ARBEITSGRUPPE DATENBANKEN UND INFORMATIONSSYSTEME

Spatial Databases: Project Documentation

SETUP-GUIDE AND DOCUMENTATION FOR SPATIAL-WEATHER-PROJECT

Johannes Dillmann (4476004)
Christian Wirth (4498611)
Jens Fischer (3923671)

February 28, 2015

Contents

List of Figures

1 Introduction

Topic

The topic of this project is to combine map data provided by Open Street Maps and weather data on a spatial database server running PostGIS.

Motivation

The motivation of this project can be seen from an educational as well as from a technical point of view.
The educational purpose of this project is to encourage the students to acquire domain knowledge concerning the problem that needs to solved with a system that is requested to be set up. In this particular case the domain is weather data. By working hands-on with real data the students expand their range of skills in order to be able to work with problems that exceed the boundary of purely computational and mathematical problems.
Once these domain related problems are understood, the students face the technological challenges in order to find a solution to the given problem which will be explained in the following subsection.

Goal

The goal is to find suitable sources for weather forecast and historical weather data as well as storing it on a PostGIS server. In the end the system shall be able to overlay the map and the collected weather data. In order to achieve this goal several technical problems have to be solved, like designing a data model, a suitable system architecture as well as setting up and configuring the whole system.

2 Data Sources

We are using three different data sources for administrative borders, historical and forecast weather data. Since the historical data is feature based but the forecasts are provided as regular grid, we decided to use irregular tessellation to display weather information for german states and districts and a voronoi diagram based on official german weather stations.

2.1 Deutscher Wetterdienst (DWD)

The DWD provides historical measurement data from 509 weather stations all over Germany. The data is freely available as comma-separated values (CSV) from a FTP server. Every station is identified by a unique id, has a name, a spatial location and an altitude and provides multiple daily measurements.
As mentioned above we use the stations locations to generate a voronoi tessellation covering Germany and assign each resulting cell the measurement from the associated station. To get values for districts or states, we are calculating a weighted mean based on the area a voronoi cell is contributing. Since this operation is costly we did apply some optimisations as described in section 5.2.
The tool used to download and import DWD data is based on work of a fellow student working on the same topic (cf. [**website:cholin**]). Its usage is described in section 8.4.

2.2 Global Forecast System (GFS)

We use data from the Global Forecast System (GFS) in our application for weather forecasts. The GFS is a weather forecast model computed and freely distributed by the National Oceanic and Atmospheric Administration (NOAA) of the United States. The GFS model is calculated every 6h hours and covers the entire globe at a resolution of 28 kilometres for weather predictions within 16 days. Furthermore it provides forecast up to two weeks with a lower resolution of 70 kilometres.
The data is free is free of charge an can be downloaded as Gridded Binaries (GRIB) from the NOAA servers. They provide among other access methods like FTP services, a simple webinterface[1] to extract certain levels and variables for a subregion.
Because we are only interested in the temperature and rainfall forecasts for Germany, we are using that interface to download the GRIB data. Because the webinterface does not allow to download data older than one month, we provide separate download and import scripts as described in section 8.5 that can be run periodically.

2.3 OpenStreetMap (OSM)

OSM is used to get the administrative borders of Germany, german states and districts. Since we are using tiles from external providers, we didn't have to render them

[1] http://nomads.ncep.noaa.gov/txt_descriptions/grib_filter_doc.shtml

ourselves from the OSM data. We recommend to download compressed PBF files for Germany from Geofabrik and import them with imposm as described in section 8.3.

3 Data Model

3.1 Entity Relationship Diagram (ERD)

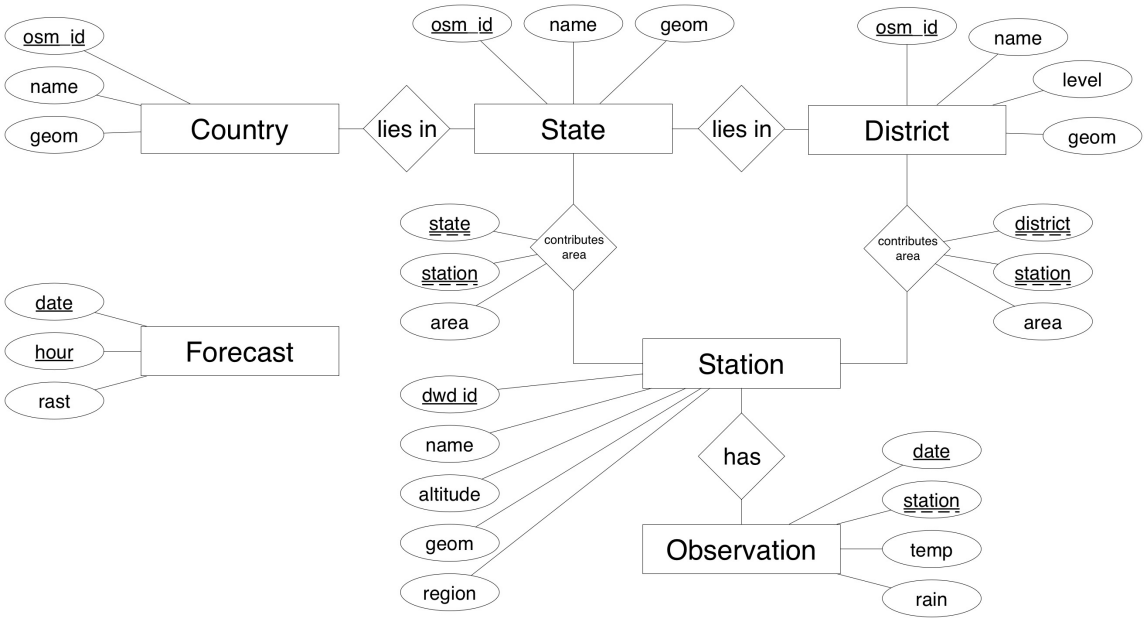

Figure 3.1: ER-Diagram

3.2 Relational Schema

Country : {[osm_id: biginteger, name: string, geom: geometry]}
State : {[osm_id: biginteger, name: string, geom: geometry]}
District : {[osm_id: biginteger, name: string, geom: geometry]}
Station : {[dwd_id: integer, name: string, altitude: integer, geometry: point, region: geometry]}
Observation : {[station_id: integer, date: timestamp, temperature: double, rainfall: double]}
Forecast : {[date: timestamp, hour: integer, rast: raster]}
ContribState : {[state_id: biginteger, station_id: integer, area: double]}
ContribDistrict : {[district_id: biginteger, station_id: integer, area: double]}

The relational schema is following [**Kemper2011**]. The schema is not identical to

the schema used in the actual PostgreSQL DB where for example additional fields used solely for the import are still remaining. The presented schema is given for illustrating conceptual ideas.

As can be seen in the relational schema there are no foreign keys or junction tables used to reference the relation between countries, states, districts and stations. Those entities are related by their spatial component and will be joined by using spatial joins provided by the PostGIS extension.

4 Architecture

The architecture as shown in fig. 4.1 consist of 4 major parts:
The data sources and their corresponding downloaders and importers, the PostGIS-Server, running in a virtual machine, the backend and a webapp. In the following four subsections 4.1 to 4.4 it will be explained how those components work together and what parts they are made of as well as which technologies have been used to implement them.

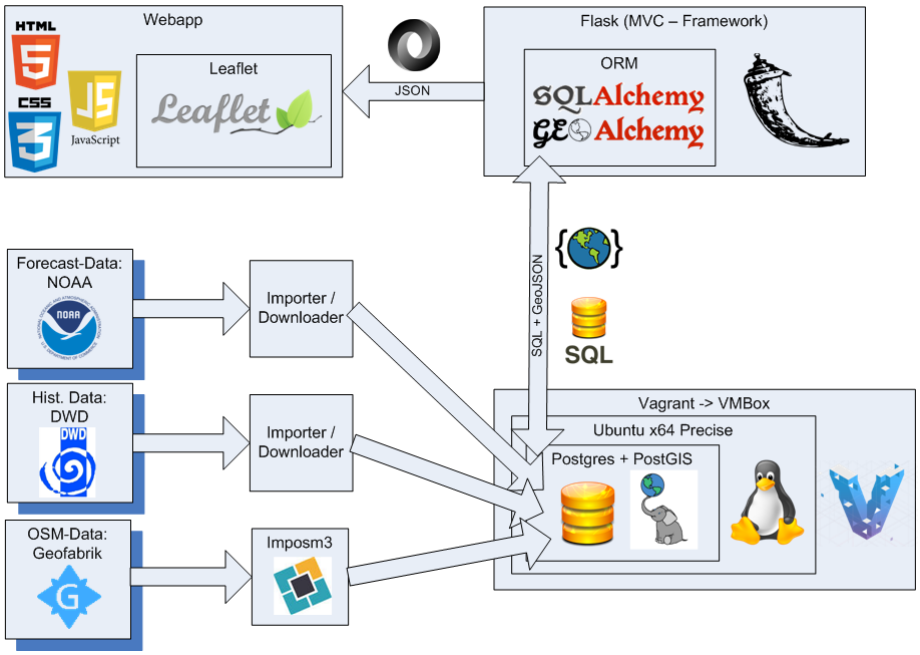

Figure 4.1: Architecture

4.1 Data Sources and Importer

The three data-sources Deutscher Wetterdienst [2], Global Forecast System [3] and OpenStreetMap [4] have been used in this project and were discussed in section 2. The importers write the collected data directly to the spatial database server so they can be used to answer the queries. For further information on this part please refer to section 2 and the setup guide at subsections 8.3 to 8.5

4.2 Database Server

The database server is provided as Vagrant virtual machine. Vagrant is a tool to automate the setup of virtual machines. In this case vagrant initializes a Ubuntu x64 instance and installs postgres and PostGIS along with other required packages and configures most of the things needed for the server to be used.

4.3 Backend

The backend runs Flask, a model-view-controller based web framework. This framework contains methods to invoke data collection and data import as well as an ORM based on SQLAlchemy and GEOAlchemy to send queries to the database to retrieve data requested by the user via the frontend. The queries are written in python, the ORM translates those queries to SQL and the the response from the database is converted to GeoJSON, which is then forwarded to the frontend.

4.4 Webapp (Frontend)

The frontend in implemented as a webapp written in HTML5, CSS3 and Javascript and provides a user interface which is described in detail in section 8.3. To display the weather and OSM data the library Leaflet is used. The frontend receives the requested data from the backend in (Geo)JSON.

5 Optimisations

5.1 Indices

Indices were used on all primary keys, all geometry columns and some frequently queried attributes, like dates. All indices were used from the beginning, so no information on the performance gains is available.

5.2 Materialised Computations

When querying the forecast information for all the states, districts or stations our first implementation used nested queries within the ORM (i.e. multiple queries

[2] http://www.dwd.de/
[3] http://www.ncdc.noaa.gov/data-access/model-data/model-datasets/global-forcast-system-gfs
[4] http://www.openstreetmap.org/

where send to the database). This queries took extremely long, several minutes in the case of districts. To improve performance, we reimplemented this query as one nested query (i.e. the nesting was done within one query). This already improved performance significantly. But we also noted that one of the nested subqueries was basically a static computation, namely the computation of the area contribution of the region of a weather station (i.e. the Voronoi cell of a weather station) to the area of a state or district. Therefore we decided to materialise this computation into two tables (`Contrib_State`, `Contrib_District`), which brought down the query time significantly.

5.3 Border Simplifications

When profiling the application further, we noticed that the huge amount of detail of the country, state and district borders seriously affected the performance, not only in terms of querying but also the sheer amount of data transmitted from the server to the client.
Initially, after the import from Open Street Maps, all the country, state and district geometry columns had a combined size of 38 MiB. In an attempt to further improve performance we wanted to simplify the geometries, especially as they were only used for querying and overlaying the respective regions, not for the rendering of the map itself.
The main problem here is that the simplification needs to preserve the topological relationships between the different polygons. PostGIS provides the function `ST_Simplify`, but this function works on an object-by-object basis. Using this to simplify the borders produces holes and overlaps between the borders. Although the name seems to indicate otherwise, `ST_SimplifyPreserveTopology` doesn't solve the problem (it only tries to preserve topologic relationships of multilines and multipolygons).
The way to achieve simplification and preserve topological relationships is to use the topology feature of Postgis. This means to create a topology, add a layer for the borders, populate the topology from the polygons, simplify the borders within the topology and convert the borders back to polygons. This method was adapted from [**website:strks-blog**]. The complete Script can be found in the Appendix (section 9.2). Figure 5.1, 5.2, 5.3 and 5.4 compare the effects of the simplification. We aimed at achieving meaningful reduction in size but still maintain the basic characteristics of the borders. As one can see at fig. 5.1 and 5.2, at the smallest level the differences in detail are clearly notable. When looking at a larger zoom level, as in fig. 5.3 and 5.4, one can hardly spot the difference. To total combined size of the geometry columns after the simplification was 895 KiB! A further optimisation could be to provide different levels of detail for different zoom levels.

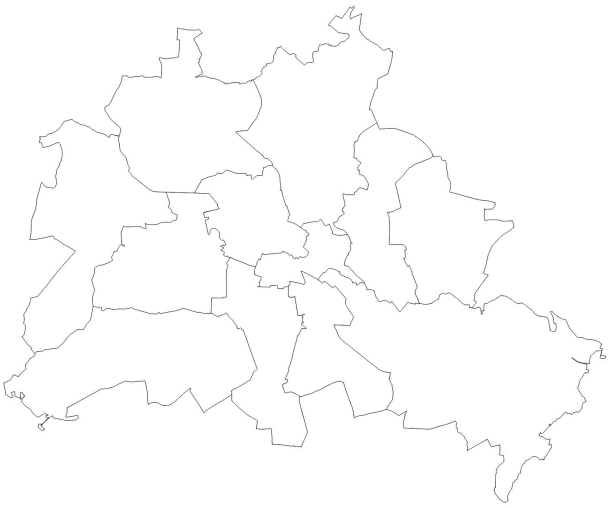

Figure 5.1: Berlin Full Detail

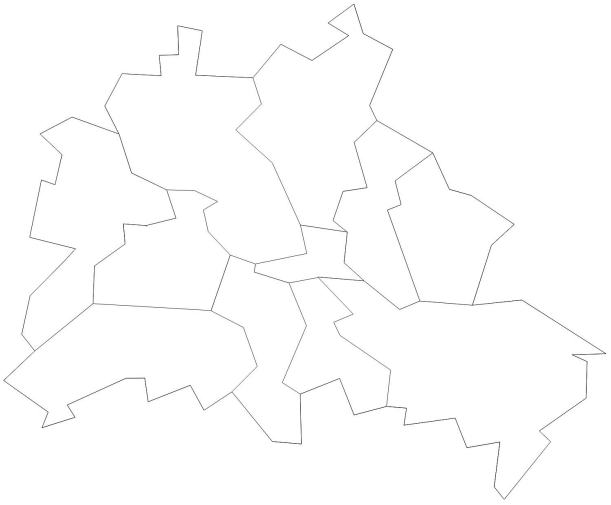

Figure 5.2: Berlin Simplified

Figure 5.3: Germany Full Detail

Figure 5.4: Germany Simplified

Figure 5.5 summarises the performance gains for the most intensive query we perform. As you can see, the simplification achieved significant performance improvements.

forecast for all districts query				
Original		Simplified		
Time / Sec	For	Time / Sec	For	
1.28	query	0.28	query	
1.59	building dict	0.30	building dict	
3.40	jsonification	0.75	jsonification	
6.27	Sum	1.33	Sum	471,43 %

Figure 5.5: Performance Gains From the Simplification

6 Usage

As mentioned in the architecture section, the weather map is displayed with HTML5 and controlled with JavaScript. It has been developed as a control for the Leaflet library and tested on Google Chrome and Chromium. Other browsers are not officially supported.

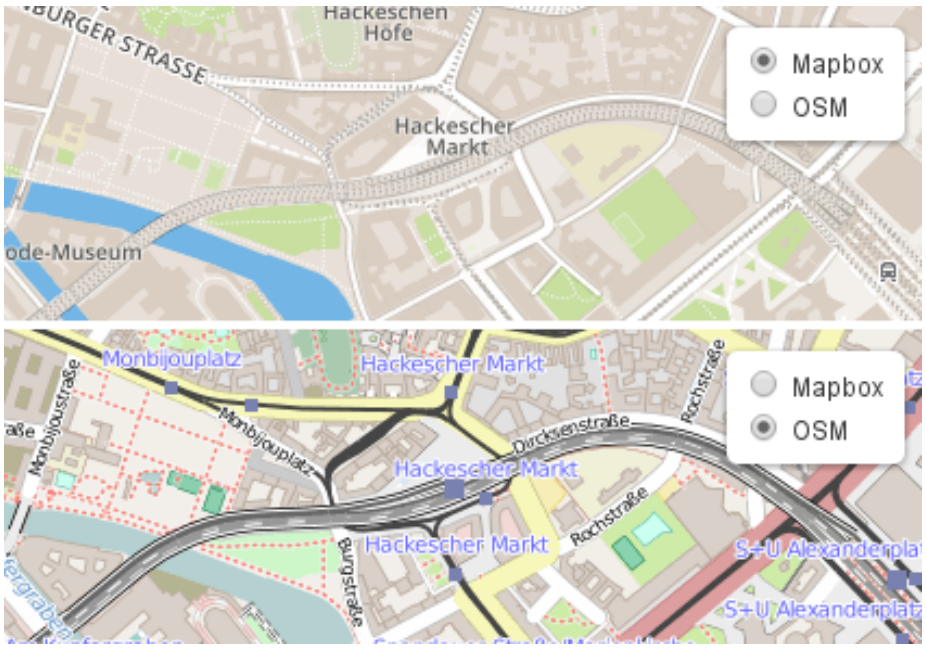

Figure 6.1: Layer selection

By default Leaflet is using a zoom and a layer selection control. Instead of rendering the base tiles ourselves, we are using the freely available OSM Tile Server [5] and tiles generated by Mapbox [6]. The desired basetile layer can be selected as seen in fig. 6.1. The map can also be panned and zoomed by using a pointing device (e.g. mouse). The displayed weather data can be set by an additional control as seen in fig. 6.2. It allows to choose either temperatures or reciprocal rainfall for a certain day. The day can be selected with an interactive date-time picker as seen in fig. 6.3.

[5] http://wiki.openstreetmap.org/wiki/Tile_usage_policy
[6] https://www.mapbox.com/

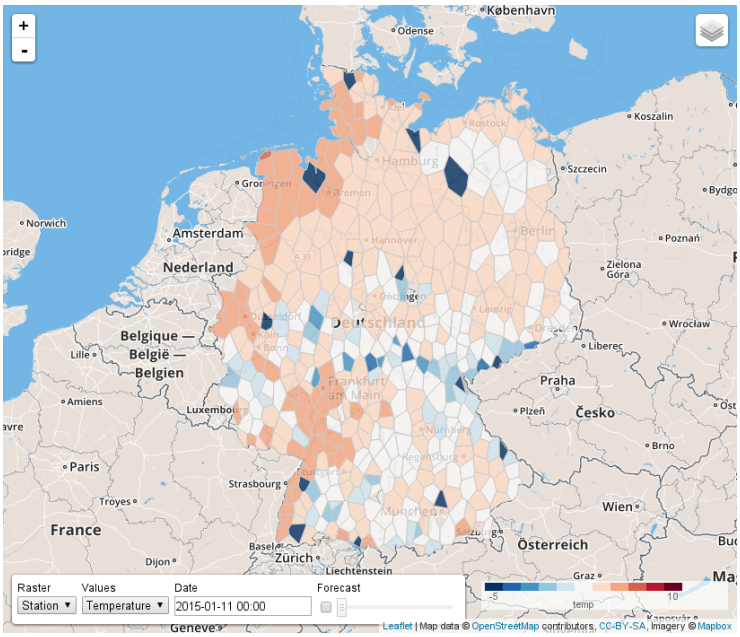

Figure 6.2: Weather control

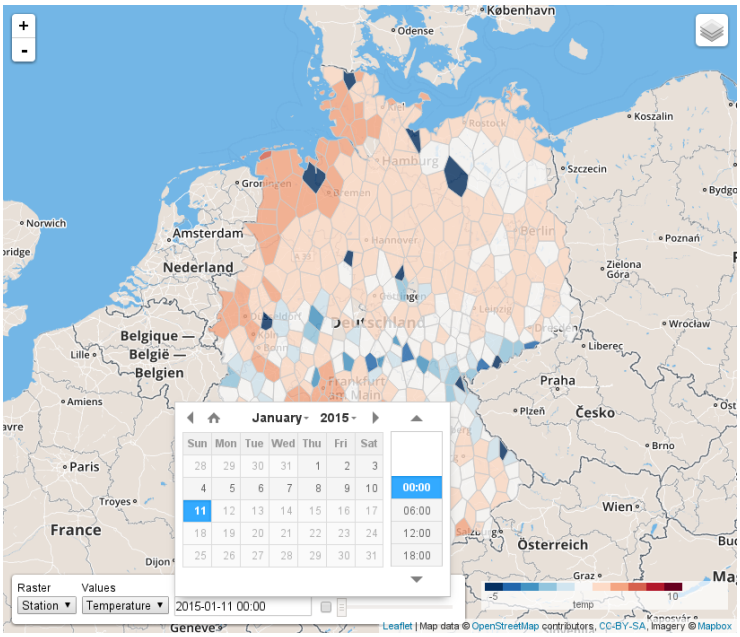

Figure 6.3: Date-time picker

The selected date and time is also used to pick the computed GFS used to display forecasts. If the forecast box is checked, the forecasts slider is activated and allows to select point of forecast in hours, starting from the chosen date and time.
As mentioned before, the data can be displayed for different rasters: a voronoi tessellation based on official weather stations, german states or districts as shown in fig. 6.4. The desired raster can be selected with the control. Once the selection did change, the weather control is loading the matching data from the backend via a JSON interface and displays the raster on the map.

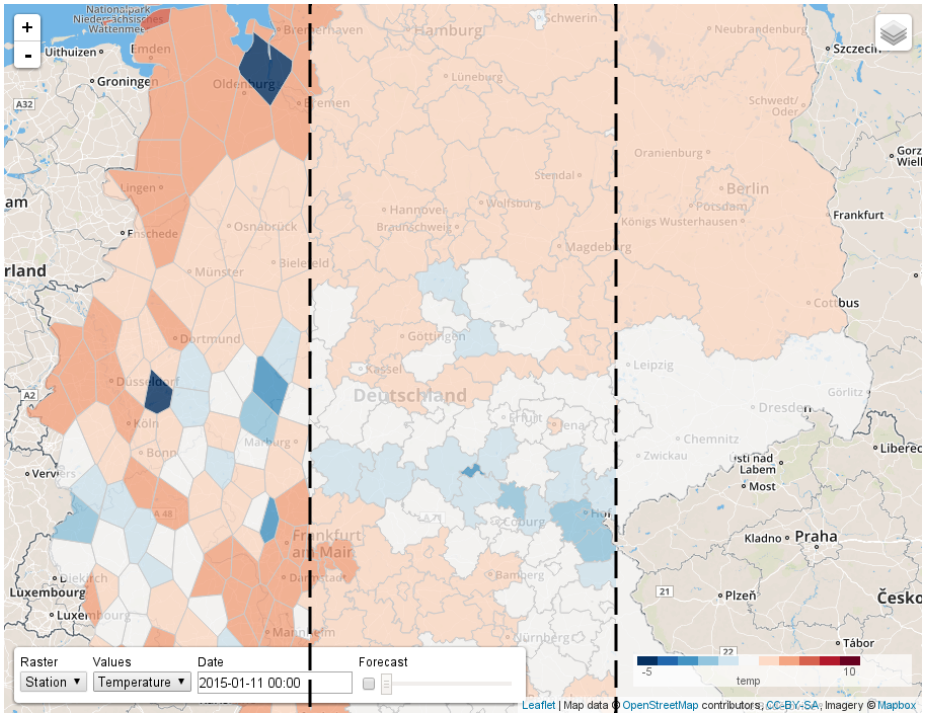

Figure 6.4: Raster comparison

The cells are coloured according to selected data and a legend is shown on the lower right for reference (see fig. 6.5 and 6.6)

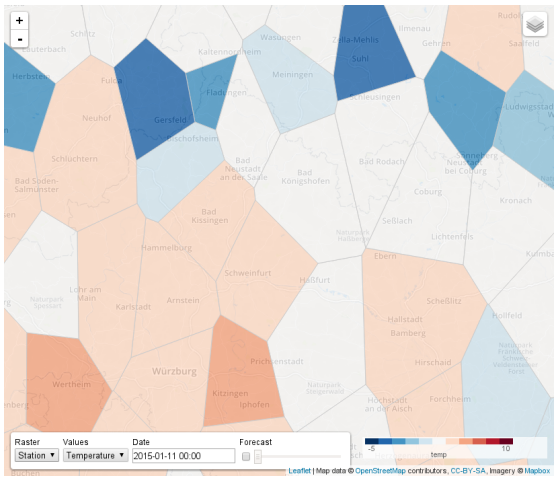

Figure 6.5: Temperature legend

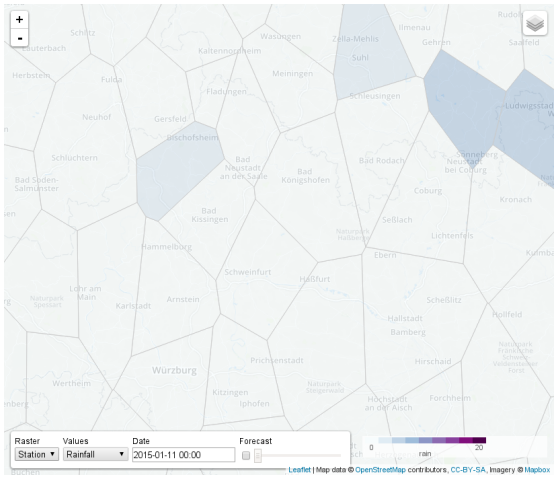

Figure 6.6: Rainfall legend

Each cell is clickable and highlights it's border or, in case of the voronoi tessellation, the position of the weather station. Furthermore a pop-up as in fig. 6.7 is shown, which provides additional data associated to the selected cell by calling the back-end's JSON interface.

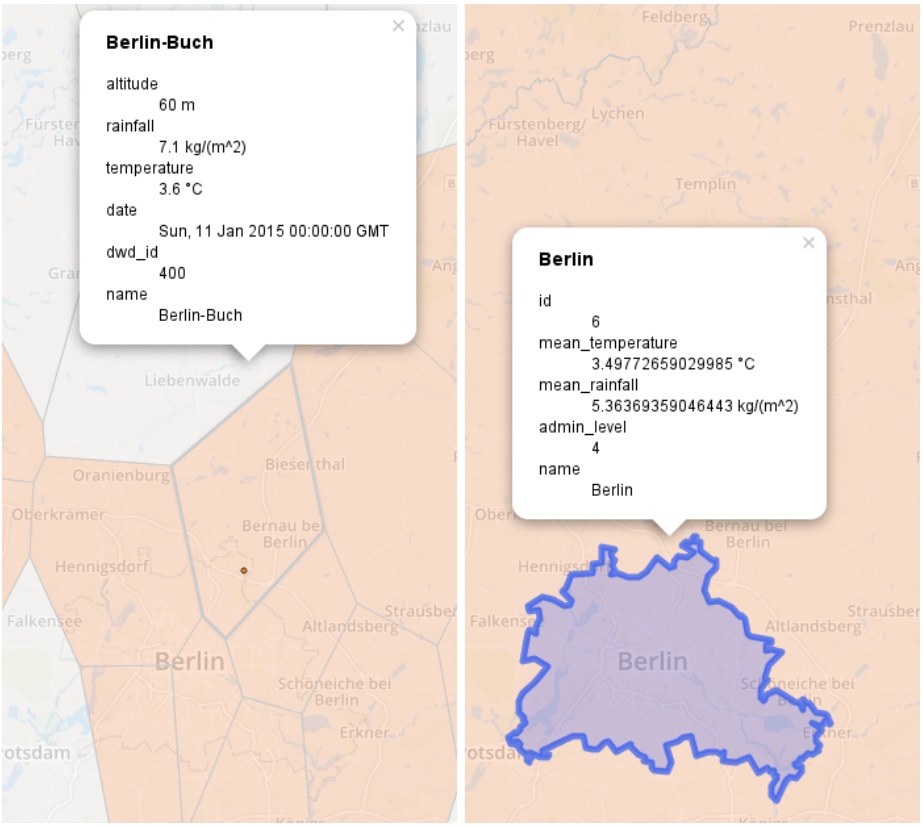

Figure 6.7: Pop-up

Currently this is just the alphanumerical attributes, but could be used to show automatically generated temperature timelines or recent webcam photos.
For future development it would be great to add features that allow a comparison and evaluation of forecast- and historical data. This could be achieved, by overlaying the forecast-data with different offsets for a particular point in time with the historical data. The resulting graphical representation could then allow an evaluation of the correctness of forecasts.

7 Conclusion

In this section we're discussing the lessons learned during this project.

- Bringing together different data-sources in one data base – especially if they include spatial references – can be quite a challenge and one has to consider more aspects than it seems right from the start. This is not only a technical problem, but one need to understand the underlying meaning of the data and the format it's stored in, to put the data together in one database and display it accordingly. Without this knowledge about the data, it would have not been possible to display it in a meaningful way.

- Creating a suitable data model at an early stage is as important as it is difficult, as long as one has very little knowledge about the data to be processed. By creating this data one develops a better understanding of the information to be delivered to the user. Even if the final technical representation might differ from the early concept, it helps keeping track of goals we want to achieve.

- Virtual Machines, especially when set up with Vagrant or other provisioning-tools, help a lot with allowing other programmers to easily reproduce the steps of the setup-process. The work put in an automated setup-routine can drastically reduce the complexity of a setup-guide, as many steps are bundled into one and therefore the provisioning-scripts act as documentation themselves. These scripts can then be examined for a deeper understanding, while still allowing to just use them, if the reader simply wishes for a working system, without going through all the pain of the manual setup.

- When dealing with spatial databases a lot can go wrong, if you don't know what you're doing. Once we got to the point where we had it up and running, we realized how slow queries on bigger data-sets are and discovered the power of system- and query optimization we're discussing in section 5.

8 Setup Guide

8.1 Prerequisites

This Setup-Guide has been written for and tested with Ubuntu 14.10 'Utopic Unicorn'.
The Following Packages or Programs need to be installed before you proceed with this setup-guide: Git, Vagrant, Python 3, PostGIS and the postgres database drive for Python.

```
sudo apt-get install gdal-bin postgis git vagrant
   python3 python3-dev python3-psycopg2 libpq-dev
```

8.2 Step-by-step Setup

Switch to directory that should later contain your project-source-code and clone Repository by typing into console:

```
git clone https://github.com/jvf/spatial-weather.git
```

Change directory to spatial-weather

```
cd spatial-weather
```

Install and configure the virtual machine by entering in console:

```
vagrant up
```

You can open a new console and proceed with the following steps on your local machine:
Install miniconda from the website: http://conda.pydata.org/miniconda.html and create a virtual environment with the following command:

```
conda create -n spatial-weather --file requirements.
    conda
```

The virtual environment can be activated by executing the following command. Remember that the environment has to be active to start the server successfully.

```
source activate spatial-weather
```

The first time the environment is activated it is required to install additional requirements that are not available within the conda distribution. The installation is handled by pip and can be called with the following command.

```
pip install -r requirements.txt
```

The weather app requires a database with the PostGIS and `postgis_topology` extensions enabled. The following command can be used to create the default database with the required access levels within the previously provisioned virtual machine.

```
vagrant ssh -c "sudo -u postgres psql -c \"CREATE
    DATABASE spatial OWNER myapp LC_COLLATE 'en_US.UTF
    -8' LC_CTYPE 'en_US.UTF-8';\""
vagrant ssh -c "sudo -u postgres psql -d spatial -c \"
    CREATE EXTENSION postgis; CREATE EXTENSION
    postgis_topology;\""
vagrant ssh -c "sudo -u postgres psql -d spatial -c \"
    GRANT ALL ON DATABASE spatial TO myapp; ALTER
    DATABASE spatial OWNER TO myapp; ALTER TABLE
    topology OWNER to myapp; ALTER TABLE layer OWNER to
     myapp; ALTER SCHEMA topology OWNER TO myapp;\""
```

8.3 Import OSM Data

Imports the OSM data for Germany. The following tables are created: Country, State, District and Cities. The data is imported from `germany-latest.osm.pbf`, which can be obtained from Geofabrik [7]. The pbf file needs to be placed in `data/` located in your Project-Folder.
The importer used is `imposm3`, which is included in the vagrant setup (and will be executed within the VM). The import uses a custom mapping, which is provided in `importer/mapping.json`. The import is conducted in two steps: First, the data is imported to the tables Osm_Admin and Osm_Places. Second the Country, State, District and Cities tables are created from these tables.

Usage

Prerequisite: No contrib tables, delete if existing:

```
python manage.py drop_tables -t contrib
```

To import all the OSM data use the manage script. To invoke the whole pipeline use the following command (will take several hours):

```
python manage.py import_osm --imposm --simplify --
    load --drop_tables
```

- `--imposm` the first import step (see above)
- `--simplify` simplify all map data (borders)
- `--load` the second import step (see above)
- `--drop-tables` deletes the Osm_Admin and Osm_Places tables after a successful import

All the above steps can be invoked separately.

8.4 Import DWD Data (Historical Weather Data)

Imports weather observation data from the DWD (Deutscher Wetterdienst). The importer is a adopted version from [**website:cholin**] . Per default, it downloads all the recent daily observations [8]. Details on the importer from [**website:cholin**]:

> The importer downloads the station summary file to get a list of all weather stations. After that it downloads for each station the corresponding zip file (with measurement data), extracts it in-memory and parses it. To get information about which weather station is the nearest for a given point, it also calculates a region polygon for each station. This is done by computing the voronoi diagram for all stations. The resulting regions may be outside of the country germany. To avoid this there is a polygon

[7] http://download.geofabrik.de/europe/germany-latest.osm.pbf
[8] ftp://ftp.dwd.de/pub/CDC/observations_germany/climate/daily/kl

of the border of Germany (data is from naturalearthdata.com - country extraction and exportation as geojson with qgis). For each region we calculate the intersection with this polygon and use the result as final region (Multi)Polygon.

Usage

Use the importer with the `manage.py` script:
To download all data and import all observation data:

```
python manage.py import_dwd
```

Create an intermediate result in `data/weather.json`:

```
python manage.py import_dwd --to_json
```

Import the intermediate result from `data/weather.json`:

```
python manage.py import_dwd --from_json
```

8.5 Download and Import NOAA GFS Data (Forecasts)

Importing the Forecast Data is done in two steps. First you have to download the GRIB files from the NOAA FTP servers. Then you have to import them as PostGIS Raster.

Download

For downloading the GFS data a date range and a target directory has to be specified. The format for the start and enddate is `YYYYMMDDHH` or `latest` for the most recently available GFS calculation. Optionally the forecast hours can be specified as a range (Defaults to download from 0 to 129 in 3 hour steps).

```
usage: run_gfs.py download [-h]
  [--hours_start HOURS_START]
    [--hours_stop HOURS_STOP]
    [--hours_step HOURS_STEP]
    [startdate] [enddate] datadir
```

For example, assuming data should be stored to `data/forecasts`:

```
./run_gfs.py download 2014121112 2015011306 data/
    forecasts
```

Import

To import the downloaded data, the download directory and a data range has to be specified:

```
usage: run_gfs.py import [-h] datadir [startdate]
  [enddate]
```

For example, assuming the data is stored in `data/forecasts`:

```
./run_gfs.py import data/forecasts 2014121112
   2015011306
```

Build Contrib Tables

To speed up some queries, the area contribution of the region (voronoi cell) of weather stations to states and districts is precomputed and materialized. Run

```
python manage.py calculate_contrib_area
```

to create and fill the ContribState and ContribDistrict tables.

8.6 Start Webapp-Server

Remember to set the virtual environment first:

```
source activate spatial-weather
```

Then you can run the Webserver:

```
python manage.py runserver
```

9 Appendix

9.1 Link to Repository

All the source code can be found at Github [9].

9.2 Border Simplification Script

```
1  -- Delete all unneeded admin levels
2  DELETE FROM osm_admin
3    WHERE admin_level != 2 AND
4      admin_level != 4 AND
5      admin_level != 6 AND
6      admin_level != 9
7  ;
8
9  -- Delete all unneeded rows with admin level 9 (keep
       only rows of admin level 9 contained in the states
       hamburg and berlin)
10 WITH
11 berlin AS
12 (
13     SELECT geometry
14     FROM osm_admin
15     WHERE admin_level = 4 AND name = 'Berlin'
16 ),
17 hamburg AS
18 (
19     SELECT geometry
20     FROM osm_admin
21     WHERE admin_level = 4 AND name = 'Hamburg'
22 ),
23 quarter AS
24 (
25     SELECT a.id, a.osm_id, a.name, a.type, a.
           admin_level, a.population, a.geometry
26     FROM osm_admin a, berlin b, hamburg h
27     WHERE a.admin_level = 9 AND ST_Contains(ST_Union(b
           .geometry, h.geometry), a.geometry)
28 )
29
30 DELETE FROM osm_admin
31 WHERE admin_level = 9 AND id NOT IN (SELECT id FROM
       quarter);
```

[9] https://github.com/jvf/spatial-weather

```
32
33 -- Change Projection
34 ALTER TABLE osm_admin ALTER COLUMN geometry TYPE
       geometry(Geometry);
35 UPDATE osm_admin SET geometry = ST_Transform(geometry,
       4326);
36 ALTER TABLE osm_admin ALTER COLUMN geometry TYPE
       geometry(Geometry, 4326);
37
38 -- Install SimplifyEdgeGeom function
39 CREATE OR REPLACE FUNCTION SimplifyEdgeGeom(atopo
       varchar, anedge int, maxtolerance float8)
40 RETURNS float8 AS $$
41 DECLARE
42   tol float8;
43   sql varchar;
44 BEGIN
45   tol := maxtolerance;
46   LOOP
47     sql := 'SELECT topology.ST_ChangeEdgeGeom(' ||
           quote_literal(atopo) || ', ' || anedge
48       || ', ST_Simplify(geom, ' || tol || ')) FROM '
49       || quote_ident(atopo) || '.edge WHERE edge_id =
           ' || anedge;
50     BEGIN
51       RAISE DEBUG 'Running %', sql;
52       EXECUTE sql;
53       RETURN tol;
54     EXCEPTION
55      WHEN OTHERS THEN
56       RAISE WARNING 'Simplification of edge % with
           tolerance % failed: %', anedge, tol, SQLERRM;
57       tol := round( (tol/2.0) * 1e8 ) / 1e8; -- round
           to get to zero quicker
58       IF tol = 0 THEN RAISE EXCEPTION '%', SQLERRM;
           END IF;
59     END;
60   END LOOP;
61 END
62 $$ LANGUAGE 'plpgsql' STABLE STRICT;
63
64 -- Create a topology
65 SELECT topology.CreateTopology('osm_admin_topo',
       find_srid('public', 'osm_admin', 'geometry'));
66
67 -- Add a layer
```

```
68 SELECT AddTopoGeometryColumn('osm_admin_topo', 'public
       ', 'osm_admin', 'topogeom', 'MULTIPOLYGON');
69
70 -- Populate the layer and the topology
71 UPDATE osm_admin SET topogeom = toTopoGeom(geometry, '
       osm_admin_topo', 1);
72
73 -- Simplify all edges up to 0.01 units
74 SELECT SimplifyEdgeGeom('osm_admin_topo', edge_id,
       0.01) FROM osm_admin_topo.edge;
75
76 -- Convert the TopoGeometries to Geometries for
       visualization
77 ALTER TABLE osm_admin ADD geomfull Geometry(Geometry,
       4326);
78
79 UPDATE osm_admin
80    SET geomfull = geometry,
81        geometry = topogeom::geometry;
```

9.3 Troubleshooting (Mac OS X)

UnicodeEncodeError

Python inherits the standard locale from the current shell environment. If this is not set to utf8 it tries to convert to ASCII, which produces.
`UnicodeEncodeError: 'ascii' codec can't encode character`
Test with `$ locale`, this should show utf-8. If not, fix with

```
export LANG=en_US.UTF-8
export LC_ALL=en_US.UTF-8
```

libssl / libcrypto Error from psycopq

The `libssl` version Mac OS X uses might be too old for `psycopg`, resulting in an error like the following:

```
...
ImportError: dlopen(...lib/python3.4/site-packages/
    psycopg2/_psycopg.so, 2): Library not loaded:
    libssl.1.0.0.dylib
  Referenced from: ...lib/python3.4/site-packages/
      psycopg2/_psycopg.so
  Reason: image not found
```

This can be solved by changing the dynamic shared library install names in the psycopq binary (cf. [**website:superuser**]). First, find out the version psycopq is using:

```
otool -L /Users/jvf/miniconda3/envs/env-sw/lib/python3
    .4/site-packages/psycopg2/_psycopg.so
$ /Users/jvf/miniconda3/envs/env-sw/lib/python3.4/site
    -packages/psycopg2/_psycopg.so:
     /usr/local/lib/libpq.5.dylib (compatibility
         version 5.0.0, current version 5.6.0)
     libssl.1.0.0.dylib (compatibility version 1.0.0,
         current version 1.0.0)
     libcrypto.1.0.0.dylib (compatibility version
         1.0.0, current version 1.0.0)
     /usr/lib/libSystem.B.dylib (compatibility version
         1.0.0, current version 1213.0.0)
     /usr/lib/libgcc_s.1.dylib (compatibility version
         1.0.0, current version 283.0.0)
```

Now, change the the shared libraries for `libssl` and `libcrypto` (using the libraries provided by Postgres.app):

```
install_name_tool -change libssl.1.0.0.dylib /
    Applications/Postgres.app/Contents/Versions/9.3/lib
    /libssl.1.0.0.dylib /Users/jvf/miniconda3/envs/env-
    sw/lib/python3.4/site-packages/psycopg2/_psycopg.so
```

```
install_name_tool -change libcrypto.1.0.0.dylib /
    Applications/Postgres.app/Contents/Versions/9.3/lib
    /libcrypto.1.0.0.dylib /Users/jvf/miniconda3/envs/
    env-sw/lib/python3.4/site-packages/psycopg2/
    _psycopg.so
```

psycopq now uses the correct libraries:

```
otool -L /Users/jvf/miniconda3/envs/env-sw/lib/python3
    .4/site-packages/psycopg2/_psycopg.so
$ /Users/jvf/miniconda3/envs/env-sw/lib/python3.4/site
    -packages/psycopg2/_psycopg.so:
     /usr/local/lib/libpq.5.dylib (compatibility
         version 5.0.0, current version 5.6.0)
     /Applications/Postgres.app/Contents/Versions/9.3/
         lib/libssl.1.0.0.dylib (compatibility version
         1.0.0, current version 1.0.0)
     /Applications/Postgres.app/Contents/Versions/9.3/
         lib/libcrypto.1.0.0.dylib (compatibility
         version 1.0.0, current version 1.0.0)
     /usr/lib/libSystem.B.dylib (compatibility version
         1.0.0, current version 1213.0.0)
     /usr/lib/libgcc_s.1.dylib (compatibility version
         1.0.0, current version 283.0.0)
```

It is strongly recommended to do all this in an virtual environment to not mess up your system!