IFS INSTITUTE FOR SOFTWARE

OST Ostschweizer Fachhochschule

Analysis Platform for OpenStreetMap

Department of Computer Science
OST - University of Applied Science
Campus Rapperswil-Jona

Student Research Project
Autumn Term 2020

Authors: Marc Scherrer & Philipp Bolliger
Advisor: Prof. Stefan Keller
Project Partner: Institute for Software
Version: 1.0-62-gb6172e1

This document was typeset using LaTeX
December 17, 2020

Abstract

Big Data analysis and especially geospatial data analysis can reveal useful information about geographical phenomena and its own composition. One of the most commonly used source in open source data science is OpenStreetMap. There are numerous community tools to handle OpenStreetMap (OSM) data and to use it for an analytical approach. The EOSMDBOne offers up-to-date OSM data from Switzerland and Liechtenstein. It uses tools that are no longer maintained and has a sub optimal schema. The purpose of this project, is to create a newer version of EOSMDBOne with an additional GraphQL API and a SQL web frontend application.

First the domain of GIS, OSM and some OSM tools were researched and evaluated. Subsequently a prototype was created for the core functionality. Then the GraphQL API was created and tuned to work with the database. For the web application a design concept in the form of wireframes and a proof of concept was created. The remaining time was dedicated to increase the performance of the database itself. The developement of the SQL web frontend application was continued outside of the project scope.

The resulting solution consists of a database with up-to-date OSM data called EOSMDBTwo and a GraphQL API with a custom function, which allow users to make some case specific queries.

The EOSMDBTwo in conjunction with the SQL web frontend application is ready to replace the EOSMDBOne, although for reliability purposes, a query limit should be added. The GraphQL interface offers a template for custom functions to add in the future.

Abstract in German

Big Data Analyse, und vor allem raumbezogene Datenanalyse kann nützliche Informationen über geografische Phänomene und die eigene Datenkomposition liefern. Eine der meistbenutzten Datensourcen in der open source Datenwissenschaft ist OpenStreetMap. Es gibt viele Community-Werkzeuge welche das Nutzen von OSM Daten in einem analytischen Ansatz vereinfachen. Die EOSMDBOne bietet aktuelle OSM Daten von der Schweiz und Liechtenstein. Sie allerdings benutzt Werkzeuge, die nicht mehr gepflegt werden und hat ein suboptimales Datenschema. Das Ziel dieses Projektes ist es, eine neue Version der EOSMDBOne zu erstellen, mit einer zusätzlichen GraphQL Application Programming Interface (API) Anbindung und einer SQL Web Applikation.

In einem ersten Schritt wurde der Bereich von GIS, OSM und einigen OSM Werkzeugen recherchiert und ausgewertet. Anschliessend wurde ein Prototyp mit der Kernfunktionalität erstellt. Danach wurde die GraphQL API erstellt und so eingerichtet, dass sie mit der Datenbank kommuniziert. Für die SQL Web Applikation wurden wireframes und eine kleine Applikation als Proof of Concept erstellt. Die verbleibende Zeit wurde für die Steigerung der Performanz der Datenbank investiert. Die Entwicklung der SQL Web Applikation wurde ausserhalb des Projektumfangs weitergeführt.

Die resultierende Lösung besteht aus einer Datenbank namens EOSMDBTwo, welche aktuelle OSM Daten enthält und eine GraphQL API bereit stellt, welche eine benutzerdefinierte Funktionen enthält, die es Benutzern erlaubt fallspezifische Queries auszuführen.

Die EOSMDBTwo ist zusammen mit der SQL Web Applikation bereit, die EOSMDBOne zu ersetzen, obwohl es für die Zuverlässigkeit noch sinnvoll wäre, eine Query-Limitation hinzuzufügen. Das GraphQL Interface bietet eine Vorlage für benutzerdefinierte Funktionen, die in der Zukunft hinzugefügt werden können.

Management summary

Situation

Big data contains a lot of statistical information which can reveal interesting insights about its domain. Geospatial data is no exception, which makes it attractive to run statistical analysis on it. OSM provides such data open source. The efficient processing of such data remains a big data problem. To accomplish that, a database which contains the data and some way to query the information desired is required.

The Extended OSM Database One (EOSMDBOne) offers a database that contains geospatial information from Switzerland and Liechtenstein and can be queried with modern SQL. However, the schema used is sub optimal for certain geospatial queries, and it uses tools that are no longer maintained. Additionally it offers an SQL Graphical User Interface (GUI) tool called PostGIS Terminal.
This is the reason why a newer version of the EOSMDBOne is necessary.

In the context of this project, the Extended OSM Database Two (EOSMDBTwo), should be created to replace the EOSMDBOne. The newer version should have a more sensible and adaptable schema. Furthermore it should allow meaningful configuration options, especially considering the data sources. To guarantee the responsiveness and prevent the EOSMDBTwo from being blocked from a single query, it should abort queries that take too long and return a useful error code. GraphQL is an explorative alternative to a RESTful API, which gained traction in the last few years, especially in the front end area. Therefore it was decided that a GraphQL endpoint should be available to provide an alternative to query the database. For educational purposes, demonstrations and exploration of gepgraphical analysis, a SQL GUI tool that allows users to query and visualize geospatial data on a map, should also be available. This was called OSM SQL Terminal.

There are several solutions that provide similar functionality. Googles Big Query Geographic Information System (GIS) is highly performant and provides functionality to query geospatial data from the whole world. However keeping the data up to date is the users concern, and the solution is quite expensive.
Overpass(-turbo) operates on up-to-date OSM data. The query language of Overpass is proprietary. It is not widely used, has a limit on complexity and also has poor performance.

The first few weeks of this project were dedicated to gain information on standards, tools and other projects. With the gained knowledge, a first, simple version of EOSMDBTwo could be created. The next step was to create the GraphQL endpoint, which is connected to the new

EOSMDBTwo. Once the GraphQL endpoint was up and running, the EOSMDBTwo schema was improved to meet the needs of analytical applications. At this point only very little time was left to create the OSM SQL Terminal and improve the performance and maintainability of EOSMDBTwo. This is why it was decided to only create a simple proof of concept and some wireframes for the GUI application. The rest of the time was dedicated to improve performance and maintainability of the EOSMDBTwo. The development of the OSM SQL Terminal was continued outside of the context of this project.

The possibility that none of the community tools provide the functionality that is needed for this project, was a constant risk during the development. The occurrence of this risk would mean, that an own solution for the problem had to be implemented. This would exceed the defined scope of the project. Since both project members have very little experience with OSM and its community tools, the risk of misjudging the time needed to implement certain features could easily cause a deviation from the project plan.

The project members consist of Philipp Bolliger and Marc Scherrer. Professor Stefan Keller supervised the whole project, helped with important decisions and provided information about OSM and its community tools. Nicola Jordan was available for questions and helped out with the setup of continuous integration and continuous deployment. Joël Schwab continued the development of the GUI application and was the first pilot user of the EOSMDBTwo. The EOSMDBOnes source code was available to the project team as reference.

Results

The resulting systems can easily be deployed on any server via docker. A first version is deployed on a server provided by the Institute for Software (IFS). The GraphQL endpoint and a pgAdmin 4 instance are available. Its deployment is explained in figure 0.1.

The configuration of EOSMDBTwo can be done by simply changing a JSON file. The EOSMDBTwo contains functionality for configuring the interval in which the data should be updated as well as country extract and update sources. A program initializes the database by downloading the data provided by the country sources and keeps the data up to date by downloading the update files from the update sources. For each country a initial source and an update source needs to be configured. Through experience with the EOSMDBOne, a revised database schema was created. The database can be queried with modern SQL.To improve query execution times, indices were created and the database configuration was optimized for OSM data. An example query output is seen in figure 0.2.

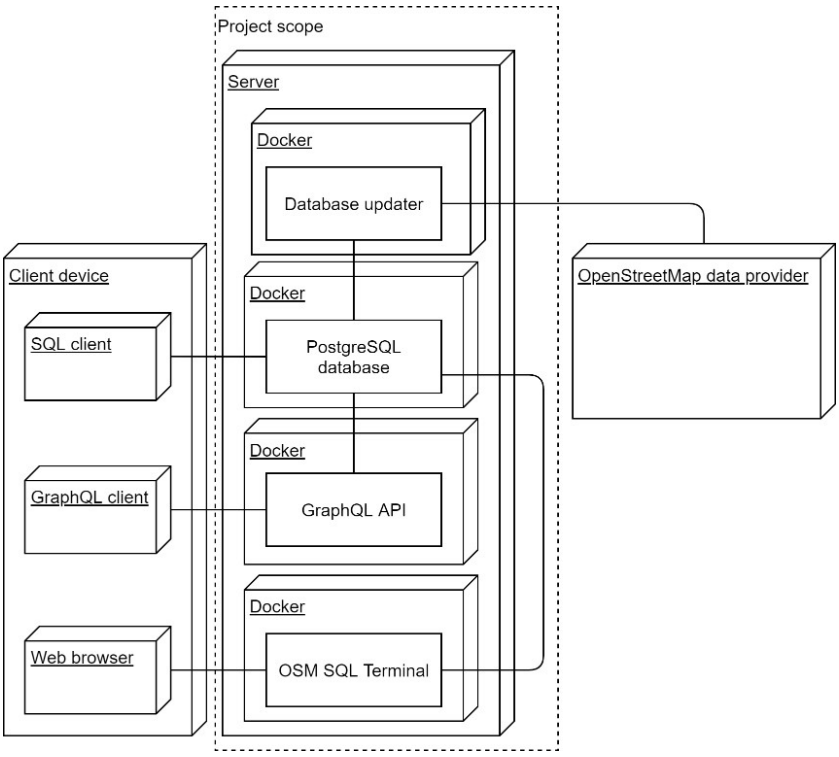


Figure 0.1.: Deployment Diagram

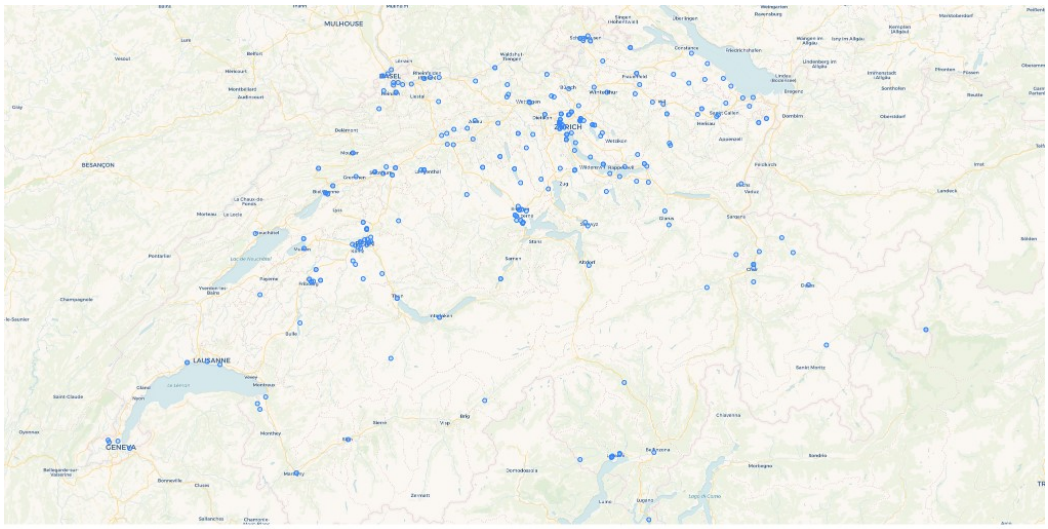


Figure 0.2.: All shops within 20 meters of a fuel station

Additionally a GraphQL API was created, providing functionality to query the EOSMDBTwo. Since the GraphQL language is not as powerful as SQL, a custom function was provided to allow for some case specific geospatial queries. This function can be seen in figure 0.3. The function can be used as a template for further custom functions.

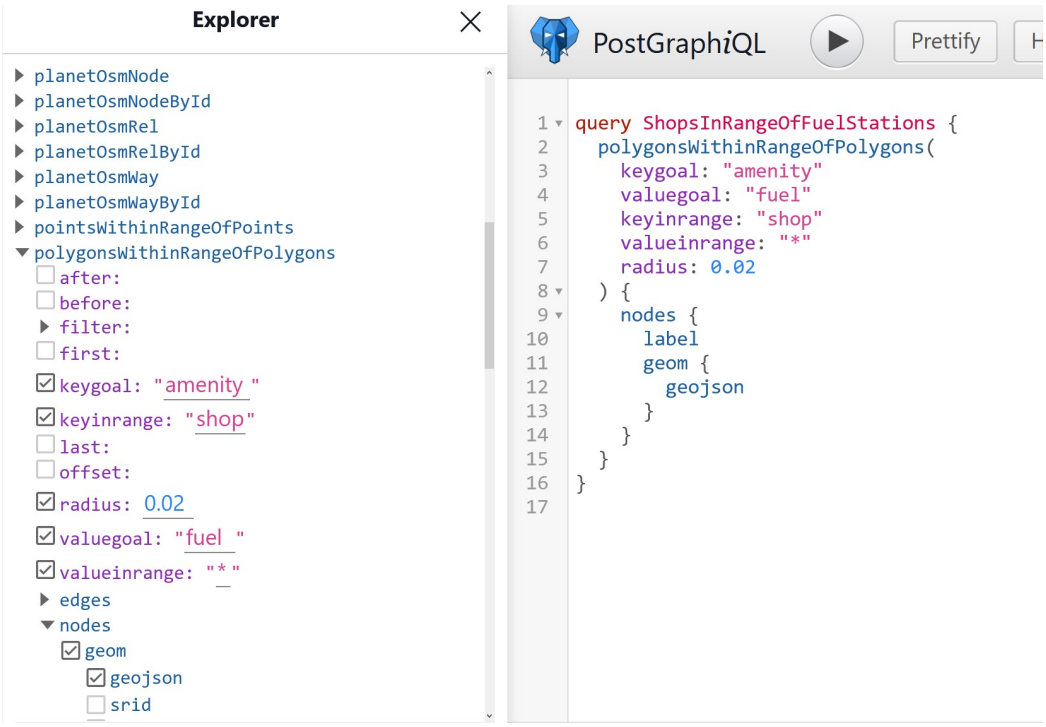


Figure 0.3.: GraphQL visualisation

The visual concept of the OSM SQL Terminal was used to continue its development.

Because of the limited time and resources, the query time is not restricted. This means, that complex queries may take several minutes or more. This could lead to an overload of the server even though only a few users are using it.
In conjunction with the GUI application, the EOSMDBTwo is ready to replace the EOSMDBOne.

Outlook

The EOSMDBTwo offers a lot of opportunities to expand. Certainly a limit on the query run time should be added in the near future.

To guarantee constant availability of the database, it would be necessary to have multiple instances running. A minimum of two instances have to be deployed. While some of them are updating their data, others are handling user requests. A load balancer could ensure that all user requests get routed to instances that are not currently updating.

Another interesting idea is to allow users to create collaborative tables that contain a specific subset of the data in the database.

It would also be possible to include historical OSM data into the database. This would allow even more analytical perspectives.

Task Definition

The Following is The task Definition, given to the team on the 13.10.2020: On the 10.11.2020, a scope change was approved, due to lack of functionality from third party tools (namely Osm2pgsql). Due to the focus on the EOSMDBTwo part of this project, the OSM SQL Terminal was removed.

An Analysis Platform for OpenStreetMap with Spatial SQL and GraphQL

- Student research project in autumn term 2020 Bachelor of Computer Science
- Authors: Philipp Bolliger and Marc Scherrer
- Supervisor: Prof. Stefan Keller, HSR Rapperswil
- Industry partners: -

Task Definition

Big Data is on everyone's minds - and with OpenStreetMap there is comprehensive, open data about the whole world. "Big Data" means that data is analyzed with complex, typically parallel loading operations and queries. This requires a high-performance software-hardware platform. OSM already has an ecosystem of services and tools in place, but they have great potential for improvement.

Here are some keywords for the required topics and activities of the project: Collect use cases and complete requirements. Evaluation of existing technologies and open source projects. Implementation of a platform for analysis of OSM data. Implementation of a web-friendly API with frontend (GraphQL). Implementing a frontend using SQL.

~~The SQL frontend "OSM SQL Terminal" has the following requirements:~~

- ~~SQL input window (is necessary with cookies to save the last query).~~
- ~~Result data in a map window and a textual (table) window.~~
- ~~Option to export the Result as GeoJSON.~~

- ~~Implementation based on experience with the existing "PostGIS Terminal", server-side limitation of read-only access, query duration or result size.~~

~~Boundary: Management of data as vector tiles and with history is only part of the software evaluation, discussion, and outlook and not part of the implementation.~~

Deliverables

1. Documentation, including text abstract (English and German), management summary (English), technical report and software engineering project (English); appendices (bibliography, content).
2. Operational platform, "Extended OSM Database Two" ("EOSMDBTwo") with two OSM databases worldwide ("Planet") and here ("Switzerland and Liechtenstein").
3. GraphQL-Frontend (English) with GraphQL API with an own, if necessary adapted, connection to EOSMDBTwo.
4. ~~A freely accessible web application "OSM SQL Terminal" (English) to run SQL queries on the EOSMDBTwo database. The results should be marked on a map.~~
5. The delivery objects required or recommended by the study course: Poster (digital only), brochure abstract, no short video.

Specifications / Conditions

- Python 3, HTML5/CSS/Javascript, and PostgreSQL/PostGIS.
- ~~Libraries: Python Flask, lightweight Javascript libraries, map library leaflet.~~
- Modern SW development (including unit testing, repositories/versioning, continuous integration, docker), where appropriate.
- Non-functional requirement (NFR): Query duration no longer than 120 seconds.

Procedure and working methods: After consultation, students choose a procedure model for software development. There are weekly meetings with prepared documents; exceptions can be agreed upon.

Documentation

For documentation (see also delivery objects above):

- The submission is to be structured in such a way that the above contents are clearly recognizable and findable (uniform numbering).

- Quotations must be marked, and the source must be indicated.
- Documents and literature used must be listed in a bibliography (do not list Wikipedia links exclusively).
- Documentation of project progress, planning etc.
- Further documents (e.g. short description/abstract, declaration of independence, terms of use) according to the requirements of the study course and in consultation with the supervisor.

Form of documentation for the supervisor (see separate instructions for the study course): Bound Report (1 copy). All documents and sources of the created software on a USB stick.

Evaluation

The usual regulations of the Computer Science study course apply, but with special emphasis on modern software development and executable and tested software, as follows:

- Project organization (weighting approx. 1/5).
- Report, structure, language (weighting approx. 1/5).
- Content: (executable) code (weighting approx. 2/5).
- Overall impression including communication with industrial partner (weighting approx. 1/5).

Other Parties Involved

- Nicola Jordan: Technical adviser
- Joël Schwab: Pilot user

Contents

Part I.

Technical Report

1. Introduction

1.1. Problem

Geospatial data is a classic example for big data and therefore contains a lot of statistical information. It is likely that a lot of correlations can be found in such a mass of data, which means, that analyzing geospatial data can reveal interesting insights on modern civilization. The efficient processing of such data remains a big data problem to this day. In order to query this data, a database is needed, which supports a query language powerful enough to do meaningful queries and is able to return results in an acceptable time. Even though there are some solutions available, there is no overall good option, which can be queried with modern SQL and is up to date. This is why EOSMDBOne was created. It provides a PostgreSQL database with geospatial data of Switzerland and Liechtenstein, which is updated every day. EOSMDBOne uses Osm2pgsql to load data into the PostgreSQL database. When EOSMDBOne was created, Osm2pgsql did not yet support a customized schema. This is why the EOSMDBOne schema is sub optimal for meaningful queries and raises the need of an updated version.

1.2. Vision

This project should create a improved version of EOSMDBOne. A database to run meaningful SQL queries on up-to-date geospatial data. It should support an easy way of deployment, in order to easily deploy the database on a powerful server. Since not all servers offer the same kind of performance, it is also desirable to have a simple way to configure which regions should be saved within the database. Users can access this database over pgAdmin 4, a web application or even a GraphQL API. GraphQL is included, because it gained a lot of support over the last few years.

1.3. Goals and Subgoals

1.3.1. EOSMDBTwo

The EOSMDBTwo is the central database. It saves up-to-date geo data and provides a sensible schema with geospatial capabilities. Queries need to deliver a response within 120 seconds. If a query takes longer than the 120 seconds it needs to be aborted and a appropriate response needs to be sent to the client.

1.3.2. GraphQL API

The GraphQL API is connected to EOSMDBTwo and allows users to query the data of EOSMDBTwo with normal GraphQL queries.

1.3.3. OSM SQL Terminal

The OSM SQL Terminal is a web application where end users can query the EOSMDBTwo with modern SQL and results are presented on a map. After the change request, the OSM SQL Terminal was removed from the project scope. Nonetheless, wireframes and a tiny prototype for it were created and Joël Schwab continued the OSM SQL Terminal concurrently to this project.

1.4. Conditions, Environment, Definitions and Restrictions

It was requested, that the project is made from scratch. The EOSMDBOne sources were available for reference.

According to the task definition, the developers used Python 3 as the programming language and PostgreSQL in combination with PostGIS for the database. The developers were free to choose the development environment.

Docker is used to enable an easy deployment of the product. The IFS provided a server with a Portainer instance to deploy the product. The veracity of OSM data is no concern of this project.

1.5. Methods and Structure

The project approach was very strongly influenced by Scrum, with 2 week sprints. In the first phase, information about OSM and tools for transformation of data were gathered. Then a first version of EOSMDBTwo was created, after which a GraphQL API was implemented. Finally the EOSMDBTwo got improved with indices and schema changes.

2. State of the Art

In this chapter, already existing tools are discussed in the context of this project. These range from fully fledged geospatial analytical database, over OSM to database tools, to frontend technologies.

2.1. Geospatial Data

Having good geographical data is a must to succeed in this field. The evaluation of which data source to use was not explicitly made, nonetheless the reasoning behind the choice will be made more clear when listing the different kind of geographical data sources.

2.1.1. Google, Apple, Bing, Etc.

The most used geographical data source for private users is definitely Google Maps. Apple and Bing also have a pretty reliable geographical data catalog. The reason why these services are not used is that they are not open source, and in big data usage scenarios they are pretty costly.

2.1.2. Admin Maps

Geographical data collected by the government would also be an option. This data is generally very reliable and complete, but not at all current. Another downside is that if you wanted to build a GIS database over many countries, you have to know every administrative geospatial data source.

2.1.3. OpenStreetMap

OpenStreetMap data is free, actual and globally available. They make no promises for completeness, but the community is expanding and tackling these problems.

2.2. Existing Geospatial Analytical Applications

There are some existing GIS applications, all with their drawbacks.

2.2.1. Google BigQuery GIS

Googles BigQuery service with the BigQuery GIS functinality provides easy geospatial calculation and loading functions. With an OSM file one could use the database easily for geospatial analysis, at the cost of BigQuerys pricing. Additionally one has to keep the database updated.

2.2.2. Overpass

Overpass(-turbo) offers a free API for analyzing geographical data, using its own query language. However there are two very big downsides of this application. The first downside is the speed. Overpass is pretty slow in contrast to a locally running database. The second downside is its own query language which does not allow modern SQL queries and therefore limits the query complexity

2.2.3. Extended OSM Database One

The EOSMDBOne is the first iteration of an analytical database used for data of Switzerland. This is used in lectures and exercises by students.

2.3. OSM Data Sources

There are some OSM data sources, that all draw originally from planet.openstreetmap.com. There are tools that can limit an extraction region from there, but it is way more performant to use proprietary sources like download.geofabrik.de or download.openstreetmap.fr and get specific countries or regions from there.

2.4. Databases

The most familiar and probably most used database for GISs is PostgreSQL with the PostGIS extension. Even Google Big Query GIS uses this, or at least the same notation. Another option would be to use a Online Analytical Processing database like Apache Spark. This would be an option for further development.

2.5. OSM Loading Tools

For loading the actual OSM data into the database, there exists multiple approaches. One approach would be to use tools like Osmium in a program and decide for every OSM object how to save it into the database. Another approach would be to use a tool like Osm2pgsql that automatically does that, and also allows to give it specific rules on how to save it.

2.5.1. Osmium

Osmium is a C++ toolkit to process OSM data.

2.5.2. Osm2pgsql

Osm2pgsql is a tool used to load OSM files into a PostgreSQL database. It is a C++ library, that is maintained quite frequently and the development team and professor Keller contacted one of the main contributors of Osm2pgsql for advice.

2.6. OSM Update Tools

For keeping the OSM data up to date, there are two ready made tools for it. One (Osmupdate) is used by EOSMDBOne and another (PyOsmium) was made specifically for working with Osm2pgsql

2.6.1. Osmupdate

Osmupdate is a commandline tool to have up-to-date OSM data. It was not worked on since 24.01.2014.

2.6.2. PyOsmium

PyOsmium is a Python tool to process OSM files of different formats. For this project the functionality to update an OSM file or create a new update file is especially interesting since the goal is to have up to date OSM data. The main contributor of this tool is coincidentally the same person as the contributor of the Osm2pgsql which we had contact with.

2.7. Database Clients

To actually use this updated data, a client for this database must be used or created.

2.7.1. GraphQL with Postgraphile

GraphQL is an explorative modern query language and Postgraphile is one implementation of this. As is explained in the chapter 3, Postgraphile has the advantage that a PostGIS extension exists and is pretty easy to use. The downside here is that in order to use PostGIS functions, custom functions have to be created, since the extension only supports PostGIS datatypes.

2.7.2. GraphQL with Hasura

Hasura is another GraphQL client, but has no easy way to customize its custom functions.

2.7.3. pg_featureserv

Pg_featureserv offers a RESTful API for a PostGIS database. It provides ressource and hypermedia control.

2.7.4. PostgREST

PostgREST is also a RESTful API for a database, but offers no PostGIS support itself. However, creating custom functions that use PostGIS should be no problem.

2.8. Frontend Technologies

To build a client with a GUI there are many technologies and libraries to consider. The evaluation only considered three contenders: Node.js, Django and Flask. All of these are web client libraries.

2.9. Standards

2.9.1. OSM

OpenStreetMap has created its own standard for abstracting geographical data into data. To abstract geospatial data, everything is separated into three categories: nodes, ways and relations.

With nodes OSM is able to represent objects that have no important size, with only the exact location is important. Also, all ways use nodes to determine their start, end and corners. Examples for this would be single trees, lanterns or house corners.

Ways are used to represent objects with a length and/or a path. Examples would be streets, trails or the trace of the outer wall of a house. A special kind of way is one that has its start and endpoint at the same location and has an area. Examples would be houses, parks etc.

Relations could possibly be anything consisting of multiple nodes or ways. This could be things like highways that consist of multiple sections, bus lines, campuses with buildings and parks in it etc.

Every OSM object also has a list of key-value pairs called tags. In these tags information like house number, building height, campus name etc. can be stored. Further information about the OSM data model is given in chapter 7.1.1.

2.9.2. PostGIS

To store these OSM objects and other types of geographical data, PostGIS has a few geometry and geography types. For nodes, PostGIS has the type point, for ways without area it has line, and for ways with area it has polygon. As for relations, it is a bit more complicated. If the relation consists of multiple nodes, PostGIS has a multipoint type, for multiple lines it is a multilinestring and for multiple polygons a multipolygon.

Further information about the PostGIS data model is given in chapter 7.1.2.

2.10. Conclusion

In this project PyOsmium in conjunction with Osm2pgsql is used for loading the OSM data and Postgraphile is used for the GraphQL API. The two RESTful clients are not used, as they are not in the scope of this project.

3. Evaluation

3.1. GraphQL API Engine Evaluation

There are several ways to add a GraphQL API to a PostgreSQL database. Common technologies and tools to achieve this are Hasura, Postgraphile, Prisma in combination with Apollo and the Node.js API Starter Kit. Most of which are very JavaScript heavy. Since the stakeholders prefer a solution without JavaScript only Hasura Core and Postgraphile are considered. In this chapter we look at some characteristics, that are important in the context of the analysis platform to evaluate which of the two is more suitable.

Both are fairly easy to set up, have a good documentation, a solid user base, create a GraphQL API based on the existing database, can be easily installed with docker and even support PostGIS functions and types.

Other than Postgraphile, which completely relies on its CLI, Hasura provides a UI for configuration. However, Postgraphile can be extended with custom plugins and reuses as much functionality from PostgreSQL as possible. This includes roles, which are useful for access control, whereas Hasura has its own role system. But the biggest difference is the hstore support. Hasura has only little support for hstore, while Postgraphile provides a plugin for basic hstore filter operations.

In the table 3.1 the most important characteristics are listed. For each characteristic a weight is defined. The weight is a number from one to ten and indicates how important the characteristic is in the context of the analysis platform. Both Hasura and Postgraphile get a score for each characteristic. The score indicates how good the API fulfills the corresponding characteristic. A score is a number from one to five. The final result in the last row equals to the sum of the product of the score and the weight of each characteristic.

Postgraphile has the higher result value and therefore is chosen to provide the GraphQL API for the analysis platform.

Characteristic	Weight	Hasura	Postgraphile
PostGIS support	10	5	5
hstore support	8	1	4
Ease of use	6	5	5
Reuses PostgreSQL	6	3	5
Extendability	5	3	5
Documentation	5	5	4
User base	3	4	2
Support	2	3	3
UI for configuraion	1	5	1
Available with docker	1	5	5
Result		158	**191**

Table 3.1.: Evaluation of GraphQL API engines

3.2. OSM Update Tool Evaluation

From the countless ways to keep OSM data up to date, the most common suggested ways are using either Osmupdate, Osmosis, or PyOsmium. Since we use Osm2pgsql, it is suggested in the manual [osma] that we use PyOsmium and specifically the 'pyosmium-get-changes' helper module. Nevertheless research about the other two options was undertaken too.

Osmosis was quickly abandoned, since it has no functionality to automatically download the corresponding update files. The effort to write the logic for that was deemed too expensive. Osmosis was rejected before the evaluation.

Osmupdate is a rather slim, easy to use tool to have OSM files continuously updated. It pretty much supports exactly our use case, with the caveat that it can either have a OSM file up to date only within graphical borders, or it can create a change file, but **not** only within graphical borders. However the biggest issue with this tool is that it was not maintained since January 2014 [osmd].

Finally PyOsmium, which is recommended by the Osm2pgsql creators. It has the same capabilities as Osmupdate, only without being able to restrict updates geographically. After contacting the main contributor of PyOsmium it was established that it would not gain the functionality to geographically constrain updates.

In the two tables 3.2 and 3.3 the most important characteristics are listed and weighted as already described in section 3.1. Both strategies (keeping an OSM file up to date or writing change files) were weighted separately to find the best possible setup.

For areas that are not countries or similar objects like cantons, PyOsmium would not suffice. Since we want to use it for countries, we can circumvent the geographical constraint using openstreetmap.fr with its minutely updates and get it from there (instead of openstreetmap.com).

Characteristic	Weight	Osmupdate	PyOsmium
Performance on large Countries	10	2	2
Future prospects	8	1	7
Performance on small Countries	6	8	8
Community	5	3	8
Result		91	154

Table 3.2.: Evaluation of OSM update tools updating a file

Characteristic	Weight	Osmupdate	PyOsmium
Performance on large Countries	10	7	8
Future prospects	8	1	7
Performance on small Countries	6	7	7
Community	5	3	8
Result		135	**218**

Table 3.3.: Evaluation of OSM update tools using a change file

3.3. Frontend Framework Evaluation

Another evaluation has to be made for the frontend framework. It is clear that JavaScript has to be used for asynchronous communication, so the only decision to make is what framework to use to communicate with the database. In discussions, three contenders for this position were made clear: Node.js, Flask and Django. As explained in the table 3.4 Flask would be the winner, although only by a very small margin. An additional difference between those three frameworks, which can not be put in numbers, would be what those framework were built for. Node.js was built for event-driven applications, which would benefit this use case, but in our experience this is a bit overkill for what is is used for in this project. Django is for fast development of web-apps, which is not really what we want. Flask was the favorite from the beginning, and was built for Web Server Gateway Interface applications.

Characteristic	Weight	Node.js	Flask	Django
Community (GitHub stars)	8	8	6	6
docker container size	7	3	9	8
Documentation	6	10	7	8
Ease of use	5	8	7	8
Templating	5	10	8	8
Testing	4	7	6	6
Functionality overview	3	5	9	7
Result		278	**279**	277

Table 3.4.: Evaluation of frontend frameworks

4. Implementaion Concept

The following steps describe how the EOSMDBTwo gets initialized and how the data is kept up to date. It is visualized in the figure 4.1

1. Determine which countries or files should be in the database

   This is done using a JSON configuration file. In the configuration file, a list of objects containing the URL of OSM files and the replication URL of their corresponding update files can be provided.

2. Download the OSM files.

   The application wget is used to download all files.

3. Combine all OSM files

   Osmium is used to combine all OSM files into one.

4. Load the OSM file into the database

   Osm2pgsql with the new flex backend is used. The flex backend allows to create a customized schema and process the data before writing it into the database.

5. Create OSM update file

   PyOsmium provides the functionality to find out which updates are missing, download them and create an update file with those changes.

6. Combine all OSM update files

   Osmium combines all OSM update files into one.

7. Loading the updates into the database

   Osm2pgsql is used again to apply the changes to the database.

The steps 5,6 and 7 are then repeated at a given interval to keep the data up to date.

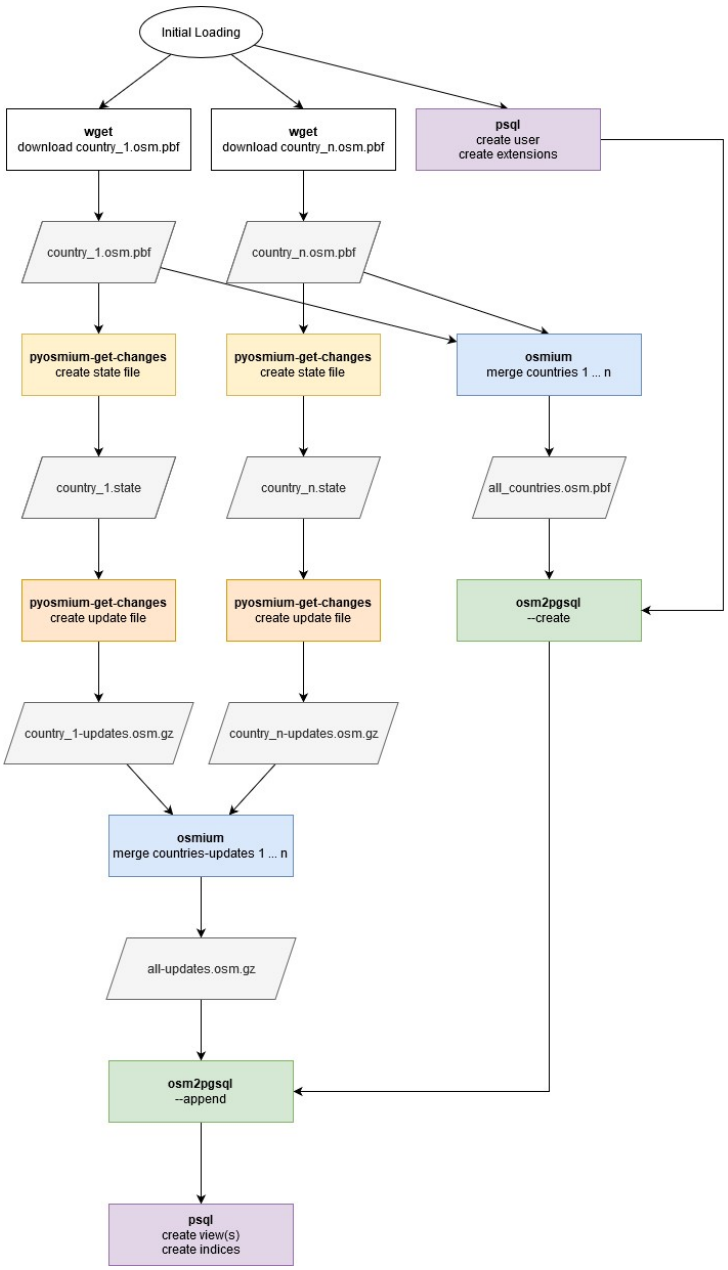


Figure 4.1.: Data Flow Diagram

5. Results

5.1. Achieved Goals

5.1.1. Continuous Updater

The continuous updater is the component which ensures, that the data in the EOSMDBTwo is up to date. Administrators have to opportunity to configure the data sources. For each data source a download URL to the OSM file and a URL for the corresponding updates needs to be provided. In normal conditions, the system uses its own database, but it is possible to configure the updater to connect to another existing database. If the default projection interval is not suitable, it can be changed, by altering the projection number in the configuration file. The default update interval can be change in the same way.

Time intensive operations are done in parallel if possible to minimize the initialization and update time. The updater creates a table with meta data about its topicality and a function to get the date and time of the latest update.

5.1.2. EOSMDBTwo

The resulting EOSMDBTwo is the central component of the whole system. It is tuned for minimum query time. To achieve this, indices were created on every column and all the tables are clustered by the label. Since every server has different capabilities, a database configuration file allows administrators to easily change database settings in order to get the best performance. The database contains a default user "guest" that may query the data, but not change it in any way.

To use this database effectively with the GraphQL API, an assortment of PostGIS functions was wrapped to enable the use from GraphQL. Additionally a few custom functions that allow a GraphQL user to run complex queries in one function, were created.

5.1.3. GraphQL

The GraphQL API was implemented with Postgraphile and the Postgraphile PostGIS extension. It automatically connects to the EOSMDBTwo and creates a GraphQL API from the public schema of the database, including all tables and functions.

5.1.4. OSM SQL Terminal

Since the OSM SQL Terminal was removed from the project scope, Joël Schwab dedicated some time to create a first version. The wireframes created for it can be seen in figure 5.1.

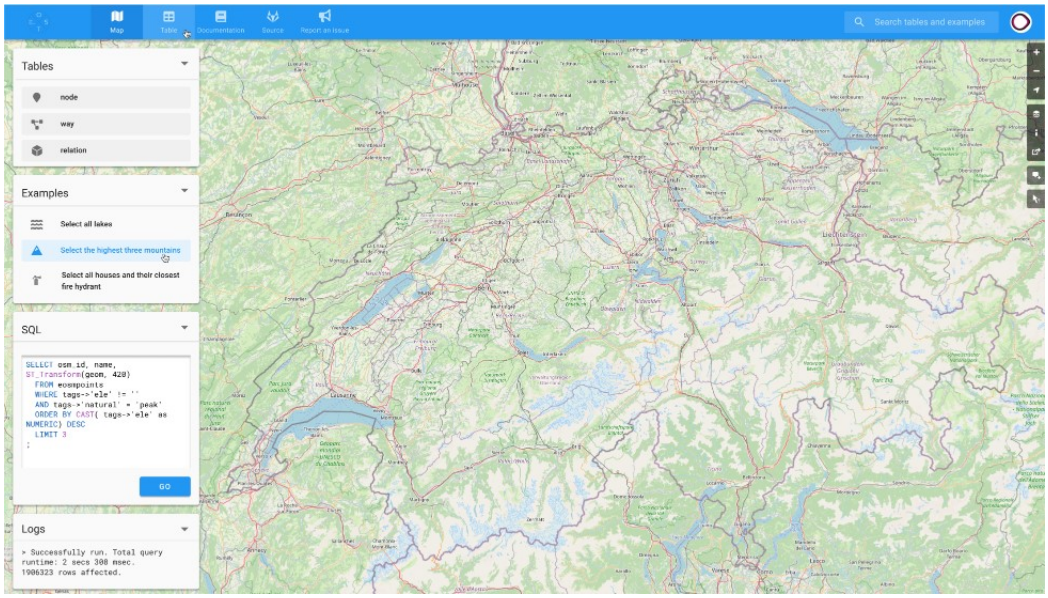


Figure 5.1.: wireframes for the OSM SQL Terminal

A screen shot of the version at mid December 2020 can be seen in figure 5.2.

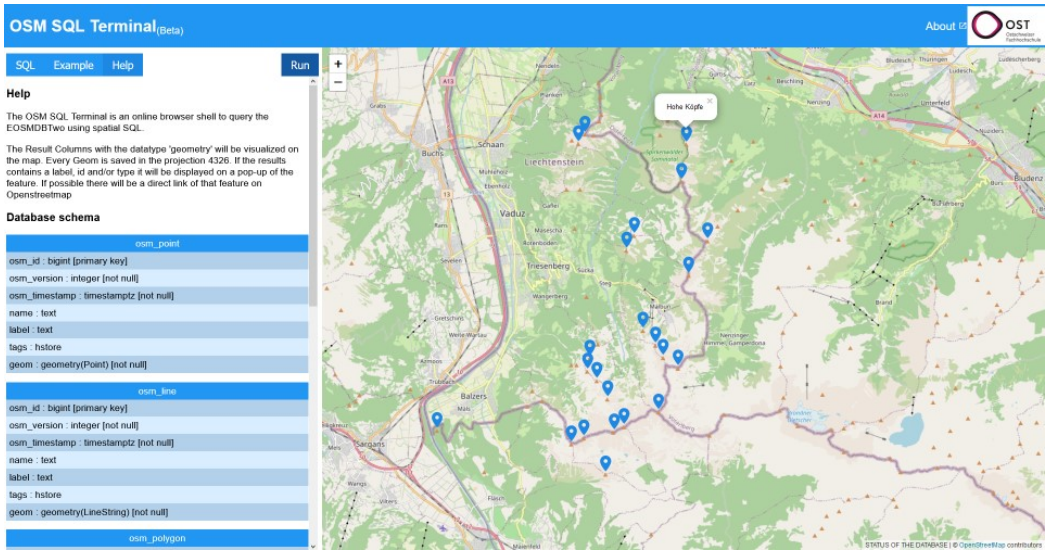


Figure 5.2.: OSM SQL Terminal at 09.12.2020

5.2. Future Prospects

This chapter is dedicated to the potential of this product. The ideas were categorized either as extension of functionality or extension of performance.

5.2.1. Extenstion of Functionality

5.2.2. Limiting Query Length

One of the first things to improve in the EOSMDBTwo is to limit the query length. This would be to prevent one single query to block the whole database. This was a requirement for this project, but could not be implemented due to time constraints.

5.2.3. RESTful Clients

Another interesting addition would be a RESTful client, in contrast to the GraphQL client. As mentioned in the chapter 2, there exist some ready made tools for that.

5.2.4. More Custom Functions

The EOSMDBTwo currently offers a few custom functions, specifically for querying with the GraphQL client. These functions can be used as templates for more custom functions, to allow simpler usage of queries that get made often.

5.2.5. Collaborative Tables

The basic idea is that users of EOSMDBTwo can create custom tables consisting of OSM data from the database. To clarify the idea, the following example is given: If a user wants to select all highways, he has to select all roads, and from all roads filter the highways. Now if users could make their own tables or views, such a select statement could be saved and future users could just select this highway table and use it for their calculations.

The main consideration for this feature is how to save this user defined table. The three main solutions are tables, views or materialized views. Tables are not a good idea, since you would have to save how the table content is calculated and at that point it is just a materialized view. Materialized views would be pretty fast for querying, but would take a lot of time updating, when the database is updated. Views don't need to be updated, but would be much slower while querying.

A hybrid solution could also be possible, but would either rely on the users decisions or require separate input from an admin.

5.2.6. Handling of Historical OSM Data

The scope of this project handles the contemporary data, however OSM data could also be analyzed through a historical lens. All the tools that were used in this project are not designed for working with OSM history data, and must therefore be substituted in a database for historical analysis.

5.3. Extension of Performance

5.3.1. Multiple Database Replications

To guarantee the responsiveness of the EOSMDBTwo, a setup with at least two databases would be helpful. With such a setup, at least one database could always be handling user queries, even if another database is currently updating its data. For a big user base this approach could also yield better performance results when used in conjunction with a load balancer.

5.3.2. Full Parallelisation of Updater

The continuous updater could benefit from parallelizing every task in some cases. This could be achieved through extension of the existing code, or splitting it into multiple services. Regardless, it would add a lot of complexity for a small performance boost.

5.3.3. More Edge Cases

Currently the dataflow is the same for every size of file. For some sizes it would be better if the data flow would be modified. For example, for very small files, the first upload into the database could contain the updated data. This would delay the initial upload of data, but cut down the time needed to do the first load and update task.

5.3.4. Distribution of Calculation

PostGIS is currently not able to meaningfully distribute its calculations. Nonetheless some tasks could be parallelizable, especially when they are geographically constrained. If you only wanted to calculate something within a given border it would be quite easy to have one independent database per region, and one scheduler that distributes tasks and collects results. However, if something must be calculated beyond region borders, it is quite challenging.

A useful tool for distributing such a database could be Greenplum.

5.3.5. Analytical Backend

In the beginning of this project, the idea was to have the whole analytical database based on an analytical tool, for example Apache Spark. This could greatly reduce the calculation speeds and offer a familiar environment for developers that are already accustomed to Apache Spark.

5.4. Personal Reports

5.4.1. Philipp Bolliger

Even though the project was not what I expected, it was very interesting and I learned a lot. It was very captivating to dive into the domain and learn about the common problems of the field. Implementing the solution was most of the time very frustrating. This is because we both had no experience with geospatial data and only little experience with databases, which lead to a slow progression. Because of the frustrating process, the satisfaction was even greater when we finally got something up and running, keeping me motivated to continue working.

5.4.2. Marc Scherrer

Initially we expected something different when we signed up for this project. Nonetheless, it became very interesting very fast. The deep dive into geospatial data and specifically the OSM community was fascinating.
The implementation of these tools was fun at the beginning, but we had to spend much time with correctly using community tools, comparing different functionalities of them and in the

end tracking bugs. This project was very consuming. Looking back at our results, they don't seem that big, but the constant rewriting of code was not easy.

5.5. Acknowledgments

We would like to take the opportunity to express our gratitude to everyone involved in this project. To Stefan Keller for explaining the various OSM topics, helping with decision making and taking the time to review our progress. To Nicola Jordan for setting up the continuous deployment (CD) pipeline and being available for other dev-ops questions. To Joël Schwab for taking over the OSM SQL Terminal and serving as pilot user of the EOSMDBTwo. To Raphaql Das Gupta for the introduction to Python and answering several questions about it and finally to Samuel Herzog for proof-reading the documentation. And last but not least to Sarah Hofmann, for spontaneously helping with the usage of Osm2pgsql and PyOsmium.

Part II.

Software Project Documentation

6. Requirements Analysis

The Requirements are taken from the task definition.

6.1. Functional Requirements

The functional requirements are classified as MUST, SHOULD or CAN. Requirements classified as MUST are mandatory, while requirements classified as SHOULD or CAN would be nice to have, but not necessary. Requirements classified as SHOULD have a higher priority than the those classified as CAN.

FR 001 An administrator starts the system, which automatically downloads the OSM data of Switzerland and Liechtenstein and inserts it into the database | MUST

FR 002 The system updates the data (Switzerland and Liechtenstein) in an interval of 60 minutes | MUST

FR 003 A user queries the SQL API to retrieve data | MUST

FR 004 A user queries the GraphQL API to retrieve data | MUST

FR 005 A user queries the GraphQL API with PostGIS functions | SHOULD

FR 006 A user queries the web interface with PostGIS functions which are older than 1h | SHOULD

FR 007 An administrator starts the system, which automatically downloads the OSM data of the whole world and inserts it into the database | CAN

FR 008 The system updates the data of the whole world in an interval of 1 day | CAN

FR 009 A user queries the database in modern SQL (lateral join etc.) | SHOULD

FR 010 ~~A user logges in automatically as a guest (no authentication) | MUST~~

FR 011 ~~A user chooses SQL predictions in the web interface | CAN~~

FR 012 ~~A user looks at returned data as a table | MUST~~

FR 013 ~~A user looks at returned data as a map | SHOULD~~

6.2. Non-Functional Requirements

NFR 001 The system must deliver a response in 120 seconds to any user queries when operating under normal conditions to ensure a good user experience.

NFR 002 The system needs to be able to handle 100 user queries at once under normal operation.

NFR 003 95% of valid user queries deliver a response without any error messages.

NFR 004 The system has to deliver a response to user queries even if the database is being updated to ensure a good user experience.

NFR 005 Users can only read from the database with the public user to prevent users form modifying the database.

NFR 006 The system does not allow more that 10 queries per minute per user to prevent denial of service attacks.

NFR 007 The system must be prepared for the docker-compose environment, to ensure a easy deployment.

NFR 008 The system needs to provide a documentation to ensure the maintainability of the project. It should at least describe how to start and use the system.

NFR 009 The system must provide tests that can be executed to ensure that the system meets the functional and non functional requirements.

6.3. Restrictions

According to the task definition, the following programming languages and technologies are considered as mandatory:

- Python 3
- PostgreSQL
- PostGIS

The OSM history will not be loaded into the database. Additionally the veracity of OSM data is not discussed and it is assumed that the data from the sources is correct. After the

scope change on 10.11.2020, the OSM SQL Terminal is no longer part of this project. The developers only create a very simple Flask application which forwards an SQL statement to the database as a proof of concept.

7. Analysis

7.1. Data Model

7.1.1. OSM Data Model

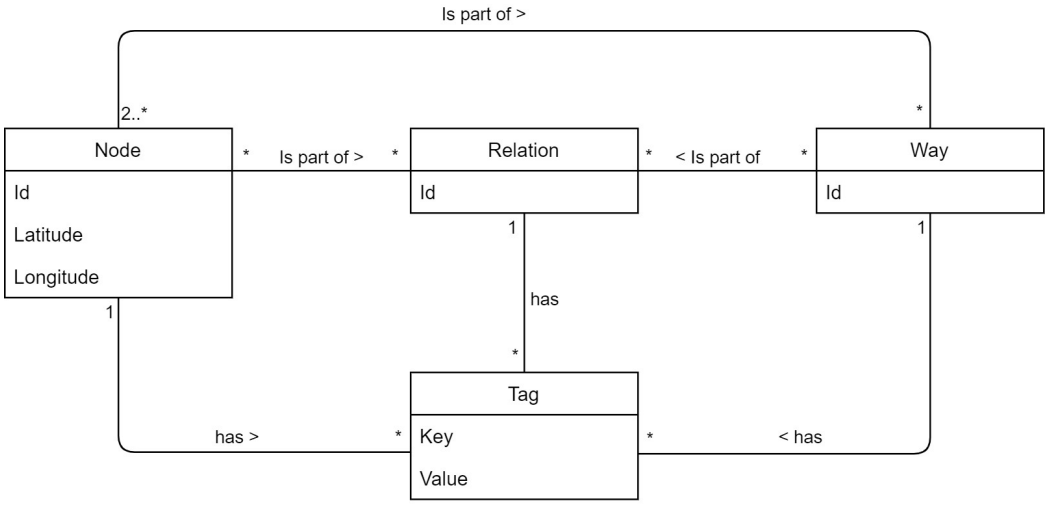


Figure 7.1.: Conceptual data model of OSM as UML class diagram [RT08]

The figure 7.1 shows the data model of OSM as it is relevant to the project. The node is the most basic entity, which includes latitude and longitude. Ways consist of two or more nodes and relations can contain several ways and nodes. Nodes, ways and relations can all have several tags, which describe specific features about the elements, such as the height of a building, speed limit of a road or simply a name. This information can be very useful when querying the data, because it allows to search for specific key value pairs. [osme]

The ids are only unique for the specific element type. This means, that all nodes have unique ids, but there are probably ways with the same id as some nodes.

Additionally to the Id there are several other meta data fields for nodes, ways and relations, that were omitted on the figure for simplicity. Important for this project are the timestamp, which gives information about when the element was changed the last time and the version, which provides information about how many times the element already has been changed.

7.1.2. PostGIS Data Model

This project uses a PostgreSQL database with the PostGIS extension. The PostGIS turns the PostgreSQL database into a spatial database by adding support for spatial types, indexes and functions. [pos]

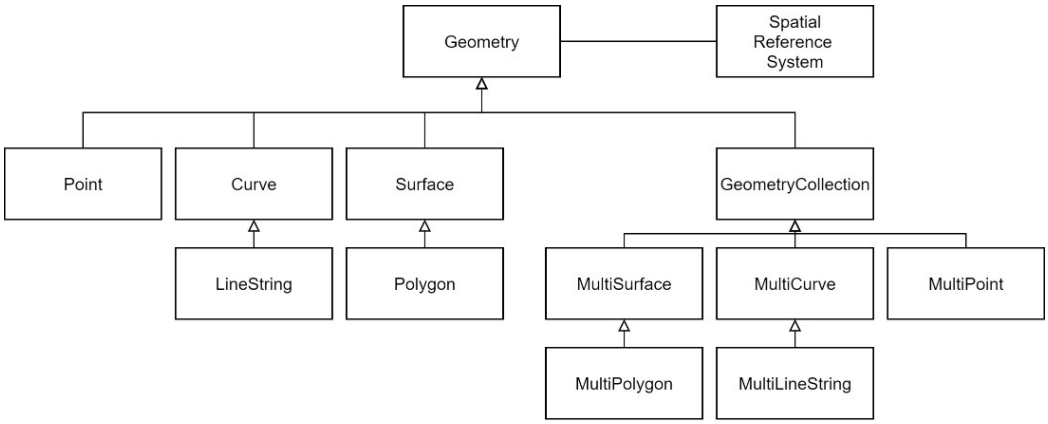


Figure 7.2.: PostGIS geometry hierarchy [pos]

The figure 7.2 shows the geometry hierarchy of the additional data types added by the PostGIS extension. These data types holds information about what projection is used, where it is located and its shape.

7.1.3. EOSMDBTwo Data Model

With Osm2pgsql it is possible to transform data from OSM to PostGIS data. With the new flex back end, it is even possible to write a customized Lua script, that specifies how every OSM feature is processed. A custom Lua script was written to take advantage of this technology. The script processes every OSM feature individually to create a database with the data model described in figure 7.3

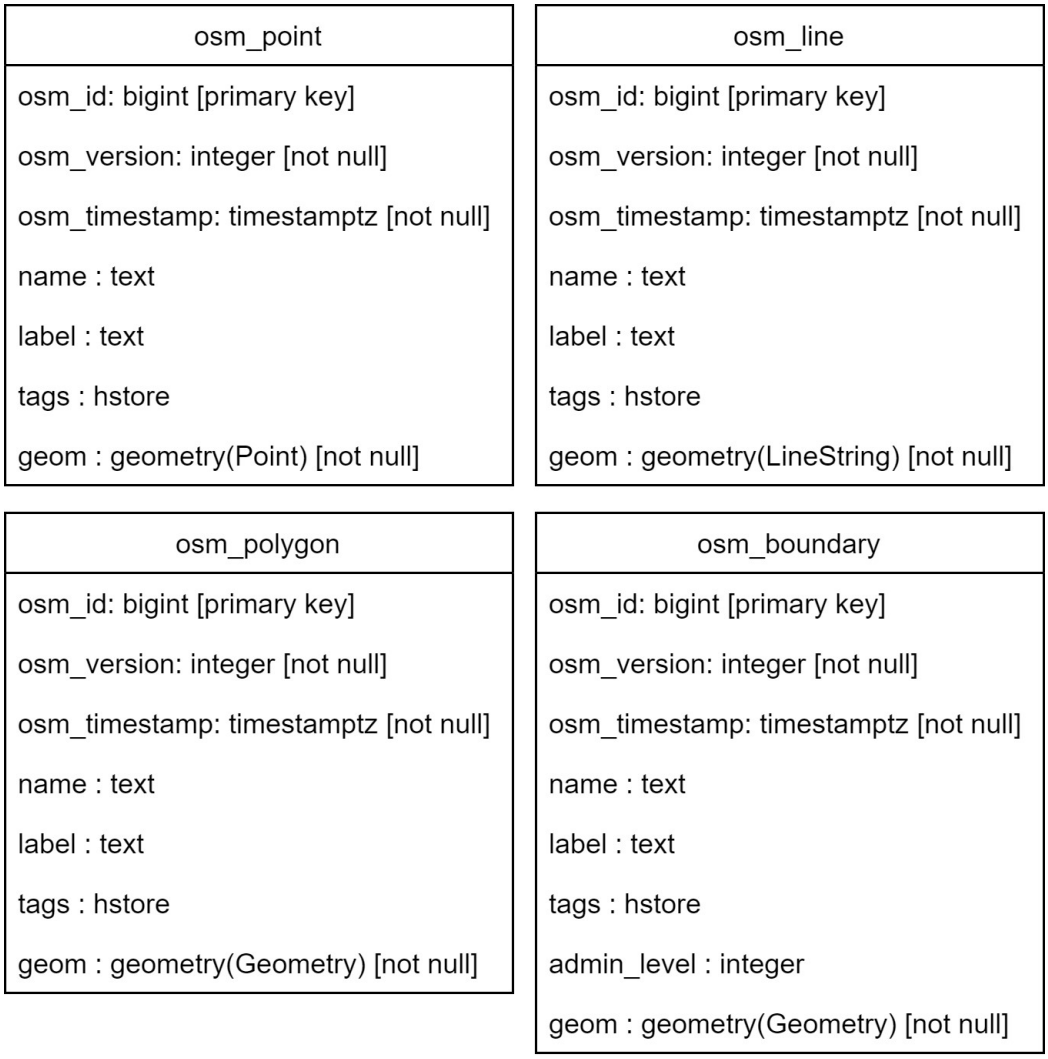


Figure 7.3.: Analytic data model

OSM Feature Processing

Every entry in one of those tables in the resulting database origins from one OSM feature. Depending on the type of the OSM feature and its values, it is added to a suiting table.

ID

Every table has the column `osm_id` as primary key. The `osm_id` is used for indexing and to find the corresponding feature on OSM. For the tables `osm_point`, `osm_line` and `osm_boundary` the `osm_id` is the same as the `id` of the OSM feature the column origins

from. Members of the table `osm_polygon` can be created from either a relation or a way. Simply using the `id` of the OSM feature as `osm_id` would lead to loss of information and lead to primary key collisions. Osm2pgsql solves this problem, by taking the negative id, if the polygon originates from a relation.

$$For\ Ways : osm\_id = osm.id$$
$$For\ Relations : osm\_id = -osm.id$$

Name

The name is the value of the key `name` in the tags of the OSM feature. It was extracted from the tags, because it is often used in queries and using the hstore would slow them down.

Label

The label is calculated depending on the chosen language. The algorithm to calculate the label first gets the name in the chosen language from the tags. If no language is chosen, it just uses the default name to proceed. If a value is available and is in latin script, it uses it as label. Otherwise a fallback value will be calculated similar to OSMaxx [osmb]. It checks the English, French, Spanish, German and International name in this exact order and uses the first one as label which is present. If at this point still no value was found for the label, the name is used.

```
label ← NULL
if preferred language is set then
  label ← languageSpecificName
else
  label ← defaultName
end if
if label is not NULL and latin script then
  return label
else
  if English name is set then
    return englishName
  else if French name is set then
    return frenchName
  else if Spanish name is set then
    return spanishName
  else if German name is set then
    return germanName
  else if International name is set then
    return internationalName
  else
    return defaultName
  end if
end if
```

Figure 7.4.: Pseudo code of a function that calculates and returns the label.

Since it always checks, if the value of the name in the chosen language is in latin script, only languages that only use latin script characters are usable.

Geometry

The geom is the geometrical information about the object. This is where the PostGIS data types come into play. Every geom is saved in the projection that is given by the configuration. For `osm_point` the geometry type `Point` is used. For `osm_line` the type `MultiLineString` is used. The tables `osm_polygon` and `osm_boundary` use the general `Geometry` type. This is because some polygons and boundaries consist of more than one part. In such cases an object of the type `MultiPolygon` is created, which can only be saved in a `Geometry`, `GeometryCollection` or `MultiPolygon` column. It would also be possible, to wrap every single polygon in a `GeometryCollection` or `MultiPolygon` but since the occurrence of cases, where a single polygon isn't sufficient is very rare, this would be too much of an overhead for both inserting and reading the data.

Location View

In addition to the tables, a view is provided, that combines points and polygons. For the geom column the PostGIS type `point` is used. For polygons, the new geom is calculated by using the PostGIS function `st_centroid`, which computes the geometric center of a geometry.

The `id` is used to have a unique value and is calculated by multiplying the original OSM id by ten. Then depending on the type of feature the object originates from, a number is added. For nodes add zero, for ways add one and for relations add two. The resulting number is the new poi_id.

$$
\begin{aligned}
For\ nodes : id &= osm.id * 10 \\
For\ ways : id &= osm.id * 10 + 1 \\
For\ relations : id &= osm.id * 10 + 2
\end{aligned}
$$

The `osm_id` is the normal id of the OSM feature the geometry origins from. The `osm_type` shows from what OSM feature type the entry origins from. For this purpose an additional enum with the values `N` for node, `W` for way and `R` for relation was created.

Figure 7.5.: Additional location view

8. Design

8.1. Architecture

The project includes several components, which all run in separate docker containers. For simplicity reasons it was decided, that for this project all containers run on the same server. The database updater and the PostgreSQL database are the central components of this project. Depending on the clients needs, he can either use the OSM SQL Terminal or the GraphQL client to query the data.

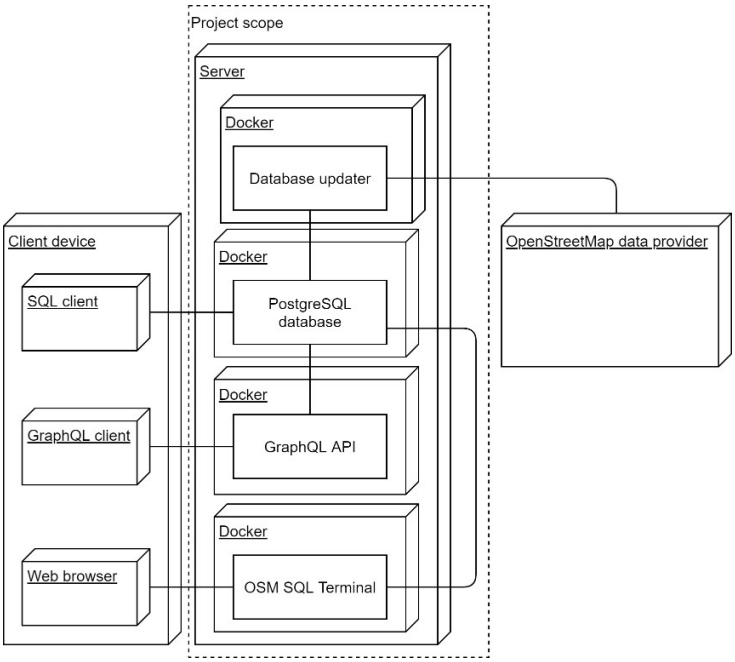


Figure 8.1.: Deployment diagram

The figure 8.2 captures the layers of the project. The Flask application and Postgraphile are both in the Application Layer and Data Access Layer. That is the case because they mainly access data from the PostgreSQL database and they do not contain much logic themselves.

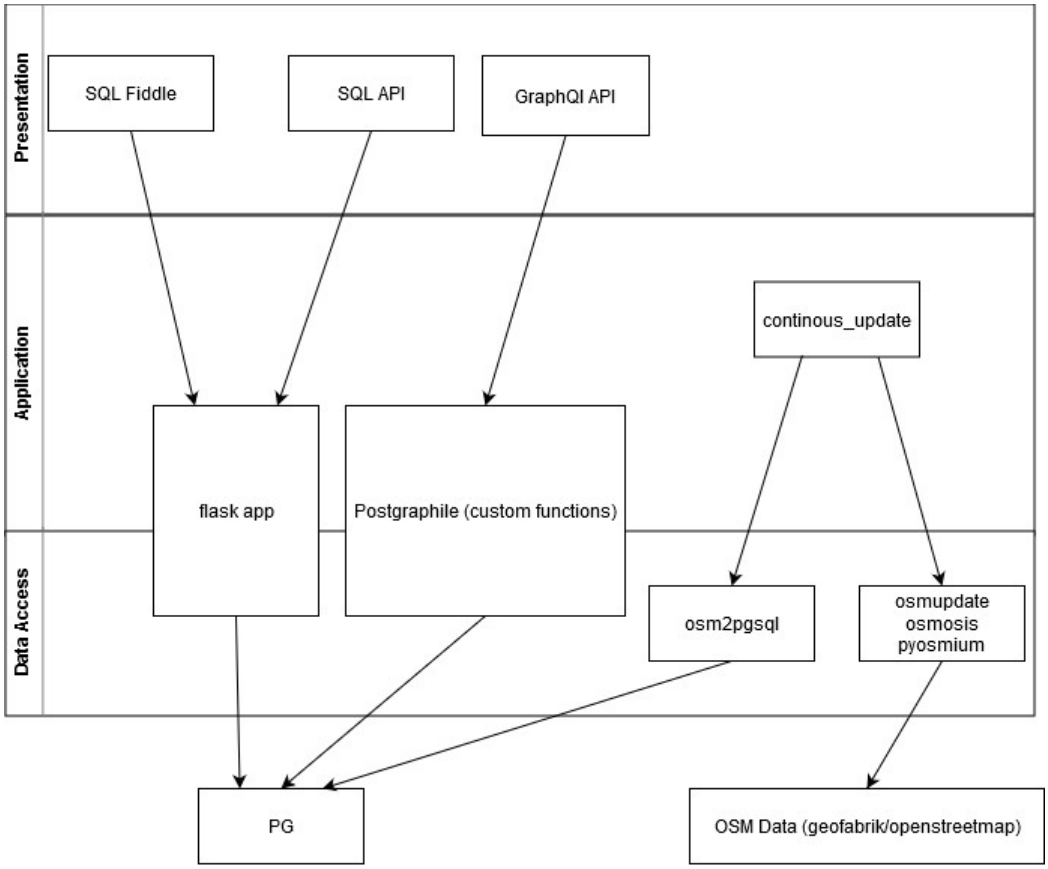


Figure 8.2.: Layer Diagram

8.2. Data Flow

The data flow itself had to be changed numerous times due to lack of in-depth knowledge of the tools. After discussing directly and indirectly with the creator of PyOsmium the data flow as shown in figure 8.3 is implemented.

The OSM files need to be combined before initializing the database, because the only other option would be to load them one after the other, which would mean, that Osm2pgsql would have to append the data for all OSM files but the first. Since the append mode is way slower this would slow down the initialization by a significant factor.

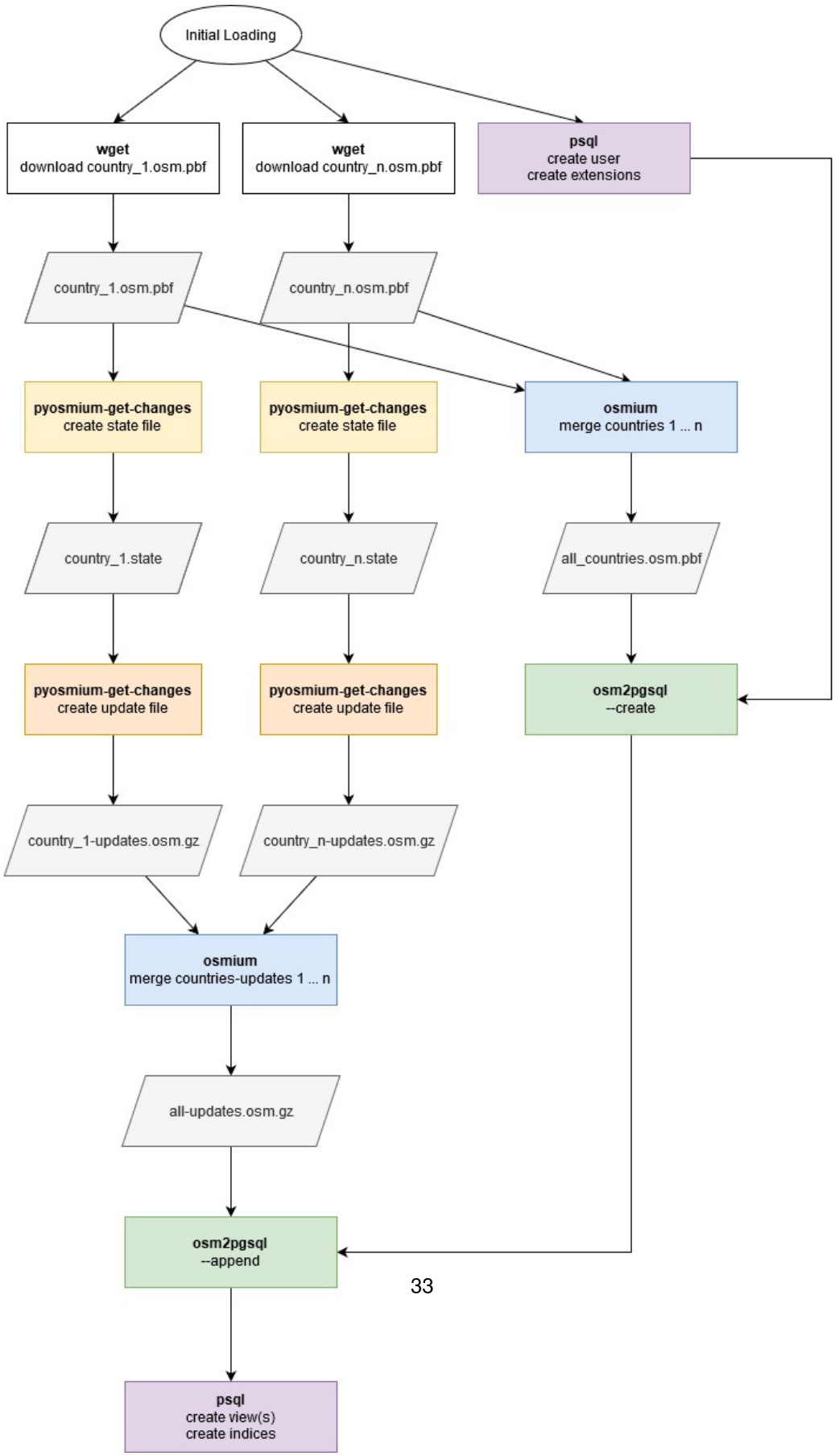


Figure 8.3.: Data Flow Diagram

8.3. Important Concepts

8.3.1. OsmFile

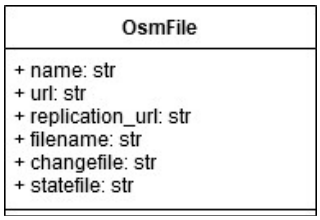


Figure 8.4.: OsmFile

The OsmFile represents the data source file. It does not contain and data itself, but rather includes information about the data can be found. This includes the URL from where the OSM file can be downloaded, the URL where the update files can be downloaded and paths where the downloaded files are saved on the local machine. It is used to access the data for handling the OSM file. For each country one OsmFile is created.

8.3.2. OsmFile Service

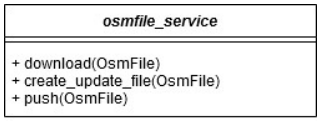


Figure 8.5.: OsmFileService

The OsmFileService provides functionality to download files, create update files and push data to the database. It collaborates with the OsmFile to know where to download files and where they are on the local machine.

8.3.3. CommandlineHelper

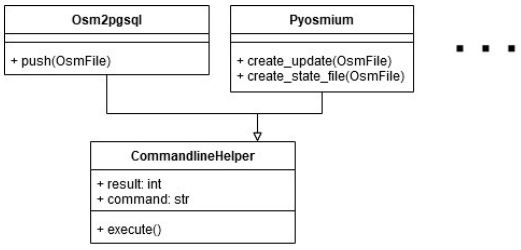


Figure 8.6.: CommandlineHelper

The CommandlineHelper is an abstract class that is used to execute commands via the commandline. For every tool or command a concrete class is implemented that wraps the needed functionality. The OsmFileService uses implementations of the Commandline-Helper.

8.3.4. DatabaseUpdater

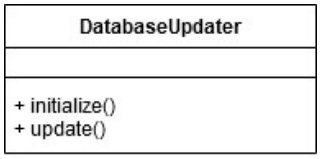


Figure 8.7.: DatabaseUpdater

The DatabaseUpdater class is responsible for realizing the data flow. It uses the OsmFile and OsmFileService to carry out the steps that are defined in the data flow.
The initialize function downloads the data from the source, pushes it into the database and carries out the first update. The update function creates the update files and pushes them into the database. This class also ensures that the database is ready to accept the data. As soon as the data is pushed, it (re)creates indices, functions, etc. using the eosmdb processor.

8.3.5. Eosmdb Processor

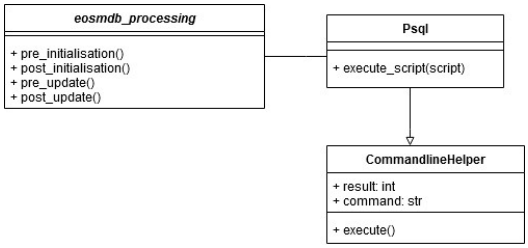


Figure 8.8.: EosmdbProcessing

The Eosmdb processor provides functions for pre- and post initialisation as well as pre- and post update processing. These include to creating users, creating indices, creating functions, clustering tables, etc. The specific steps performed are defined in SQL scripts.

8.3.6. Communication Sequence

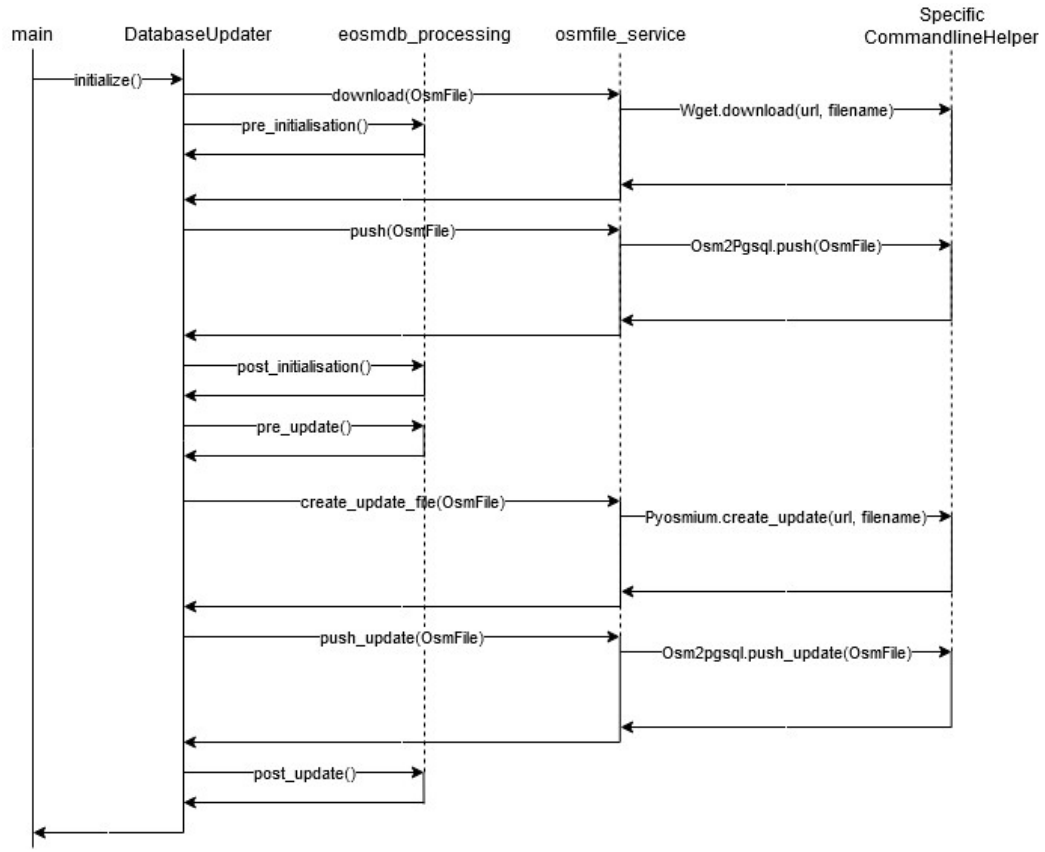


Figure 8.9.: Sequence Diagram

The sequence diagram for the initial loading of the data is shown in figure 8.9. Some steps can be parallelized. The scale of this diagram is not in proportion with the time needed by these steps.

8.4. Architectural Descisions

We used Olaf Zimmermanns "Y-Statements" [Zim20] approach to document and explain the design decisions made in this project.

In the context of <use case and/or component>,
facing <non-functional concern>,
we decided for <option>,
and neglected <options>,
to achieve <quality>,

accepting the downside <consequence>.

8.4.1. Service split

In the context of how to define and split services, facing reliability and maintainability, we decided for splitting services into four components, and neglected splitting services into one service per one job or having a monolith, to achieve overview and not to have an elaborate communication concept, accepting the downside that the continuous update component is very big.

8.4.2. Loading updates

In the context of how to write updates into the database with Osm2pgsql, facing update speed and availability, we decided for "–append" for updates, and neglected "–create" with the updated OSM files, to achieve reduced update time and increased data availability throughout the update process, accepting the downside that pre- and post-update scripts had to be written.

8.4.3. Implementing the data flow

In the context of implementing the download and update dataflow, facing i/o intensive tasks, we decided for working with threads, and neglected strictly procedural programming, to achieve less waiting time, accepting the downside of a bit more work and less clarity.

9. Testing

EOSMDBTwo is a combination of several tools and technologies, making it hard to create automated tests, without having to create a lot of overhead by implementing mocks. Therefore it was decided, that for this project testing shall be done completely manually. In order to reproduce test results, test protocols are written, with detailed steps and expectations. Each test protocol tests one specific feature, such that all test protocols together guarantee a working system according to the functional and non functional requirements. Testing shall be done before every merge from the development to the master branch.

The test templates can be found in the appendix and the completed tests can are in a separate folder.

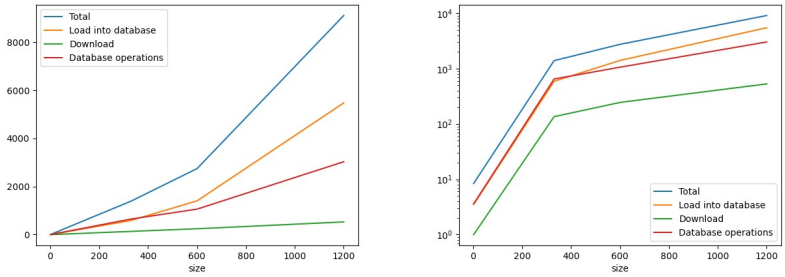
10. Results and Further Development

10.1. Results

The general results are mentioned in chapter "Results" 5.

To test the performance and how the system scales, the initial loading and the first update was measured for four different countries. The countries were chosen such that the next bigger one is approximately double the size (except the first one). Chosen candidates were "Lichtenstein" with 2.3 MB size, "Switzerland" with 331 GB size, "Austria" with 601 GB size and "Poland" with 1.2 GB size. These four countries were loaded sequentially and the resulting load times were compared.

From the performance comparison plots 10.1, it is clearly visible that our solution scales non-linearly with the size of the OSM file, but not quite logarithmically.



(a) Performance scaled linearly (b) Performance scaled logarithmically

Figure 10.1.: Performance comparison plots

10.2. Further Development

The Ideas for further development are discussed in the chapter "Future Prospects" 5.2.
These include

- Limiting query length
- RESTful clients

- More custom fucntions
- Collaborative tables
- Handling historical OSM data
- Multiple database replications
- Parallelisation of the updater
- Handling more edge cases
- Distribution
- Analytical backend

10.3. Further Development Procedure

The most pressing issue is certainly the limitation of query length. The best way would be to limit it in the database. There it would affect all clients.
The custom functions can be modeled after the ones already implemented.
All other further developments would require more investigation then is done in this project.

11. Continuous Integration / Continuous Deployment

For CI/CD GitLab CI is used. GitLab uses a special file called ".gitlab-ci" which uses the .yaml format to define CI steps. In this project, there are mainly two task-chains, one for the continuous deployment of the product itself and one for building and deploying the documentation pdf.

The documentation is quite easy to understand since there are only four transparent steps that have to be made. In the first step the timereports are generated. The timereports were used by the project members to track their time during the project. The second step simply writes the Git information to a file which is later included in the pdf. In the third step the pdf is generated from the latex source. The fourth step is then to publish the pdf with GitLab pages [1] through the "pages" job.

The deployment of the application is a bit more complicated. The technical adviser (Nicola Jordan) has setup the pipeline and the developers have configured it with him. First the product images are built and pushed to the GitLab images registry. Then a Portainer instance is informed that a new version exists and it pulls the current images. Deployment variables are defined separately in the Portainer service.

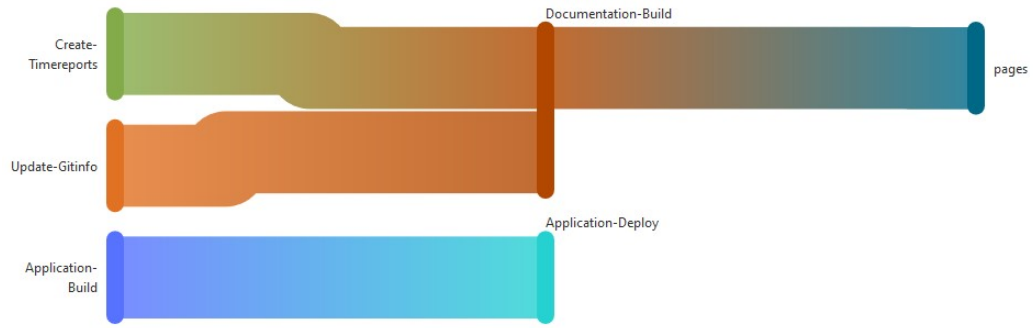


Figure 11.1.: GitLab CI DAG

[1] GitLab Pages | Gitlab, accessed on 07.11.2020, https://docs.gitlab.com/ee/user/project/pages/

12. Project Management

12.1. Process

Since this is a very exploratory project and it is very likely that requirements will change, we decided to base our development approach on Scrum. The iteration length is two weeks.

Every Tuesday the development team meets with the customer. The team presents the current state of the project and the customer can suggest changes and new ideas.

Afterwards the development team holds the retrospective, the product backlog refinement and if its the beginning of a sprint, the sprint planning, as described in the Scrum guide. [scra]

12.1.1. Issues

Issues are managed using GitLab. Every issue is assigned to one or more labels to categorize the work that needs to be done. This is useful for the time report that can be found in chapter 13.1. The full list of labels and their purpose can be found in the GitLab repository.

New issues may be created by developers at any time. Both project members are responsible to create new issues from the suggestions of the customer. Newly created issues are automatically added to the product backlog. The product backlog refinement consists of estimating the new issues, splitting large issues into several smaller issues, as well as discussing if existing issues are still relevant and deleting those that are not, which in turn may lead to new issues.

When estimating issues, the developers discuss what needs to be done and use this information to guess its size. Each size specifies how long it takes to complete an issue of that size. An issue be categorized into one of the following sizes.

Size	Time in hours
XS	1
S	2
M	4
L	8
XL	16+

Table 12.1.: Issue sizes and how long it takes to complete them

12.1.2. Collaboration

GitLab is used as Git-Repository manager to store all Files of the project. To ensure consistent commit messages, the developer should use the following template:

if this commit is applied it should <commit message>

The developers also use a shared OneNote notebook for everything that does not belong in the GitLab repository, like useful links, drawings, etc. For communication, the developers frequently use discord to talk and share screens. Also Microsoft Teams is used to semi-formally discuss issues with the adviser.

12.1.3. Time Tracking

To track the time the developers spend on the project, the GitLab time tracking tool is used, where developers can simply book time on issues. There exists a special management overhead issue, that is intended to book time for minor work, that does not fit to any other issue and it's not worth to make a specific issue for this task. An example would be a catch-up discussion for developers. If time is booked on that issue, one must also write a comment to explain what was done.

12.1.4. Reviews

Every tuesday, the developers and the adviser meet up and discuss the status of the project, and try to answer questions that came up that week. As a preparation the developers send an email to the adviser with the information on what was worked on that week, and what problems they had.

Protocols with professor Keller are called "Meeting <date>" and protocols for the meeting of the developers "Weekly <date>". All can be found on the repository as issues with the "Meetings" tag at gitlab.dev.ifs.hsr.ch/sa2/osm-data-pipeline.

12.2. Roles

12.2.1. Developer

- Philipp Bolliger
- Marc Scherrer

The developers are responsible for the creation and deployment of the product.

12.2.2. Project Owner

- Marc Scherrer

His duty is to create new issues, prioritize them and keep the backlog up to date. Since he is both developer and project owner, he has to represent these roles interchangeably.

12.2.3. Scrum Master

Since the team only consists of two members, there is no Scrum master.

12.2.4. Supervisor and Customer

- Professor Stefan Keller

Prof. Stefan Keller initiated the project and is also responsible for the idea. As supervisor he is responsible to control the project flow and its progress, while his role as customer is more concerned about the product.

12.2.5. Technical Adviser

- Nicola Jordan

The technical adviser is at our disposal for any technical questions regarding the domain.

12.2.6. Pilot User

- Joël Schwab

The pilot user was assigned on the 2nd November and is here to try out the product. He also continued the development of the OSM SQL Terminal

12.3. Project Plan

12.4. Milestones

12.4.1. Minimal Viable Product

The Minimal Viable Product (MVP) is a SQL-Database, filled with up to date OSM data of Switzerland. The OSM data is updated every hour and stored as geometry types.

12.4.2. GraphQL API

This milestone is set very competitively, since all PostGIS functions have do be mapped in GraphQL. The result is a GraphQL connection to the database, over which one can execute ST_ functions from PostGIS.

12.4.3. OSM SQL Terminal

This is the first version for the terminal in which one should be able to query OSM data with SQL and the PostGIS extenstion. This milestone was removed due to the scope change.

12.4.4. EOSMDBTwo

EOSMDBTwo is the heart of this project and because of this the OSM SQL Terminal was cut out of this project.

12.5. Iterations

Iteration	Start	End	Description	Version
1	29.09.2020	13.10.2020	MVP	0.1
2	13.10.2020	27.10.2020	GraphQL API	0.2
3	27.10.2020	10.11.2020	GraphQL API Refinement	0.2.1
4	10.11.2020	24.11.2020	~~OSM SQL Terminal~~ EOSMDBTwo refinement	0.3
5	24.11.2020	8.12.2020	Overall Refinement	1.0
6	8.12.2020	18.12.2020	Documentation (shorter sprint due to end of semester)	1.0

Table 12.2.: Iterations

12.6. Risks

Following is a list of possible risks for this project.

1. Risk: Person gets sick for 1 week
    - Measure: -
    - Max. Damage: 17h
    - Probability: 10%
    - Weighted Damage: 2h
    - Behavior: Scope reduction
2. Risk: Unfamiliar technologies require more time than expected
    - Measure: Estimate time for training
    - Max. Damage: 200h
    - Probability: 10%

- Weighted Damage: 20h
- Behavior: Ask advisers or community for help

3. Risk: Technologies do not provide desired functionality
    - Measure: Evaluate technologies based on requirements
    - Max. Damage: 200h
    - Probability: 7%
    - Weighted Damage: 14h
    - Behavior: Use other technology, write own functionality or reduce scope
4. Risk: Data loss due to shutdown of GitLab server
    - Measure: have the repo locally
    - Max. Damage: 250h
    - Probability: 0%
    - Weighted Damage: 0h
    - Behavior: Use GitHub
5. Risk: Requirements change during the project
    - Measure: Working agile
    - Max. Damage: 50h
    - Probability: 50%
    - Weighted Damage: 25h
    - Behavior: Scope reduction, ask for help
6. Risk: Communication problems
    - Measure: Try to be as clear as possible
    - Max. Damage: 50h
    - Probability: 30%
    - Weighted Damage: 15h
    - Behavior: Use retrospectives at sprint ends to improve communication.
7. Risk: Unrealistic scope
    - Measure: Set small goals & always have a working product
    - Max. Damage: 50h
    - Probability: 5%
    - Weighted Damage: 3h
    - Behavior: Scope Reduction

8. Risk: OSM Standard changes
    - Measure: -
    - Max. Damage: 200h
    - Probability: 0.1%
    - Weighted Damage: 0h
    - Behavior: Use old standard
9. Risk: Import times are too slow for fast evaluation
    - Measure: Develop with small dataset
    - Max. Damage: 20h
    - Probability: 20%
    - Weighted Damage: 4h
    - Behavior: Work slower
10. Risk: Hardware request is not granted in time
    - Measure: Work only with Switzerland
    - Max. Damage: 40h
    - Probability: 10%
    - Weighted Damage: 4h
    - Behavior: Scope reduction

During the development time the risk number three "Technologies do not provide desired functionality" has occurred. As described in the behavior, the scope was changed as documented. The damage is very hard to estimate, but the scope reduction that was approved, cut the project by approximately 80 hours.

13. Project Monitoring

13.1. Timereport

In the table 13.1 the hours spent per week is visualized. The numbers may fluctuate a bit since the sprints started and ended on Tuesdays. Since the time tracking tool was not ready at the beginning of the first week, it seems like not much time was invested.

Week	Time spent
14.09.2020 - 20.09.2020	3h
21.09.2020 - 27.09.2020	38h
28.09.2020 - 04.10.2020	26h
05.10.2020 - 11.10.2020	33h
12.10.2020 - 18.10.2020	38h
19.10.2020 - 25.10.2020	34h
26.10.2020 - 01.11.2020	38h
02.11.2020 - 08.11.2020	30h
09.11.2020 - 15.11.2020	37h
16.11.2020 - 22.11.2020	31h
23.11.2020 - 29.11.2020	34h
30.11.2020 - 06.12.2020	36h
07.12.2020 - 13.12.2020	44h
14.12.2020 - 18.12.2020	26h
Total	**444h**

Table 13.1.: Time spent per week

During the project about 2/5 of the time was invested in the actual products, another 2/5 for the documentation and 1/5 for meetings.

13.2. Code Statistics

The code consists of 52% Python code, 28% SQL scripts, 18% Lua and 2% HTML.

Acronyms

API Application Programming Interface. I, II, III, VI, VIII, IX, XI, XII, XIII, 2, 3, 5, 7, 8, 9, 10, 15, 22, 44, 50, 54, 73

CD continuous deployment. IV, 20, 41

CI continuous integration. IV, 41, 53

DAL Data Access Layer. 31

EOSMDBOne Extended OSM Database One. I, II, III, IV, VI, XI, 2, 3, 5, 6

EOSMDBTwo Extended OSM Database Two. I, II, III, IV, VI, VII, VIII, IX, XI, XII, XIII, XIV, 2, 3, 13, 15, 17, 18, 20, 26, 38, 45, 50

GIS Geographic Information System. I, II, III, XI, 4, 5

GUI Graphical User Interface. III, IV, VI, 7

IFS Institute for Software. IV, 3

MVP Minimal Viable Product. XIII, 44

OLAP Online Analytical Processing. 5

OSM OpenStreetMap. I, II, III, IV, VI, VII, VIII, IX, XI, XII, XIII, 3, 4, 5, 6, 7, 8, 10, 11, 13, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 30, 31, 32, 34, 37, 39, 40, 44, 45, 47, 49, 50, 51, 53, 54, 57, 69

WSGI Web Server Gateway Interface. 11, 51

Glossary

GraphQL Is a Open-source data query and manipulation language. I, II, III, IV, VI, VIII, IX, XI, XII, XIII, 2, 3, 6, 8, 9, 10, 15, 17, 22, 31, 44, 51, 53, 54, 73

SQL Stands for "Structured Query Language". It is a Language for querying and creating data and data structures. I, II, III, IV, VI, VIII, IX, XI, XII, XIII, 2, 3, 5, 16, 17, 20, 22, 24, 31, 35, 44, 45, 48, 50, 53

wireframes Graphical sketches. I, II, IV, 3, 16, 53

PostGIS Is a PostgreSQL extension that adds geometry data types and functions. III, IX, XII, XIII, 3, 5, 6, 7, 8, 9, 10, 15, 19, 22, 23, 26, 29, 30, 44, 45, 51, 52, 53

RESTful A programming interface standard. REST is an Acronym for REpresentational State Transfer. III, XII, 7, 17, 39, 51

OSM SQL Terminal This is the Terminal where a user can query the EOSMDBTwo and view the results. III, IV, VI, VIII, IX, XI, XII, XIII, 3, 16, 17, 20, 24, 31, 44, 45, 53

Overpass A API to query OSM data of the whole world. III, XI, 5

docker Is a program for OS-level virtualization. IV, 12, 31, 56, 57, 59, 62, 66, 71, 73

pgAdmin 4 A frontend for PostgreSQL database connections. IV, 2, 56, 57, 62, 66, 69

JSON A data-interchange format [jso]. IV, 13, 58

Osm2pgsql Is a program to create a PostgreSQL database from a OSM file. VIII, XI, 2, 5, 6, 8, 10, 13, 20, 26, 28, 32, 37, 69

PostgreSQL is a relational database implementation. IX, 2, 3, 5, 6, 9, 23, 26, 31, 50, 51

Osmium Is a library for processing OSM data. XI, 5, 6, 13

Osmupdate Is a tool [osmd] to keep .osm files up to date. XI, 6, 10, 11

PyOsmium Is a tool [pyo] for processing OSM data. XI, 6, 8, 10, 11, 13, 20, 32

Postgraphile A GraphQL application for PostgreSQL. XI, 6, 8, 15

Hasura A GraphQL application for PostgreSQL. XI, 6

pg_featureserv A lightweight RESTful web service for accessing a PostgreSQL database, offering PostGIS support. XI, 7

PostgREST A lightweight RESTful web service for accessing a PostgreSQL database. XI, 7

Python Is a programming language. 3, 6, 20, 23, 48, 51

Portainer A docker container manager. 3, 41

Scrum Is a agile, iterative framework for developing complex and unpredictable products [scrb]. 3, 42

Apache Spark Apache Spark is a unified analytics engine for large-scale data processing. 5, 19

Node.js Is an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications [nod]. 7, 11, 12

Django Is a high-level Python Web framework that encourages rapid development and clean, pragmatic design [dja]. 7, 11, 12

Flask Is a lightweight WSGI web application framework [fla]. 7, 11, 12, 24

JavaScript A common programming language. 9, 11

Osmosis Is a tool [osmc] for processing OSM data. 10

GitHub Is a Git repository manager, which provides time and issue tracking. 12, 46

Greenplum A parallel processing platform for PostgreSQL. 19

Lua A lightweight, high-level programming language. 26, 48

OSMaxx A tool to get OSM data in a specific format. 28

GitLab Is a Git repository manager, which provides time and issue tracking. 41, 42, 43, 46, 53

Git Is a version-control system, which helps collaborating. 41, 43, 51

ST_ functions "ST_" is the prefix of PostGIS' geometry functions (e.g: ST_Buffer(geom) to select a buffer zone around the geom). 44

PostGraphiQL Is a GraphQL front end application. 60

Bibliography

[dja] The web framework for perfectionists with deadlines | django. https://www.djangoproject.com/. Accessed: 2020-10-31.

[fla] Flask - pypi. https://pypi.org/project/Flask/. Accessed: 2020-10-31.

[jso] json.org. https://www.json.org/json-en.html. Accessed: 2020-11-21.

[nod] About | node.js. https://nodejs.org/en/about/. Accessed: 2020-10-31.

[osma] Osm2pgsql manual. https://osm2pgsql.org/doc/manual.html#getting-and-preparing-osm-data. Accessed: 2020-10-14.

[osmb] Osmaxx data schema. https://github.com/geometalab/osmaxx/blob/master/docs/osmaxx_data_schema.md. Accessed: 2020-11-15.

[osmc] Osmosis - openstreetmapwiki. https://wiki.openstreetmap.org/wiki/Osmosis. Accessed: 2020-10-14.

[osmd] Osmupdate - openstreetmapwiki. https://wiki.openstreetmap.org/wiki/Osmupdate. Accessed: 2020-10-12.

[osme] Tags. https://wiki.openstreetmap.org/wiki/Tags. Accessed: 2020-11-15.

[pos] PostGIS introduction. https://postgis.net/workshops/postgis-intro/introduction.html. Accessed: 2020-11-22.

[pyo] Pyosmium - osmcode. https://osmcode.org/pyosmium/. Accessed: 2020-10-14.

[RT08] Frederik Ramm and Jochen Topf. OpenStreetMap, pages 42–44. Lehmanns Media, 1. edition, 2008.

[scra] The scrum guide. https://www.scrumguides.org/scrum-guide.html. Accessed: 2020-10-06.

[scrb] What is scrum? https://www.scrum.org/resources/what-is-scrum. Accessed: 2020-10-03.

[Zim20] Olaf Zimmermann. Y-statements. medium, 2020.

List of Figures

List of Tables

Part III.

Appendices

A. Test Protocols

In this chapter all test protocol templates are listed. The completed protocols are in a separate folder.

Requirements

For all test protocols the following preconditions are required:

- docker and docker-compose is installed
- repository cloned
- pgAdmin 4 is installed

Success condition

A test is considered successfully completed when all expected results match the actual results

Starting the System

This test should verify, that the system automatically downloads OSM data from Switzerland at start-up and writes the data into the database.

Date:
Branch:
Commit:
Tester:

Steps

Step 1

Open a terminal and navigate to the Product folder of the repository and run the command

```
sudo docker-compose up --build
```

to create and start the docker containers.

Expected Result:
Docker is able to build and start all containers without any errors. Within the logs of each container no errors are to be found.

Actual Result:

Step 2

Start pgAdmin 4 and connect to the database with the user osmuser.

Expected Result:
The connection can be established, the tables osm_point, osm_line, osm_polygon and osm_boundary and a view osm_location are present.

Actual Result:

Step 3

This step needs to be done for osm_point, osm_line, osm_polygon and osm_boundary and osm_location individually. Open the query tool and run the following select statement:

```
SELECT count(*) FROM <name>;
```

Expected Result:
The number `count` is be greater than ten thousand for osm_point, osm_line and osm_polygon. For the table osm_boundary it is be greater than ten. The sum of `count` of osm_point and osm_polygon equals the `count` of osm_location.

Actual Result:

Step 4

This step needs to be done for osm_point, osm_line, osm_polygon and osm_boundary and osm_location individually. In the query tool, replace the present statement with the following:

```
SELECT osm_id, ST_AsGeoJSON(geom) as geom FROM <name>;
```

Pick a random row and copy the value of the `geom` column into the JSON tab of `https://geojson.io/`.

Expected Result:
The resulting geometry is either in Liechtenstein or Monaco or not more than ten kilometers from their borders.

Actual Result:

Test Result

Label Calculation

This test verifies that the label calculation works as intended.

Date:
Branch:
Commit:
Tester:

Steps

Step 1

Open a terminal and navigate to the Product folder of the repository and run the command

```
sudo docker-compose up --build
```

to create and start the docker containers.

Expected Result:
Docker is able to build and start all containers without any errors. Within the logs of each container no errors are to be found.

Actual Result:

Step 2

Visit `http://localhost:5439/graphiql` and run the following query:

```
query MyQuery {
  allOsmPoints {
    edges {
      node {
        osmId
        name
        label
        tags
      }
    }
  }
}
```

Then search for `name:en` and choose the first result entry, where the `name` does not match with the `name:en` in `tags`.

Expected Result:
The `label` matches the `name:en` in tags.

Actual Result:

Step 3

Find any entry, where the `name` is in latin script and no `name:en` is present.

Expected Result:
The `label` equals the `name`.

Actual Result:

Step 4

Go to the file `config-dev.json` in the continuous update folder and replace the content of the countries array with a single entry for Bhutan. The URLs are:

```
URL:
http://download.geofabrik.de/asia/Bhutan-latest.osm.pbf

Replication URL:
http://download.geofabrik.de/asia/Bhutan-updates
```

It is also necessary, to change the `PEFLANG=en` to `PEFLANG=es` int the .evn file. Then run the following commands to restart the docker containers with the new configuration:

```
sudo docker-compose down
sudo docker-compose up --build
```

After the containers are running again, rerun the query in PostGraphiQL to get the updated data and search for `1466919186`.

Expected Result:
One entry is found. The label equals the name. Both are clearly not latin script

Actual Result:

Step 5

Search for `1376501203`

Expected Result:
The `label` equals the `name:es` in tags.

Actual Result:

Step 6

Search for `1467096122`

Expected Result:
The `label` equals the `name:en` in tags.

Actual Result:

Test Result

View

This test should verify, that the location view works correctly.

Date:
Branch:
Commit:
Tester:

Steps

Step 1

Open a terminal and navigate to the Product folder of the repository and run the command

```
sudo docker-compose up --build
```

to create and start the docker containers.

Expected Result:
Docker is able to build and start all containers without any errors. Within the logs of each container no errors are to be found.

Actual Result:

Step 2

Start pgAdmin 4, connect to the database with the user osmuser and run the following query in the query tool.

```
SELECT
        *
FROM
        osm_point p,
        osm_location l
WHERE
        p.osm_id = l.osm_id
        AND NOT(p.geom = l.geom);
```

Expected Result:
The query returns no results.

Actual Result:

Step 3

Run the following query in the query tool.

```
SELECT
        count(*)
FROM
        osm_polygon p,
        osm_location l
WHERE
        (abs(p.osm_id) = l.osm_id)
        AND NOT(st_centroid(p.geom) = l.geom);
```

Expected Result:
The query returns no results.

Actual Result:

Step 4

Run the following query in the query tool.

```
SELECT
        p.osm_id,
        l.id as calculated,
        l.osm_id as original
FROM
        osm_point p,
        osm_location l
WHERE
        p.osm_id = l.osm_id
LIMIT 1;
```

Expected Result:
There is only one row in the result set. The `original` is equals to `osm_id` and the `calculated` value equals

$$osm\_id * 10$$

Actual Result:

Step 5

Run the following query in the query tool.

```
SELECT
        p.osm_id,
        l.id as calculated,
        l.osm_id as original
FROM
        osm_polygon p,
        osm_location l
WHERE
        (abs(p.osm_id) = l.osm_id)
ORDER BY 1 ASC
LIMIT 1;
```

Expected Result:
There is only one row in the result set with a negative `osm_id`. The `original` value is equal to the absolute value of `osm_id` and the `calculated` value equals

$$osm\_id * 10 + 2$$

Actual Result:

Step 6

Run the following query in the query tool.

```
SELECT
        p.osm_id,
        l.id as calculated,
        l.osm_id as original
FROM
        osm_polygon p,
        osm_location l
WHERE
        (abs(p.osm_id) = l.osm_id)
ORDER BY 1 ASC
LIMIT 1;
```

Expected Result:
There is only one row in the result set with a positive `osm_id`. The `original` value is equal to `osm_id` and the `calculated` value equals

$$osm\_id * 10 + 1$$

Actual Result:

Test Result

Locations Within Range of Locations Function

This test verifies that the function `locations_within_range_of_locations` works properly.

Date:
Branch:
Commit:
Tester:

Steps

Step 1

Open a terminal and navigate to the Product folder of the repository and run the command

```
sudo docker-compose up --build
```

to create and start the docker containers.

Expected Result:
Docker is able to build and start all containers without any errors. Within the logs of each container no errors are to be found.

Actual Result:

Step 2

Start pgAdmin 4, connect to the database with the user osmuser and run the following query in the query tool.

```
SELECT *
FROM
        locations_within_range_of_locations(
                'shop',
                'supermarket',
                'amenity',
                'post_office',
                25
        );
```

Check the locations of the resulting geometries.

Expected Result:
All results are within 25 meter range of a post office.

Actual Result:

Step 3

Run the following query in the query tool.

```
SELECT *
FROM
        locations_within_range_of_locations(
                'shop',
                'supermarket',
                'amenity',
                '*',
                25
        );
```

Expected Result:
All results are within 25 meter range of an amenity location(bars, pubs, restaurants, parking, car wash...)

Actual Result:

Step 4

Run the following query in the query tool.

```
SELECT *
FROM
        locations_within_range_of_locations(
                'shop',
                '*',
                'amenity',
                'post_office',
                25
        );
```

Expected Result:
All results are within 25 meter range of a post office. All results have a tag with the key "shop" but not all of them have the value "supermarket".

Actual Result:

Step 5

Run the following query in the query tool.

```
SELECT *
FROM
        locations_within_range_of_locations(
                'shop',
                '*',
                'amenity',
                '*',
                25
        );
```

Expected Result:
All results are within 25 meter range of an amenity location (bars, pubs, restaurants, parking, car wash...). All results have a tag with the key "shop" but not all of them have the value "supermarket".

Actual Result:

Test Result

•

Updating the System

This test should verify, that the system automatically downloads OSM updates from Switzerlandand writes it into the database.

Date:
Branch:
Commit:
Tester:

Steps

Step 1

Open a browser and navigate to openstreetmap.com. Zoom to Switzerland and open the "Chronik"/"Changes" Tab. Keep that tab open for later comparison. If the last change is older than the timespan in the configuration file, this test may yield no valuable result. Additionally, check the configuration, that the replication url is set to openstreetmap.fr, because that is the minutely updates.

Expected Result:
Visible changeset.

Actual Result:

Step 2

As soon as the Osm2pgsql command is finished, start pgAdmin 4 and connect to the database with the user osmuser.

Expected Result:
The connection can be established, the tables osm_point, osm_line, osm_polygon and osm_boundary and a view osm_location are present.

Actual Result:

Step 3

This step needs to be done after the second Osm2pgsql command (write updates) is finished. If you have time in between the database writes, you can optionally compare the two results. Run the following command and check if the latest update is at the same time and date as the latest update on the openstreetmap.com tab.

```
SELECT * FROM latest_osm_change();
```

Expected Result:
the output matches the latest change in the openstreetmap.com tab.

Actual Result:

Test Result

Performance

This test should help monitor the performance cost of each component in the data flow.

Date:
Branch:
Commit:
Tester:
Countries:
Replication urls:

Steps

Step 1

Open a terminal and navigate to the Product folder of the repository and run the command

```
sudo docker-compose up --build
```

to create and start the docker containers.

Expected Result:
Docker is able to build and start all containers without any errors. Within the logs of each container no errors are to be found.

Actual Result:

Step 2

After every update is in the database, execute the log graph (gantt_for_logs.py) and save the graph.

Expected Result:
Graph that shows when each command was started and how much time it needed.

Actual Result:

Step 3

Analyze the most time consuming steps.

Expected Result:
The most time consuming steps should be the download or the osm2pgsql –create com-

mand. If you are working with smaller countries like monaco and get the updates from a minutely replication url the update step can be lengthy too.

Actual Result:

Test Result

GraphQL API

This test should verify, that the GraphQL API works.

Date:
Branch:
Commit:
Tester:

Steps

Step 1

Open a terminal and navigate to the Product folder of the repository and run the command

```
sudo docker-compose up --build
```

to create and start the docker containers.

Expected Result:
Docker is able to build and start all containers without any errors. Within the logs of each container no errors are to be found.

Actual Result:

Step 2

Open `http://localhost:5439/graphiql` in a web browser.
Expected Result:
The PostGraphiQL application is visible. In the explorer the following nodes are all present:

- allLastUpdates
- osmPoint
- osmPointByOsmId
- allOsmPoints
- osmLine
- osmLineByOsmId
- allOsmLines
- osmPolygon
- osmPolygonByOsmId

- allOsmPolygons
- osmBoundary
- osmBoundaryByOsmId
- allOsmBoundaries
- allOsmLocations
- planetOsmNode
- planetOsmNodeById
- allPlanetOsmNodes
- planetOsmWay
- planetOsmWayById
- allPlanetOsmWays
- planetOsmRel
- planetOsmRelById
- allPlanetOsmRels
- latestOsmChange
- locationsWithinRangeOfLocations
- stArea
- stAreaGeography
- stAsGeoJson
- stAsGeoJsonGeography
- stBuffer
- stBufferGeography
- stBufferGeographySeg
- stBufferSeg
- stCentroid
- stDWithin
- stDWithinGeography

- stDistance
- stDistanceGeography
- stIntersection
- stIntersectionGeography
- stIntersects
- stIntersectsGeography
- stTransform
- stTransformFromTo
- stTransformToText
- stTransformFromText
- stWithin

Actual Result:

Step 3

Run the following query.

```
query MyQuery {
  allOsmPoints {
    edges {
      node {
        name
        label
        geom {
          geojson
        }
      }
    }
  }
}
```

Expected Result:
The query returns statements. Copying the content of the `geojson` field into `geojson.io` and removing the backslash characters results in single point in Liechtenstein or Monaco or no more than 10 km away from the border of either of those country.

Actual Result:

Step 4

Add the following to the `QUERY VARIABLES` section.

```
{
  {
  "mygeom":{
    "type": "Polygon",
    "coordinates": [[
          [100, 0 ],
          [101, 0 ],
          [101, 1 ],
          [100, 1 ],
          [100, 0 ]
       ]]
     }
   }
}
```

Then run the following query.

```
query ($mygeom: GeoJSON){
  stCentroid(geom: $mygeom){
    geojson
  }
}
```

Expected Result:

```
{
  "data": {
    "stCentroid": {
      "geojson": "{\"type\":\"Point\",\"coordinates\":[100.5,0.5]}"
    }
  }
}
```

Actual Result:

Test Result