Cartography M.Sc.

Vector tile cache connecting effective spatial communication and geospatial AI

Sharon Chawanji

2 November 2020

Outline

1. Background and Motivation
2. Research Objectives
3. Related Work
4. Methods
5. Prototype Implementation
6. Discussion
7. Conclusion

Motivation

- According to Li et al., (2016), nearly 80% of data are published on the Web.
- The data is often published as Linked Data embedded with geographic information that can be mapped and geovisualised.
- Linked data is structured data which is interlinked with other data.

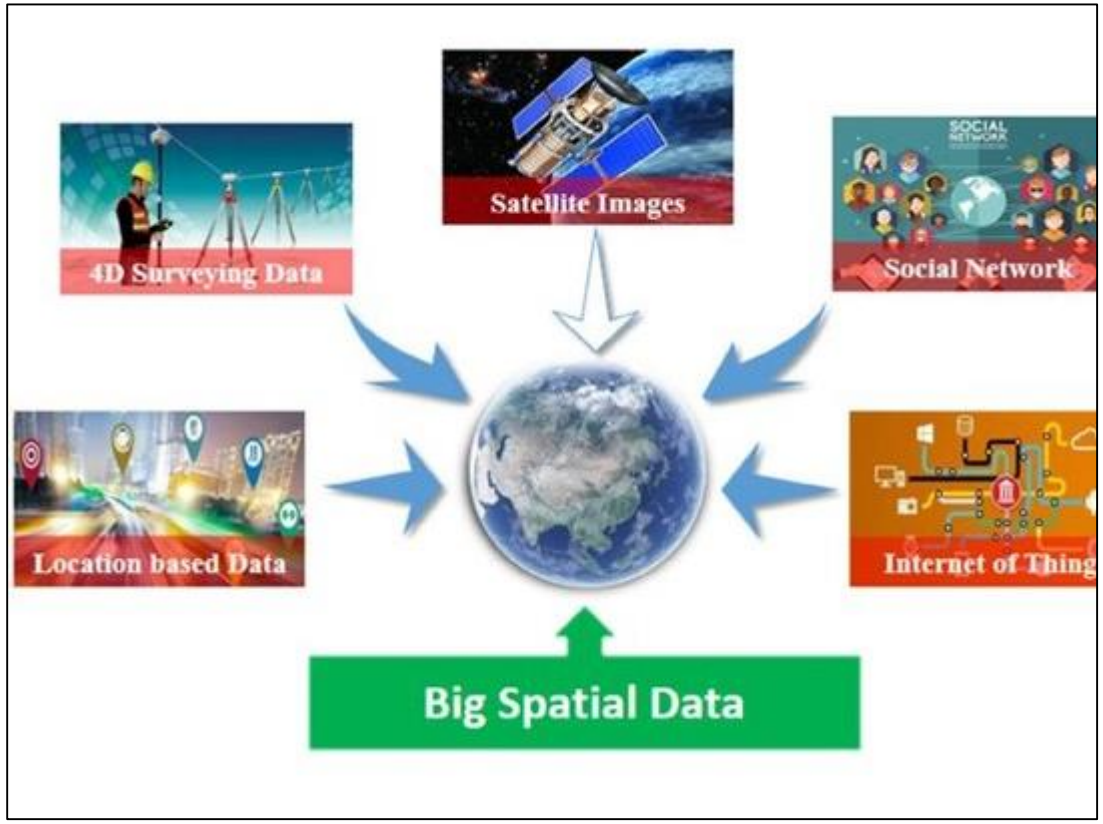


Figure 1. The classification diagram of big spatial data (Yao and Li, 2018).

Motivation

- Linked Data is useful for semantic queries to retrieve new knowledge.
- The Linked data is build on technologies such as
    1. RDF (Resource Description Framework) – Structured data publication on the web as a triple store.
    2. **SPARQL** – query language for RDF data stores.

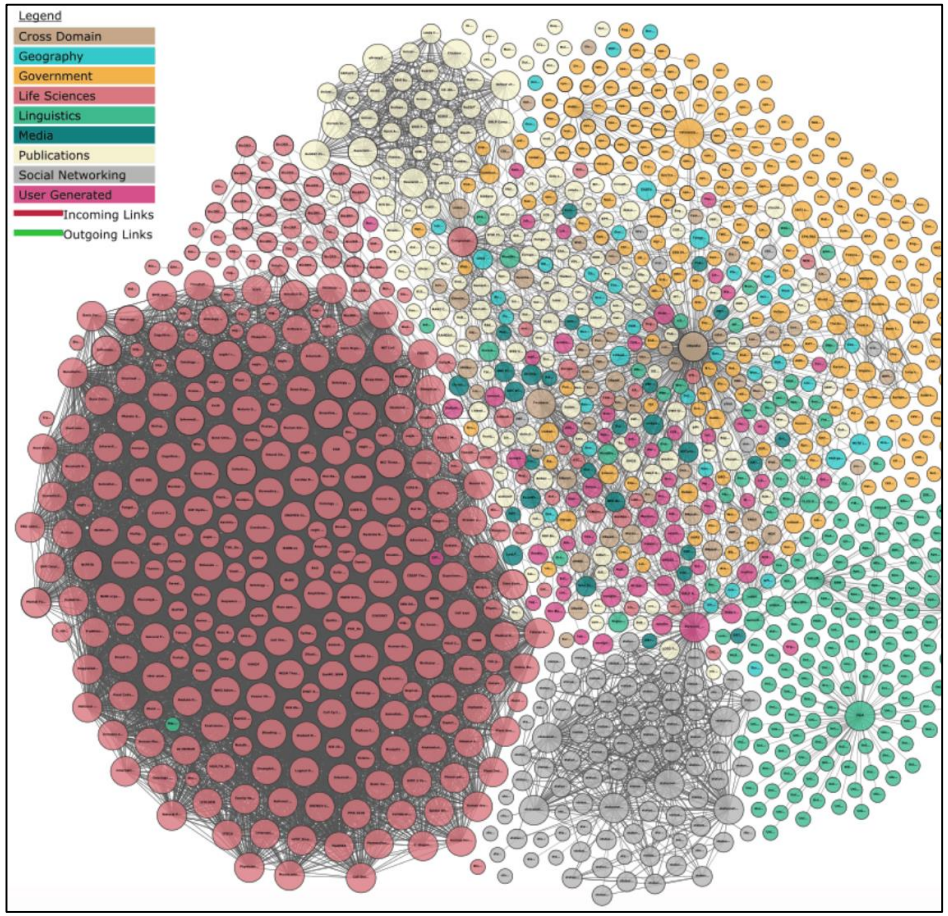


Figure 2. Linked Open data cloud

(Source :https://lod-cloud.net/clouds/lod-cloud.svg)

Motivation

- Through applying **Geospatial Artificial Intelligence (GEO AI)** it is possible to get concealed knowledge from Linked data that is linked to geography.
- For example we can ask a computer :
  - **“What are the GDP values and population values for UN member states? The results should name the UN member states based on ISO 3166-1 numeric standard codes”**
- The result is achieved by integrating and analysing interlinked data using Semantic Web technologies.

Motivation

- **SPARQL endpoint services** provides an interface to query Linked Data and make it available in common data formats such as CSV data.
- **CSV data** is integrated with geospatial frameworks so that it can be visualised on Web GIS Application.
- **Table Joining service** (TJS) is an OGC open standard that can be used to integrate attribute data in tabular format with geographic data.

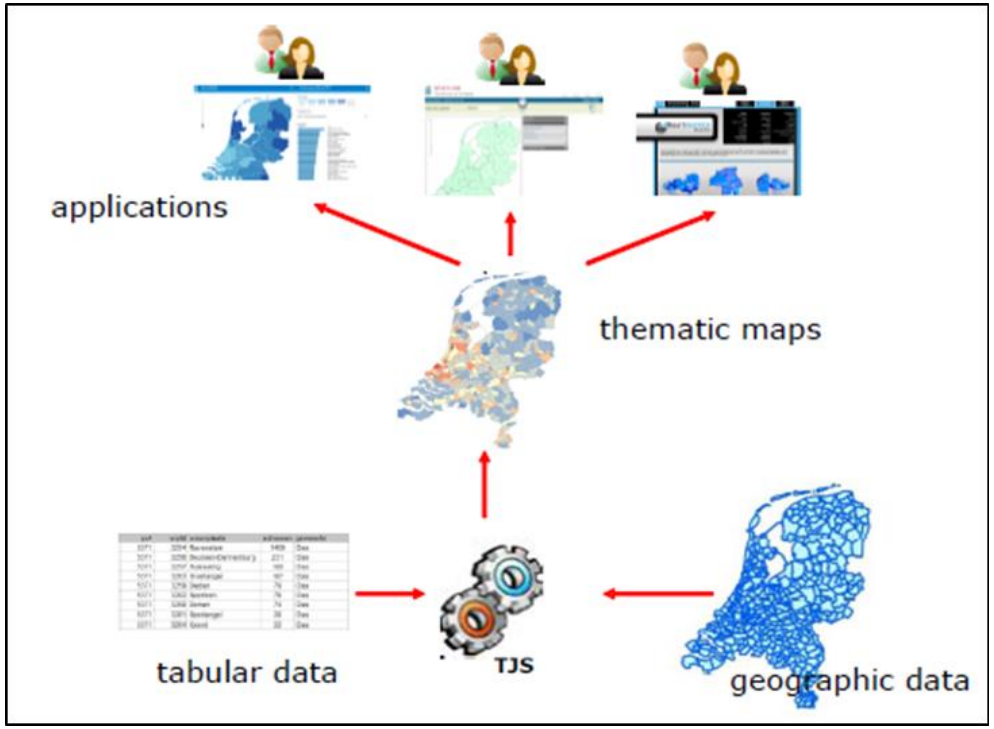


Figure 3. TJS Concept

(Source :Grothe and Brentjens, 2013).

Motivation

- **Table Joining Service** - Web service interface that defines how attribute and geospatial data can be accessed from different servers on a computer network and join them together.
- **Prerequisite for the integration**

1. Common framework keys also known as **geographic identifiers**.
2. Attribute data in tabular format.
3. Vector based geospatial framework data.

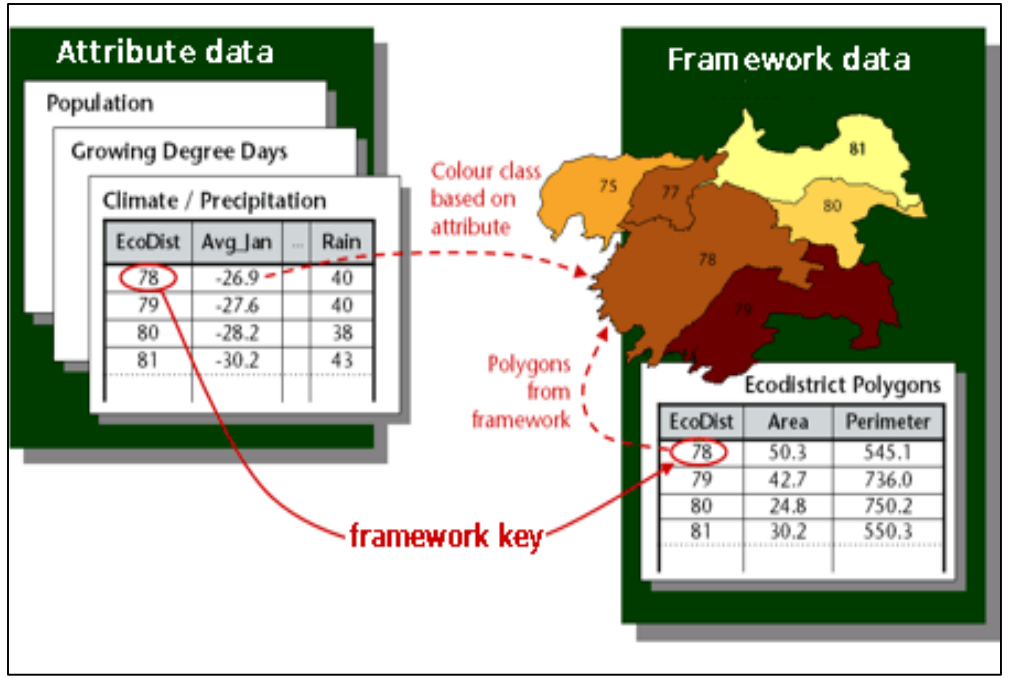


Figure 4. TJS inputs.

(Source :http://geoprocessing.info/tjsdoc/Overview)

Research Objectives

1. Examine the possibility of using **cached vector tiles** as a **geospatial framework** to be integrated with **attribute data** by means of a **Table Joining Service**.
2. Develop a **TJS API** prototype that will be used as a tool for integrating **cached vector tiles** with attribute data.
3. Display the results of **cached vector tiles** **joined** with attribute data, using the prototype TJS API, on a web map application .

Related work

- Integrating data attribute data with geospatial data for Web maps has been done in several ways.
- OGC standards that deliver Web thematic maps
  1. Web Map Service and Styled Layer Descriptor.
  2. Geolinking Service Standard (GLS) and Geolinked Data Access Service (GDAS).
  3. **Table Joining Service** a combination of GLS and GDAS.

Related Work :Table Joining Service

- Previous research focus on Web Feature Service (WFS) as framework data.
- WFS is an OGC standard that provide an interface for requesting geographic features across the web.
- Geographic features are encoded as shapefiles, GeoJSON or Geographic Markup Language (GML).
- WFS become inefficient and slow to respond with large volumes of data (EFGS and Eurostat, 2019).

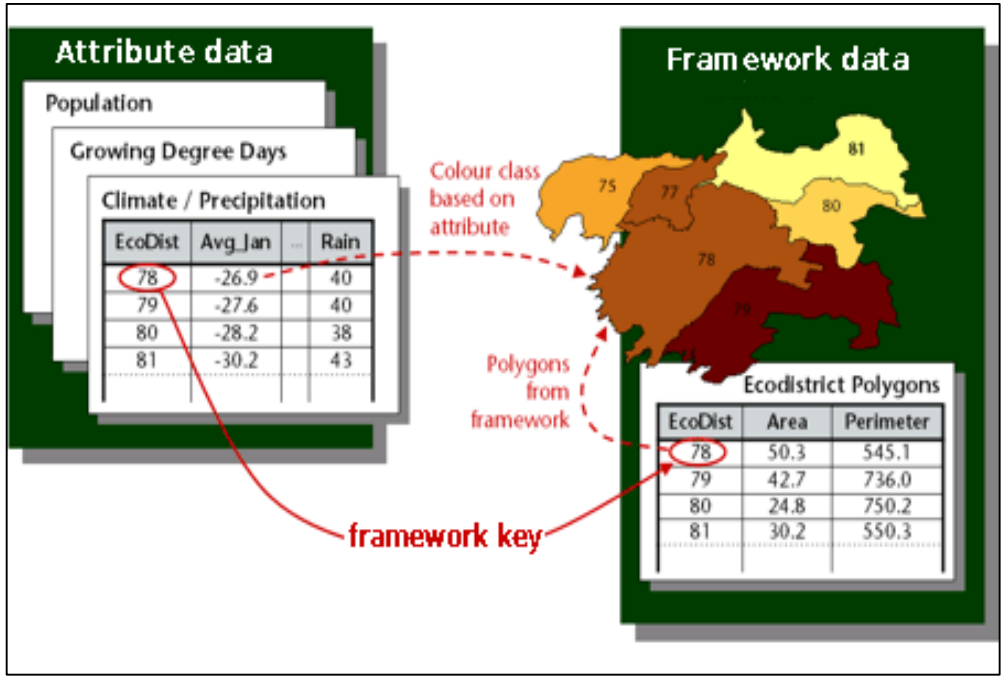


Figure 5. TJS inputs

(Source :http://geoprocessing.info/tjsdoc/Overview)

Related Work :Table Joining Service

- Attribute data format has been inconsistence.
- Specified format for TJS is GDAS .
- Common attribute data formats used are: CSV ,SDMX, ODATA .
- Framework keys: area codes, population grids.
- Naming of framework keys not consistent.
- Use of persistent identifiers as framework keys is nonexistent.

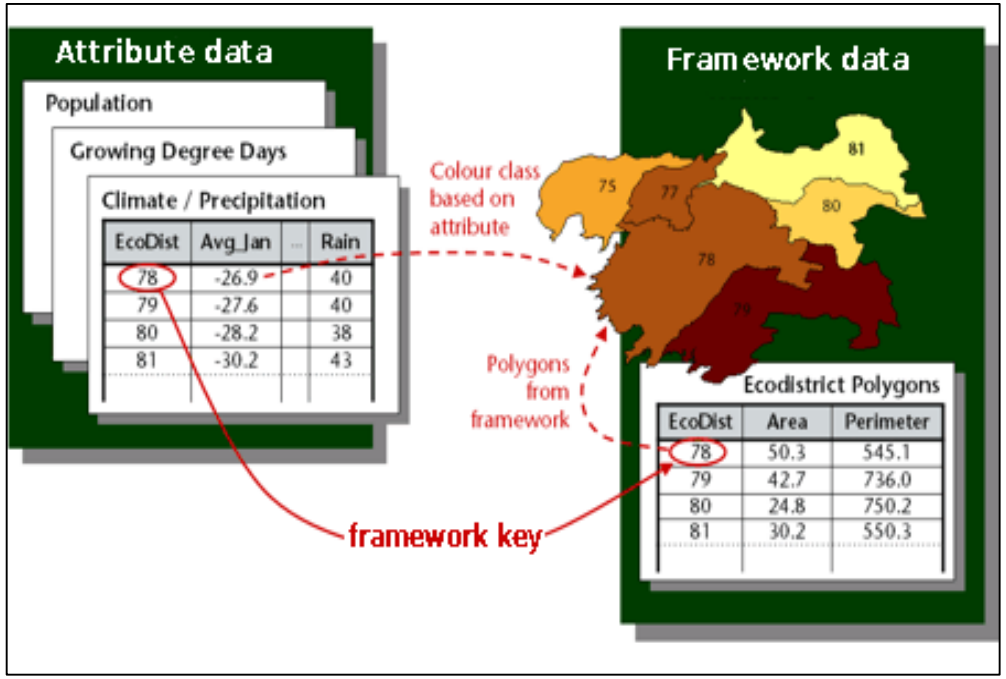


Figure 5. TJS inputs

(Source :http://geoprocessing.info/tjsdoc/Overview)

Related work :TJS implementations

- At the time of writing a Google search shows that two TJS software implementations are available :
  1. Géoclip
  2. Geoserver TJS Extension

Software Products	Client/Server	TJS operations	Technology	Type of Software
Géoclip	Client server	TJS-access TJS-join	?	Proprietary
Geoserver TJS Extension	Server	TJS-access TJS-join	Java	Open source

Table 1. TJS Implementations

(Source :Grothe and Brentjens, 2013).

Related Work :Vector tiles

- Vector tiles are relatively new technology.
- Considered the most efficient way of transmitting geospatial data over the web.
- Mostly optimized for rendering.
- Can be efficiently cached reducing access rate on the server.
- Rendered by the client and can also be cached on the browser.

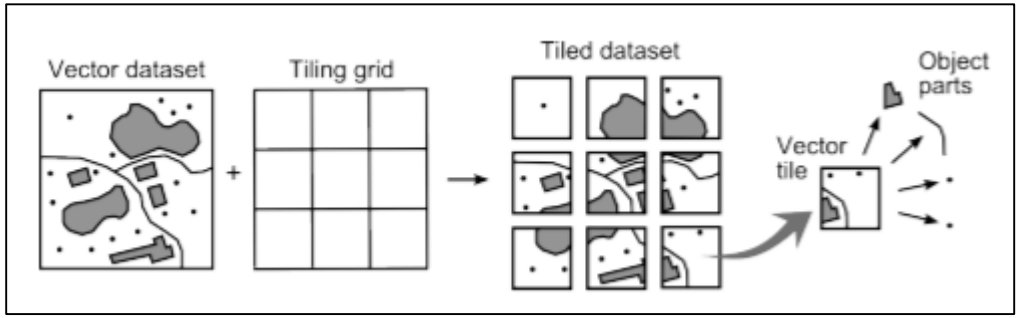


Figure 6. The procedure for generating vector tiles according to Gaffuri (2012).

Related Work :Vector tiles (continued)

- Allow feature manipulation and styling on the client side.
- They can be published as **Tile Map Services** (TMS).
- TMS is a **RESTful Web service** that gives access to underlying textual data.
- Mostly available is three data formats namely GeoJSON , TopoJSON and MVT.
- **GeoJSON** – JSON object both machine and human readable and can be used as a dataset.

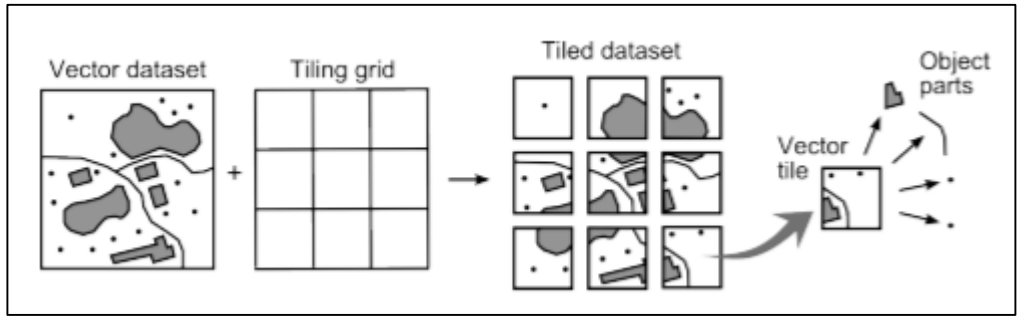


Figure 7. The procedure for generating vector tiles according to Gaffuri (2012).

Related Work :Vector tiles (continued)

- OGC is a process of adopting Vector tile specification.
- Previous studies focused on optimized rendering , efficient transfer and storage.
- Potential of using vector tiles as source of geographic data for geoprocessing and spatial analysis is not well understood.

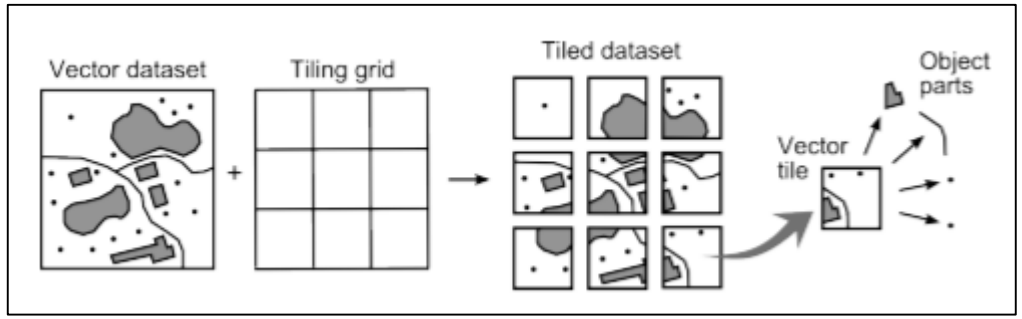


Figure 8. The procedure for generating vector tiles according to Gaffuri (2012).

Methods

Workflow stage	Software and /or programming language
Pre-processing and editing CSV data	Microsoft Excel ,Postgres database SQL, Python 3.8.5 SPAQRLWrapper
Geospatial data pre-processing editing	QGIS 3.4.6
Vector tile generation	GeoServer 2.15
Vector tile caching	GeoWebCache integrated in Geoserver 2.15
Develop mock server for storing CSV data	Python 3.8.5 Flask framework
Develop TJS server and TJS JoinData API	Python 3.8.5 Flask framework
Develop web map application	OpenLayers v6.4.3, JavaScript and Node.js

Table 1. Software and programming language used.

Methods: Experimental data

- Vector data – 1:10 Million ESRI shapefile from Natural Earth
- Attribute data – Wiki data Knowledge graph and UN statistical data in CSV format
- Geographic identifiers/framework keys - ISO 3166-1 numeric standard codes also the same area codes used in the UN M49 series country codes for UN member states and regions.
  - The ISO 3166-1 numeric codes can be accessed from Wiki data knowledge graphs as **persistent identifiers**.
  - Persistent identifiers are references to an object that can accessed over the internet.

Methods

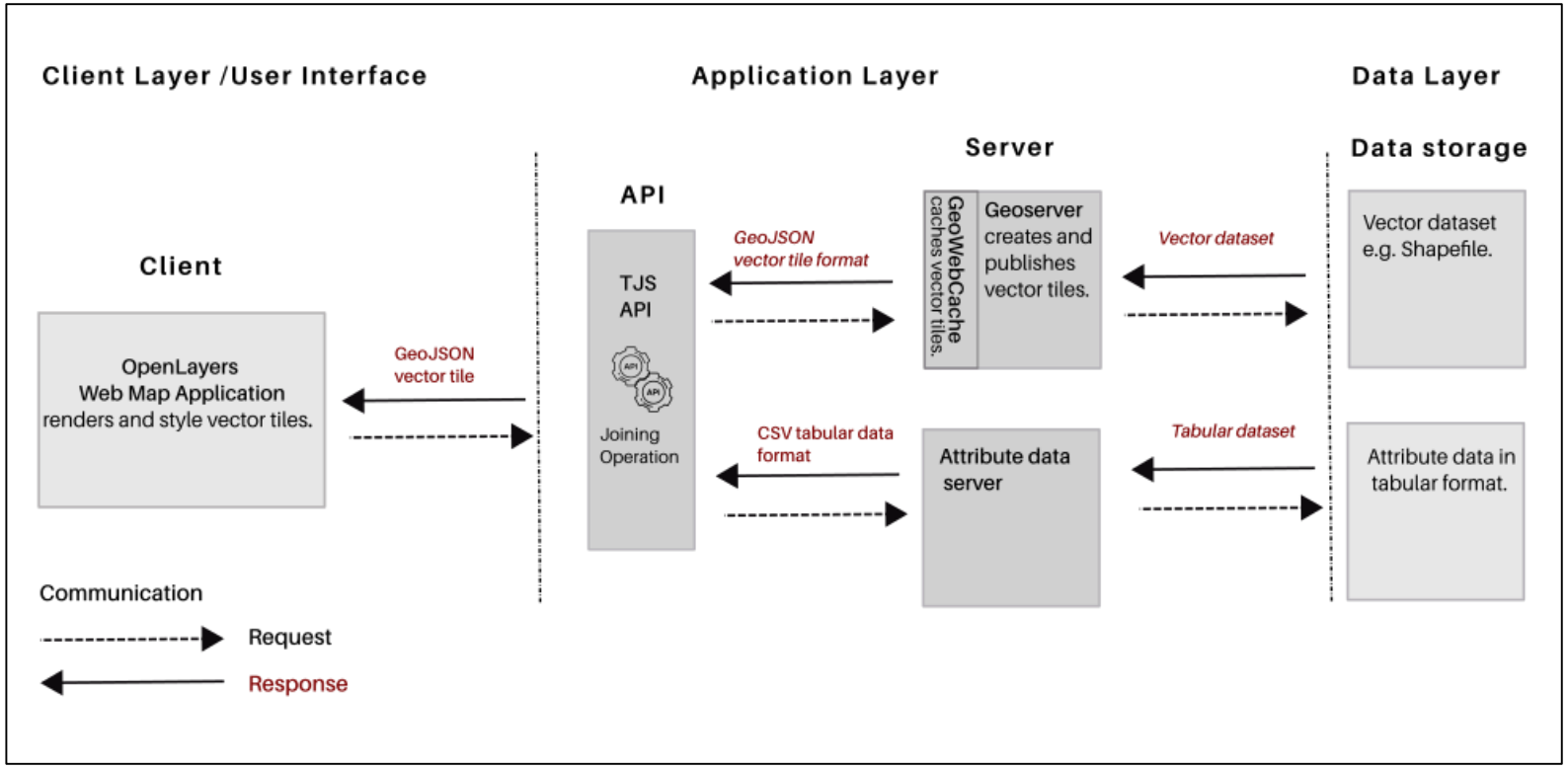


Figure 9. To achieve the research objectives a prototype is built based on an three-layer software architecture.

Prototype Implementation

- TJS Joining Operation uses three python modules:
  1. Request – Manages http requests of GeoJSON and CSV data files.
  2. GeoPandas – handles GeoJSON data and the joining operation.
  3. Pandas – handles the CSV data.

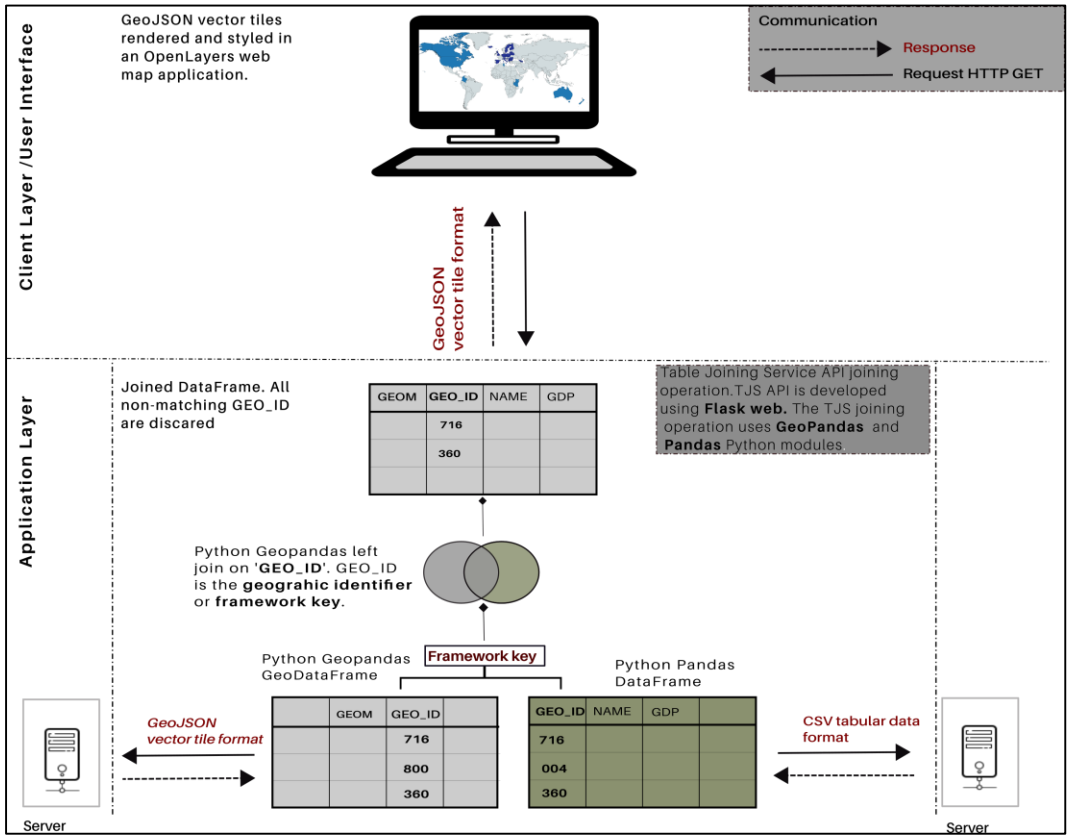
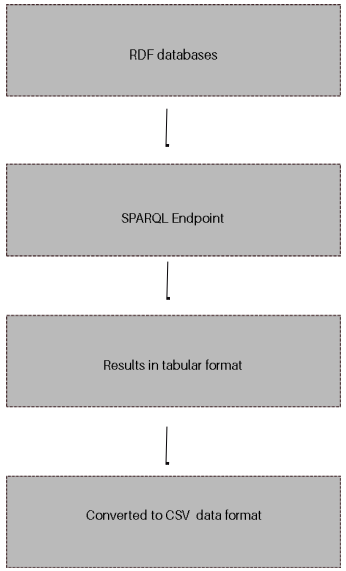


Figure 10. Table joining operation workflow illustration.

Prototype Implementation : Data Layer
Attribute data acquisition

```
SELECT distinct? countryLabel ?UN_A3 ?ISO_A2 ?Population
WHERE
{
  ?country wdt:P299 ?UN_A3 .
  ?country wdt:P297 ?ISO_A2 .
  ?country wdt:P1082 ?Population.
  SERVICE wikibase:label { bd:serviceParam wikibase:language
"[AUTO_LANGUAGE],de" }
} order by ?ISO_A2
```




```python
import sys
from SPARQLWrapper import SPARQLWrapper, JSON
import pandas as pd

un_df = pd.read_csv("static\sample-csv.csv")
endpoint_url = "https://query.wikidata.org/sparql"
query = """
SELECT distinct ?countryLabel ?UN_A3 ?ISO_A2?Population
WHERE
{?country wdt:P299 ?UN_A3 .
  ?country wdt:P297 ?ISO_A2 .
  ?country wdt:P1082 ?Population.
  SERVICE wikibase:label { bd:serviceParam wikibase:language
  "[AUTO_LANGUAGE],de" }
} order by ?ISO_A2"""


def get_results(endpoint_url, query):
    user_agent = "WDQS-example Python/%s.%s" % (sys.version_info[0],
sys.version_info[1])
    sparql = SPARQLWrapper(endpoint_url, agent=user_agent)
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    return sparql.query().convert()


results = get_results(endpoint_url, query)
results_df = pd.io.json.json_normalize(results['results']['bindings'])
wiki_df = results_df[['countryLabel.value', 'UN_A3.value', 'ISO_A2.value',
'Population.value']]
# Convert column names
wiki_df.columns = ['DE_name', "UN_A3", 'ISO_A2', "Population"]
wiki_df_columns = ["UN_A3", 'ISO_A2', 'DE_name', "Population"]
un_df_columns = ["UN_A3", "Country_name", 'GDP(2005)']
un_df = un_df.reindex(columns=un_df_columns)
wiki_df = wiki_df.reindex(columns=wiki_df_columns)
un_df['UN_A3'] = un_df['UN_A3'].astype(int)
wiki_df['UN_A3'] = wiki_df['UN_A3'].astype(int)
result = un_df.join(wiki_df.set_index('UN_A3'), on=["UN_A3"])

# Save results in a static file folder
result.to_csv(r'sample-csv.csv', index=False)
```

Figure 11. SPARQL endpoint Python interface.

Prototype Implementation : Application Layer

- **Geoserver** is used for generation and publication of vector tiles .
- **GeoWebCache** (GWC) caches vector tiles.
- Vector tiles encoded in GeoJSON can be accessed using the URL below:

```
http://localhost:8080/geoserver/gwc/tms/1.0.0/layername@grisetId@formatExt
ension/z/x/y.format

z is zoom level

x and y define a given tile coordinates

gridsetId refers to the coordinate reference system

format refers to format of the vector tiles

GeoJSON vector tiles requests from GeoWebCache

http://localhost:8080/geoserver/gwc/service/tms/1.0.0/countries%3Ane_110m_
admin_0_countries@EPSG%3A3857@geojson/{z}/{x}/{y}.geojson
```

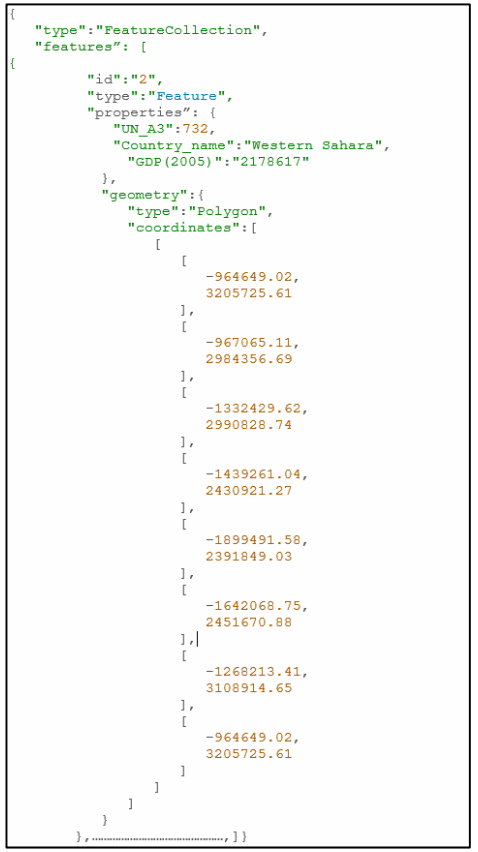


Figure 12. GeoJSON structure.

Prototype Implementation : Application Layer

- Attribute web server built using Flask.
- Flask is a web framework for building webservices, web resources.

```
1. # URL for accessing csv data files
2. http://127.0.0.1:8000/static/sample-csv.csv
```

```
43. from flask import Flask
44.
45. app = Flask(__name__)
46.
47. @app.route('/')
48. def index():
49.     return '<!DOCTYPE html>'
50.            '<html>'
51.            '<head>'
52.            '<title>Attribute data server</title>'
53.            '</head>'
54.            '<body>'
55.            '<h1>Attribute data web server</h1>'
56.            '<p>Attribute data Server is running on http://localhost:8000 . '
57.            'To request a file from the server add name of the CSV in from of
58.            'the web server e.g. http://127.0.0.1:8000/static/sample
   csv.csv</p>'
59.            '</body>'
60.            '</html>'
61.
62.
63. app.run(debug=True, port=8000)  # run app in debug mode on port 8000
```

Figure 13. Python script for developing a server using Flask.

Prototype Implementation Application Layer

- TJS API Flask endpoint

```
http://127.0.0.1:5000/tjs/api?

FrameworkURI=http://localhost:8080/geoserver/gwc/service/tms/1.0.0/countri
es%3Ane_110m_admin_0_countries@EPSG%3A3857@geojson/{z}/{x}/{y}. geojson&

GetDataURL=http://127.0.0.1:8000/static/sample-csv.csv&

FrameworkKey=UN_A3&

attribute1=Country_name&

attribute2=GDP (2005)
```

```python
import geopandas as gpd
import pandas as pd from flask
import Flask, request

app = Flask(__name__)

def get_framework_data(FrameworkURI):
    gdf = gpd.read_file(FrameworkURI)
    return gdf

def get_attribute_data(GetDataURL):
    df = pd.read_csv(GetDataURL)
    return df

def get_framework_key(FrameworkKey, attribute1, attribute2):
    FrameworkKey = str(FrameworkKey)
    attribute_1 = str(attribute1)
    attribute_2 = str(attribute2)
    return [FrameworkKey, attribute_1, attribute_2]

@app.route('/tjs/api', methods=['GET'])
def join_data():
    # Input parameters required
    FrameworkURI = request.args.get('FrameworkURI')
    GetDataURL = request.args.get('GetDataURL')
    FrameworkKey = request.args.get('FrameworkKey')
    attribute1 = request.args.get('attribute1')
    attribute2 = request.args.get('attribute2')
    # Joining operation.
    gdf = get_framework_data(FrameworkURI)
    adf = get_attribute_data(GetDataURL)
    keys = get_framework_key(FrameworkKey, attribute1, attribute2)
    gdf_columns = ['UN_A3', 'geometry']
    adf_columns = keys
    adf = adf.reindex(columns=adf_columns)
    gdf = gdf.reindex(columns=gdf_columns)
    adf['UN_A3'] = adf['UN_A3'].astype(int)
    gdf['UN_A3'] = gdf['UN_A3'].astype(int)
    geometry = gdf[['geometry', 'UN_A3']]
    attributes = adf[keys]
    geometry = geometry.merge(attributes, on='UN_A3').reindex(gdf.index)
    geojson = geometry.to_json()
    return geojson


app.run(debug=True, port=5000)  # run app in debug mode on port 5000
```

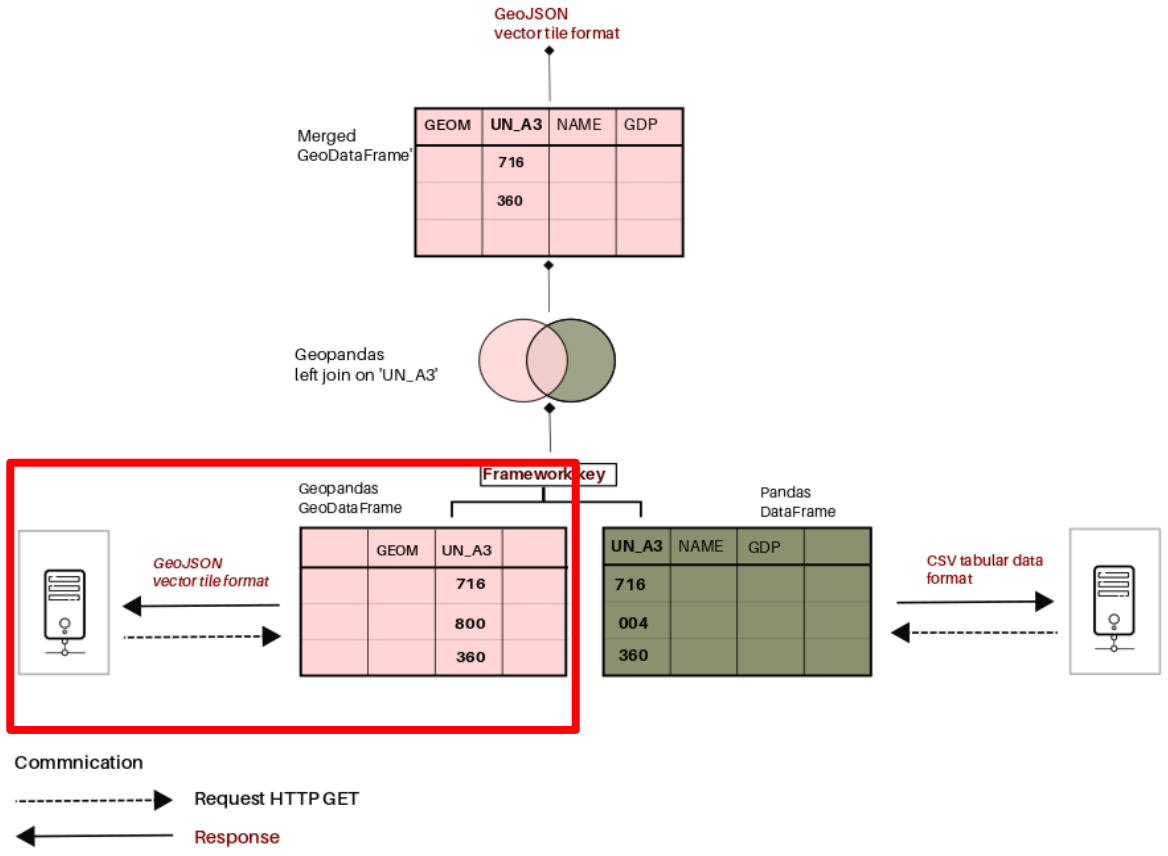
Figure 14. Python script for developing a TJS API using Flask .

Prototype Implementation :Application Layer

- Converting GeoJSON to dataframe using GeoPandas.

geometry	UN_A3	NAME_EN
POLYGON ((20037508.34 -1812498.41 ….	242	Fiji
POLYGON ((3774143.87 -105758.36 ….	834	Tanzania
POLYGON ((-964649.02 3205725.61	732	Western Sahara
POLYGON ((-13674486.25 6274861.39	124	Canada
POLYGON ((-13674486.25 6274861.39	840	United States of America

Figure 11. GeoPandas dataframe.

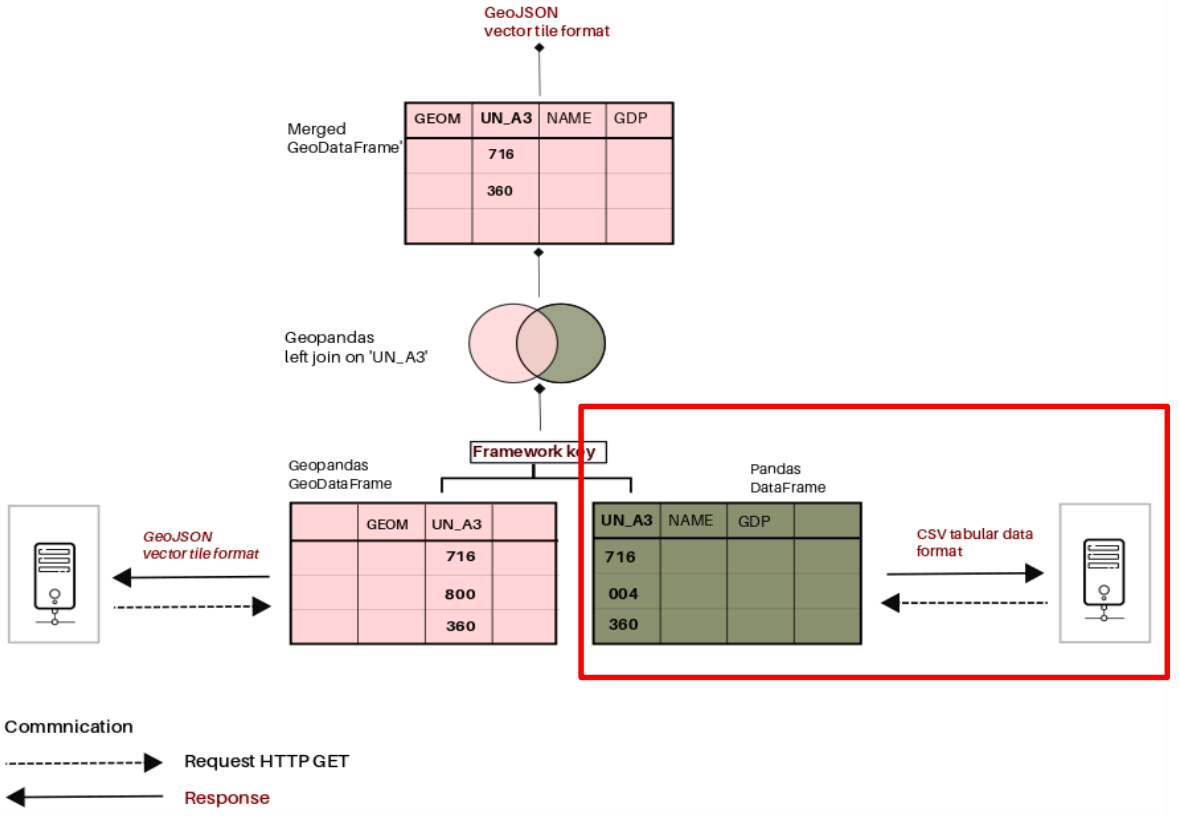


Prototype Implementation :Application Layer

- Converting CSV to dataframe using Pandas.

UN_A3	Country_name	GDP (2005)
4	Afghanistan	17140
8	Albania	19895
12	Algeria	248534
20	Andorra	48395
24	Angola	94874

Figure 12. Pandas dataframe from CSV data conversion.

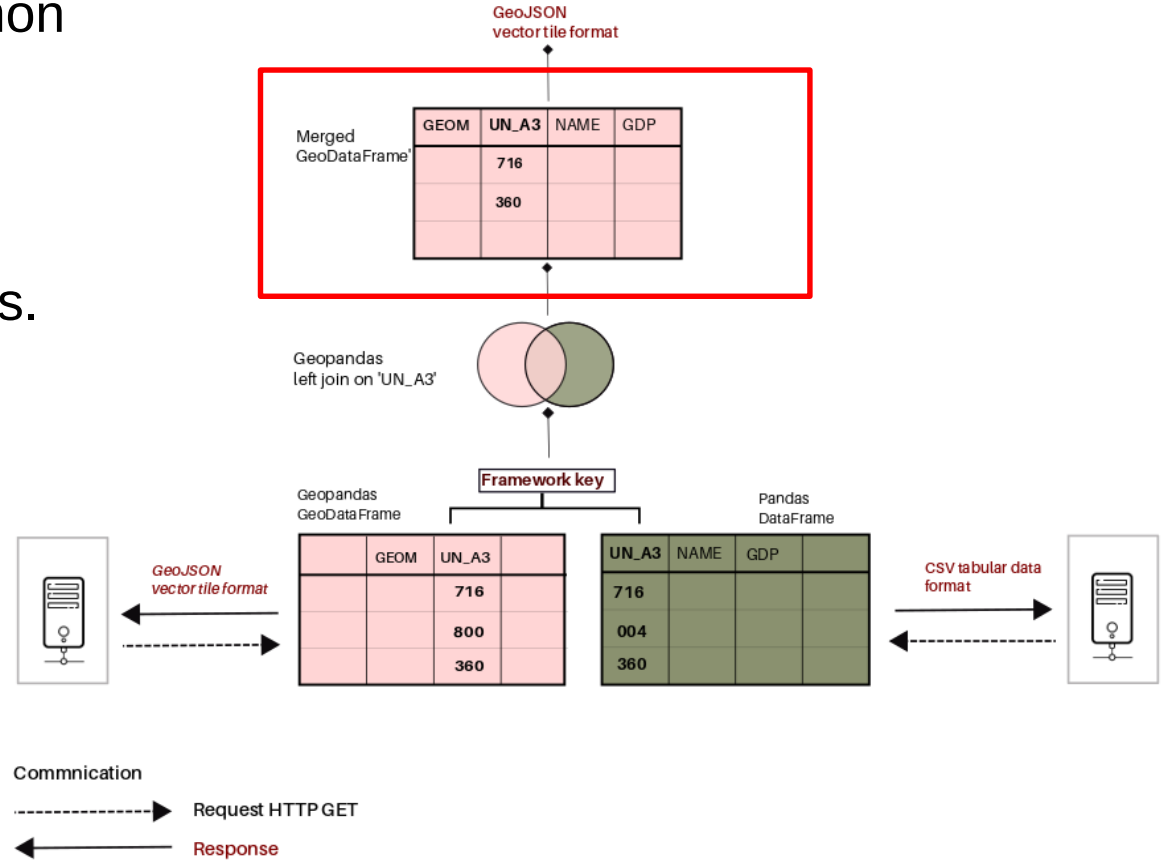


Prototype Implementation :Application Layer

- Merging of the dataframes is based on a common geographic identifier.
- In this case column with name UN_A3, which contain the ISO 3166-1 numeric standard codes.

geometry	UN_A3	Country_name	GDP(2005)
POLYGON ((20037508.34 -1812498.41 ….	242	Fiji	17140
POLYGON ((3774143.87 -105758.36 ….	834	Tanzania	19895
POLYGON ((-964649.02 3205725.61	732	Western Sahara	248534
POLYGON ((-13674486.25 6274861.39	124	Canada	48395
POLYGON ((-13674486.25 6274861.39	840	United States of America	94874
POLYGON ((20037508.34 -1812498.41 ….	398	Kazakhstan	18627

Figure 13. Result merged dataframe.



Prototype Implementation :Application Layer


```
{
  "type":"FeatureCollection",
  "features": [
{
        "id":"2",
        "type":"Feature",
        "properties": {
           "UN_A3":732,
           "Country_name":"Western Sahara",
            "GDP(2005)":"2178617"
         },
         "geometry":{
            "type":"Polygon",
            "coordinates":[
               [
                  [
                     -964649.02,
                     3205725.61
                  ],
                  [
                     -967065.11,
                     2984356.69
                  ],
                  [
                     -1332429.62,
                     2990828.74
                  ],
                  [
                     -1439261.04,
                     2430921.27
                  ],
                  [
                     -1899491.58,
                     2391849.03
                  ],
                  [
                     -1642068.75,
                     2451670.88
                  ],
                  [
                     -1268213.41,
                     3108914.65
                  ],
                  [
                     -964649.02,
                     3205725.61
                  ]
               ]
            ]
         }
      },.....,]}
```

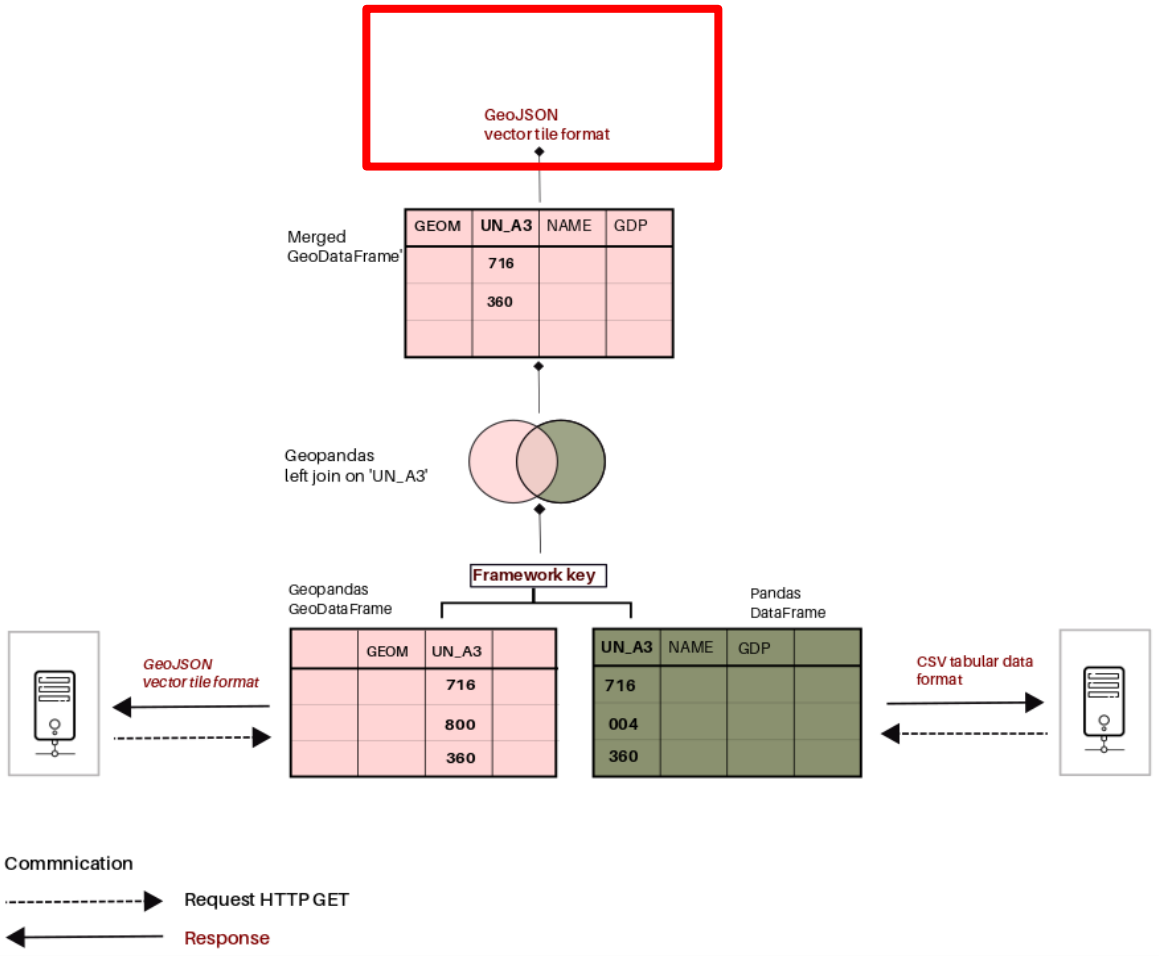
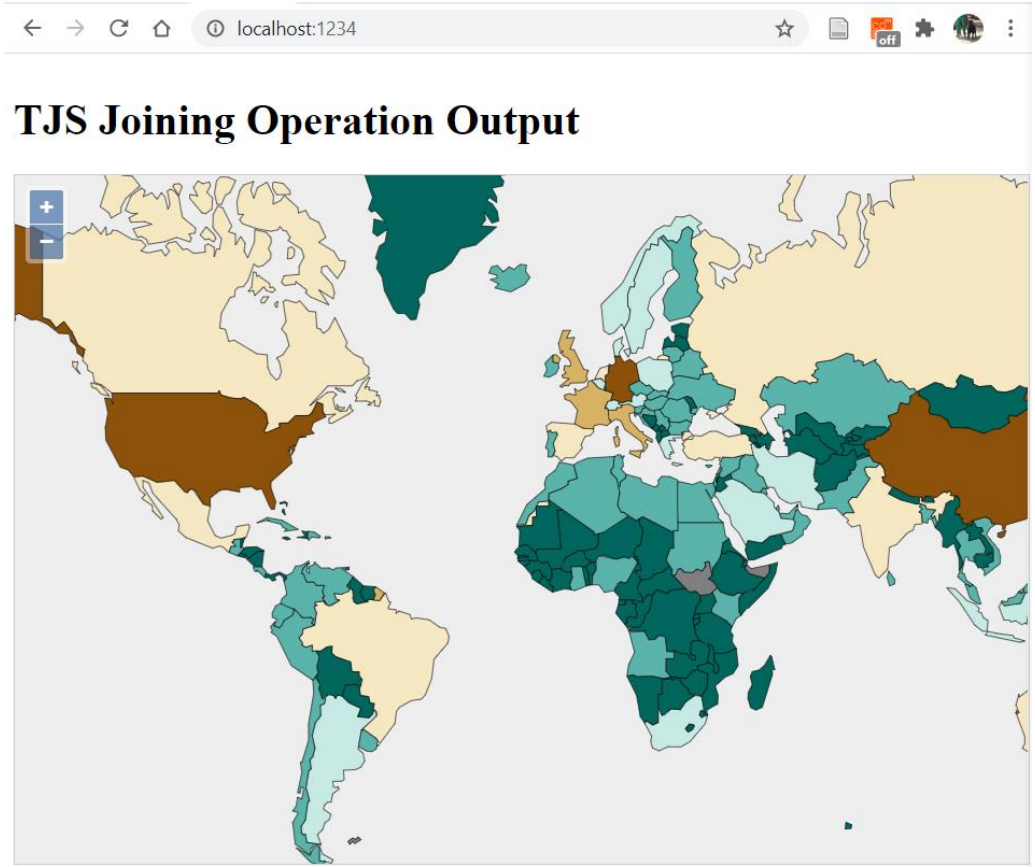



Figure 14. GeoJSON structure.

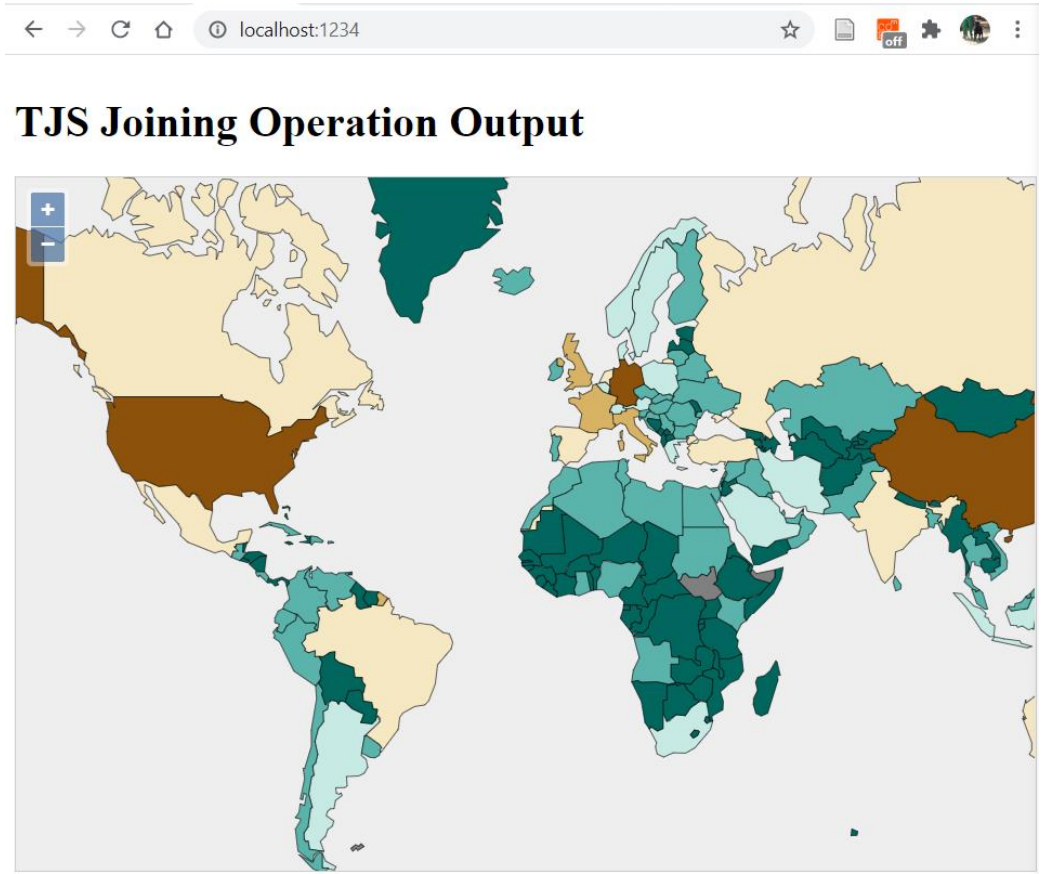
Prototype Implementation : Client Layer




```
1. import 'ol/ol.css';
2. import {GeoJSON} from 'ol/format';
3. import {Map,View} from 'ol';
4. import {Fill,Stroke,Style,Text} from 'ol/style';
5. import VectorTileLayer from 'ol/layer/VectorTile';
6. import VectorTileSource from 'ol/source/VectorTile';
7.
8. var colorGradient = [
9.   'rgb(128,128,128)', 'rgb(140,81,10)', 'rgb(216,179,101)',
10.  'rgb(246,232,195)', 'rgb(199,234,229)', 'rgb(90,180,172)',
11.  'rgb(1,102,94)']
12.
13.// map the income level codes to a colour value, grouping them
14.var gradStyle = function (feature, resolution) {
15.  var data = feature.get('GDP(2005)');
16.  var color;
17.  if (data < 50000) {
18.          color = colorGradient[6]; //low value
19.  } else if (data >= 50000 && data < 500000) {
20.          color = colorGradient[5]; //
21.  } else if (data >= 500000 && data < 1000000) {
22.          color = colorGradient[4];
23.  } else if (data >= 1000000 && data < 3000000) {
24.          color = colorGradient[3];
25.  } else if (data >= 3000000 && data < 5000000) {
26.          color = colorGradient[2];
27.  } else if (data >= 5000000) {
28.          color = colorGradient[1];
29.  } else if (data = 'null') {
30.          color = colorGradient[0];
31.  }
32.  return new Style({
33.          stroke: new Stroke({
34.                  color: 'black',
35.                  lineCap: 'butt',
36.                  lineJoin: 'miter',
37.                  width: 0.5,
38.          }),
39.          fill: new Fill({
40.                  color: color
41.          }),
42.  });
43. }
```

Figure 15. OpenLayers JavaScript: modules and styling.

Prototype Implementation : Client Layer




```
var url = 'http://127.0.0.1:5000/tjs/api?' +
  'FrameworkURI=http://localhost:8080/geoserver/gwc/service/tms/1.0.0/' +
  'countries%3Ane_110m_admin_0_countries' +
  '@EPSG%3A3857@geojson/{z}/{x}/{-y}.geojson&' +
  'GetDataURL=http://127.0.0.1:8000/static/sample-csv.csv&' +
  'FrameworkKey=UN_A3&attribute1=Country_name&attribute2=GDP(2005)'


const layer = new VectorTileLayer({
  style: gradStyle,
  source: new VectorTileSource({
          attributions: '',
          format: new GeoJSON(),
          maxZoom: 19,
          url: url,
          tileLoadFunction: function (tile, url) {
          tile.setLoader(function (extent, resolution, projection) {
          fetch(url).then(function (response) {
                          response.text().then(function (data) {
                          const jsons = JSON.parse(data);
                          const format = tile.getFormat();
                          console.log(data);
                          tile.setFeatures(format.readFeatures(data));
                                  });
                          });
                  });
          },
  })
});

var map = new Map({

  view: new View({
          center: [0, 0],
          zoom: 2,
          maxZoom: 20
  }),
  layers: [layer],
  target: 'map'
});
```

Figure 16. OpenLayers JavaScript: accessing vector tiles from TJS API.

Discussion

- The findings will be discussed with focus on;
  1. Framework data - cached vector tiles.
  2. Framework keys - geographic identifiers.
  3. Attribute data.
  4. Software architecture.

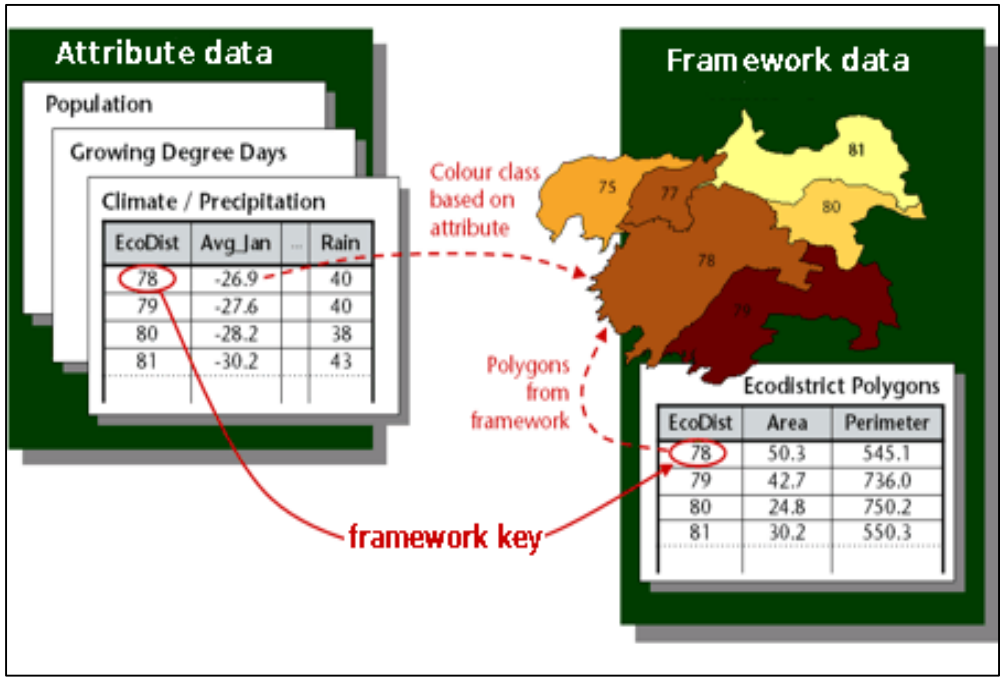


Figure 17. TJS inputs

(Source :http://geoprocessing.info/tjsdoc/Overview)

Discussion: Framework data

- Cached vector tiles can be used as framework data for TJS.
- It is vital that all geographic features be embedded with a unique geographic identifier.
- It is important to preserve all geometric and property information of GeoJSON features during and after joining operation so that the vector tiles can be properly assembled on the web GIS client.

Discussion : Framework Keys

- All non matching geographic identifiers were discarded including the geometry and attribute data which lead to missing geographic features.
- The solution was to make sure that all GeoJSON features had corresponding geographic identifiers in the attribute data even for no data attribute value.
- Using standardised framework keys/geographic identifiers i.e. ISO 3166-1 numeric standard codes removed any discrepancies.

Discussion: Framework Keys

- In comparison to previous studies; this study used standardised persistent identifiers therefore, there were no ambiguity in the naming of geographic identifiers in both attribute data and geographic data.
- In the study to examine impact analysis of TJS for Statistic Netherlands (Bresters et al., 2016).
- The two datasets would had almost similar keys for example “GM0307” as opposed to just “0307”.

Discussion: Attribute data

- Specified attribute data format for TJS is GDAS format.
- However, there are already more popular formats for attribute data.
- For example;
  - On the Web its CSV
  - For INSPIRE directive it is SDMX
- Converting common tabular data formats to GDAS adds another complexity.

Discussion: Attribute data

- For this study CSV data was used because it can be converted into Linked Data or from Linked data (Mahmud et al., 2020).
- CSV data format is a common data format in previous most studies.
- However, for the impact analysis of TJS study an API that converts CSV, Odata and SDMX data formats to GDAS was developed (Bresters et al., 2016).

Discussion : Software Architecture

- Similar to previous studies, the software architecture used in this study support data to be published once and reused many times through use of services.
- This is beneficial because statistical and spatial data providers can publish and manage their data separately.
- Geographic identifiers become vital for data integrating .

Conclusion

- This study provided a proof of concept on the potential of using cached vector tiles as frame work data for TJS.
- The research objectives were accomplished .

Conclusion

Achieved research objectives:

1. The results showed that vector tile caches can be successfully used as framework data for TJS.
2. To examine the potential of using cached vector tiles encoded in GeoJSON format as framework data for TJS; a TJS API was developed with a method that allow joining of attribute data with geographic features.
3. An OpenLayers web application was developed to display the resultant vector tiles populated with attribute data using the TJS API.

Future work

- The current version of TJS Version: 1.0.0 is under review.
- The National Land Survey of Finland is currently working on the revision of the TJS addressing some of concerns brought up by previous studies and this current study.
- These concerns includes:
  1. Defining TJS as a RESTful service,
  2. Supporting more input and output formats and services,
  3. Support use of persistent identifiers for all references to geospatial and attribute data and
  4. To define how missing keys or mismatched keys are to be treated.

References

- Yao, X., Li, G., 2018. Big spatial vector data management: a review. Big Earth Data 2, 108–129. https://doi.org/10.1080/20964471.2018.1432115
- Bresters, P., Nl, S., van Oirschot, H.K., Nl, S., Fokke, E., Nl, S., Venema, J., Brentjens, T., Grothe, M., van Pelt, B., Hogeboom, J., Kruse, D., 2016. Impact analysis Table Joining Service 35.
- Gaffuri, J., 2012. Toward Web Mapping with Vector Data, in: Xiao, N., Kwan, M.-P., Goodchild, M.F., Shekhar, S. (Eds.), Geographic Information Science, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 87–101. https://doi.org/10.1007/978-3-642-33024-7_7

References

- Mahmud, S.M.H., Hossin, M., Hasan, M.R., Jahan, H., Noori, S., Ahmed, Md.R., 2020. Publishing CSV Data as Linked Data on the Web. pp. 805–817. https://doi.org/10.1007/978-3-030-30577-2_72
- Grothe, M., Brentjens, T., 2013. Joining tabular and geographic data – Merits and possibilities of the Table Joining Service 53.
- EFGS and Eurostat, 2019. Automated Linking of SDMX and OGC Web Services [WWW Document]. URL https://www.globalhealthlearning.org/gheltaxonomy/term/1177 (accessed 7.11.20).

Cartography M.Sc.

Thank you for your attention.