<th>PLN_AREA_N</th>\s*<td>(.*?)</td>", desc or "")
    return m.group(1) if m else None

geo_df = pd.DataFrame({
    "type": ["Feature"] * len(sg_geojson["features"]),
    "geometry": [ft["geometry"] for ft in sg_geojson["features"]],
    "PLN_AREA_N": [extract_pln_area(ft["properties"].get("Description", "")) for ft in sg_geojson["features"]],
})

# Standardize join keys
geo_df['PLN_AREA_N'] = geo_df['PLN_AREA_N'].str.strip().str.upper()
pop_total['PLN_AREA_N'] = pop_total['PLN_AREA_N'].str.strip().str.upper()


sg_chart = (
    alt.Chart(geo_df, title='Singapore Resident Population by Planning Area (2019)')
    .mark_geoshape(stroke='#706545', strokeWidth=0.75)
    .transform_lookup(
        lookup='PLN_AREA_N',
        from_=alt.LookupData(pop_total, key='PLN_AREA_N', fields=['Total_Total'])
    )
    .encode(
        color=alt.Color('Total_Total:Q', title='Population', scale=alt.Scale(scheme='blues')),
        tooltip=[
            alt.Tooltip('PLN_AREA_N:N', title='Planning Area'),
            alt.Tooltip('Total_Total:Q', title='Population', format=',')
        ]
    )
    .project(type='mercator')
    .properties(width=520, height=520)
)

sg_chart
```

Out[ ]:



PROMPT: Describe your map. What patterns does it show in the data? Describe your choice of encodings. Why did you choose them?

[Your Answer Here]

The map shows how Singapore's population is distributed across planning areas. Darker blue areas represent regions with higher population

counts, while lighter areas indicate lower populations. From the map, we can clearly see that densely populated areas are concentrated in the northeastern and central parts of the island, such as Tampines, Bedok, and Jurong West, which are known for large residential estates.

For the encoding, I used:

- Color (quantitative) to represent total population, with a blue sequential scale that emphasizes higher values through darker shades.
- Tooltip to display exact population counts when hovering over each area.
- Geoshapes to outline each planning area and keep geographic boundaries clear.

I chose a sequential color scale because population is a continuous variable, and blue is often used in demographic maps for readability and contrast.