

Updatable Vector Tiles from OpenStreetMap

Lukas Martinelli, Manuel Roth

Free Vector Tiles from OpenStreetMap data
Bachelor Thesis, Spring 2016.

Lukas Martinelli, Manuel Roth: *Updatable Vector Tiles from OpenStreetMap* © Spring 2016

SUPERVISORS:

Prof. Stefan Keller

UNIVERSITY:

HSR University of Applied Science Rapperswil

DEPARTMENT:

Department of Computer Science

INSTITUTE:

Geometa Lab

LOCATION:

Rapperswil

TIME FRAME:

Spring 2016

LICENSE:

CC BY-SA 3.0 Unported

ABSTRACT

The *OSM2VectorTiles* project provides free, global and regularly updated vector tiles that can be used by anyone offline or on their own server to create state of the art maps. The vector tiles are prerendered and can be used completely offline providing new possibilities for web and mobile developers.

This bachelor thesis improves upon *OSM2VectorTiles* and adds support for prerendering the entire world, keeping the vector tiles up to date and improve quality significantly to become the best possible vector tile source from *OpenStreetMap*.

An open workflow keeping the vector tiles up to date with the millions of changes *OpenStreetMap* contributors add every day has been created and can be scaled across multiple hosts to produce vector tiles with global coverage. Due to the compatibility with Mapbox Streets it is possible for developers to switch their vector tile source to their own tile server avoiding vendor lock in and bringing down costs.

Since the vector tiles are available as download it is possible to create an offline version of OpenStreetMap for Desktop and Mobile. This will provide new possibilities for mobile developers that want to create applications that need a local map.

More information can be found on the project website <http://osm2vectortiles.org>.

MANAGEMENT SUMMARY

Situation

Digital mapping is moving towards vector tiles to create more interactive and resolution independent cartography. Instead of delivering the image of the map to the client, the vector representation of the data is sent to the map client.

Several vector tile providers such as Mapbox open the process to create vector tiles but still own the vector tile data. Developers and cartographers who want to be independent from 3rd party services or have limited internet access in their applications require free and global vector tiles to create the next generation of map applications.



Map rendered from *OSM2VectorTiles* vector tiles

Approach

The *OSM2VectorTiles* project strives to push mapping forward by providing free and high quality vector tiles with no strings attached. To make developers independent from the providers the vector tiles can be downloaded and hosted using a tileserver of their choice. This approach differs from all other vector tile providers and will enable new possibilities for desktop and mobile developers creating offline map applications.

The process to improve *OSM2VectorTiles* is completely open and the feedback and requirements of the community have been the cornerstone for improving the project with most feature requests and bug reports coming from actual users.

Result

The result of this thesis are downloadable vector tiles for the entire world and extracts for over 200 countries and 600 cities provided on the project website <http://osm2vectortiles.org>. The vector tiles are compatible with the newest Mapbox Streets v7 vector tiles which allows people to use their visual styles created with Mapbox Studio together with OSM2VectorTiles.

The workflow to create the vector tiles is available for everyone to use and is meant to be adapted by other projects. The workflow can be scaled linearly by adding more worker machines and is a unique approach to rendering offline vector tiles with global coverage.

The vector tiles can be kept up to data by continuously applying the latest *OpenStreetMap* changes and rerendering only the parts of the planet that are affected by those changes. This enables to provide downloadable vector tiles but still keep the data up to date.

The *OSM2VectorTiles* project is already used in several real world projects and by pushing the project further the result of this thesis is a living open source project that hopefully will survive this bachelor thesis.



Project website *OSM2VectorTiles* providing docs, downloads and examples

ACKNOWLEDGEMENTS

We want to thank the following people for their support and contributions to the thesis.

Prof Stefan Keller, IFS Institute for Software, for his support with regular meetings, contacts in the OSM community and time and effort in checking this thesis.

Imre Samu, *OpenStreetMap*, for the help in maintaining the project, advice on technical challenges and code contributions.

Petr Pridal, PhD, Klokantech GmbH, for initiating the project and donating the necessary cloud and CDN infrastructure for producing and hosting the vector tiles



KLOKAN TECHNOLOGIES

<http://www.klokantech.com/>

CONTENTS

i	TECHNICAL REPORT	1
1	INTRODUCTION	2
1.1	Vision	2
1.2	Problems	2
1.3	Preceding Study Thesis	3
1.4	Real World Usage Examples	3
2	THEORY	4
2.1	OpenStreetMap Data Model	4
2.2	Vector Tiles	5
2.3	Mapbox Vector Tile Specification	6
2.4	XYZ Coordinate Schema	7
3	DEFINING MAPBOX VECTOR TILES	8
3.1	Approach	8
3.2	Implementation	8
3.3	Problems and Optimizations	12
4	SCALABLE RENDERING PROCESS	16
4.1	Split Rendering Process into Jobs	16
4.2	Distributed Architecture	19
4.3	Merging Results	20
4.4	Save Space by removing identical subpyramids	20
5	UPDATABLE VECTOR TILES	22
5.1	OpenStreetMap Diff File	22
5.2	Diff Import	23
5.3	Track changes	24
5.4	Calculate changed tiles	25
6	RESULTS AND FUTURE	29
6.1	Results	29
6.2	Future	29
ii	PROJECT DOCUMENTATION	30
7	REQUIREMENTS SPECIFICATION	31
7.1	Use Cases	31
7.2	User Characteristics	31
7.3	Requirements	31
7.4	Non Functional Requirements	31
8	ARCHITECTURE AND DESIGN	33
8.1	Architecture	33
8.2	Database and Layer Schema	39
9	PROJECT MANAGEMENT	52
9.1	Software Development Process	52
9.2	Schedule	52
9.3	Milestones	53
9.4	Roles and Responsibilities	53

9.5 Risks	53
10 QUALITY MEASURES	54
10.1 Testing	54
10.2 Guidelines	56
11 PROJECT MONITORING	57
11.1 Code Statistics	57
11.2 Estimated Time vs Actual Time	57
11.3 Time per Person	58
12 DEVELOPER AND USER DOCUMENTATION	59
iii APPENDIX	61
BIBLIOGRAPHY	67

Part I

TECHNICAL REPORT

INTRODUCTION

1.1 VISION

Vector tiles are the future of web and mobile mapping. The next generation of maps is only possible with free and open vector tile sources. The goal of *OSM2VectorTiles* is to push mapping forward by providing vector tiles that are produced using an open process, completely free of charge and can be used offline. Encouraged by the existing real users of the project further improvements need to be done to meet the requirements of developers and cartographers using the project to create custom OSM maps without building up their own rendering pipeline.

1.2 PROBLEMS

The focus of this bachelor thesis lies on three major problems that need to be solved.

Defining Mapbox Vector Tiles (Chapter 3)

The vector tiles need to meet certain cartographic standards to enable cartographers to create high quality maps. Creating a global base map from scratch is a huge undertaking with several interesting problems like label placement, importance ranking and fitting the right data into less space. Focusing on quality and compatibility with Mapbox Streets v7 makes the project truly usable for the end users which expect a high quality map and lets existing users switch over to *OSM2VectorTiles* more easily.

Scalable rendering process (Chapter 4)

The global vector tiles should be rendered within a reasonable time-frame to meet project deadlines and enable developers to iterate quickly on the vector tiles. The sheer amount of tiles makes it impossible for a single process to render the entire planet. By distributing the process on multiple machines the time for rendering the planet can be significantly reduced only limited by the amount of infrastructure available. The solution to this problem will serve as example how to distribute a tile rendering pipeline and enabling vector tiles with global coverage.

Updatable vector tiles (Chapter 5)

OpenStreetMap contributors add up to three million nodes every day[7]. Keeping a map up to date is of significant relevance to the users of the vector tiles and the contributors. The vector tiles should be released in a regular interval. However rerendering the entire planet using the scalable rendering process is not feasible due to the infrastructure costs. By calculating the tiles that will change in advance and only render those tiles a single machine can keep the vector tiles up to date making the project sustainable for long term.

1.3 PRECEDING STUDY THESIS

The *OSM2VectorTiles* project originated from the preceding study thesis in the fall semester of 2015 with the same vision as the bachelor thesis to allow anyone to create custom *OpenStreetMap* maps without managing complex infrastructure.

RESULTS OF STUDY THESIS

- Workflow for generating vector tiles (not meant to scale for rendering entire planet)
- Mapbox Streets v5 compatible vector tiles of Switzerland

The scope of the study thesis was to create a repeatable workflow for generating vector tiles based on *OpenStreetMap* data. The study thesis focus lied on solving the problem of creating a basemap and implementing Mapbox Streets v5 at small scale ([Chapter 3](#)) and delivered prerendered vector tile data for Switzerland as result.

This workflow was intended to run on a single machine and was not meant to scale for rendering the entire planet ([Chapter 4](#)) or keeping the vector tiles up to date ([Chapter 4](#)).

1.4 REAL WORLD USAGE EXAMPLES

OSM2VectorTiles is already used in real projects which confirms the demand for such a project. The examples show the potential use cases and customers of prerendered vector tiles.

- MapHub.net allows you to create interactive customizable maps to organize your own geo-data. It is using *OSM2VectorTiles* based basemaps to provide a variety of basemaps to choose from for it's users.
- The GeoPortal of Mecklenburg County GIS is using *OSM2VectorTiles* in combination with custom data as a basemap to present important information to citizens.
- The Helsinki Regional Transport Authority (HSL) is using *OSM2VectorTiles* as source with a custom style to provide map services to other developers as part of the Digitransit platform.

THEORY

This chapter describes the underlying concepts which are used throughout the thesis.

2.1 OPENSTREETMAP DATA MODEL

The data model of *OpenStreetMap* consists of objects of type node, way and relation. Nodes define points in space, ways define linear features and relations define how objects relate to each other[6].

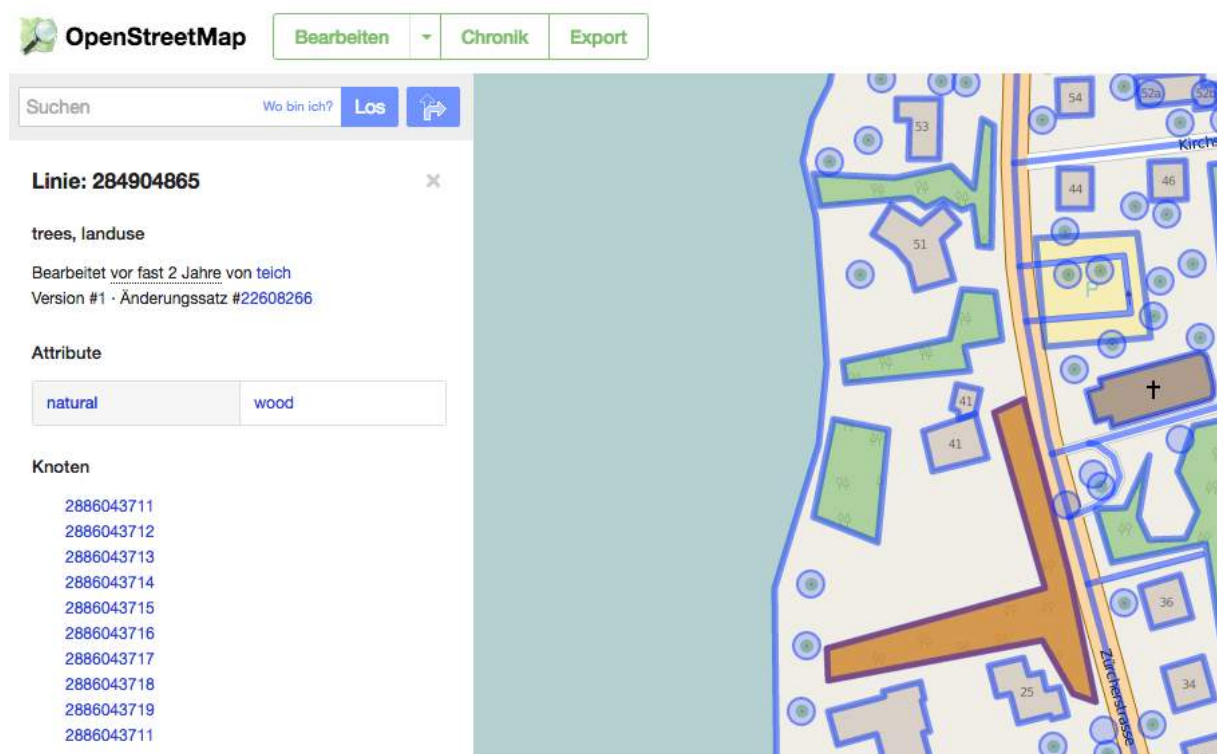


Figure 1: Example of object tagged natural=wood

Every object can have one or more tags associated with it. Tags define the meaning of a certain object. A tag consists of a key/value pair for example the tag **natural=wood** is used to define areas which are covered in trees. *OpenStreetMap* tags are not strictly defined and users can add their own tags but there are many conventions which should be followed.

The semi-structured *OpenStreetMap* data model is transformed a structured database schema as explained in [Section 3.2.1](#). Nodes, ways and relations are converted into real geometries. A way for example is turned into a polygon because it is a closed way where the first and last node are shared[4].

2.2 VECTOR TILES

Instead of delivering a image of the map to a client like a browser or mobile phone, only the vector representation of the data is sent to the client which is using less data and allows more interactive, dynamic and resolution independent cartography. This is only possible since clients have more powerful hardware and are able to render maps themselves.

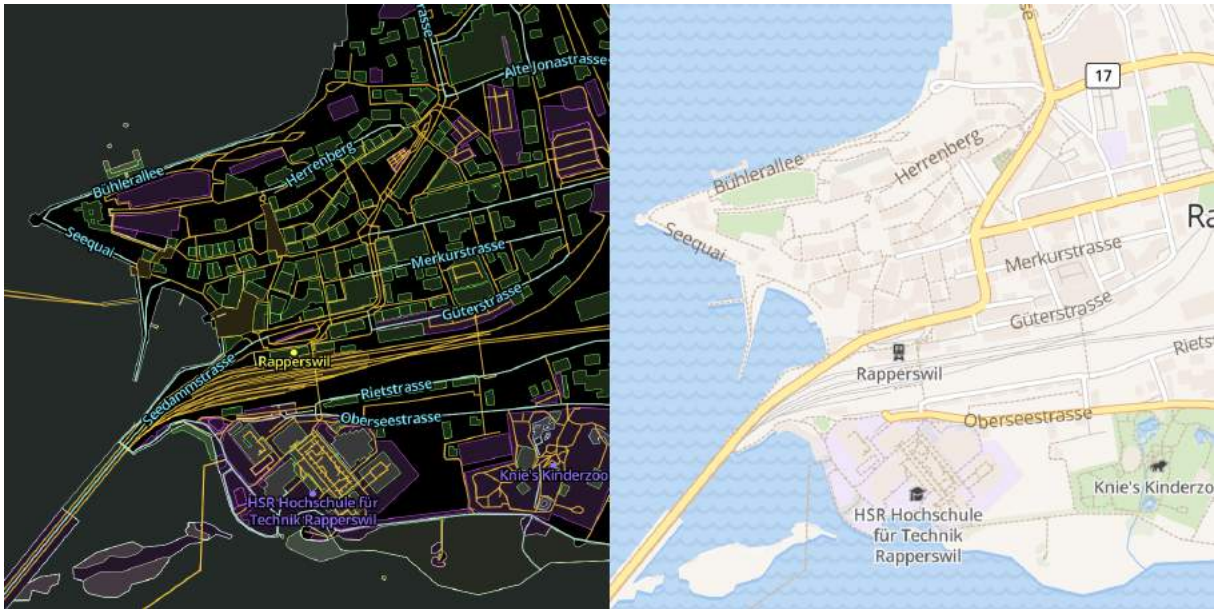


Figure 2: Vector and client representation of a map section

Google introduced the XYZ tiling scheme [8] back in 2005 because it is not scalable to deliver an image of the map tailored to the viewport of the client. The idea is to divide the map image into a grid as shown in Figure 3 where clients request idempotent raster images by using tile indices instead of coordinates. This allows caching on the browser and server side and results in a smoother map experience. The same approach can be applied to vector data. Instead of delivering the vector data for the entire viewport the vector data is sliced into tiles.

Vector tiles contain all the geometry and metadata needed for a specific tile. This makes them more flexible than serving raster tiles, because a different style can be applied on the fly. For example based on the language preference of the browser language-specific city labels can be shown.



Figure 3: Tiled raster map

2.3 MAPBOX VECTOR TILE SPECIFICATION

The Mapbox Vector Tile Specification defines how to encode tiled vector data using Protocol Buffers[14]. More details about the encoding and internal structure of vector tiles can be found in the specification[18].

The data inside of vector tiles is structured into layers and features. A vector tile can have multiple layers such as roads, landuse or water (elaborated in [Section 8.2](#)). Each layer consists of one or multiple features and must have a unique name. A feature contains a geometry field (either of type point, linestring or polygon) and metadata (tags) such as label translations. The metadata key-value pairs are stored in the layer as arrays of keys and values. The tags on the feature only reference the key and value by their index. This helps to keep the file size small since only the unique values and keys need to be stored and can be referenced by multiple features. Keys can only be strings while values can have different types such as string, double or integer.

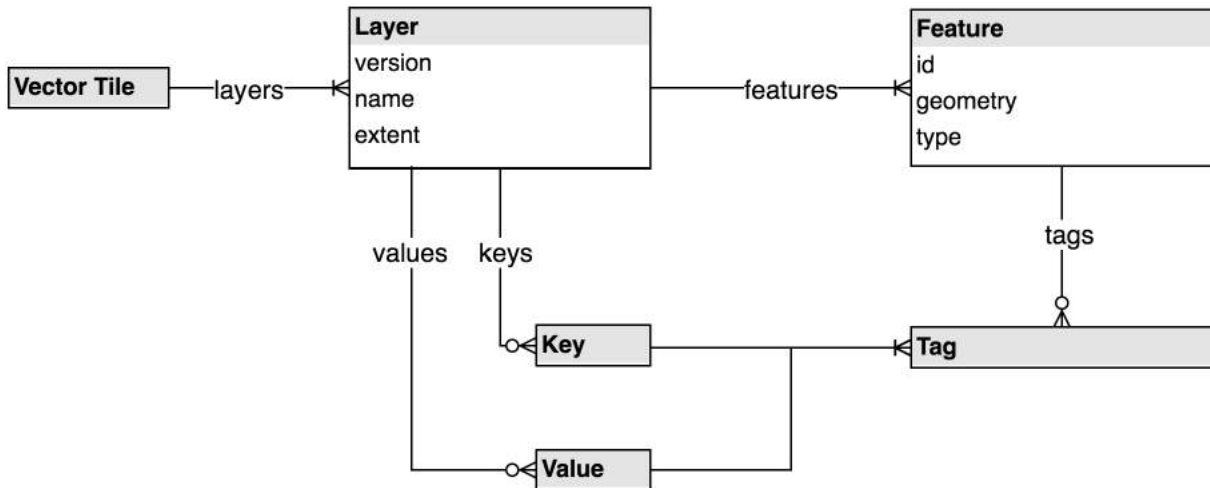


Figure 4: Vector Tile Structure

The **extent** field defines the width and height of the tile coordinates (the vector tile resolution) which is relevant for encoding the geometry of a feature as shown in [Figure 5](#). For *OSM2VectorTiles* the resolution of 4096 coordinate units has been chosen.

The geographic coordinates of the geometries are converted into relative coordinates inside the vector tile. Vector tiles are encoded as commands for a virtual pen (the rendering client). Drawing the polygon from Figure 5 results in the commands from Listing 1.

1. Move to the starting point (3,1) relative to the top left corner
2. Draw line from current pen position to (3,3). Since the target position is encoded relative to the current position the encoded geometries take up less space.
3. Draw line from current position (6,4) to (-4,2)
4. Close path of current position (2,6) with last used starting point (3,1)

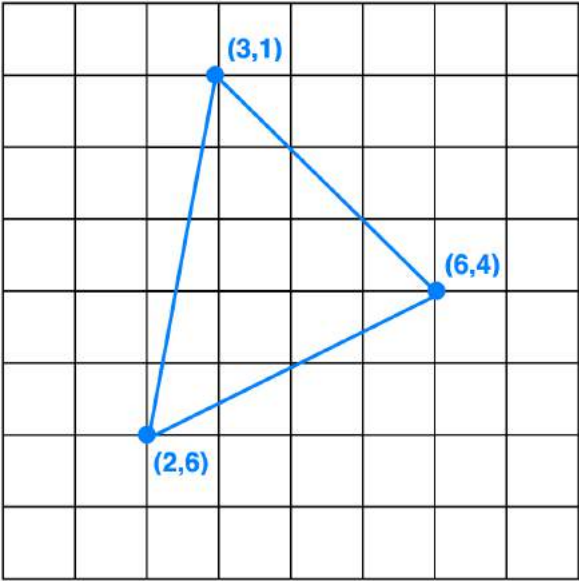


Figure 5: Vector tile grid with encoded geometry

```
LineTo(3,1)
LineTo(3,3)
LineTo(-4,2)
ClosePath()
```

Listing 1: Geometry inside vector tile encoded as drawing commands

2.4 XYZ COORDINATE SCHEMA

For tiling the vector tiles the XYZ numbering schema has been used. The tiles are organized in a 3-dimensional coordinate system (x/y/z) where x and y represent the axes and z the zoom level. As users zoom into a map each tile is replaced by four children within the tile.

The map in mercator projection is divided into the x and y axis. The x axis reaches from 0 to 2^z (from left to right edge of map) and the y axis from 0 to 2^z (from top to bottom edge of map).

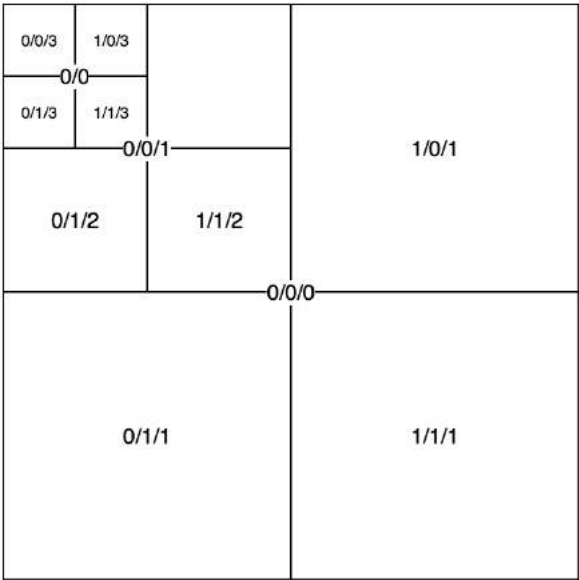


Figure 6: XYZ coordinate schema

DEFINING MAPBOX VECTOR TILES

The main requirement regarding the content of the vector tiles is to be compatible with the vector tiles of Mapbox. This allows people to seamlessly switch to *OSM2VectorTiles* and use the same visual styles created with Mapbox Studio.

3.1 APPROACH

Mapbox's tileset is called Mapbox Streets. Mapbox provides detailed documentation on what data is included in the Mapbox Streets vector tiles. The documentation contains a layer reference which defines the attributes a layer can have. However the zoom levels at which data is shown is not documented publicly as well as the *OpenStreetMap* tags and constraints describing the data. To be able to reverse engineer Mapbox Streets this information had to be retrieved by analyzing the official Mapbox Streets vector tiles at different zoom levels.

In an iterative and time consuming process the mapping and queries were continuously improved until the vector tile output matches the data from Mapbox Streets very closely.

3.2 IMPLEMENTATION

This section describes the main components which needed to be implemented in order to generate Mapbox Streets compatible vector tiles.

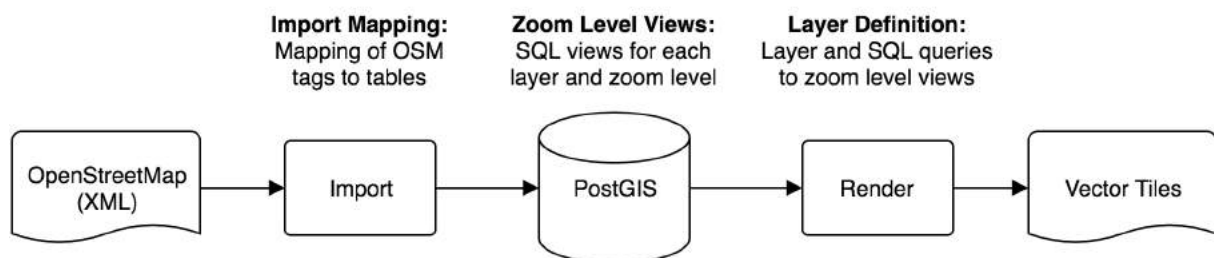


Figure 7: Simplified process from data import to vector tile rendering

3.2.1 Import Mapping

Imposm3[16] is used to import the *OpenStreetMap* data into the PostgreSQL database. Imposm3 satisfies the two purposes of filtering and mapping the *OpenStreetMap* data.

FILTER DATA The mapping allows to filter explicitly by tags and defines which data is imported. This is important since only a subset of all *OpenStreetMap* data is included in the vector tiles and therefore not all *OpenStreetMap* data needs to be imported.

MAPPING DATA The mapping allows to map *OpenStreetMap* key/value pairs to a certain database table creating a structured and organized schema from semi-structured data. It takes care of constructing actual geometries from the *OpenStreetMap* data model (Section 2.1).

The example definition in listing 2 maps *OpenStreetMap* tags with the key **building** and any value associated with that key into the table **building_polygon**. The **building_polygon** table has the columns **id**, **geometry**, **underground**, **timestamp** and **type**. The **underground** column in the database originates from the normalized value associated with the *OpenStreetMap* key **building:levels:underground**.

During the import process Imposm3 transforms the *OpenStreetMap* nodes, ways and relations to one of the geometry types point, linestring or polygon. The mapping is one of the most important aspects of the project and maps more than 400 individual tags.

```
building_polygon:
  type: polygon
  fields:
    - name: id
      type: id
    - name: geometry
      type: geometry
    - name: underground
      key: building:levels:underground
      type: integer
    - name: timestamp
      type: pbf_timestamp
  mapping:
    building:
      - __any__
```

Listing 2: YAML definition of a single table in the import mapping

3.2.2 Zoom Level Views

After the import process all *OpenStreetMap* data which is required for the vector tiles is stored in the database. Since only a subset of the data in the tables is shown on a given zoom level, SQL views for each zoom level and layer were created. The zoom level views filter the tables to the rows which are shown on a given zoom level.



Figure 8: Difference of zoom level view 13 and 14 in buildings layer

Figure 8 shows on the left side the building layer on zoom level 13 and on the right side the same layer on zoom level 14. On zoom level 13 only a subset of the data shown on zoom level 14 is visible. This is a classic example of using zoom level views to provide more details on higher zoom levels.

Listing 3 shows the definition of the SQL views on both zoom levels. The **WHERE** clause in the query of the view **building_z13** filters the rows to buildings which have an area greater than 1700. That is the reason why less buildings are shown on right side of Figure 8.

```
CREATE OR REPLACE VIEW building_z13 AS
  SELECT id AS osm_id, underground, geometry
  FROM osm_building_polygon
  WHERE ST_Area(geometry) > 1700;

CREATE OR REPLACE VIEW building_z14 AS
  SELECT id AS osm_id, underground, geometry
  FROM osm_building_polygon;
```

Listing 3: Definition of zoom level views of building layer

Additionally zoom level views help to decouple the database tables which hold the actual data and the definition of the layer. This is very helpful for example if new data is added to a layer, as only the import mapping and the zoom level views need to be modified.

3.2.3 Layer Definition

The source project contains the definition of the layers inside the vector tiles. The definition contains metadata to access the database and a query which returns the necessary data for this layer. The listing 4 shows the definition of the layer **building**. The query does not directly access the database table **building_polygon**. Instead it queries the zoom level views **building_z13** and **building_z14**.

```
- id: aeroway
  Datasource:
    type: postgis
    table: |-
      (
        SELECT osm_ids2mbid(osm_id, is_polygon(geometry)) AS osm_id,
              geometry, building_is_underground(underground) AS underground
        FROM (
          SELECT * FROM building_z13
          WHERE z(!scale_denominator!) = 13
          UNION ALL
          SELECT * FROM building_z14
          WHERE z(!scale_denominator!) = 14
        ) AS building WHERE geometry && !bbox!
      ) AS data
  properties:
    "buffer-size": 4
```

Listing 4: Definition of layer aeroway in the vector tile source project

The layer definition serves as input to the vector tile renderer (Mapnik). The tile renderer will execute every layer query for each tile and replaces expressions like **!scale_denominator!** (zoom level) and **!bbox!** (extent of the tile) with the values of the current tile. If the query in listing 4 is executed for a tile on zoom level 12 it won't return any data as the **WHERE** clause will not match in both cases. Whereas if it is executed on a tile on zoom level 13 all data of the zoom level view **building_z13** will be included in the layer building.



Figure 9: Layer road, water, and building features on same map

3.2.4 Relation between Database Tables, Zoom Level Views and Layers

The [Figure 10](#) shows how the database tables, zoom level views and layers are related to each other. This architecture helps to structure the *OpenStreetMap* data inside the database and opens the possibility to optimize single zoom levels individually. The arrow in [Figure 10](#) describes the data flow.

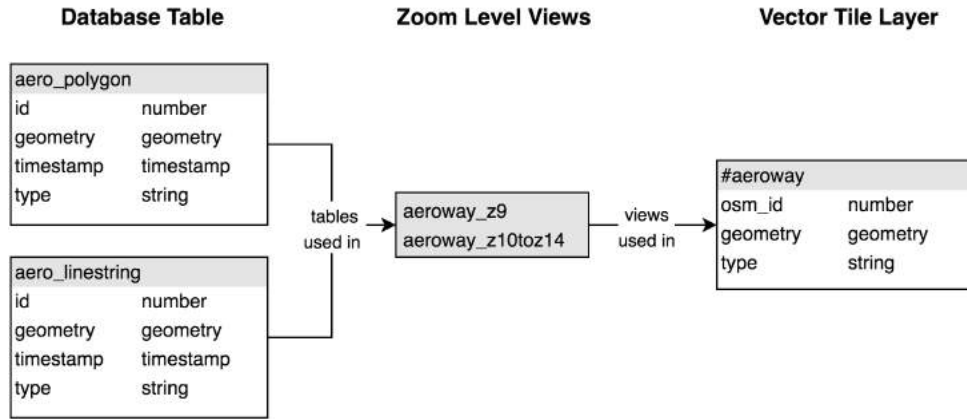


Figure 10: Data flow between database tables, zoom level views and layer

3.3 PROBLEMS AND OPTIMIZATIONS

During the development process of the map a number of problems were discovered and optimizations were implemented. This section explains the most interesting problems in detail.

3.3.1 Avoid Expensive Transformations in Zoom Level Views

The purpose of the zoom level views is to filter the data to only contain rows that are shown on a specific zoom level. Expensive calculations or transformations should be avoided in these views or made in a preprocessing step.

For example there are multiple label layers which transform a polygon geometry to a point geometry and since calculating the centroid from a polygon is expensive, transformations like these result in bad rendering performance.

The reason for this is that every time the layer query gets executed, all rows of the view get transformed even though only a subset of the data is needed for the rendered tile ([Figure 12](#)). Therefore selecting the right rows and only executing the transformation on the smallest possible subset results in much better performance ([Figure 11](#)).

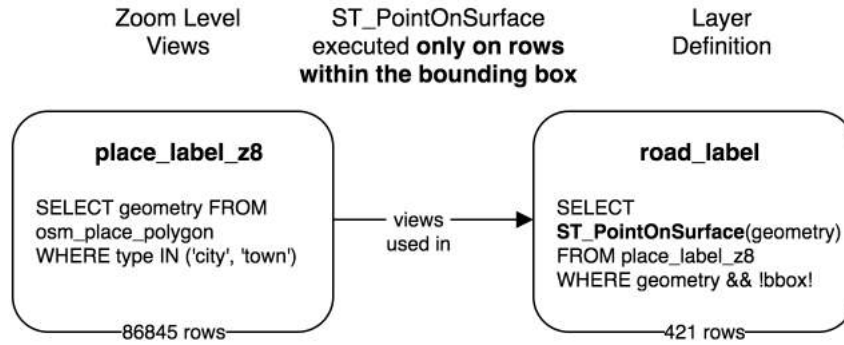


Figure 11: Point calculation in layer definition

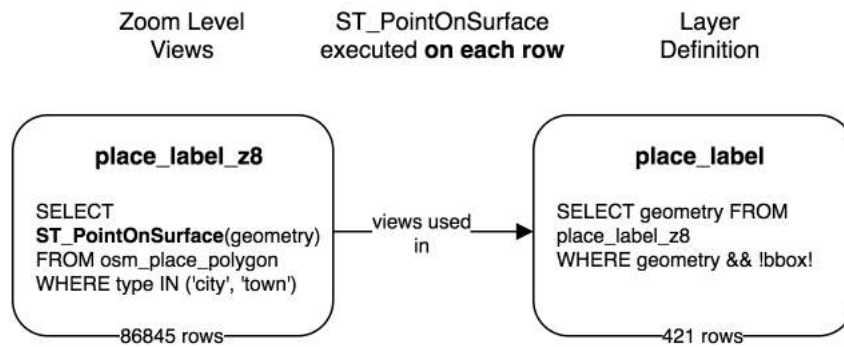


Figure 12: Point calculation in zoom level view

3.3.2 Classification Functions

OpenStreetMap contains a complex taxonomy of tags mapping to real-world objects. The tags are structured into layers but cartographers need another level of abstraction to style groups of features. For example designers want to set the color of all parks to green instead of styling tags such as dog parks, gardens and playground individually. Therefore closely related tags are grouped together and are stored in the **class** field. This process is called classification and is essentially a mapping of many tag values to a single value.

These classifications cover more than 400 individual tags grouped into more than 100 groups and are implemented by SQL functions as in listing 5. These class definitions are written in a YAML based format from which the SQL classification functions are generated.

```

CREATE OR REPLACE FUNCTION landuse_class(type VARCHAR) RETURNS VARCHAR
AS $$
BEGIN
    RETURN CASE
        WHEN type IN ('park', 'dog_park', 'garden', 'playground') THEN 'park'
        WHEN type IN ('school', 'college', 'university') THEN 'school'
        WHEN type IN ('cemetery', 'christian', 'jewish') THEN 'cemetery'
    END;
END;
$$ LANGUAGE plpgsql IMMUTABLE;

```

Listing 5: Definition of classification helper function

The listing 5 shows the simplified class function of the layer *landuse*. It takes the type value as input and returns the correct class value.

3.3.3 *OpenStreetMap* ID Transformation

The data model of *OpenStreetMap* consists of nodes, ways and relations. Every object gets its own OSM id assigned. This id is not unique across object types. Therefore one can find three objects with the same id but with a different object type.

While this works perfectly fine for *OpenStreetMap*, this represents a problem because during the import process these *OpenStreetMap* objects get transformed to PostGIS geometries. Objects of different types can get transformed to the same PostGIS geometry and therefore their ids would collide.

In order to prevent this, the ids need to be transformed according to Table 1 to make OSM ids unique within vector tiles [17].

OSM type	Geometry type	<i>OpenStreetMap</i> ID transform
node	point	$\text{id} \times 10$
way	linestring	$(\text{id} \times 10) + 1$
way	polygon + polygon label points	$(\text{id} \times 10) + 2$
relation	linestring	$(\text{id} \times 10) + 3$
relation	polygon + polygon label points	$(\text{id} \times 10) + 4$

Table 1: OSM id transformation

3.3.4 *Place Label Rank Calculation*

Ranks are important for determining at which zoom level which places should be displayed. The NaturalEarth database contains places with scaleranks assigned by humans and is the most important source for better quality labels (historic places might be much more important despite having a very small population). This dataset is merged with the imported *OpenStreetMap* data. They can also be used to limit density at lower zoom levels to decrease data density. Using the scalerank and other information such as place type and population the actual rank called **localrank** is calculated. An example of localrank calculation can be seen in figure Figure 13 (localrank is denoted as number inside circle).

Algorithm

1. Divide map into grid
2. Group labels by tile index
3. Sort labels by scalerank, type and population within group
 - a) By scalerank ascending
 - b) By type city, town, village, hamlet, suburb, neighbourhood
 - c) By population descending
4. Use the row number of the sorted labels as **localrank** for each feature

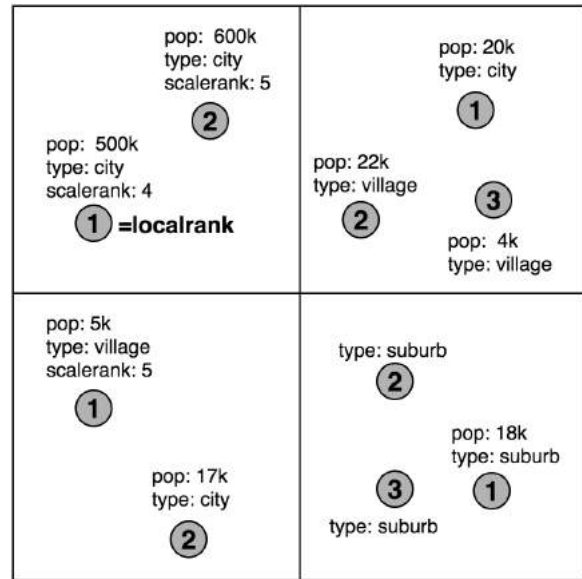


Figure 13: Localrank calculation

SCALABLE RENDERING PROCESS

Rendering the planet file from zoom level 0 to zoom level 14 requires $\sum_{i=1}^{14} 4^i = 357\,913\,941$ tiles to be rendered. Rendering this amount of tiles serially is no longer possible due to two reasons.

- The rendering process might fail and progress is lost
- A single worker process takes approximately 276 days to render the planet with a throughput of 54 000 tiles per hour

To solve these problems two measurements described in [Section 4.1](#) and [Section 4.2](#) have been taken.

4.1 SPLIT RENDERING PROCESS INTO JOBS

The rendering process (Mapnik) renders tiles within the given bounding box into a SQLite database (MBTiles). To adapt the process the global bounding box needs to be divided into many smaller bounding boxes and the many small SQLite databases need to be merged together into a large planet SQLite database at the end of the process. The unit of a job therefore is a bounding box derived from a XYZ tile index (pyramid job [Section 4.1.1](#)) or a list of tiles (list job [Section 4.1.2](#)).

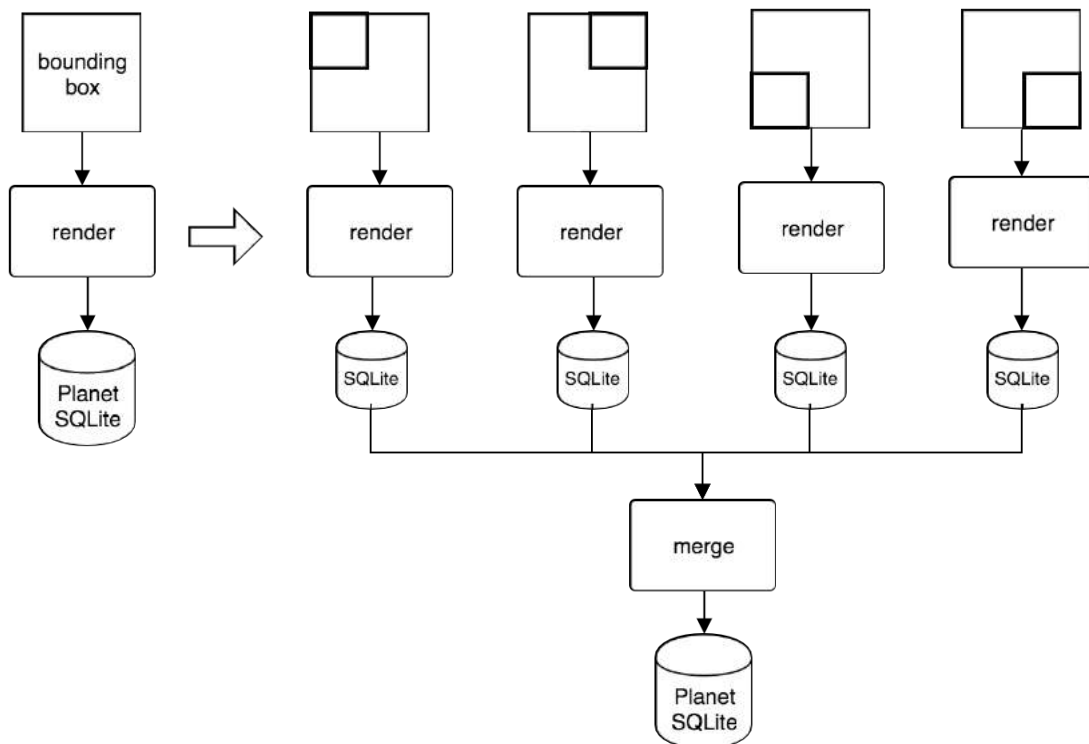


Figure 14: Adapt rendering process to divide work and merge it back together

4.1.1 Pyramid Job

To divide the work of rendering the planet into equal parts across the world, the XYZ coordinate schema described in [Section 2.4](#) is used to divide the planet into several subpyramids. A pyramid job means that the worker needs to not only render the given tile index but also all descendant tiles down to the maximum zoom level (the entire tile subpyramid).

Algorithm

1. Choose job zoom level z and maximum zoom level Z
2. Calculate the 4^z tiles for job zoom level
3. Convert XYZ tile index into a WGS84 bounding box
4. Render bounding box from zoom level z down to the maximum zoom level Z

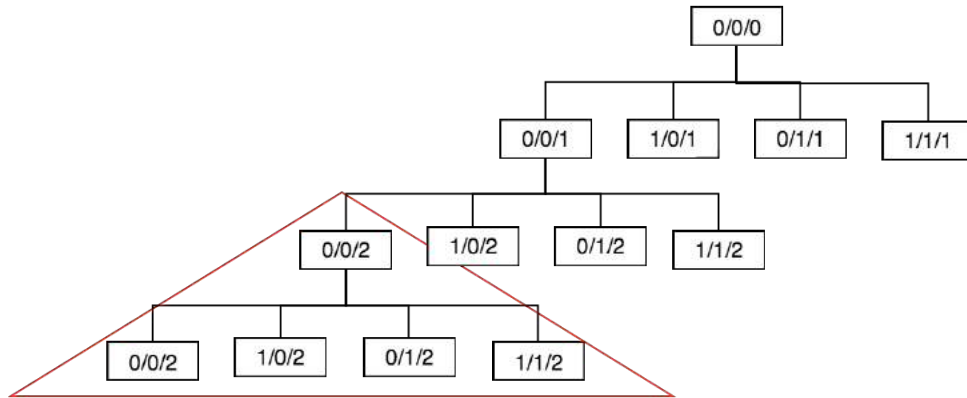


Figure 15: Pyramid job of a zoom level 2 tile and the descendants

Example

Given the job zoom level $z = 8$ and the max zoom level $Z = 14$ the planet is divided into 4^8 jobs. This means each subpyramid task consists of rendering all descendant tiles from a tile at zoom level 8. Each job therefore consists of $\sum_{i=0}^6 4^i = 5461$ tiles that need to be rendered.

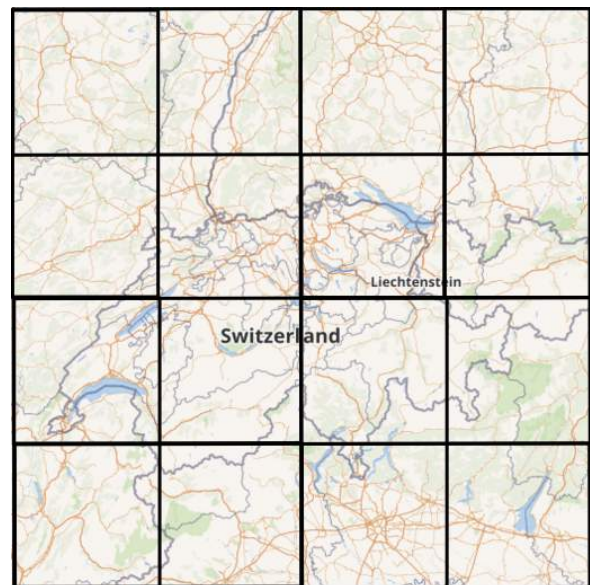


Figure 16: Map divided into tasks at job zoom level 8

4.1.2 List Job

It is important to be able to update distinct tiles to fix bugs later on or rerender all changed geometries. A list job is a batch job of tiles grouped together by their proximity and is created from a large list of tiles.

Each week a list of changed tile indizes is produced by the changed tiles detection (20 000 000 to 30 000 000 tiles). This list must be split into batch jobs where each job contains several thousand tiles that need to be rendered. Since PostgreSQL has been configured to cluster rows together by the tile index of the geometry column it is possible to gain additional performance by grouping the tiles that are close to each other into a batch job.

Algorithm

To group tiles that are in close proximity in one batch job the tiles are sorted lexically by their Quadkey[19]. Due to the properties of the Quadkey numbering scheme this results in the tiles being sorted by their parents indizes.

1. Given a large list of tiles
2. Calculate Quadkey of XYZ tile index
3. Sort lexically by Quadkey
4. Split list into sublists of batch size

XYZ Index	Quad Key
8/175/48	10321111
8/48/103	02310222
8/76/60	01223300
8/154/130	30011030
8/205/31	11023323
8/128/93	12022202
8/63/33	00311113
8/86/117	03230312
8/160/119	12320222
8/246/84	13130310

Table 2: List of changed tiles indizes

XYZ Index	Quad Key
8/63/33	00311113
8/76/60	01223300
8/48/103	02310222
8/86/117	03230312
8/175/48	10321111
8/205/31	11023323
8/128/93	12022202
8/160/119	12320222
8/246/84	13130310

Table 3: List of tile indizes split into batches

Quadkey

The Quadkey has several unique properties which makes it ideal for grouping tiles together.

1. Length indicates level of detail
2. Quadkey starts with Quadkey of parent tile

These properties make it possible to simply sort lexically by the Quadkey of a tile to group tiles together in batches.

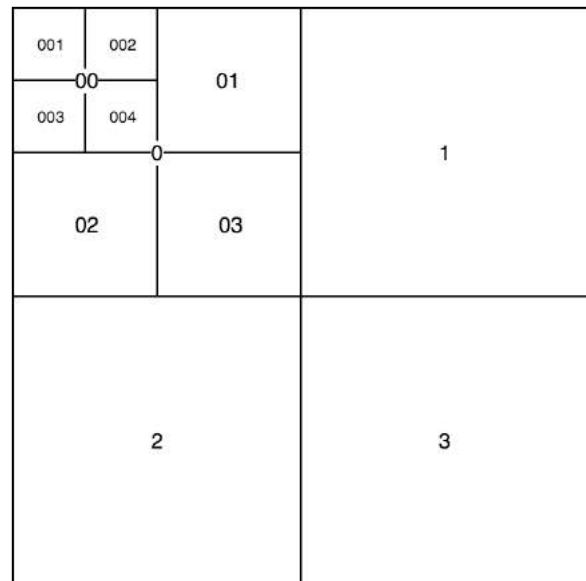


Figure 17: Quadkey indexing

4.2 DISTRIBUTED ARCHITECTURE

To distribute across several hosts a distributed architecture (Figure 18) using job queues has been implemented.

1. Pyramid or list jobs are created by generate-jobs and put into the jobs queue
2. The different worker processes on different hosts poll the **jobs** queue for new jobs and try to render them in the given time frame.
3. If the rendering does not complete in the given time frame it is put into the failed-jobs queue.
4. The resulting SQLite database is uploaded to a S3 compatible object store and linked in the result message which is stored in the **results** queue.

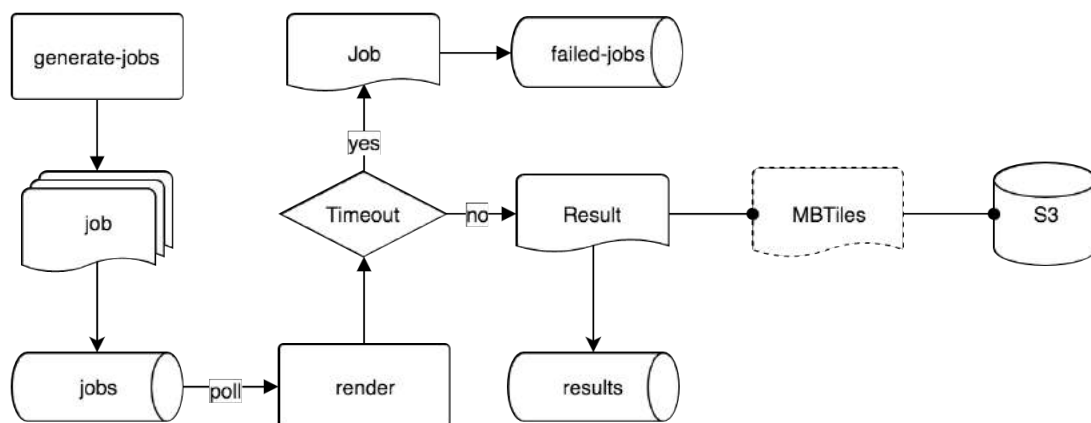


Figure 18: Distributed rendering architecture using message queues

Dealing with Errors

The message queue is using an acknowledge mechanism together with durable queues. This means if any process fails at any stage in the workflow (**render** or **merge**) the message is requeued and redelivered to the next worker. Since there are always some jobs that never complete or have very distinct problems the timeout prevents the workers from being jammed by the same failing jobs over and over again. Failed jobs can be inspected and rescheduled at a later point in time.

4.3 MERGING RESULTS

A very important part of the workflow is merging all of the 65 536 SQLite databases. The **merge-jobs** process downloads the linked SQLite database in the result message and then merges it into the specified merge target (e.g. the Planet MBTiles file).

1. Message is consumed from the **results** queue.
2. The linked SQLite database is downloaded from S3.
3. The downloaded SQLite database is attached to the merge target.
4. The data tables **map** containing the tile indices and **images** containing the actual PBF data are copied over replacing the already existing entries in the database.

```
ATTACH DATABASE 'source.mbtiles' AS source;
REPLACE INTO map SELECT * FROM source.map;
REPLACE INTO images SELECT * FROM source.images;
```

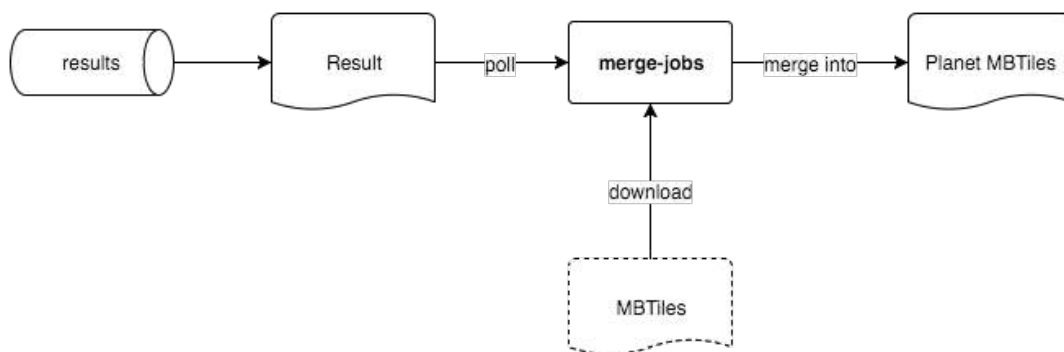


Figure 19: Merge completed MBTiles files together

This is a very fast way of distributing updates to an MBTiles file which can be applied with any SQLite client and performs for many small databases merged into one large merge target.

4.4 SAVE SPACE BY REMOVING IDENTICAL SUBPYRAMIDS

Water tiles are a large part of the resulting planet file since most of the earth is covered in water. If a tile only contains water it is not desirable to store the same water geometry on all zoom levels from z8 down to z14 (resulting in $\sum_{i=0}^6 4^i = 5461$ tiles all containing the same geometry). To prevent this issue all z8 subpyramids containing the same data on all descendant tiles are removed.

Algorithm

1. Calculate all descendant tiles of a given parent tile at the zoom level **maskLevel**
2. Calculate SHA1 checksum for each tile
3. Count occurrences of each unique checksum
4. Ensure there is only one checksum used in all descendants
5. If checksum matches parent tile checksum remove all descendants

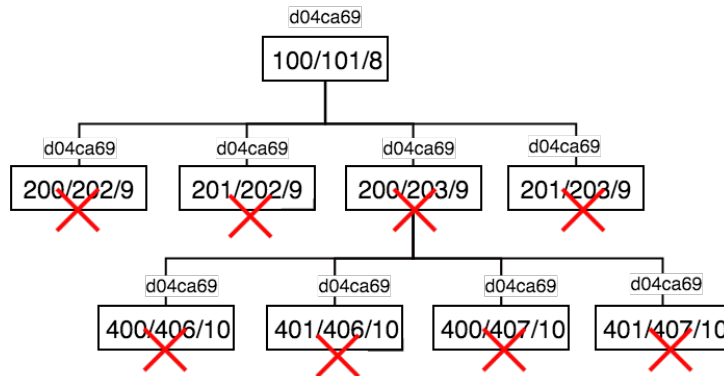


Figure 20: A z8 subpyramid with the same data hash in all descendant tiles

Tile Server Implementation

To support removed subpyramids in tile server implementations need to be able to backfill missing data in higher zoom levels with data from lower zoom levels.

1. Serve vector data of requested tile if exists
2. If tile not exists calculate parent tile at zoom level **maskLevel**
3. Serve vector data from parent tile

UPDATABLE VECTOR TILES

OpenStreetMap contributors add more than three million nodes and ways every day. In order to keep the prerendered tiles up to date this poses a challenge of looking at the changes and figuring out which tiles are affected by those changes and schedule them for rerendering.

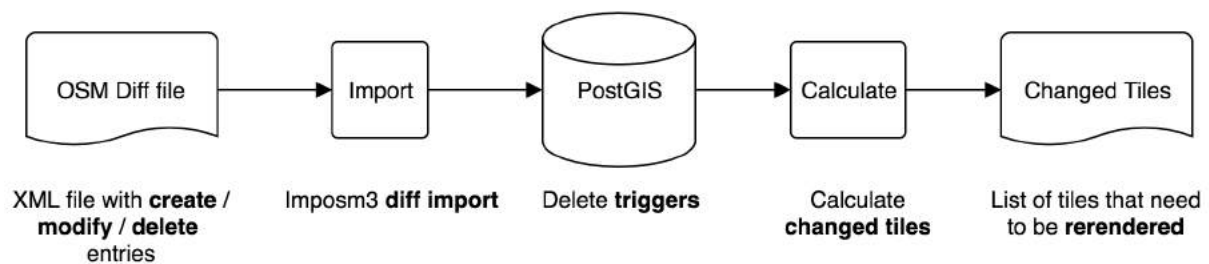


Figure 21: Simplified diagram of changed tiles detection process

Figuring out which tiles are affected by updates requires a multi-step process.

1. Find relevant *OpenStreetMap* objects affected by the changes
2. Calculate tiles covered by the object geometries

TIME CONSTRAINT In order to be able to keep up with rendering changes the update process can not take longer than the number of days the database is behind.

5.1 OPENSTREETMAP DIFF FILE

OpenStreetMap provides a single XML file, which contains every mapped object. Since the process of importing is time and resource consuming it is not feasible to redo this process to keep up with all the changes.

Therefore OSM additionally provides hourly or daily diff files in the `0smChange` format which contains the created, modified and deleted objects over a period of time. Importing only the subsequent diff files after an initial import allows to keep the database in sync with the latest changes.

Example

The listing 6 shows an example diff file which contains a create, modify and delete entry for different objects.

```

<?xml version='1.0' encoding='UTF-8'?>
<osmChange version="0.6" generator="Osmosis 0.43.1" timestamp="2016-05-20T11:32:32Z">
  <create>
    <node id="4196907493" version="1" uid="1" lat="46.9280366" lon="7.1163806">
      <tag k="amenity" v="pharmacy"/>
      <tag k="name" v="Amavita"/>
    </node>
  </create>
  <modify>
    <node id="4051684660" version="2" uid="1" lat="53.5705074" lon="9.9950888">
      <tag k="emergency" v="fire_hydrant"/>
      <tag k="ref" v="13874"/>
    </node>
  </modify>
  <delete>
    <node id="1044604768" version="2" uid="1" lat="52.6429848" lon="5.082264"/>
  </delete>
</osmChange>

```

Listing 6: Create, modify and delete example

Different Change scenarios

CREATE OBJECT A new object is created (e.g. a new point of interest is added). All tiles covered by the created object must be rerendered.

DELETE OBJECT An object gets deleted (e.g. an old house which does not exist anymore). All tiles covered by the deleted object must be rerendered.

MODIFY OBJECT An object gets modified (e.g. add additional translations to a place). All tiles covered by the modified object must be rerendered.

MOVE OBJECT An object is moved (e.g. an incorrectly mapped bus stop had to be repaired or has been moved in the real world). All tiles covered by the original object and all tiles covered by the moved object must be rerendered.

5.2 DIFF IMPORT

Additionally to importing the regular OSM Planet file, `imposm3` supports importing OSM Diff files. `Imposm3` will read the Diff file and calculate with the local cache of previous imports which nodes, ways and relations are affected by the changes.

The create entry from listing 6 results in a SQL **INSERT**, delete in a SQL **DELETE** and modify in a SQL **DELETE** followed by a SQL **INSERT** statement. This is very unfortunate, one would think that a modify results in a SQL **UPDATE** statement.

The diff import functionality of `imposm3` is only meant to keep the database in sync with the OSM changes. It is not possible to track which rows in the table have been inserted, updated or deleted.

However this information is crucial to detect changed tiles and support all change scenarios described in section 5.1. The only way to keep track of changed features is to either modify `Imposm3` or take actions at the database level.

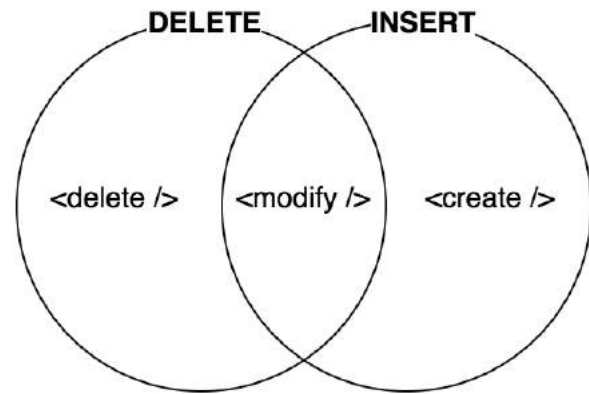


Figure 22: `Imposm3` SQL statements and OSM-Change actions

5.3 TRACK CHANGES

To track the entries at the database level the timestamp of the import is added to each object. This makes it possible to query modified and created objects by filtering for the latest import date. To keep track of entries that are no longer present in the database (like deleted and moved objects) auditing of **DELETE** actions has been implemented.

5.3.1 Track inserted rows

The **timestamp** column is used to keep a history of inserted features. The **timestamp** column contains the date of the original PBF or OSC file the feature was defined or changed. This is important for calculating the changed tiles within a timeframe later on in [Section 5.4](#).

1. The `imposm3` diff process inserts new rows for updated and added features
2. The timestamp column is now set to NULL for all new rows
3. Update the table and set the rows timestamp column to the timestamp of the import

5.3.2 Track deleted rows

If *OpenStreetMap* features are changed or removed, `Imposm3` will first delete the row from the table and then insert it again (if it is an update). This is due to performance reasons since a **DELETE** followed by an **INSERT** is faster than an **UPDATE**. To support the change scenarios of deleting, modifying and moving an object, the **track_osm_delete** trigger is enabled for each table to keep track of deleted rows as shown in listing 7.

```
DROP TRIGGER IF EXISTS osm_building_polygon_track_delete ON osm_building_polygon;
CREATE TRIGGER osm_building_polygon_track_delete
BEFORE DELETE ON osm_building_polygon
FOR EACH ROW EXECUTE PROCEDURE track_osm_building_polygon_delete()
```

Listing 7: Delete trigger on a table

The trigger in listing 8 will track the deleted row in a separate audit table before discarding it.


```

CREATE OR REPLACE FUNCTION track_osm_building_polygon_delete() RETURNS TRIGGER AS $$
BEGIN
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO osm_building_polygon_delete(id, geometry)
        VALUES($1, $2) USING OLD.id, OLD.geometry;
        RETURN OLD;
    END IF;
    RETURN NULL;
END;
$$ language plpgsql;

```

Listing 8: Logic of delete trigger

As a result each table has an additional delete table which contains all deleted and modified rows. This allows in a second step to calculate the affected tiles and rerender them.

5.4 CALCULATE CHANGED TILES

One way to determine which tiles are affected by changes in the database is to calculate the covered tiles from changed geometries by recursively descending the XYZ Quadtree and checking for intersections of the tiles and geometries.

Algorithm

1. Calculate the extent e for a given (x, y, z) tile index
2. Check if geometry g intersects with the tile extent e
3. Stop if there is no intersection in 2
4. Select tile index (x, y, z)
5. Calculate the four child (x, y, z) indices

$$\begin{pmatrix} x * 2 & y * 2 & z + 1 \\ x * 2 + 1 & y * 2 & z + 1 \\ x * 2 & y * 2 + 1 & z + 1 \\ x * 2 + 1 & y * 2 + 1 & z + 1 \end{pmatrix}$$
6. Call 1 for each child row if zoom level z has not reached max zoom level Z

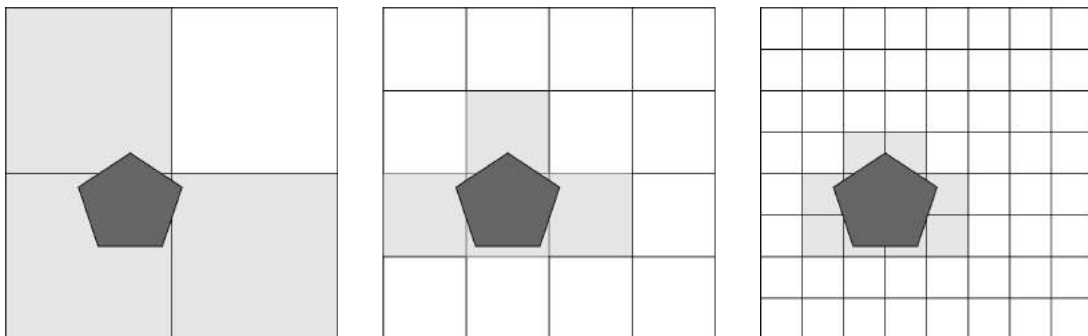


Figure 23: Recursive tile matching on polygon

Tile Buffers

Geometries in vector tiles can extend beyond the boundaries of a tile (tile buffer). To support the concept of a buffer in the algorithm the extent e for the (x, y, z) tile index is extended by a custom buffer b . The tile usually has a resolution of 256px. By adding the buffer to the tile $256 + 2 * b$ it is ensured that tiles that contain the geometries inside their buffers are detected as well.

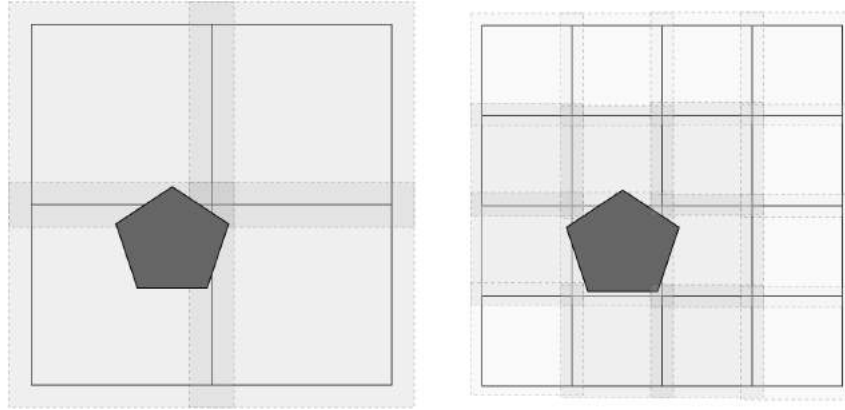


Figure 24: Recursive buffered tile matching on polygon

PostgreSQL Implementation

The PostGIS implementation makes heavy use of the `&&` operator and GiST indices on the geometry columns to check whether the tile extent and geometry intersect with each other. Although using the bounding box of the geometry is not accurate and can yield false positive changed tiles it is faster than using the correct `ST_Intersects`.

```
CREATE OR REPLACE FUNCTION overlapping_tiles(geom geometry, max_zoom_level INT, buffer_size INT)
RETURNS TABLE (tile_z INT, tile_x INT, tile_y INT) AS $$
BEGIN
    RETURN QUERY
        WITH RECURSIVE tiles(x, y, z, e) AS (
            SELECT 0, 0, 0, geom && XYZ_Extent(0, 0, 0, buffer_size)
            UNION ALL
            SELECT x*2 + xx, y*2 + yy, z+1,
                   geom && XYZ_Extent(x*2 + xx, y*2 + yy, z+1, buffer_size)
            FROM tiles, (VALUES (0, 0), (0, 1), (1, 1), (1, 0)) as c(xx, yy)
            WHERE e AND z < max_zoom_level
        )
        SELECT z, x, y FROM tiles WHERE e;
END;
$$ LANGUAGE plpgsql IMMUTABLE;
```

Listing 9: Recursive tile matching of geometry

Export changed tiles

The algorithm defined in [Section 5.4](#) is applied over all geometries that have changed since a given timestamp to find out all unique tiles that are affected by the changes and need to be rendered again.

```

SELECT DISTINCT t.tile_x AS x, t.tile_y AS y, t.tile_z AS z
FROM osm_building_polygon AS g
INNER JOIN LATERAL overlapping_tiles(g.geometry, 14, 4) AS t
ON g.timestamp >= (LOCALTIMESTAMP - INTERVAL '7 days')

```

Listing 10: Calculate all tiles containing building polygons that changed in the last 7 days

Figure 25 shows the changed tiles on zoom level 10 over the course of 10 days. A number of interesting discoveries were made.

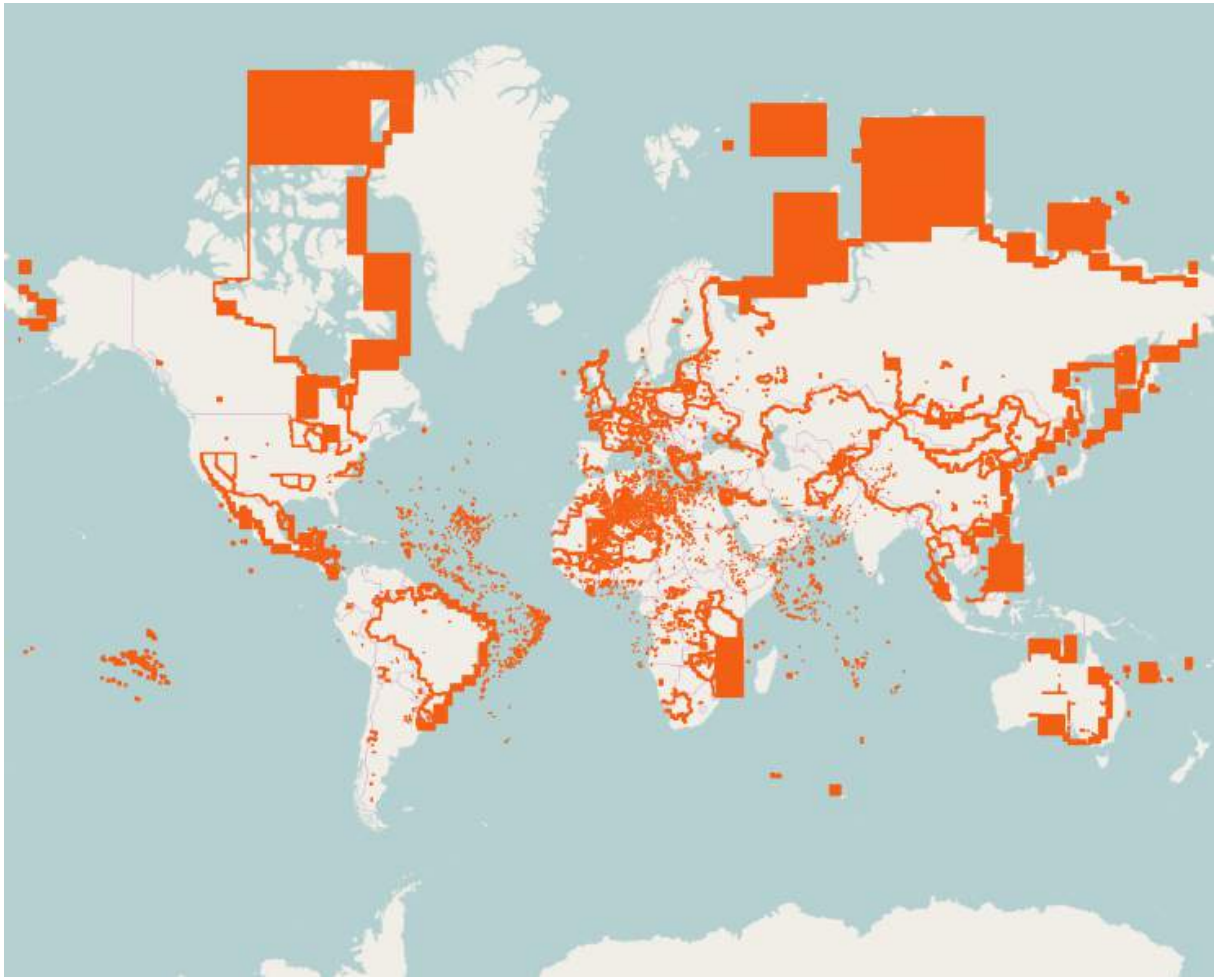


Figure 25: Changed tiles on z10 over course of 10 days

BOUNDARIES It is striking that along the administrative boundaries many tiles have changed. This leads back to the fact that in *OpenStreetMap* a single boundary is represented as multiple nodes which are part of a single relation. However in the database a boundary is represented as a single geometry of type *linestring*.

If somebody changes a single node on the administrative boundary, the Diff import process deletes this geometry and inserts it again like described in [Section 5.2](#). In a second step the changed tiles process detects that this row has changed and calculates all affected tiles. This is the reason why all tiles along the boundaries are detected as changed even though only a small part of the boundary was actually changed.

A possible solution for this problem could be to exclude administrative boundaries of the changed tile process and only update them irregularly.

CHANGES IN AFRICA It is encouraging to see that there is a lot of mapping activity in Africa. This part of the world is probably the least mapped part in *OpenStreetMap*.

BIG GENERIC BLOCKS There are a couple of big generic blocks visible. No explanation could be found for this symptom yet.

BIG RURAL AREAS WERE NOTHING CHANGES There are a lot of areas like Canada, Alaska, Russia, South America and Australia were almost nothing changes.

HYPOTHESIS It turns out that the unique changed tiles of 20 days is ~20 million tiles while unique changed tiles of 10 days has ~18 million tiles. This leads to the hypothesis that less tiles change often.

RESULTS AND FUTURE

Solutions for all problems defined in [Section 1.2](#) have been found and implemented.

6.1 RESULTS

- The quality of the vector tiles has been significantly improved and compatibility with Mapbox Streets v7 has been reached.
- An open source workflow to scale rendering of vector tiles with global coverage has been created.
- A process for updating vector tiles based on *OpenStreetMap* Diff files has been implemented.
- The project website has been significantly improved based on user feedback.
- Vector tiles for the entire planet, 219 country and 692 city extracts are provided as download on the project website.
- Detailed tutorials and a custom tile server to make getting started as easy as possible were created.

6.2 FUTURE

The interest and involvement of many people showed that this project provides great value to the FOSS community. In order for it to be valuable in the future, several challenges need to be overcome:

- An infrastructure provider needs to be found in order to be able to run the update process on a regular basis.
- More people need to know about this project.
- During the project, Mapbox released an update to their vector tile specification (version 2.0). The new standard needs to be supported, in order to properly work with their tools in the future.

For some of the challenges possible solutions are already planned in the coming months:

- The institute for software at HSR owns a server which could possibly be used for the update process, but this is not yet confirmed.
- Lukas Martinelli and Manuel Roth will give a talk about *OSM2VectorTiles* at the FOSSGIS conference in Salzburg and FOSS4G conference in Bonn. This will help to reach the broader FOSS community and further enhance the interest in this project.
- In order to support the Mapbox vector tile 2.0 specification the entire planet needs to be rerendered. If time and resources allow the project maintainers intend to do this in the future.

Part II

PROJECT DOCUMENTATION

REQUIREMENTS SPECIFICATION

This chapter describes the requirements, use cases and user characteristics for this project.

7.1 USE CASES

This section describes the two main use cases of this project.

RENDER VECTOR TILES The user renders a custom set of vector tiles. Since not all data of *OpenStreetMap* is inside *OSM2VectorTiles*, some users may want to add or remove data. Assuming users that are interested in creating their own vector tiles clone the *OSM2VectorTiles* repository these are potentially 70 users each month.

USE VECTOR TILES The user makes use of the prerendered vector tiles and wants to create a custom basemap. Assuming users that are interested in using their own vector tiles read the documentation these are potentially 600 users each month.

7.2 USER CHARACTERISTICS

LIMITED OR NO ACCESS TO THE INTERNET Users which have the constraint of limited or no access to the internet can download vector tiles for the entire planet and serve their custom basemap locally.

CAN NOT RELY ON A THIRD-PARTY SERVICE Many organizations can not afford to rely on a third-party service and want to run their map on-premise.

CUSTOMIZING THE LOOK OF THE BASEMAP In many use cases it is desirably to adjust the basemap to better match the design of a product.

7.3 REQUIREMENTS

The bachelor thesis consists out of three major requirements out of the use cases described in [Section 7.1](#). The technically interesting problems of these requirements are described in [Part i](#).

- Prerendered vector tiles from *OpenStreetMap* for the entire planet ([Chapter 4](#))
- Update functionality to keep up with future *OpenStreetMap* changes ([Chapter 5](#))
- Vector Tiles compatible with Mapbox Streets v7 ([Chapter 3](#))

7.4 NON FUNCTIONAL REQUIREMENTS

The non functional requirements are the key to success of this project. If the following requirements can be fulfilled, the specified users will be able to benefit of this project.

PERFORMANCE The initial rendering process for the entire world must be kept below two weeks. The import on the master server should take less than a day while rendering should be kept below two weeks. Updates should happen in a weekly interval.

LEARNABILITY It is important that users without previous vector tile or docker experience can get started with as few obstacles as possible.

COST The cloud instances to render the world once costs around 1500 dollar. The updating should only cost a fraction of this initial investment.

REPEATABILITY Since *OSM2VectorTiles* will provide continuous updates it is important that the process and results are repeatable.

COMPATIBILITY Compatibility with Mapbox Streets gives the users access to a wide range of styles and editors. Therefore the vector tiles must contain all features sets Mapbox Streets contains.

VECTOR TILE SIZE The size of a single vector tile should not be greater than 500 KB.

ARCHITECTURE AND DESIGN

8.1 ARCHITECTURE

The architecture of the project is structured into the import phase (ETL process), the changed tiles detection phase and the export phase (render vector tiles). This section describes the components and their purpose.

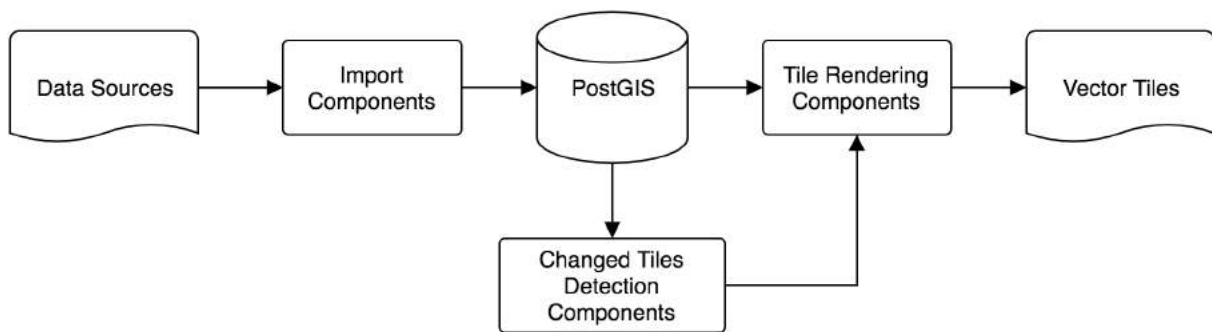


Figure 26: Workflow structured into components

Each component is a Docker image[12] with a single purpose. The Docker containers (components) are then linked against the database container and provided with additional input data to perform their functions. Isolating the components into Docker images makes it possible to ensure that the complicated installations and dependencies of some components never cause an issue. By additionally using Docker Compose it is possible to define the entire ETL process with different containers and make deployment and scaling containers easy.

8.1.1 Import Components

The import components take care of importing *OpenStreetMap* data, external data sources and SQL utilities such as views, triggers, indices and functions to help with the rendering and changed tiles detection process.

8.1.1.1 Import External

The **import-external** component is responsible for importing all data that is not mapped directly from *OpenStreetMap* into the PostGIS database. Figure 27 shows the external data sources and programs used to import the data into the PostGIS database. The GDAL tool `ogr2ogr` is used to import the various data sources into the PostGIS database.

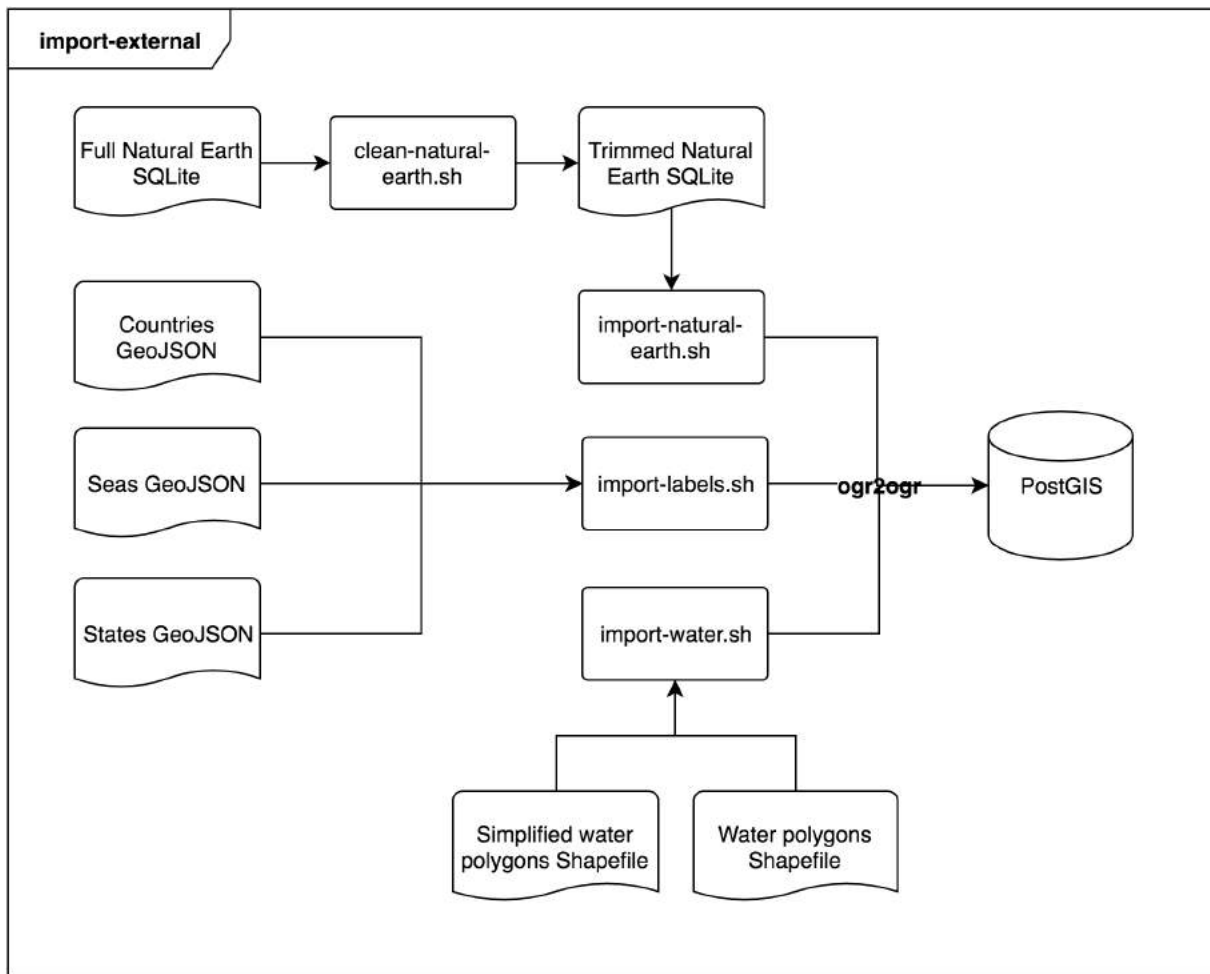


Figure 27: Import of external data sources

8.1.1.2 Import OSM

The **import-osm** component takes the first PBF file in the import folder and imports it into PostGIS. After that it updates the scaleranks using Natural Earth data from **import-external** to update the scaleranks and create generalized tables based off the imported data. The data is imported using **imposm3** diff mode and can take up to 14 hours for the entire planet file.

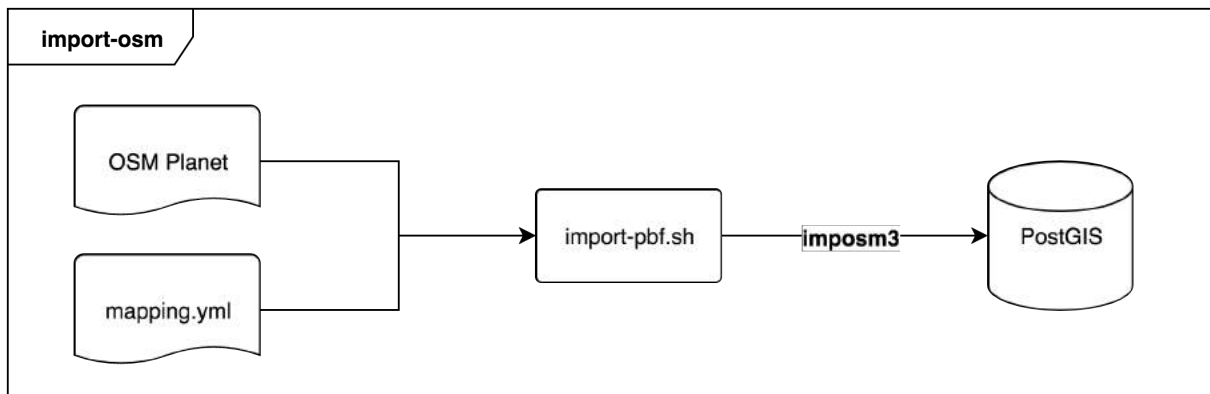


Figure 28: Import OSM diagram

8.1.1.3 Import SQL

The **import-sql** component is responsible for provisioning the SQL used in the different layers. It also generates SQL code for different classifications and code to detect changed tiles as well as table management commands for different layers.

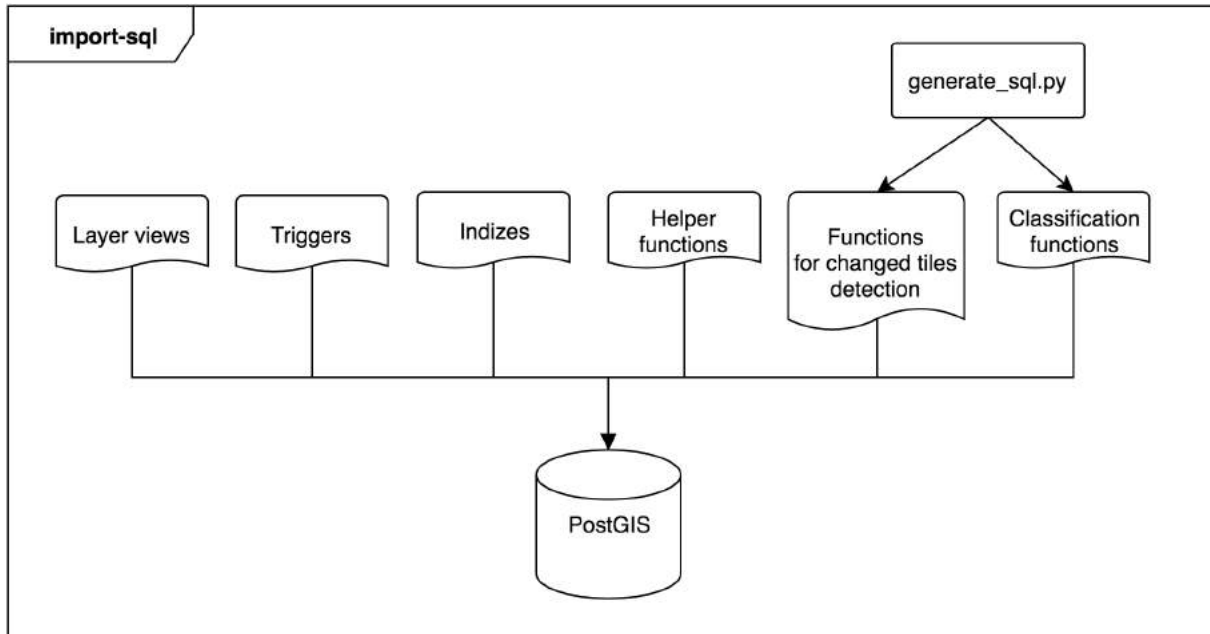


Figure 29: Import SQL diagram

8.1.2 Changed Tile Detection Components

The changed tile detection components handle creating an *OpenStreetMap* Diff file based on a certain *OpenStreetMap* planet file, importing a *OpenStreetMap* Diff file and updating outdated *OpenStreetMap* planet file with the latest changes. The following sections show what happens in every component.

8.1.2.1 Update OSM Diff

The **update-osm-diff** component takes the planet file as input and creates an *OpenStreetMap* Diff file containing all the changes happened since the planet file was downloaded. The `osmupdate` tool is used to execute this task.

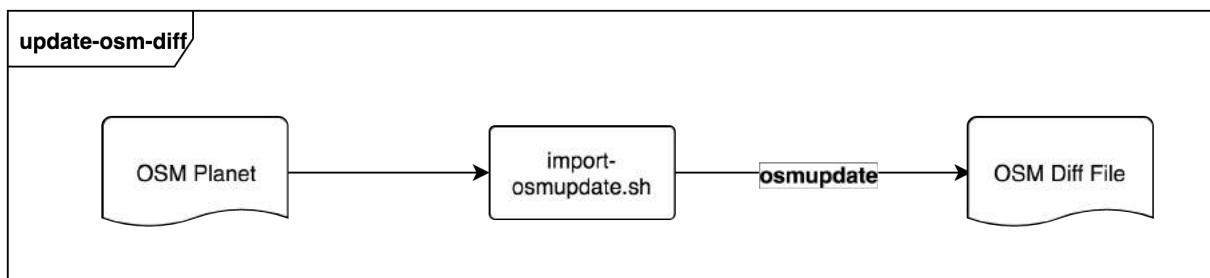


Figure 30: Update OSM Diff diagram

8.1.2.2 Import OSM Diff

The **import-osm-diff** component takes the *OpenStreetMap* diff file created with the **update-osm-diff** component as input and imports all changes into the database.

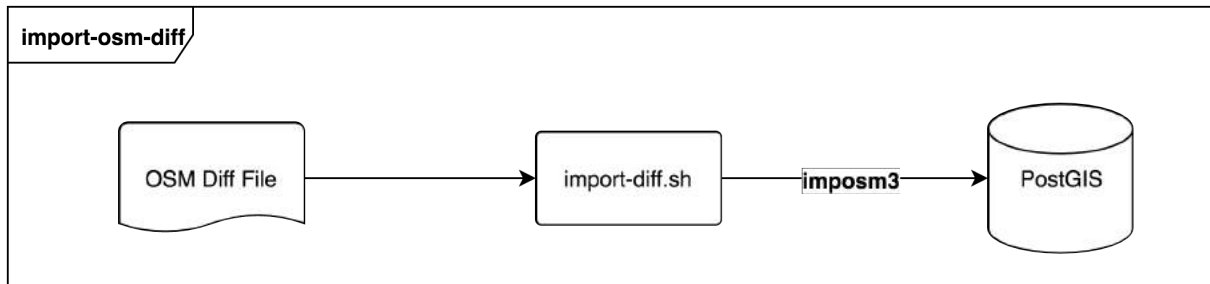


Figure 31: Import OSM Diff diagram

8.1.2.3 Merge OSM Diff

The **merge-osm-diff** component takes the old planet file and the latest diff file as input and merges all changes into the old planet file. Additionally the timestamp of the planet file is updated in order to have a correct diff file when the **update-osm-diff** process runs the next time. The *osmconvert* tool is used to merge the latest diff file into the old planet file.

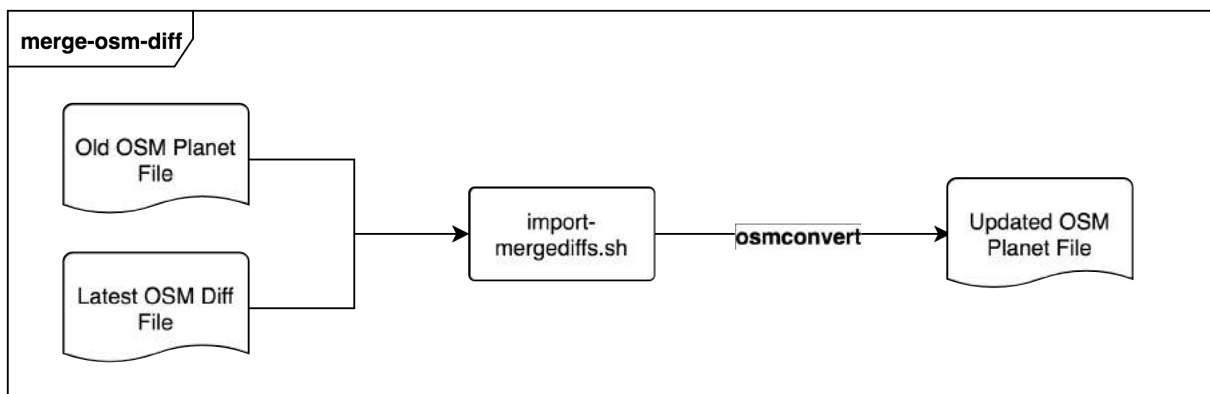


Figure 32: Merge OSM Diff diagram

8.1.2.4 Changed Tiles

The **changed-tiles** component is responsible for executing the changed tiles SQL logic and store the list of changed tiles in a text file using *pgclimb*. The actual logic for detecting the changed tiles is contained in the **import-sql** component.

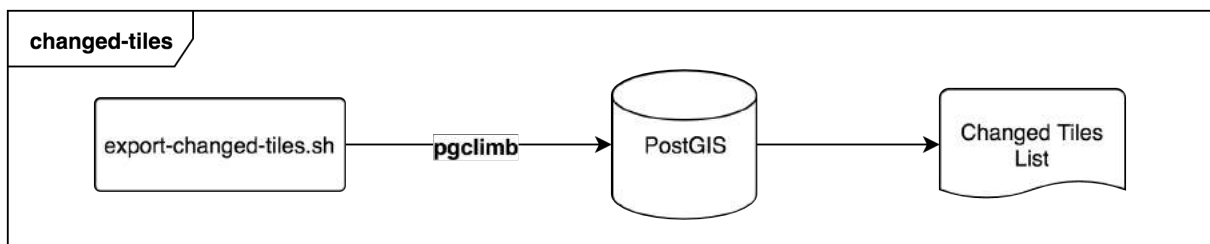


Figure 33: Changed Tiles diagram

8.1.3 Distributed Tile Rendering Components

In order to meet the performance requirements a distributed rendering architecture is needed to scale the process on to multiple hosts and process. The central component of the rendering pipeline is the message queue which contains the rendering jobs and results all other components interact with the message queue to take or confirm a job.

8.1.3.1 Generate Jobs

The generate-jobs component is responsible for creating JSON jobs consumed by the export component. It supports two types of jobs:

- **Pyramid:** Job of rendering a tile pyramid (e.g. from z8 all down to z14). Used for initial rendering of the world.
- **List:** Batch jobs of list of tiles to be rendered grouped by data locality. Used for rendering changed tiles.

Generate-jobs will output the jobs as individual JSON objects to stdout. A tool like `pipecat` can be used to schedule them on the job server.

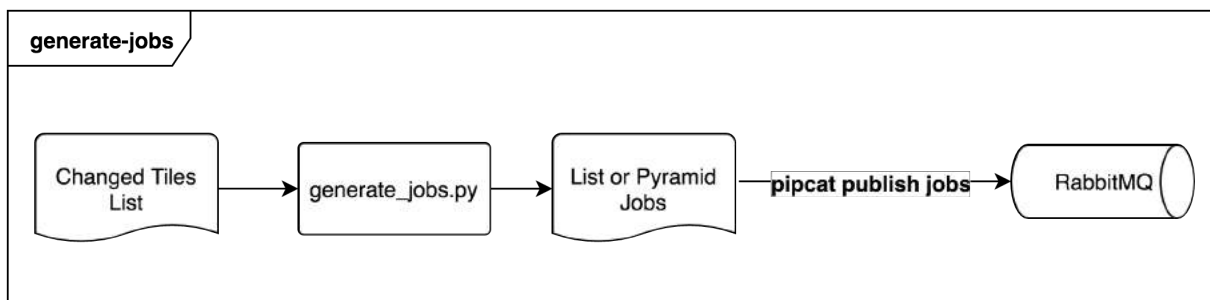


Figure 34: Generate Jobs diagram

8.1.3.2 Export

The **export** component is responsible for rendering vector tiles using `osm2vectortiles.tm2source` and the **postgis** component. Exports can be run together with a message queue like RabbitMQ or standalone for smaller extracts where it is not necessary to divide the work into several parts.

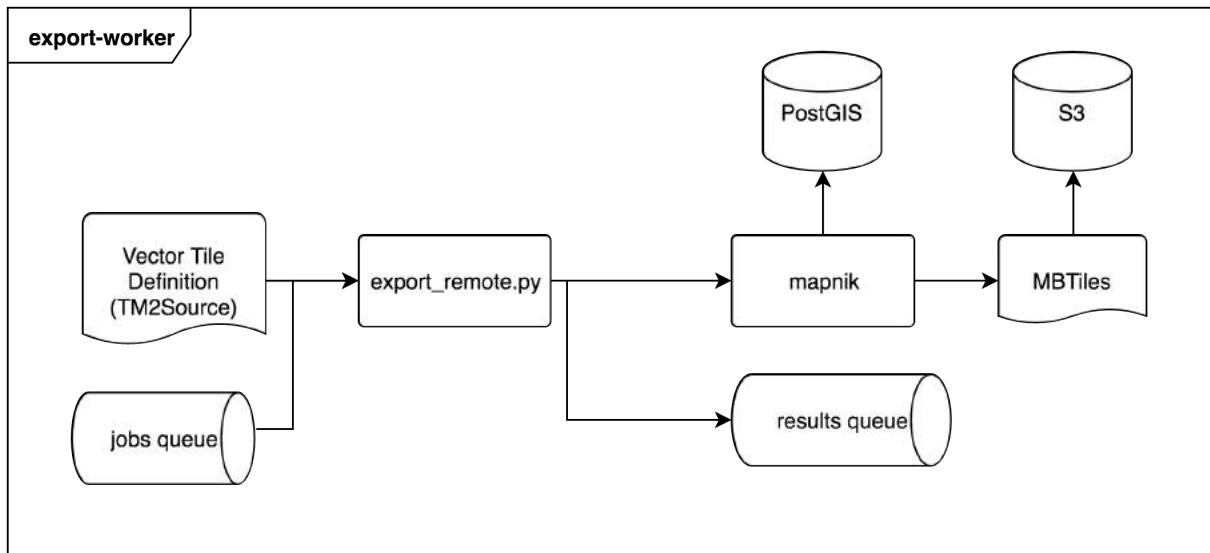


Figure 35: Export Worker diagram

8.1.3.3 Merge Jobs

The **merge-jobs** component is responsible for taking result messages from the queue, download the attached MBTiles file and merge it into the latest planet MBTiles file.

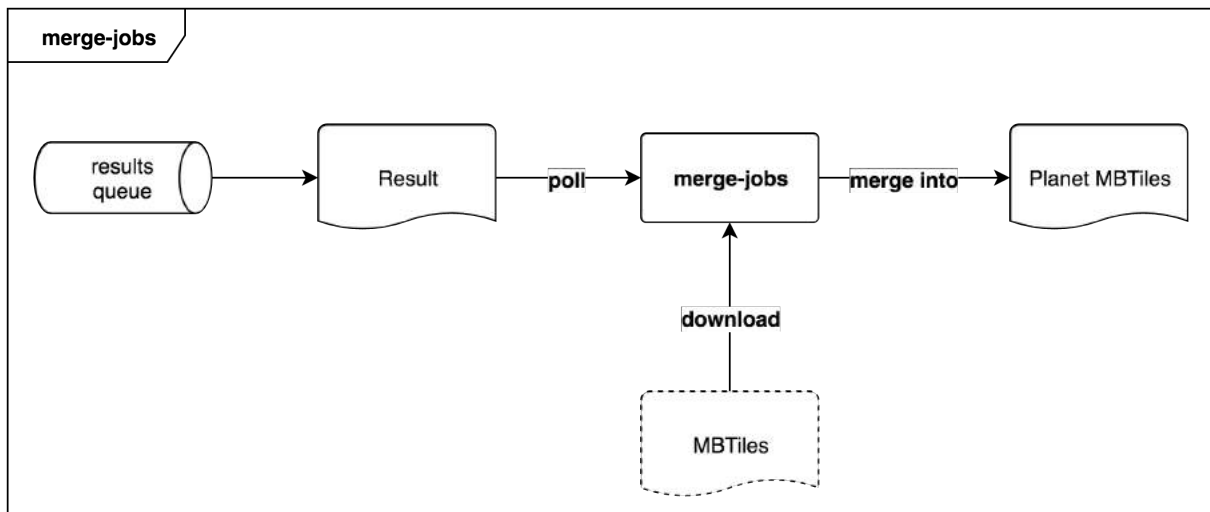


Figure 36: Merge Jobs diagram

8.2 DATABASE AND LAYER SCHEMA

In this section the different layers and the database schema related to it are explained and justified. Each layer contains a diagram showing the relations and model between tables, zoom level views (Section 3.2.2) and the vector tile layers (Figure 37). The database schema is denormalized and has no relations to fulfill the performance constraints of rendering. It is heavily optimized for fast reads since the only use case of the database schema is generating vector tiles from the PostGIS database. The arrow in Figure 37 represents the data flow.

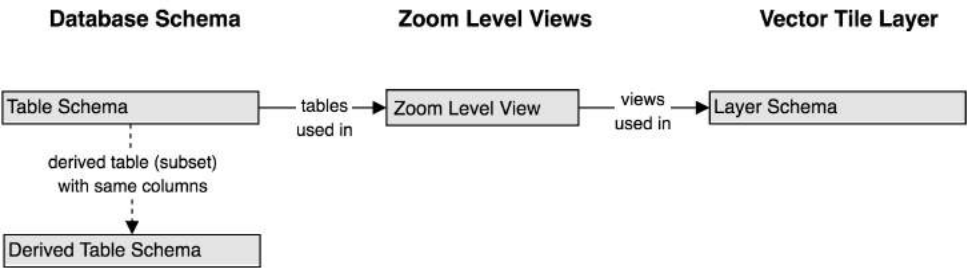


Figure 37: Database and layer schema diagram notation

8.2.1 Barriers

The layer **barrier_line** contains barriers that block a way or path. Common features are structural walls, fences or access controls like bicycle barriers and gates. Man made objects like piers or natural barriers like a cliff are contained as well in the **barrier_line** layer. Barriers are quite a detailed information and are therefore only relevant at the highest zoom level 14.

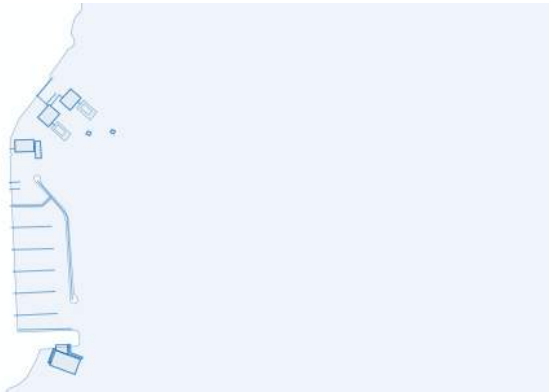


Figure 38: Barrier line layer with piers

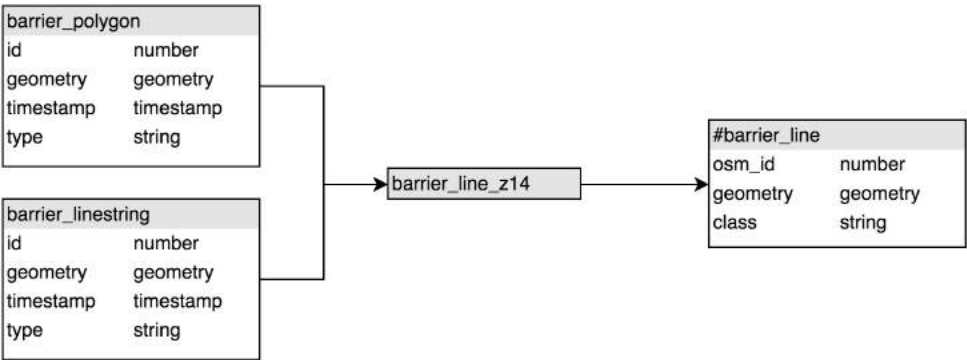


Figure 39: Barrier layer

8.2.2 Water

The **water** layer contains bodies of water like the ocean, lakes or large rivers. Since water is essential to the quality of the map, different data sources as shown in Table 4 are used on different zoom levels. The **water_label** layer shows the labels of lake bodies (not marine waters). Due to the costly calculation of centroid of very large polygons the **water_point** table is precalculated ahead of rendering time.

Coastlines in *OpenStreetMap* are sensitive for change and the *OpenStreetMapData*[21] project takes care of repairing broken coastlines and checking it thoroughly. The *OpenStreetMapData* project also takes care of splitting the ocean into several smaller tiled polygons which results in better database performance. The **water** layer uses simplified ocean polygons from **osm_ocean_polygon_gen0** on zoom level 4 and the original polygons from **osm_ocean_polygon** are used from zoom level 5 up to zoom level 14.

For choosing the right water bodies at low zoom levels the NaturalEarth data set [20] provides manually curated data of physical features such as water in the Shapefile format. For lakes and oceans water polygons of different resolutions (1:110M, 1:50M and 1:10M) were chosen on zoom level 0 to 4.



Figure 40: Ocean and water polygons in southern Europe

Table Name	Source
ne_110m_ocean	NaturalEarth
ne_110m_lakes	NaturalEarth
ne_50m_ocean	NaturalEarth
ne_50m_lakes	NaturalEarth
ne_10m_ocean	NaturalEarth
ne_10m_lakes	NaturalEarth
osm_ocean_polygon_gen0	OpenStreetMapData
osm_ocean_polygon	OpenStreetMapData

Table 4: Tables from external data sources for water layer

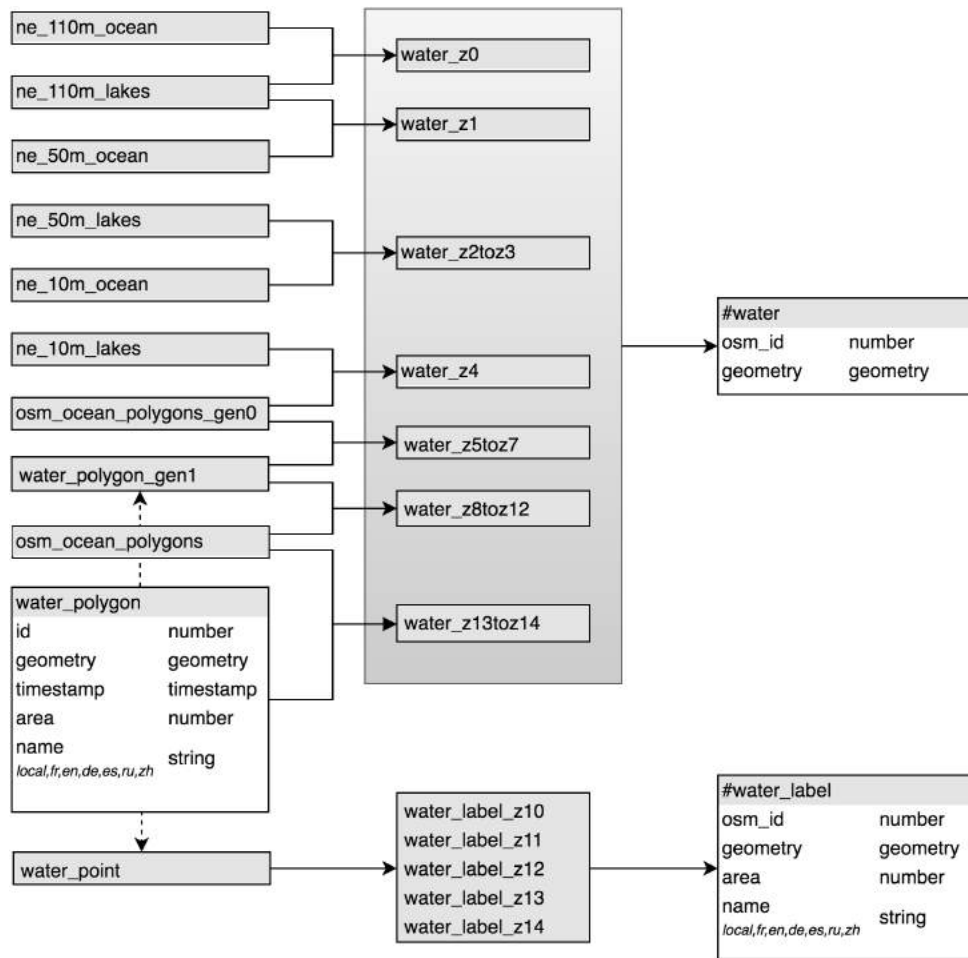


Figure 41: Water layer schema

8.2.3 Roads

Roads are one of the most essential features in maps. Roads are present across all zoom levels filtered by type. At the lowest zoom levels only motorways are shown while at higher zoom levels residential and service roads are rendered as well. Due to the complex hierarchy of roads in *OpenStreetMap* each zoom level contains custom filters to control which kind of roads get displayed on which zoom level. The **road** and **road_label** layer, query the data from the **road_geometry** table where both linestrings and polygons (for bigger avenues and squares) are present. The **road_label** layer consists of linestrings with a road name assigned and the vector tile client then takes care of drawing the road label text across the road linestring. To avoid having too many labels on roads (especially relevant for motorway signs) the roads are grouped by their vicinity and ranked by their length to reduce label density.

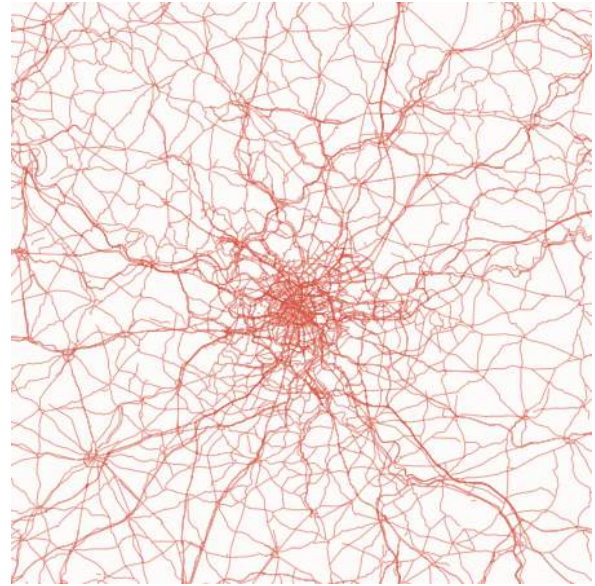


Figure 42: Road layer around Paris

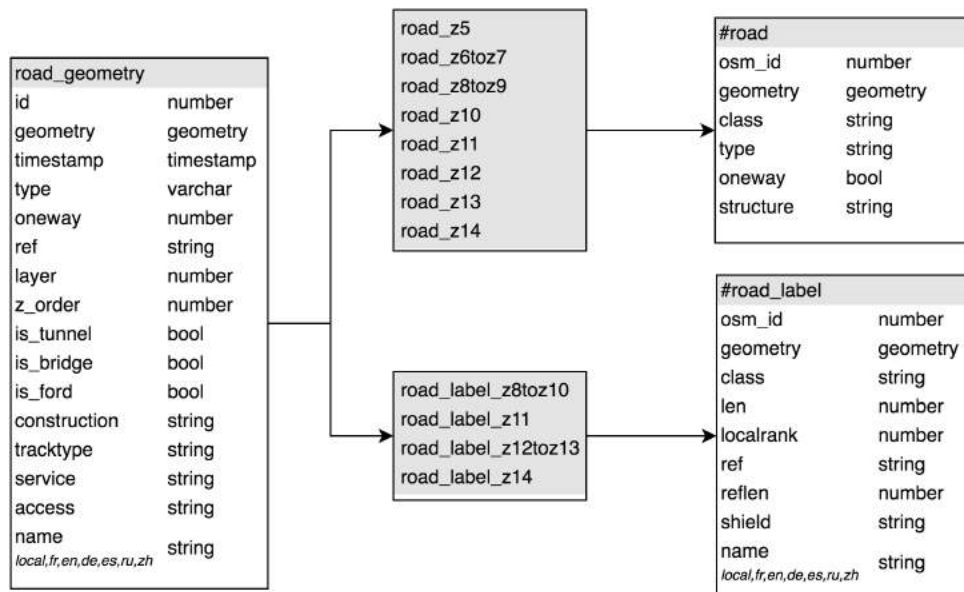


Figure 43: Road layer schema

8.2.4 Buildings and Housenumbers

Buildings are polygons such as houses or skyscrapers. They only appear on the highest zoom level 14 with the exception of large buildings already appearing at zoom level 13. Buildings are one of the most frequent features at high zoom levels.

The **housenum_label** layer contains buildings or single points tagged with a house-number. Housenumbers only appear on zoom level 14.



Figure 44: Buildings and house numbers

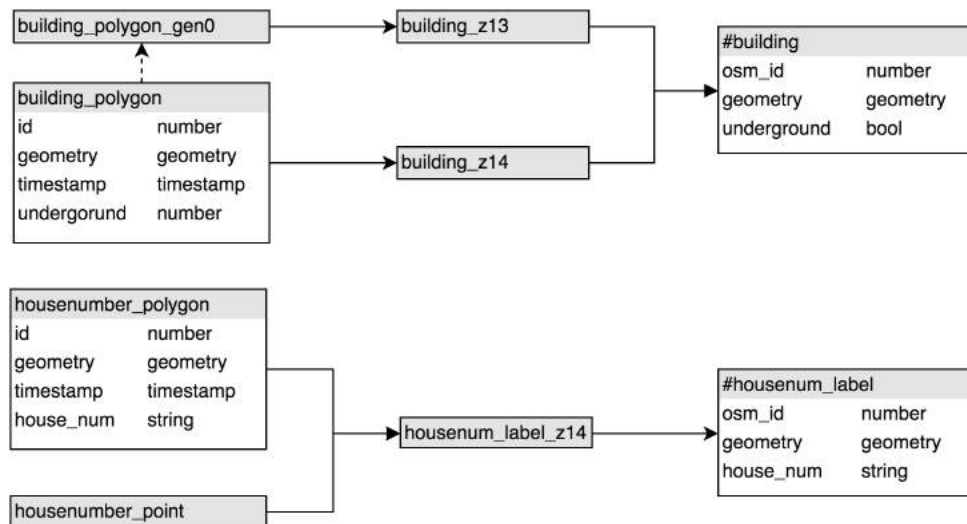


Figure 45: Building layer schema

8.2.5 Administrative Boundaries

The **admin** layer contains the linestrings of administrative boundaries such as countries, states or provinces. Boundaries are treated as linestrings because borders often break in *OpenStreetMap* and can no longer be reconstructed as polygons. Therefore it is safer to work with linestrings even though this provides less cartographic styling options.

Since a different level of detail is required at different zoom levels the cultural data set from Natural Earth [20] has been used at low zoom levels for boundaries of countries, provinces and disputed areas while at higher zoom levels the more accurate *OpenStreetMap* data is used.



Figure 46: Admin level 4 (states) in the US

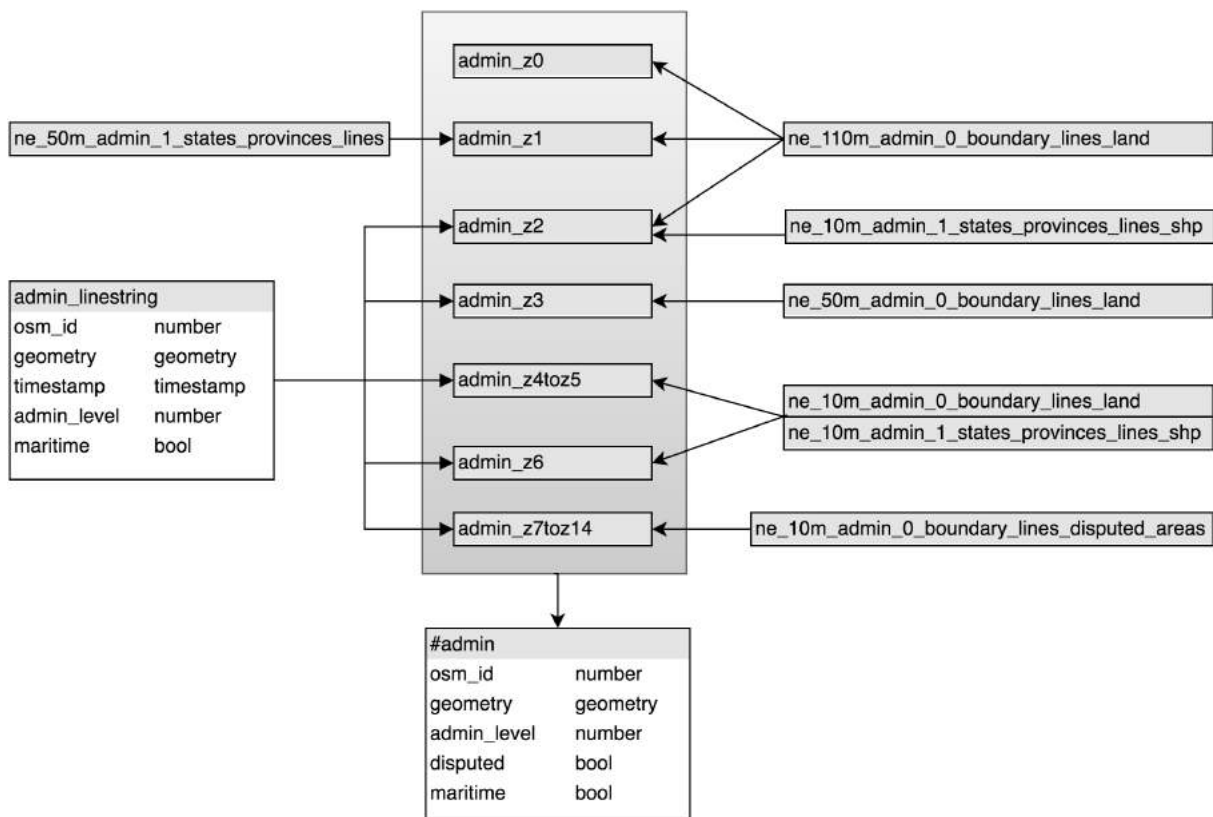


Figure 47: Admin layer schema

8.2.6 Landuse and Landuse Overlay

The layer **landuse** contains polygons of specially zoned land. The most frequent features inside **landuse** are wood areas as well as national parks, swamps, commercial, industrial and military zones. Very large polygons are split into several pieces into the **landuse_split_polygon** table and large polygons are generalized for lower zoom levels. The polygons in the **landuse** layer are filtered by area depending on the zoom level so that at low zoom levels only the biggest polygons are shown.

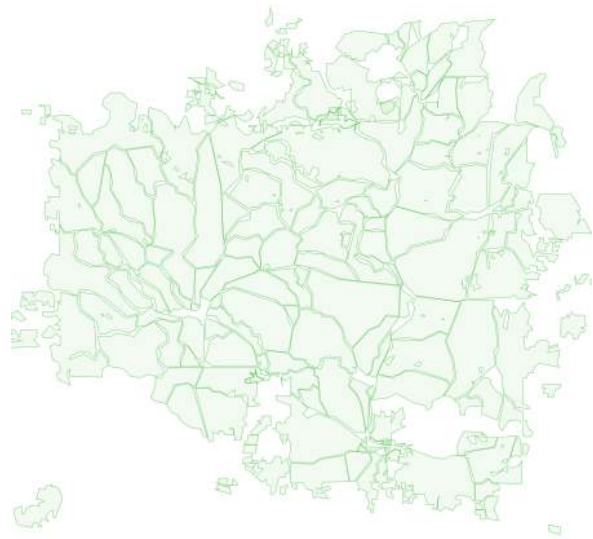


Figure 48: Landuse (wood) at zoom level 10

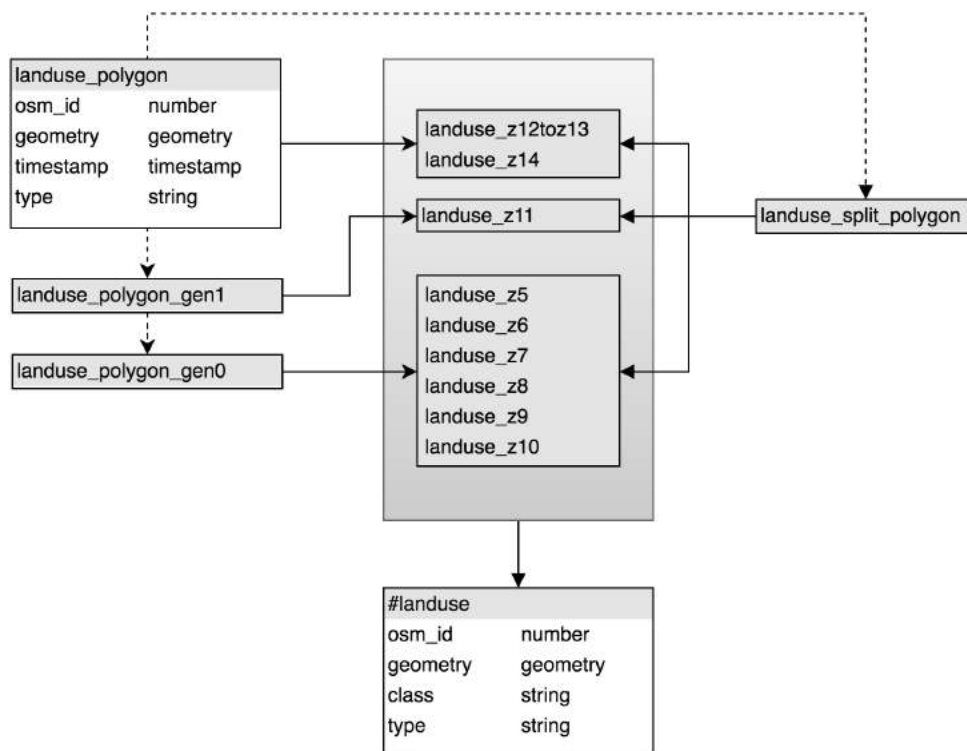


Figure 49: Landuse layer schema

8.2.7 POI Labels

A point of interest (POI) is a feature bound to a particular point of the map (e.g. churches, schools, tourist attractions, hotels, trees). Not all POIs mapped in *OpenStreetMap* are relevant for a visual map. In order to appear on the map, POIs are required to have at least a name or icon derived by the **type** field.

Users shouldn't be overwhelmed with too many point of interest icons. Therefore the field **localrank** contains an ascending importance rank which can be used by map clients to prioritize important POIs. Very prominent POIs additionally have a **scalerank** based on their covered **area**. The rank calculation for POIs works very similar to place label ranking described in [Section 3.3.4](#).



Figure 50: POI layer on top of building layer

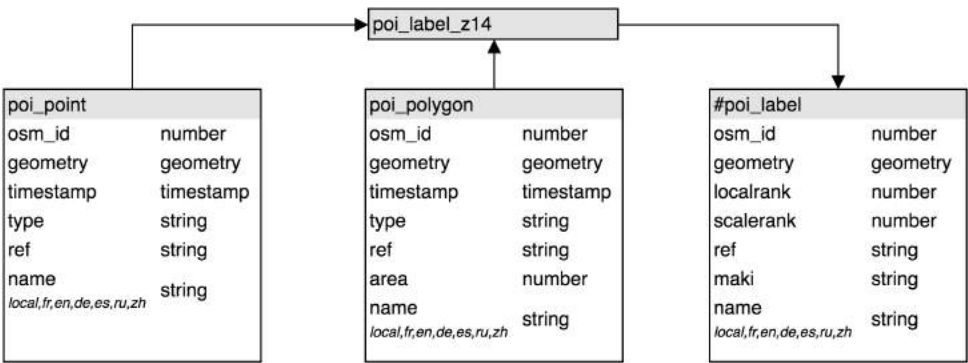


Figure 51: POI label layer schema

8.2.8 Countries and States

The placement and importance of labels of countries, states and seas matters[1] and is important to get right. Because it is difficult to ensure the quality of these features when importing directly from *OpenStreetMap*, the labels of countries and states are curated by hand.

Data from the Overpass API [5] is converted into GeoJSON and manually edited and enhanced with a label rank to get the best possible label placement and importance ranking. This effort is worth it because country and state data changes at infrequent intervals. Country labels are not present on all zoom levels and are filtered based on their **scalerank** value to show countries like Italy prior to a less important city state like the Vatican.



Figure 52: Country and state labels around Brasil

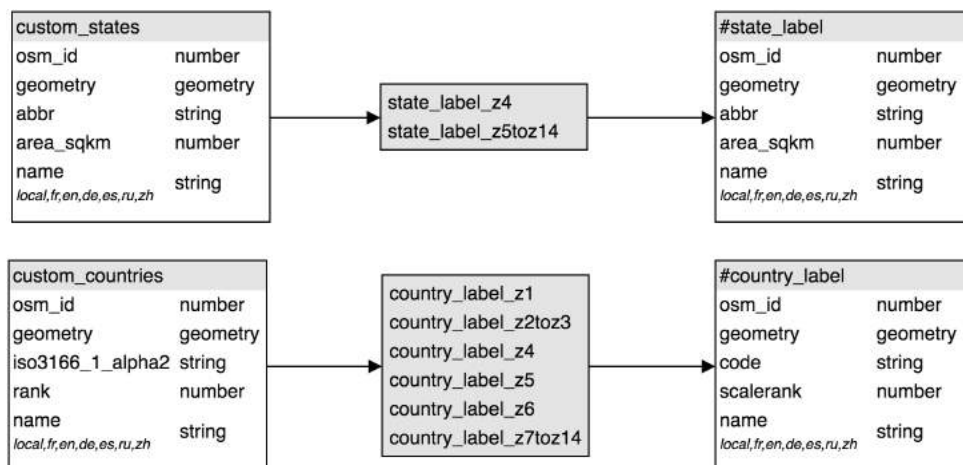


Figure 53: Country and state layer schema

8.2.9 Places

Place labels are vital for building nice maps with good text hierarchy. It is important to show only world cities at low zoom levels and then gradually show more and more local places. Filtering happens via the **scalerank** and **localrank** fields in the vector tiles. The **scalerank** field originates from the **ne_10m_populated_places** table from NaturalEarth and is merged into the **place_point** table. Most place labels are mapped as point in *OpenStreetMap* except for residential districts and islands for which the centroid of the geometry is used as label point. Since place labels are very delicate and important every zoom level has custom filters to control which places are relevant. The rank calculation for places is explained in [Section 3.3.4](#).



Figure 54: Important place labels in Europe



Figure 55: Place label layer

8.2.10 Aeroways and Airports

The layer **aeroway** contains infrastructure regarding air travel. The most common features are airports and their aprons, runways and taxiways. Airports are big landmarks and therefore all features are already present after zoom level 10. The layer consists of polygons and linestrings because runways are often polygons, while taxiways are mostly linestrings.

The layer **airport_label** contains labels of airports (either a point or the centroid of the airport polygon) since airports are important orientation points. The **scalerank** field describes the importance of the airport based on the covered area and type. Airports usually have official abbreviations (either the IATA, FAA, ICAO or custom reference code) that are stored in the **ref** field.

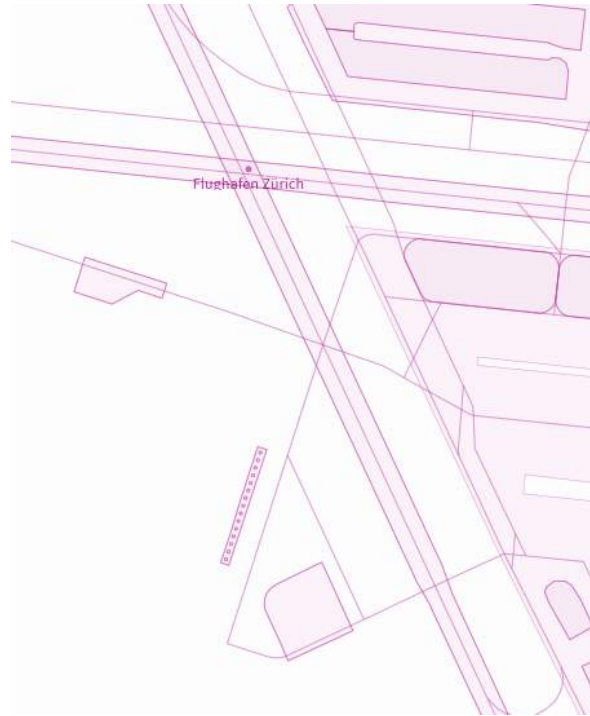


Figure 56: Aeroway and airport label layer of Zurich airport

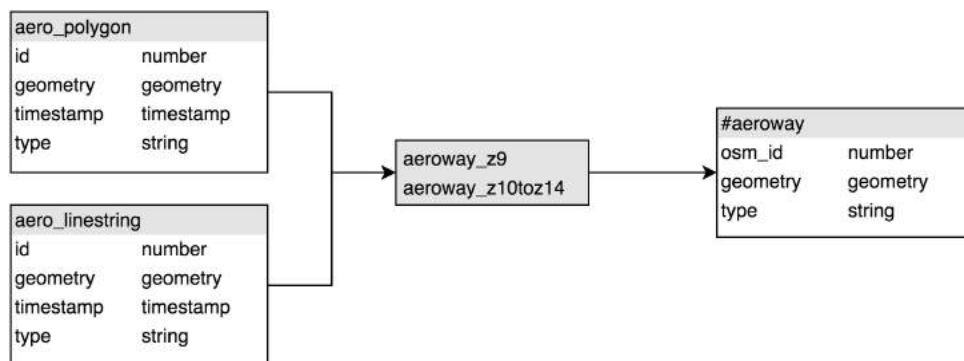


Figure 57: Aeroway layer

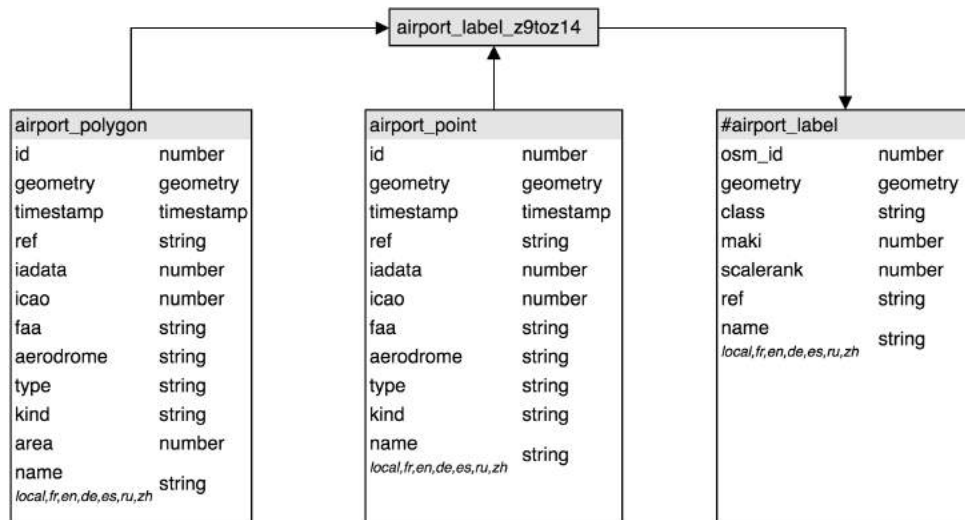


Figure 58: Airport label layer

8.2.11 Oceans, large Lakes and Bays

For oceans, large lakes and bays having a curved label along the body is advantageous because the label does not interfere with other labels like places.

Since these features are not available as polygons the label linestrings cannot be calculated but had to be drawn by hand. For the 100+ most important lakes and bays custom linestrings have been drawn. The metadata still originates from *OpenStreetMap* but the geometries are custom. Since these natural features do not change this does not pose a problem.

The linestrings are filtered ascending for the upper zoom levels based on the **rank** field and are only shown at low zoom levels 0 to 6.



Figure 59: Marine label of Mediterranean Sea

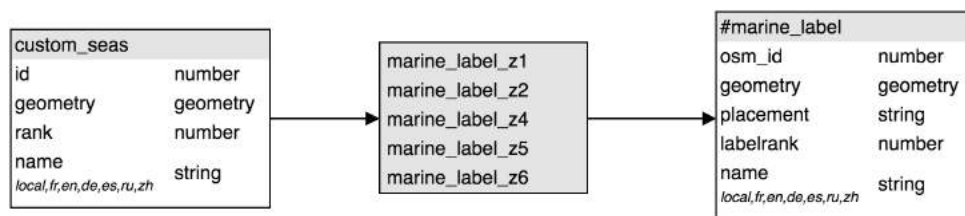


Figure 60: Marine label layer

8.2.12 Mountain Peaks

Mountain peaks are points tagged as mountains and volcanoes with a name and elevation. Because clients only have limited logic capabilities the elevation is calculated in both meters and feet.

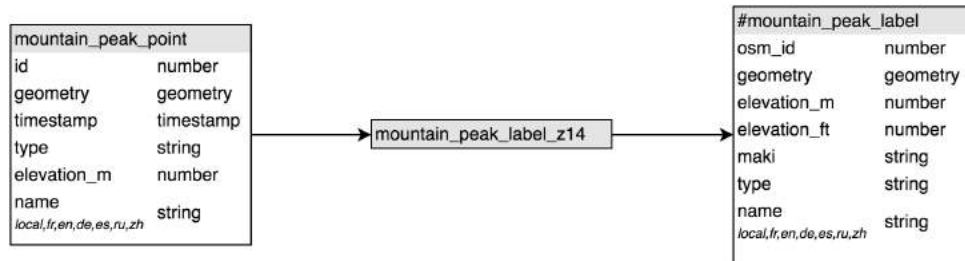


Figure 61: Mountain peak label layer

8.2.13 Rail Stations

Rail stations include railways, metros and tram stations. Since these public transport features are usually styled differently than other point of interests they are contained in a separate **rail_station_label_layer**.

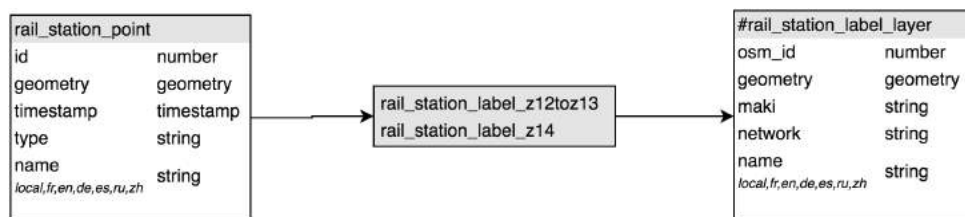


Figure 62: Rail station label layer

PROJECT MANAGEMENT

9.1 SOFTWARE DEVELOPMENT PROCESS

An agile approach based on SCRUM has been used as the process model of this project. The maintainers of the project act as joint product owners while the backlog is managed not only by the product owners but also by the community which add features to the wish list and even contribute code to the repository.

At the beginning of each sprint features of the backlog are estimated and scheduled for the next release. After each sprint (two to three weeks) a new stable version is released.

In comparison to SCRUM there are no sprint reviews and retrospectives but regular meetings with the thesis advisor to inform about project status are conducted. Regular meetings with the technical advisor help to solve problems together.

GitHub

GitHub was used for planning and tracking of the tasks and milestones. To provide a SCRUM board and burndown chart the ZenHub browser plugin has been used.

It has a big advantage over other project management tools, as revision control and issue tracking are at the same place. Non project members can understand the thoughts behind certain decisions and communicate their ideas directly to team members which is important for an open source project.

An organization named *osm2vectortiles* has been created with the following repositories:

- **osm2vectortiles** Contains the distributed workflow to create vector tiles
- **tileserver-gl-light** A trimmed down fork of *tileserver-gl* for serving MBTiles and predefined Mapbox GL styles with emphasis on serving vector tiles from *OSM2VectorTiles*
- **imposm3** Custom fork of *imposm3* to support timestamp field
- **mapbox-gl-styles** Collection of Mapbox GL styles that are compatible with *OSM2VectorTiles*.
- **rabbitmq** Custom fork of official RabbitMQ Docker image with additional support for message timeout configuration
- **bachelor-thesis** LaTeX source for bachelor thesis
- **study-thesis** LaTeX source for preceding study thesis

9.2 SCHEDULE

Because the *OSM2VectorTiles* members were already familiar with the technologies and field of work no elaboration phase was needed. Each sprint was tightly coupled to the next release.

9.3 MILESTONES

Each milestones marks a special release version of the vector tiles.

Version	Date	Resolved Issues	Merged PRs	Fixed Bugs
v1.1	Mar 4	9	3	1
v1.2	Mar 21	9	5	1
v1.3	Apr 1	6	6	1
v1.4	Apr 12	14	11	6
v1.4.1	Apr 22	10	11	10
v1.4.2	Apr 25	5	4	11
v1.5	Apr 28	5	8	3
v2.0	May 24	12	8	4

Table 5: Project sprints and statistics

9.4 ROLES AND RESPONSIBILITIES

Prof Stefan Keller	Thesis advisor responsible for supervising work and assess the thesis.
Petr Pridal, PhD	Technical partner responsible for providing infrastructure
Manuel Roth	Maintainer
Lukas Martinelli	Maintainer

Table 6: Thesis contributors and their roles

9.5 RISKS

In contrast to the preceding study thesis the bachelor thesis is less risky due to the increased knowledge of the field.

Risk	Measurement	Probability (1-6)
Too slow update process	Early measurements of solution	4
Vector tiles oversized	Continuously measure vector tiles	4
Infrastructure not sufficient	Switch to non school infrastructure and rely on external sponsors	3
Unwanted Features	Open roadmap and feedback of community	2
Lacking quality	Regularly control whether defined quality measurements were complied with	2

Table 7: Risks and measurements

QUALITY MEASURES

In order to ensure proper quality a number of measurements were taken. The following section describes every measurement separately.

10.1 TESTING

OSM2VectorTiles consists of many different components in different languages that all work together which makes testing quite challenging. This section describes how testing was approached in this project.

10.1.1 Mapping Report Tool

The mapping report tool was created to verify that all imported *OpenStreetMap* data is used inside the vector tiles. This helped to identify *OpenStreetMap* key/value pairs which should be removed from the import mapping because they are not used inside the vector tiles.

OSM Tables	Vector Tile Layers	All Records	Used Records	Percent
osm_admin_*	admin	11284	813	7.2
osm_aero_*	aeroway	643	643	100
osm_airport_*	airport_label	199	199	100
osm_barrier_*	barrier_line	22880	22880	100
osm_building_*	building	604304	604304	100
osm_housenumber_*	housenum_label	108376	108376	100
osm_landuse_*	landuse (overlay)	79337	79337	100
osm_mountain_peak_*	mountain_peak_label	3235	3235	100
osm_place_*	place_label	12269	6921	56.41
osm_poi_*	poi_label	120793	75108	62.18
osm_rail_station_*	rail_station_label	2990	2990	100
osm_road_*	road (label)	522469	522469	100
osm_water_linestring	waterway (label)	31075	20458	65.83
osm_water_polygon	water (label)	10496	10496	100

Table 8: Example output of Mapping Report Tool

[Table 8](#) shows an example output of the mapping report tool. It lists the number of rows per table and compares them with the number of rows used in a certain layer. If the percentage value is below 100 percent, the layer does not use all the features specified in the import mapping. Therefore these *OpenStreetMap* key/value pairs should be removed.

10.1.2 Integration Test

In Travis CI[3] the entire workflow was completed for a small data sample on each commit. Because the entire workflow is configured with Docker Compose [11] the CI server had to execute all import steps in serial order. This is a straightforward way to check if all components work together correctly and although it is a simple setup it has helped tremendously during project development, catching bugs like missing tables or SQL typos.

script:

```
# Test import
- docker-compose up -d postgis
- sleep 10
- docker-compose run import-external
- docker-compose run import-osm
- docker-compose run import-sql
# Test export
- docker-compose run export
# Test changed tiles
- docker-compose run update-osm-diff
- docker-compose run import-osm-diff
- docker-compose run changed-tiles
```

10.1.3 Structural Test

The Vector Tile Compare tool was created to analyze the content of single vector tiles. This helped to identify which type of data is shown on which zoom levels and later to ensure that the same amount of features are present in OSM2VectorTiles compared to Mapbox Streets v7.

10.1.4 Visual Test

In addition to comparing the content of the vector tiles the Visual Compare tool was created to visually preview and compare the map on different zoom levels.



Figure 63: Visual Compare Tool

Figure 63 shows a screenshot of the Visual Compare tool. On the left hand side is OSM2VectorTiles and on the right hand side Mapbox Streets v7. Both use the OSM Bright visual style.

10.2 GUIDELINES

To have homogeneous software the contributors have settled on common guidelines in the beginning of the project.

10.2.1 *Releases*

Semantic versioning [22] should be used for releases. At the end of each milestone a new release will be created. For each release a changelog with all closed issues, merged pull requests and fixed bugs should be created. This makes it easier for people to track the progress of the project.

10.2.2 *Git*

COMMIT MESSAGES The seven rules of great git commit messages[2] should be used.

REWRITING Git history should be kept clean and therefore local branches should be squashed meaningfully.

PULLING To avoid unnecessary merge messages one should always use the `--rebase` parameter.

10.2.3 *Workflow*

The Feature Branch Workflow[25] should be used. Every project member has a local repository with a copy of the remote repository. For each feature ticket in GitHub a separate branch will be created. Once a ticket has been completed a pull request will be created and needs to be reviewed and merged into the master branch by an other member.

Coding Standards

BASH Bash was used for the Docker image entrypoints and follow the rules of Defensive Bash Programming [15].

PYTHON Python code should stay PEP-8[23] compliant and write idiomatic Python code according to PEP-20[24].

JAVASCRIPT The JavaScript code is checked using ESLint[13]

SQL The PostgreSQL code is using upper case for the key words. Apart from nice formatted SQL code and functions should be used to keep the queries DRY[9].

DOCKERFILE Dockerfiles follow the best practices[10] defined by Docker.

PROJECT MONITORING

11.1 CODE STATISTICS

Table 9 shows how many lines of code were written in which language. A lot of SQL code is generated during the **import-sql** process. Most of the YAML definitions are used as input to generate SQL functions resulting in a much higher LOC count for generated code. Most of the logic is written in SQL with Python programs used for the distributed workers and generating SQL code. Since the ETL process requires a lot of programs to work together there is also a significant portion of small Bash scripts used as start scripts inside the Docker containers.

Language	LOC
SQL	1155
Bash	836
Python	728
Javascript	209
Dockerfile	201
YAML	
Vector Tile Definition	1160
SQL Classifications Definition	1928
Docker Compose File	169
Other	18
Total	6404

Table 9: Lines of code (LOC) per language

11.2 ESTIMATED TIME VS ACTUAL TIME

The estimations are based on the required work hours per week (20 hours per person) multiplied by the amount of weeks that are planned for a certain sprint.

Table 10 shows that every sprint took longer than planned. This has many reasons sometimes urgent issues had to be resolved in the same sprint or some task just took longer than expected. The issue management and communication with the community also turned out to be a very time intensive task.

Sprint	Estimated	Actual
v1.1	80	87
v1.2	80	99
v1.3	80	98
v1.4	100	121
v1.4.1	40	53
v1.4.2	40	52
v1.5	100	126
v2.0	80	87
v2.1	80	89
Total	720	812

Table 10: Estimated vs actual time for different sprints

11.3 TIME PER PERSON

The invested time per person was very balanced. This was thanks to the agreement that both contributors meet three times a week in school to work together on the project.

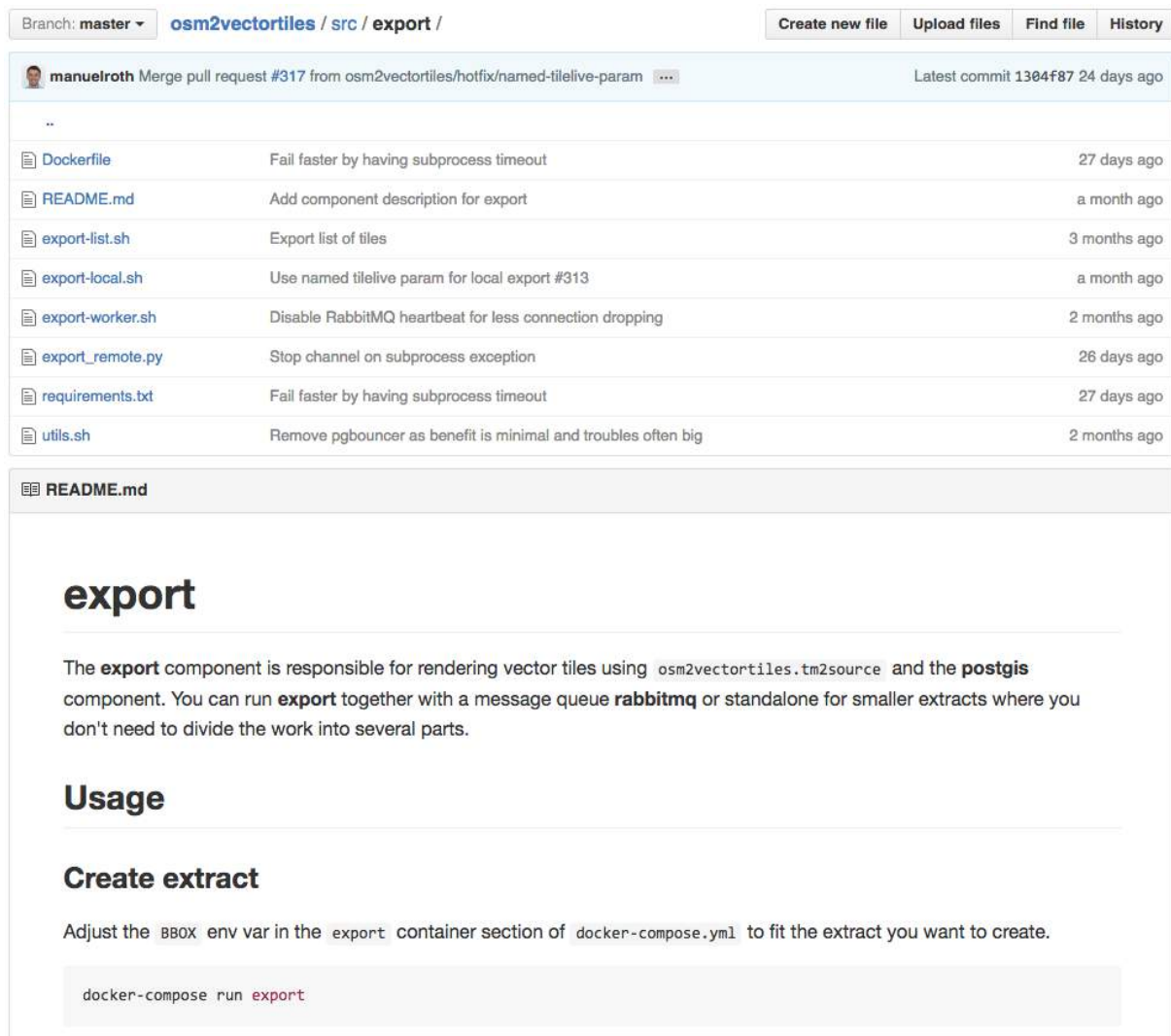
Sprint	Lukas Martinelli	Manuel Roth	Total
v1.1	43	44	87
v1.2	47	52	99
v1.3	50	48	98
v1.4	62	59	121
v1.4.1	28	25	53
v1.4.2	26	26	52
v1.5	62	64	126
v2.0	43	44	87
v2.1	46	43	89
Total	407	405	812

Table 11: Time for each contributor for each sprint

During the study thesis detailed documentation and tutorials for the *OSM2VectorTiles* project has been created. The documentation was separated into a user-centric and developer-centric part. All of this information was provided on the project website (<http://osm2vectortiles.org>). However when the project was publicly released and people started to get interested in the project, two main problems occurred:

- The tutorials targeted at regular users were too complicated and error prone
- Developers want to have the documentation on GitHub right next to the code and not on the project website

With this knowledge it was decided to move the developer-centric documentation into README files right next to the code and simplify the user-centric tutorials to eliminate most beginner errors.



Branch: master osm2vectortiles / src / export / Create new file Upload files Find file History

manuelroth Merge pull request #317 from osm2vectortiles/hotfix/named-tilelive-param Latest commit 1304f87 24 days ago

File	Commit Message	Time
Dockerfile	Fail faster by having subprocess timeout	27 days ago
README.md	Add component description for export	a month ago
export-list.sh	Export list of files	3 months ago
export-local.sh	Use named tilelive param for local export #313	a month ago
export-worker.sh	Disable RabbitMQ heartbeat for less connection dropping	2 months ago
export_remote.py	Stop channel on subprocess exception	26 days ago
requirements.txt	Fail faster by having subprocess timeout	27 days ago
utils.sh	Remove pgbouncer as benefit is minimal and troubles often big	2 months ago

export

The **export** component is responsible for rendering vector tiles using `osm2vectortiles.tm2source` and the **postgis** component. You can run **export** together with a message queue **rabbitmq** or standalone for smaller extracts where you don't need to divide the work into several parts.

Usage

Create extract

Adjust the `BBOX` env var in the `export` container section of `docker-compose.yml` to fit the extract you want to create.

```
docker-compose run export
```

Figure 64: Documentation of the export component using a README file

Figure 64 and Figure 65 show the documentation in the repository on GitHub and on the project website. The documentation can be found either on GitHub (<https://github.com/osm2vectortiles/osm2vectortiles>) or on the project website (<http://osm2vectortiles.org>).

Documentation

The following tutorials explain how you can use the OSM2VectorTiles project. Feel free to submit an [issue](#), if you think something is missing.

Getting Started

Quickly get started using OSM2VectorTiles to display maps in your browser. This tutorial explains how to serve downloaded Vector Tiles and use them in your browser.

Design beautiful maps with Mapbox Studio

Design beautiful maps in Mapbox Studio and use it together with OSM2VectorTiles to host your own map.

Generate your own Vector Tiles

Run the process to generate Vector Tiles yourself. In this tutorial you will learn how to use OSM2VectorTiles to create Vector Tiles for your desired region.

Create your own Extract

Can't find your desired region in the downloads section? No problem, this tutorial will show you how to create your own extract.

Figure 65: Simplified overview of user documentation on the project website

Part III

APPENDIX

Updatable Vector Tiles from OpenStreetMap

- Bachelor Thesis, Spring Semester 2016, Dept. of Informatics
- Authors: Lukas Martinelli and Manuel Roth
- Advisor: Prof. Stefan Keller, Geometa Lab at Institute for Software HSR
- Industry Partner: -

Introduction

In the previous semester thesis vector tiles from OpenStreetMap for Switzerland were created and publicly released. The feedback of the community confirmed the demand for downloadable vector tiles. Therefore it was decided to continue the project and further develop based on the feedback of the community. The most demanded feature was to provide regular updates to the vector tiles based on the Diff files of OpenStreetMap.

Goals

The primary goal of this thesis is to create a distributed process to generate vector tiles for the entire planet. These vector tiles should be compatible with the latest specification of the Mapbox Streets tileset.

As the OSM community adds constantly more data to OpenStreetMap the vector tiles quickly get outdated. Therefore the second goal is to implement the possibility to identify and update specific vector tiles which have changed.

Deliverables

- MBTiles files for the entire planet (snapshot)
- Update functionality to keep the vector tiles in sync with recent changes from OSM database
- Documentation:
 - The thesis will be written in English
 - The usual documents and rules of the dept. of informatics apply (e.g. poster in digital version)

Environment

Technologies: PostGIS, Docker, Message Queues

Preferred programming languages: PostgreSQL, Python, JavaScript, Bash

Evaluation

The thesis evaluation scheme there is a special emphasis of modern software development aspects.

Other involved parties

Consultation: Dr. Petr Pridal, Klokkan Technologies GmbH, Unterägeri

LIST OF FIGURES

Figure 1	Example of object tagged natural=wood	4
Figure 2	Vector and client representation of a map section	5
Figure 3	Tiled raster map	5
Figure 4	Vector Tile Structure	6
Figure 5	Vector tile grid with encoded geometry	7
Figure 6	XYZ coordinate schema	7
Figure 7	Simplified process from data import to vector tile rendering	8
Figure 8	Difference of zoom level view 13 and 14 in buildings layer	10
Figure 9	Layer road, water, and building features on same map	11
Figure 10	Data flow between database tables, zoom level views and layer	12
Figure 11	Point calculation in layer definition	13
Figure 12	Point calculation in zoom level view	13
Figure 13	Localrank calculation	15
Figure 14	Adapt rendering process to divide work and merge it back together	16
Figure 15	Pyramid job of a zoom level 2 tile and the descendants	17
Figure 16	Map divided into tasks at job zoom level 8	17
Figure 17	Quadkey indexing	19
Figure 18	Distributed rendering architecture using message queues	19
Figure 19	Merge completed MBTiles files together	20
Figure 20	A z8 subpyramid with the same data hash in all descendant tiles	21
Figure 21	Simplified diagram of changed tiles detection process	22
Figure 22	Imposm3 SQL statements and OSMChange actions	24
Figure 23	Recursive tile matching on polygon	25
Figure 24	Recursive buffered tile matching on polygon	26
Figure 25	Changed tiles on z10 over course of 10 days	27
Figure 26	Workflow structured into components	33
Figure 27	Import of external data sources	34
Figure 28	Import OSM diagram	34
Figure 29	Import SQL diagram	35
Figure 30	Update OSM Diff diagram	35
Figure 31	Import OSM Diff diagram	36
Figure 32	Merge OSM Diff diagram	36
Figure 33	Changed Tiles diagram	36
Figure 34	Generate Jobs diagram	37
Figure 35	Export Worker diagram	38
Figure 36	Merge Jobs diagram	38
Figure 37	Database and layer schema diagram notation	39
Figure 38	Barrier line layer with piers	39
Figure 39	Barrier layer	39
Figure 40	Ocean and water polygons in southern Europe	40
Figure 41	Water layer schema	41
Figure 42	Road layer around Paris	42
Figure 43	Road layer schema	42

Figure 44	Buildings and house numbers	43
Figure 45	Building layer schema	43
Figure 46	Admin level 4 (states) in the US	44
Figure 47	Admin layer schema	44
Figure 48	Landuse (wood) at zoom level 10	45
Figure 49	Landuse layer schema	45
Figure 50	POI layer on top of building layer	46
Figure 51	POI label layer schema	46
Figure 52	Country and state labels around Brasil	47
Figure 53	Country and state layer schema	47
Figure 54	Important place labels in Europe	48
Figure 55	Place label layer	48
Figure 56	Aeroway and airport label layer of Zurich airport	49
Figure 57	Aeroway layer	49
Figure 58	Airport label layer	50
Figure 59	Marine label of Mediterranean Sea	50
Figure 60	Marine label layer	50
Figure 61	Mountain peak label layer	51
Figure 62	Rail station label layer	51
Figure 63	Visual Compare Tool	55
Figure 64	Documentation of the export component using a README file	59
Figure 65	Simplified overview of user documentation on the project website	60

LIST OF TABLES

Table 1	OSM id transformation	14
Table 2	List of changed tiles indizes	18
Table 3	List of tile indizes split into batches	18
Table 4	Tables from external data sources for water layer	40
Table 5	Milestones	53
Table 6	Thesis contributors and their roles	53
Table 7	Risks and measurements	53
Table 8	Example output of Mapping Report Tool	54
Table 9	Lines of code (LOC) per language	57
Table 10	Estimated vs actual time for different sprints	57
Table 11	Time for each contributor for each sprint	58

GLOSSARY

Vector Tiles Packets of geographic data, packaged into predefined roughly-square shaped "tiles" for transfer over the web

Mapbox Streets Name of Mapbox's vector tile set

Mapbox GL Clientside rendering engine

Mapbox Studio Mapbox's map style editor

Imposm3 Tool which allows to import the OSM Planet file into a database

Mapnik Tool to generate vector tiles

MBTiles Container format for storing map tiles in a single file

GeoJSON File format for encoding a variety of geographic data structures

Web GL Javascript API for the graphics library in browsers

Natural Earth Public map dataset

OpenStreetMapData OpenStreetMap-derived dataset

OSM Planet file XML file containing all OpenStreetMap data

BIBLIOGRAPHY

- [1] Axismaps. Labeling and text hierarchy in cartography, 2015. URL <https://axismaps.github.io/thematic-cartography/articles/labeling.html>. Visited on 2016-05-06.
- [2] Chris Beams. How to write a git commit message, 2015. URL <http://chris.beams.io/posts/git-commit/>. Visited on 2016-05-06.
- [3] Travis CI. Travis ci, 2016. URL <http://travis-ci.org>. Visited on 2016-05-06.
- [4] OpenStreetMap Community. Way - openstreetmap wiki, 2016. URL http://wiki.openstreetmap.org/wiki/Way#Combined_closed-polyline_and_area. Visited on 2016-05-07.
- [5] OpenStreetMap Community. Overpass api - openstreetmap wiki, 2016. URL https://wiki.openstreetmap.org/wiki/Overpass_API. Visited on 2016-05-06.
- [6] OpenStreetMap Community. Elements, 2016. URL <http://wiki.openstreetmap.org/wiki/Elements>. Visited on 2016-05-27.
- [7] OpenStreetMap Community. Database statistics, 2016. URL http://wiki.openstreetmap.org/wiki/Stats#Database_statistics_-_Active_reports. Visited on 2016-05-27.
- [8] OpenStreetMap Community. Slippy map, 2016. URL http://wiki.openstreetmap.org/wiki/Slippy_Map. Visited on 2016-05-06.
- [9] Wikipedia Community. Don't repeat yourself, 2016. URL https://en.wikipedia.org/w/index.php?title=Don%27t_repeat_yourself&oldid=691047461. Visited on 2016-05-06.
- [10] Docker. Best practices for writing dockerfiles, 2015. URL https://docs.docker.com/engine/articles/dockerfile_best-practices/. Visited on 2016-05-06.
- [11] Docker. Docker compose, 2015. URL <https://docs.docker.com/compose/>. Visited on 2016-05-06.
- [12] Docker. Docker - the open-source application container engine, 2016. URL <https://github.com/docker/docker>. Visited on 2016-04-19.
- [13] Eslint. Eslint - pluggable javascript linter, 2015. URL <http://eslint.org/>. Visited on 2016-05-06.
- [14] Google. Protocol buffers, 2016. URL <https://developers.google.com/protocol-buffers/>. Visited on 2016-05-06.
- [15] Kfir Lavi. Defensive bash programming, 2012. URL <http://www.kfirlavi.com/blog/2012/11/14/defensive-bash-programming/>. Visited on 2016-05-06.

- [16] Mapbox. Vector tile specification, 2015. URL <https://github.com/mapbox/vector-tile-spec/tree/master/1.0.1>. Visited on 2016-05-10.
- [17] Mapbox. Mapbox streets v7 | mapbox, 2016. URL <https://www.mapbox.com/vector-tiles/mapbox-streets-v7/>. Visited on 2016-05-10.
- [18] Mapbox. Vector tile specification, 2016. URL <https://www.mapbox.com/vector-tiles/specification/>. Visited on 2016-05-19.
- [19] Microsoft. Bing maps tile system, 2016. URL <https://msdn.microsoft.com/en-us/library/bb259689.aspx>. Visited on 2016-05-09.
- [20] Naturalearthdata. Natural earth, 2016. URL <http://www.naturalearthdata.com/>. Visited on 2016-05-06.
- [21] Openstreetmapdata. Openstreetmapdata, 2016. URL <http://openstreetmapdata.com/>. Visited on 2016-05-06.
- [22] Tom Preston-Werner. Semantic versioning, 2016. URL <http://semver.org/>. Visited on 2016-05-06.
- [23] Python.org. Python pep 8, 2016. URL <https://www.python.org/dev/peps/pep-0008/>. Visited on 2016-05-06.
- [24] Python.org. Python pep 20, 2016. URL <https://www.python.org/dev/peps/pep-0020/>. Visited on 2016-05-06.
- [25] Atlassian Git Tutorial. Feature branch workflow, 2016. URL <https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow/>. Visited on 2016-05-06.

DECLARATION

Hereby we acknowledge,

- that we conducted this thesis by ourselves and without any external help, except with those, which are explicitly mentioned,
- that all used sources are cited academically correct, and
- that I didn't use any copyright protected materials (e.g. images) in an unauthorized manner.

Rapperswil, Spring 2016

A handwritten signature in dark ink, appearing to read 'L. Martinelli', written in a cursive style.

Lukas Martinelli, June 16, 2016

A handwritten signature in dark ink, appearing to read 'M. Roth', written in a cursive style.

Manuel Roth, June 16, 2016