Master's Thesis
Computer Science

# Creating a RDF Knowledgebase from OpenStreetMap Data

Axel Lehmann

Examiner: Prof. Dr. Hannah Bast
Advisers: Patrick Brosi

Albert Ludwig University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair of Algorithms and Data Structures

Mai 5th, 2021

**Writing Period**
05. 11. 2020 — 05. 05. 2021

**Examiner**
Prof. Dr. Hannah Bast

**Second Examiner**
Prof. Dr. Fabian Kuhn

**Advisers**
Patrick Brosi

# Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

| | |
|---|---|
| Place, Date | Signature |

# Abstract

We present *osm2ttl*, a tool for converting OpenStreetMap data into valid RDF data for use in SPARQL engines. The output we generate retains all data provided by the OpenStreetMap, and we add some explicit spatial relations *ogc:intersects*, *ogc:contains*, *ogc:intersects_ area*, and *ogc:contains_ area.*

The spatial and non-spatial data stored in the OpenStreetMap is curated by volunteers providing mostly accurate and recent data. The internal representation of non-spatial data can not enforce certain data types, formats, or the use of special characters and thus all data is treated as text. To conform with the rigid grammars of the RDF standard, we implemented conversions into both the *N-Triple* and *Turtle* grammar.

The *ogc:intersects_ area* and *ogc:contains_ area* describe the spatial relation of named areas with other named areas. *ogc:intersects* and *ogc:contains* describe the spatial relation of all other types with named areas. To generate these relations more efficiently, we use a directed acyclic graph storing named areas.

Through the addition of these relations, a SPARQL engine without explicit GeoSPARQL support can answer spatial queries.

# Zusammenfassung

Wir präsentieren *osm2ttl*, ein Werkzeug zur Umwandlung von OpenStreetMap Daten in gültiges RDF zur Verwendung in Triplestores. Das von uns erzeugte Ergebnis beinhaltet alle Daten, welche von der OpenStreetMap zur Verfügung gestellt werden. Wir fügen außerdem einige räumliche Relationen (*ogc:intersects*, *ogc:contains*, *ogc:intersects_ area* sowie *ogc:contains_ area*) hinzu.

Die Daten der OpenStreetMap werden von Freiwilligen gepflegt und aktualisiert. Wir wandeln die Daten in gültige RDF-Triple um und nehmen dafür nötige Ersetzungen vor.

Die Relationen *ogc:intersects_ area* und *ogc:contains_ area* beschreiben die Beziehungen zwischen benannten Flächen. Durch *ogc:intersects* und *ogc:contains* werden die Beziehungen anderer Typen mit diesen benannten Flächen ausgedrückt.

Durch diese Relationen kann eine SPARQL Engine ohne explizite GeoSPARQL-Unterstützung Geo-Anfragen beantworten.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Listings

# 1. Introduction

Knowledgebases like Wikidata store vast amounts of factual data. This data be can made explorable through SPARQL-Engines like QLever [BB17] using the SPARQL Query Language [SH13]. There are many different triplestores which provide Resource Description Framework (RDF, Section 3.2) input data as knowledge graphs [Ali+21] which have different indexing and storage capabilities.

To support geospatial data the GeoSPARQL [PH12] standard was published in 2012, but support in triplestores is still lacking [JHS21].

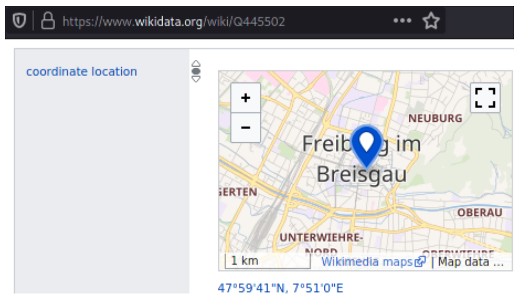
In this thesis we present *osm2ttl*, a tool for converting OpenStreetMap data into RDF. To ensure compatibility with triplestores, we implemented the N-Triples [SC14] and Turtle [CP14] standards. We enrich the data with explicit geospatial relations regarding areas.

To improve usability inside knowledge graphs, we differentiate between named areas, and all other types of OpenStreetMap data. This distinction allows us to generate a transitively reduced directed acylic graph of the named areas. With the directed acylic graph we can reduce the number of comparisons, and thus the number of triples stored in the RDF file.

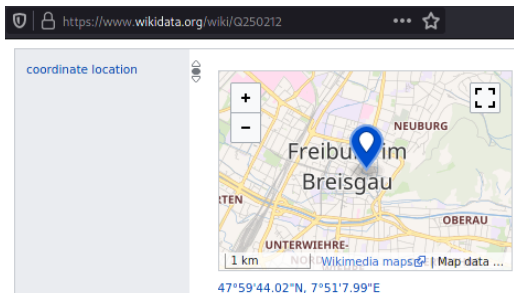
All other relations are calculated with respect to the directed acylic graph, and for each element only relations with the smallest named areas are added. The directed acylic graph speeds up computation since bigger areas which are known to contain the current area can be skipped. It also reduces the number of relations stored in the RDF file.

## 1.1. Problem

Knowledgebases such as Wikidata provide huge amounts of factual data. For example, the entry of the *Freiburg Minster* (Q250212) as shown in Listing 1 contains data about *architectural style* (P149), *coordinate location* (P625), *located in the administrative territorial entity* (P131), and many more. However, it does not contain the information that the minster is located inside *Altstadt Freiburg* (Q445502). *Altstadt Freiburg* has a different *coordinate location* (P625) but the same value for *located in the administrative territorial entity* (P131) as *Freiburg Minster*. It is not possible to infer the spatial



(a) Altstadt Freiburg coordinates


https://www.wikidata.org/wiki/Q250212
coordinate location
NEUBURG
Freiburg im Breisgau
OBERAU
UNTERWIEHRE-
1 km
Wikimedia maps | Map data ...
47°59'44.02"N, 7°51'7.99"E

(b) Freiburg Minster coordinates

**Figure 1.: Point coordinates of *Altstadt Freiburg* (a) and *Freiburg Minster* (b) provided by *Wikidata*.** © Wikidata: All structured data from the main, Property, Lexeme, and EntitySchema namespaces is available under the Creative Commons CC0 License; text in the other namespaces is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply.

relation of these two entities since both only share a link to their parent (P131) and have a single point (P625) location, each as shown in Figure 1.

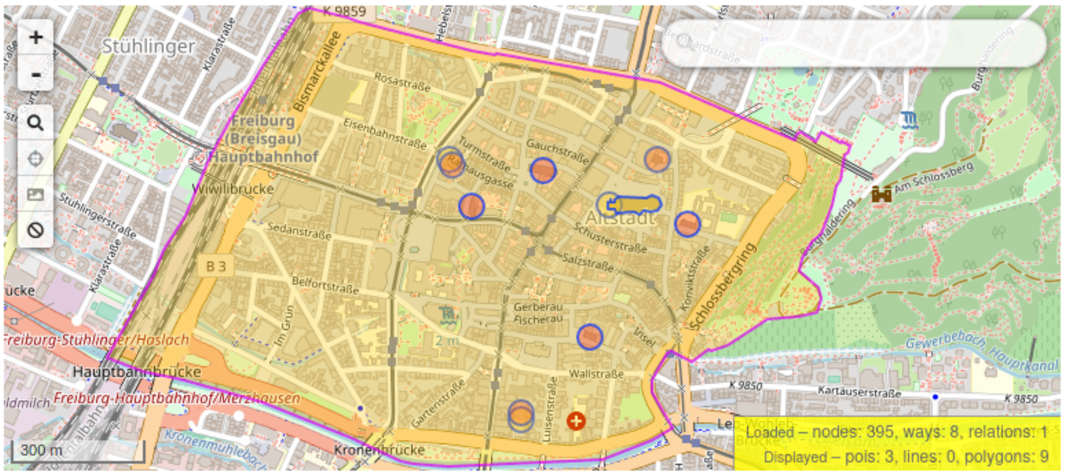
Given the data stored in Wikidata we can not formulate a SPARQL query which selects all *places of worship* inside *Altstadt Freiburg* which are build in a *gothic architectural style* (Q176483). To formulate this query, spatial information about the area of *Altstadt Freiburg* would be required.

OpenStreetMap provides spatial information about the whole planet, but does not contain much factual and historic data. The OpenStreetMap entry for the *Freiburg Minster* as shown in Listing 2 contains information about *wheelchair* accessibility and *opening_hours*, but no information about *architectural style*. The Overpass API query as provided in Listing 3 yields all *places of worship* inside *Altstadt Freiburg* as shown in Figure 2.

## 1.2. Contribution

Our tool *osm2ttl* provides a way to transform OpenStreetMap data into RDF. This transformation generates valid RDF files in both N-Triples and Turtle format. The RDF data can then be fed into a triplestore, allowing the usage of the SPARQL query language. The Overpass query from Listing 3 can then be expressed as shown in Listing 4, and the result can be found in Figure 3.

We explicitly link OpenStreetMap entities to Wikidata entities if the information is provided in the OpenStreetMap input data. Through this link, the data of both independent knowledgebases can be joined, enabling us to answer queries neither could answer alone.



```
Loaded – nodes: 395, ways: 8, relations: 1
Displayed – pois: 3, lines: 0, polygons: 9
```

**Figure 2.:** ***Places of worship*** **inside** ***Altstadt Freiburg*** **provided by** ***Overpass.*** This is the result of the query from Listing 3. © Overpass API; Base map and data from OpenStreetMap and OpenStreetMap Foundation

Jovanovik, Homburg, and Spasic [JHS21] show that GeoSPARQL support in triplestores is lacking. To alleviate this problem, we calculate the intersection and containment for OpenStreetMap nodes, ways, and unnamed areas inside named areas.

Combining the knowledgebases with the explicit containment relations allows us to execute the query from Listing 5, yielding the *Freiburg Minster* as the singular result as shown in Figure 4.

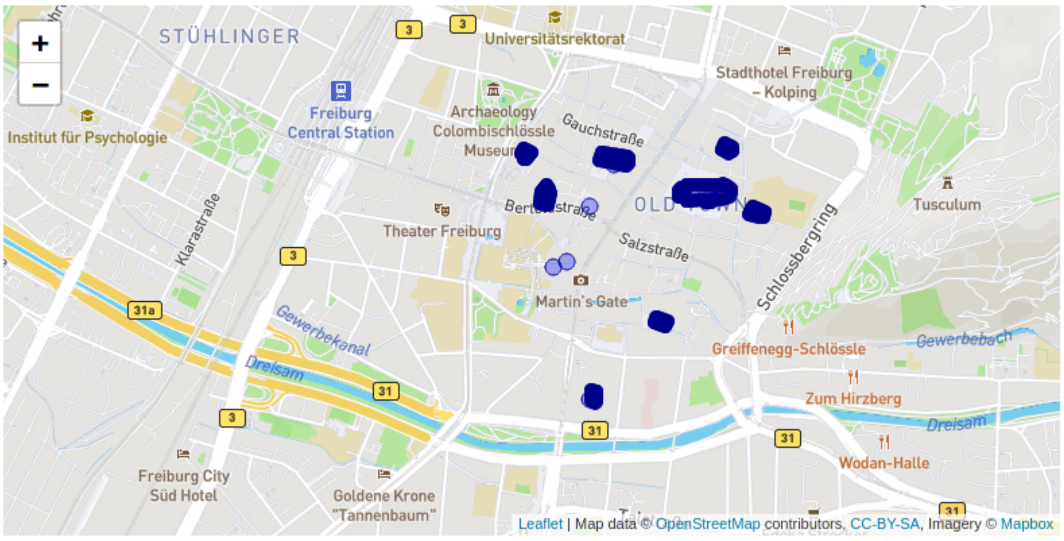


**Figure 3.:** ***Places of worship* inside *Altstadt Freiburg* provided by *QLever* using *osm2ttl*.** This is the result of the query from Listing 4.

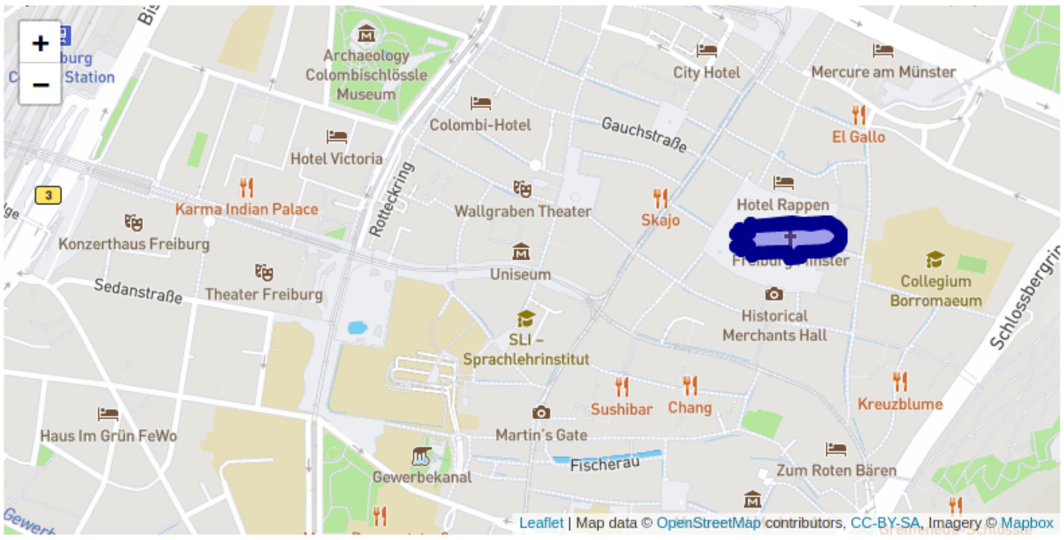


**Figure 4.:** ***Freiburg Minster* inside *Altstadt Freiburg* provided by *QLever*.** This is the result of the query from Listing 5.

**Listing 1:** Excerpt Freiburg Minster in Wikidata (Turtle)

```
wd:Q250212 rdfs:label "Freiburg Minster"@en ;
  skos:prefLabel "Freiburg Minster"@en ;
  schema:name "Freiburg Minster"@en ;
  rdfs:label "Katedralo Nia Sinjorino"@eo ;
  skos:prefLabel "Katedralo Nia Sinjorino"@eo ;
  schema:name "Katedralo Nia Sinjorino"@eo ;
  ...
  wdt:P373 "Freiburg Minster" ;
  wdt:P625 "Point(7.852222 47.995556)"^^geo:wktLiteral ;
  wdt:P131 wd:Q2833 ;
  wdt:P17 wd:Q183 ;
  wdt:P646 "/m/09vk99" ;
  wdtn:P646 <http://g.co/kg/m/09vk99> ;
  wdt:P31 wd:Q2977 ;
  wdt:P1004 "9cd57b23-0e8f-4b4d-bd2d-8013f9a7e0b6" ;
  wdt:P1612 "Cathedral, Freiburg" ;
  wdt:P227 "4132384-1" ;
  wdtn:P227 <https://d-nb.info/gnd/4132384-1> ;
  wdt:P149 wd:Q176483 ;
  wdt:P214 "132561921" ;
  wdtn:P214 <http://viaf.org/viaf/132561921> ;
  wdt:P708 wd:Q260287 ;
  wdt:P244 "n83135744" ;
  wdtn:P244 <https://id.loc.gov/authorities/names/n83135744> ;
  wdt:P454 "20014414" ;
  wdt:P1435 wd:Q11691318 ;
  wdt:P856 <http://www.freiburgermuenster.info/> ;
  wdt:P571 "1200-01-01T00:00:00Z"^^xsd:dateTime ;
  wdt:P910 wd:Q9528736 ;
  wdt:P140 wd:Q1841 ;
  wdt:P112 wd:Q690990 ;
  wdt:P5383 "7843" ;
  wdt:P691 "kn20070305011" ;
  wdt:P417 wd:Q345 ;
  wdt:P7561 wd:Q75106529 ;
  wdt:P138 wd:Q859115 ;
  wdt:P7859 "lccn-n83135744" ;
  wdt:P825 wd:Q345 ;
  wdt:P186 wd:Q121649 ;
  wdt:P2048 "+116"^^xsd:decimal ;
  wdt:P2971 "3221" ;
  wdt:P8596 wd:Q104549317 ;
  p:P373 s:q250212-EAF588A6-F346-417C-942E-3C45FEAD618C .
```

**Listing 2:** Excerpt Freiburg Minster in OpenStreetMap (XML)

```xml
<way id="110404213" visible="true" ...>
  <nd ref="1160187359"/>
  ...
  <nd ref="1160187359"/>
  <tag k="addr:city" v="Freiburg im Breisgau"/>
  ...
  <tag k="amenity" v="place_of_worship"/>
  <tag k="building" v="cathedral"/>
  <tag k="contact:website" v="https://www.freiburgermuenster.info"/>
  <tag k="denomination" v="roman_catholic"/>
  ...
  <tag k="name:en" v="Freiburg Minster"/>
  ...
  <tag k="opening_hours" v="Mo-Sa 10:00-17:00; PH,Su 13:00-19:30"/>
  <tag k="religion" v="christian"/>
  <tag k="tourism" v="attraction"/>
  <tag k="wheelchair" v="limited"/>
  ...
  <tag k="wikidata" v="Q250212"/>
</way>
```

**Listing 3:** Overpass query for all places of worship in Altstadt Freiburg

```
// Altstadt Freiburg
relation(1960176);
// Print outline
out geom;

map_to_area;

(
  way[amenity=place_of_worship](area);
  node[amenity=place_of_worship](area);
  relation[amenity=place_of_worship](area);
);
(._;>;);
out;
```

**Listing 4:** SPARQL query for all places of worship in Altstadt Freiburg

```
PREFIX osmt: <https://www.openstreetmap.org/wiki/Key:>
PREFIX osm: <https://www.openstreetmap.org/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX ogc: <http://www.opengis.net/rdf#>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
SELECT ?osm_id ?hasgeometry WHERE {
  osmrel:1960176
      ↪ (ogc:contains_area+/ogc:contains)|ogc:contains_area+|ogc:contains
      ↪ ?osm_id .
  ?osm_id osmt:amenity "place_of_worship" .
  ?osm_id geo:hasGeometry ?hasgeometry .
}
```

**Listing 5:** SPARQL query for all places of worship in Altstadt Freiburg with Wikidata entry and Gothic architecture

```
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
PREFIX osmt: <https://www.openstreetmap.org/wiki/Key:>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osm: <https://www.openstreetmap.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ogc: <http://www.opengis.net/rdf#>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?osm_id ?wikidata_id ?hasgeometry WHERE {
  ?osm_region rdf:type osm:relation .
  ?osm_region osm:wikidata wd:Q445502 .
  ?osm_region
      ↪ (ogc:contains_area+/ogc:contains)|ogc:contains_area+|ogc:contains
      ↪ ?osm_id .
  ?osm_id geo:hasGeometry ?hasgeometry .
  ?osm_id osmt:amenity "place_of_worship" .
  ?osm_id osm:wikidata ?wikidata_id .
  ?wikidata_id wdt:P149 wd:Q176483 .
}
```

# 2. Related Work

We provide a generic tool for the conversion from OpenStreetMap data into RDF triples. The problem with the generic conversion is defining clear boundaries for relatedness. We focus on the *OpenStreetMap and RDF* aspects of the tool, but provide some insight into the usage regarding *SPARQL* with the added context of *GeoSPARQL*.

## 2.1. OpenStreetMap as RDF

In this section we introduce other projects which convert OpenStreetMap data into RDF. We highlight some key aspects of the respective implementation, and provide short comparisons and differences to our implementation.

### 2.1.1. Sophox

The currently defunct Sophox [Wik21b] project uses Python, and the Python libosmium bindings, to convert the OpenStreetMap data into RDF. They dump the spatial data for each OpenStreetMap object as well-known text (WKT) [IEC16], and use a well designed set of TTL prefixes derived from OpenStreetMap URLs, see Listing 11.

They do not convert possibly invalid tag-keys, but only record the occurrence of such keys. The associated values are thus lost and not represented in the dumped data. They have special rules for handling `wikidata` and `wikipedia` tags to provide the option to link these datasets with the OpenStreetMap data. The RDF data is feed into Blazegraph which implements basic *circle*/*distance* and *box* queries using GeoSpatial [Beb21].

We convert all OpenStreetMap tags into triples using the NT [SC14] or Turtle [CP14] grammar. We reuse the prefixes defined by *Sophox* (Listing 11), with slight modifications regarding the prefix names.

### 2.1.2. LinkedGeoData

Stadler et al. [Sta+12] describe a system for transforming the OpenStreetMap into RDF. They interlink the data with *DBpedia*, *GeoNames* and other datasets providing a *SPARQL endpoint*. They want to provide better support for spatial queries using PostGIS in the future. Using a different set of prefixes, as shown in Listing 12, they provide their own interpretation of the OpenStreetMap data.

We do not introduce a new namespace, and reuse the prefixes defined by *Sophox*, as stated in Subsection 2.1.1, since these are based on the URLs provided by the OpenStreetMap project. Further we try to replicate the data as provided by the OpenStreetMap as-is without changing the semantics, or other aspects of the data.

## 2.2. Linking Knowledge Graphs with OpenStreetMap

Tempelmeier and Demidova [TD20] discuss problems of linking OpenStreetMap data with equivalent entities in knowledge graphs. They show that many *OSM* entities in Germany are not linked to the respective *Wikidata* entities. They looked at different categories of entities, namely *cities*, *train stations*, *mountains*, and *castles*. Within these categories, *cities* have the highest percentage of link coverage with approximately 70%, mountains the smallest with approximately 5%.

We can not improve this link coverage, since we do not correlate knowledgebase entities with the data provided by the OpenStreetMap in any other way. This linkage can be improved outside of our work, and the resulting triples can be merged with ours.

## 2.3. Usage of OpenStreetMap data

Bast, Brosi, and Näther [BBN20] use OpenStreetMap data to provide suggestions for changes to the data. They do not evaluate spatial data as their suggestions are only based on the name-tags in the OpenStreetMap.

We use the provided possibility to link *Wikidata* and *Wikipedia* entries directly using the respective tags. This allows for a reverse search of all OpenStreetMap objects linking to the same knowledgebase entity. These reverse links could be used to provide more insight into the data and suggestions to where other changes could be necessary.

## 2.4. Query languages

In this section we introduce two query languages and highlight their differences and usage. We introduce them here as the support of the GeoSPARQL extension in current software was a factor deciding the scope of our tool.

### 2.4.1. Overpass API

The Overpass API [Wik20] [Lin16] provides a read-only access to OpenStreetMap data. This API allows for the search inside an area or bounding box, near previously selected elements, and by tags. The syntax of this API is focused on spatial data and especially on working with the data provided by the OpenStreetMap project. An example of this syntax is shown in Listing 3.

### 2.4.2. SPARQL and GeoSPARQL

The *SPARQL Protocol and RDF Query Language* [SH13] describes a structured way of retrieving RDF data. RDF triplestores, and search engines like Blazegraph [Beb21] and QLever[BB17] represent the data as graphs. The basic implementations do not handle spatial relations explicitly, and rely on the presence of a applicable predicate linking the requested entities. If such explicit links are missing, a basic triplestore can not be used to query this data.

The GeoSPARQL [PH12] extension defines capabilities and querying methods for the retrieval and usage of spatial data.

Jovanovik, Homburg, and Spasic [JHS21] compare triplestores for their GeoSPARQL capabilities. The proposed compliance benchmark shows that triplestores can be improved in regards of GeoSPARQL.

Version 9.4 of Ontotext GraphDB implements the GeoSPARQL support as an optional plugin [Ont21]. This plugin supports the *coveredBy* function which we store as the *contains* relation. They also provide support for simplifying geometries using the *simplify* function using the Douglas-Peuker algorithm. We provide the same functionality using the same algorithm, but we implemented a minimal threshold for the number of nodes a geometry needs to have in order to be simplified. This threshold allows us to keep objects with fewer nodes exact.

## 2.5. OSCAR

Bahrdt [Bah20] presents "OSCAR: a textual and spatial exploratory search engine for OpenStreetMap data". *OSCAR* requires the use of a custom query language. This enables the author to optimize the data storage and query parser to resolve spatial queries.

*OSCAR* provides text search capabilities using inverted indices and other index structures.

We do not provide search capabilities but the data generated by *osm2ttl* can be added to a generic triplestores. Then all search capabilities provided by the triplestore can be used.

*OSCAR* partitions the world into cells and stores information in them. This allows the system to perform queries based on the relation of these cells. The internal representation allows the author to provide additional search functions not defined in the GeoSPARQL standard. These additional functions are *Along Path*, *Nearby*, *Cardinal Direction*, and *Betweenness*.

We store the spatial relations of the OpenStreetMap data for the duration of the computation. Named areas and their relations are kept in-memory for use in the generation of relations of other objects. All objects which are not named areas are only kept in-memory for the computation of these relations. After the relations for a given object are dumped the object is remove from memory. We therefore convert the spatial information into triples, and allow any RDF capable triplestore to use the transformed data. We only calculate *intersects* and *contains*, and thus a subset of the *Simple Features Topological Relations* defined by the GeoSPARQL standard [PH12].

# 3. Background

In this chapter we introduce OpenStreetMap, what kinds of data are stored, and how the data is stored. We introduce the Resource Description Framework, and well-known text, as formats to store the data. Further, we introduce the concepts of a directed acyclic graph and rectangle trees.

## 3.1. OpenStreetMap

In this section we provide an overview on how data is stored and organized inside the OpenStreetMap. We introduce the relevant data types, data structures, and their relations with each other.

Data is curated by many volunteers around the globe, and changes are made regularly. In this thesis, we ignore these minute-by-minute changes and focus on the more static dumps.

We present some aspects of stored data, and the usage thereof. In the following subsections we introduce the spatial and non-spatial storage used by the OpenStreetMap. We introduce the different storage objects in the order they are used and required, starting with a generic key-value store. We then introduce the geometric objects node, way, and relation. Afterwards, we introduce a non-explicitly stored geometric object area which is used for our approach as explained in Chapter 4.

### 3.1.1. Stored data and download sources

OpenStreetMap stores many variants of data ranging from height lines and speed limits, to opening hours, and public transport data. This multitude of data is represented in the number of projects associated with the OpenStreetMap. As an example, we mention the OpenRailwayMap [Wik21a] as a project documenting the rail infrastructure of the world. They incorporate this data into the OpenStreetMap, allowing anyone access to it. They introduce their own extensive set of tags, and related values providing detailed data[1].

[1] `https://wiki.openstreetmap.org/wiki/OpenRailwayMap/Tagging`

The data stored inside OpenStreetMap is provided under the umbrella of the *OpenStreetMap Foundation* [Fou21] with its basic mission statement[2]:

> The OpenStreetMap Foundation is an international, not-for-profit, democratic organisation with the tasks of supporting the OSM project, running and protecting the OSM database, and making it available to all. [...]
>
> The OpenStreetMap Foundation is there to protect the OSM data to keep it Free and Open.

Downloads for the whole planet are provided free of charge by the OpenStreetMap project. Partial downloads can be obtained from Geofabrik [Kar21]. Geofabrik strips the data of metadata to ensure GDPR compliance[3], which also reduces the dataset size.

> The OpenStreetMap data files provided on this server do not contain the user names, user IDs and changeset IDs of the OSM objects because these fields are assumed to contain personal information about the OpenStreetMap contributors and are therefore subject to data protection regulations in the European Union.

### 3.1.2. Non-spatial data: tags

All data not contributing to spatial shapes is stored as key-value pairs known as tags. These tags are associated with spatial objects (nodes, ways, and relations). To ensure that all current and future data can be stored, all keys and all values are stored as UTF-8 strings. Validation and conversion of data must be handled in the programs consuming OpenStreetMap data.

The choice of using UTF-8 strings allows the OpenStreetMap project to store names and other data in many languages. We omitted most of them in Listing 2, but *name:en* is the key for the name in the English language, and *name:de* is the key for storing the name in the German language. The unsuffixed key *name* stores the locally used name, since translations are provided in the suffixed variants.

The missing enforcement of value type and/or explicit values requires the handling of invalid entries in all consuming products. To illustrate we present the key *admin_level*[4]. This key shall be used to represent the administrative hierarchy inside a country. These levels should always be a positive integer $n \in \{1, 2, \ldots, 11\}$, but e.g. there are 18 nodes in India with `admin_level=A`[5].

[2] `https://wiki.osmfoundation.org/w/index.php?title=Mission_Statement&oldid=7116`
[3] `https://download.geofabrik.de/`
[4] `https://wiki.openstreetmap.org/wiki/Key:admin_level`
[5] `https://overpass-turbo.eu/?w=%22admin_level%22%3D%22A%22+global&R`

### 3.1.3. Spatial data: node, way, and relation

Spatial data is stored in specific objects inside the OpenStreetMap. The simplest object is a *single point* (latitude and longitude) named *node*. A node has *x, y* values representing the *longitude and latitude* respectively. Tags can be associated with a node, but are not required to. In Table 2 we present the number of nodes in the source file in the *nodes (src)* row, and the number of nodes with at least one tag in the *nodes (fact)* row. Most nodes do not contain anything besides the latitude and longitude values.

To represent streets, buildings, and other simple geometries, the *way* data structure is used. It has optional tag storage, but does not contain latitude and longitude data and instead a list of *NodeRef* entries which are node ids. This approach ensures that changes on the nodes are propagated to all ways which they are part of. Through the storage of nodes as an ordered list inside the ways, ways can only represent single lines or closed areas (if the first and last node are the same) consisting of a single shape without holes.

For more complex geometric shapes and other relations, the *relation* structure can be used. It has a similar base structure as ways, but stores a sorted list of *object, role* pairs. For spatial relations the roles *outer* and *inner* are most commonly used. Spatial data is evaluated explicitly in the OpenStreetMap, which means no special order of *inner* and *outer* members is required. Libraries such as *Boost.Geometry* [Boo] require a specific order of these elements as we will explain in Section 4.5.

Other roles such as *member* and *label* exist, e.g. the *label* role is usually applied to a node, and represents where the *name* tag should be located in the rendered tiles.

Organizational relations exist as well, e.g. *Germany (51477)* is a *member_state* of the *European Union (2668952)*. It is also part of *Germany, highway default values (8131479)*, and *Germany, federal public holidays (2188155)*, in both cases with the role *apply_to*.

### 3.1.4. Area

Areas are not stored explicitly inside OpenStreetMap data, but are used by the rendering layer. A relation or way can be marked as an area by using the tag `area:yes`, but other values are used even though they are not encouraged[6]. Relations with `type=boundary` shall be marked as areas as well as closed ways[7].

[6]`https://taginfo.openstreetmap.org/keys/area`
[7]`https://wiki.openstreetmap.org/wiki/Area`

**Listing 6:** Structure of a single RDF line with highlighted space characters.

```
Subject␣Predicate␣Object␣.
```

## 3.2. Resource Description Framework (RDF)

Hayes and Patel-Schneider [HP14] define RDF triples as simple elements separated by a single space (`0x20` only) between each element, and finished by a dot as shown in Listing 6. To allow the storage of data in different languages, Unicode code points are represented in either verbatim UTF-8, a two byte escape sequence (`\uXXXX`), or four byte escape sequence (`\UXXXXXXXX`) depending on its value.

The N-Triple (NT) [SC14] dialect is more verbose than the Turtle (TTL) [CP14] dialect, which is shorter but more complex in terms of syntax rules. Both dialects share the same *Subject*, *Predicate*, and *Object* definitions used for the line structure as shown in Listing 6. TTL allows for multiline data where the subject is omitted, and only predicates and objects are present since the subject does not change. Further, they share definitions for the elements representing these more abstract concepts. These equalities follow from the representation of the triples as a knowledge graph. *Subjects* are internal nodes, *Predicates* are edges, and *Objects* can be internal nodes and leaves.

We will now introduce the explicit types and their usage.

The most universal entry type is an *Internationalized Resource Identifier* (*IRI*). IRIs are primarily used to represent relations and entities, and can be used in all places of an RDF triple. They are the only allowed type for predicates. IRIs can be generic nodes inside the knowledge graph as well as the edges. This allows for predicates to act as subjects, and to store information about the predicate itself inside the same structures.

Subjects can be either an *IRI* or a *Blank Node*. *Blank Nodes* are used to represent more complex relations and relations with metadata.

An example for this kind of relations would be a list of spatial coordinates (nodes) forming a way. In this case, the order of nodes is important and must be stored inside the RDF file. A solution without blank nodes would be to introduce a new relation for each position in the node list, as shown in Listing 7. This would introduce many unique relations (*ex:member_X*), and selecting all nodes of any given way would result in either a complex union, or the inclusion of any triple since the predicate could not be fixed to a single value.

The alternative solution introduces *blank nodes* and uses up to three predicates as shown in Listing 8. The predicate *ex:member* links the way to a *blank node* for each

**Listing 7:** A way as a node list with unique predicates (TTL)

```
osmway:42 ex:member_0 osmnode:23
osmway:42 ex:member_1 osmnode:3
osmway:42 ex:member_2 osmnode:21
osmway:42 ex:member_3 osmnode:42
osmway:42 ex:member_4 osmnode:1337
osmway:42 ex:member_5 osmnode:4711
osmway:42 ex:member_6 osmnode:815
```

real node. This *blank node* links to each real node via *ex:node*, and to the index inside the node list, which is represented as an integer literal via *ex:index*.

Subjects on their own do not contain much data but are required to differentiate between distinct entities which share the same attribute values. The explicit values are often stored inside the literals, or through a combination of literals and relations.

*Literals* store quoted values which are not nodes inside the knowledge graph. They represent factual data such as numbers or names. *Literals* can have suffixes detailing the type of value or in the case of strings the language of the value. A typed literal is of the form `"value"^^IRI`, where *IRI* represents the type the value should be interpreted as, e.g. *xsd:integer* for an integral value. A literal with language information is of the form `"value"@language`, where language is a valid RFC 3066 [Alv01] language tag.

In Figure 5 we provide an overview on where each type can be used. The most universal type is the *IRI*, which can be used in most places. *Literals* have the most specific usage, and can be further specified by a type annotation or a language tag.

## 3.3. Well-known text (WKT)

Introduced as *ISO:13249-3* [IEC16] the *well-known text* standard defines textual representations of spatial features. To represent the data provided by the OpenStreetMap project, a subset of the defined features is sufficient. In our implementation we use *Point*, *LineString*, *Polygon*, and *MultiPolygon* strings to map the spatial structures.

These primitives are contained in a hierarchical order represented by the number of used parenthesis, for $x$ and $y$ values. Integers and floating point numbers can be used as values for coordinates.

A *Point* is represented as `POINT(x y)` where the coordinates are separated by a single whitespace. A *LineString* is a list of *points*, but does no introduce additional parentheses as no further differentiation of features is required. Therefore, a *LineString* has the form `LINESTRING(x0 y0, x1 y1, ...)` where each `xi yi` pair is a *Point*.

**Listing 8:** A way as a node list with blank nodes (TTL)

```
osmway:42 ex:member _:10
_:10 ex:node osmnode:23
_:10 ex:index "0"^^xsd:integer
osmway:42 ex:member _:11
_:11 ex:node osmnode:3
_:11 ex:index "1"^^xsd:integer
osmway:42 ex:member _:12
_:12 ex:node osmnode:21
_:12 ex:index "2"^^xsd:integer
osmway:42 ex:member _:13
_:13 ex:node osmnode:42
_:13 ex:index "3"^^xsd:integer
osmway:42 ex:member _:14
_:14 ex:node osmnode:1337
_:14 ex:index "4"^^xsd:integer
osmway:42 ex:member _:15
_:15 ex:node osmnode:4711
_:15 ex:index "5"^^xsd:integer
osmway:42 ex:member _:16
_:16 ex:node osmnode:815
_:16 ex:index "6"^^xsd:integer
```

Well-known text *Polygons* allow for exclusions inside, thus they introduce a second level of parentheses to group the used *LineStrings*. A polygon describing a simple square with sides of length 10 has the form `POLYGON((0 0, 0 10, 10 10, 10 0))`. In contrast to the OpenStreetMap area, the first corner is not repeated as the last element. To remove a triangular area from the previously introduced square the following form is used: `POLYGON((0 0, 0 10, 10 10, 10 0),(2 3, 5 7, 7 4))`. The area of a *Polygon* is always defined by the first *LineString*, and each subsequent *LineString* removes regions of the polygon.

*MultiPolygons* combine multiple *Polygons*, adding another level of parentheses. They are written as `MULTIPOLYGON(((x0 y0,...), ...), ...)`, allowing for the addition with the first *LineString* of each entry, and exclusions via all other *LineString* entries, of multiple areas.

Other WKT primitives are defined, but are not used in this thesis.

## 3.4. Directed Acyclic Graph (DAG)

In this section we introduce the concept of a directed acyclic graph. This variant of a generic directed graph allows for the usage of some properties in the implementation

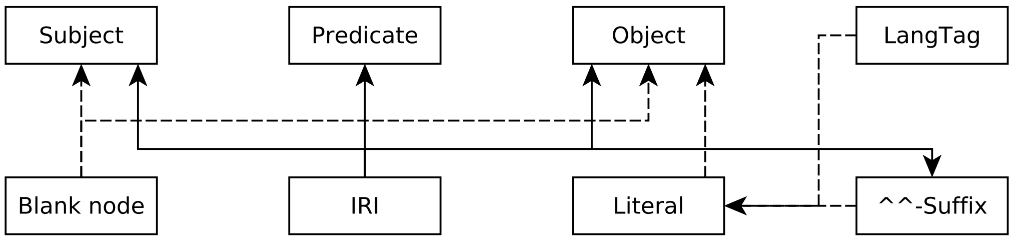


**Figure 5.:** ***RDF* overview.** This diagram shows the possible usage for *IRIs* (everywhere except LanguageTags), *Blank nodes* (as Subject and Object), and *Literals* (only as objects).

of this thesis, and is further explained in Subsection 4.7.3. We provide a visual representation of the various graph types in Figure 6.

### 3.4.1. Undirected and directed graph

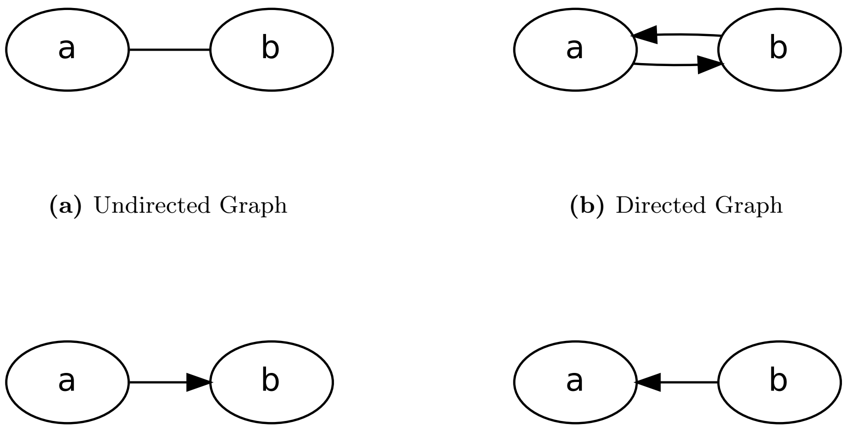
A undirected graph is a graph $G = (V, E)$ where $V$ is the set of vertices, and $E$ is the set of edges $E = \{\{x, y\} \mid x, y \in V\}$. Edges are represented as sets of two vertices, thus $\{x, y\} = \{y, x\} \mid x \neq y$.

A directed graph is a graph $G = (V, E)$ where $V$ is the set of vertices, and $E$ is the set of edges $E \subseteq V \times V$. Edges are represented as ordered pairs of two vertices, thus $(x, y) \neq (y, x) \mid x \neq y$.

A vertex $v$ is reachable from $u$ when there exists a path $(e_0, e_1, \dots)$ with $e \in E$, such that there exists a sequence of vertices $(v_0, v_1, \dots)$ for which there are edges such that every edge following another edge starts with the same vertex as the previous ends. This ensures the direction and connectivity of the path in the directed graph setting. In the undirected graph setting, each edge must only contain one node from the previous edge to ensure connectivity.

### 3.4.2. Directed Acyclic Graph

A directed acyclic graph introduces the requirement that there exists no vertex $v$ which can reach itself via a path. This restricts the set of edges $E$ to not contain any self-loops $(x, x) \mid x \in V$. Further, it prohibits the addition of edges to the graph $G$ which would enable the creation of a path with a sequence of vertices $(v_0, v_1, \dots, v_n) \mid v_0 = v_n$.



**(a)** Undirected Graph

**(b)** Directed Graph

**(c)** Directed Acyclic Graph Variant 1

**(d)** Directed Acyclic Graph Variant 2

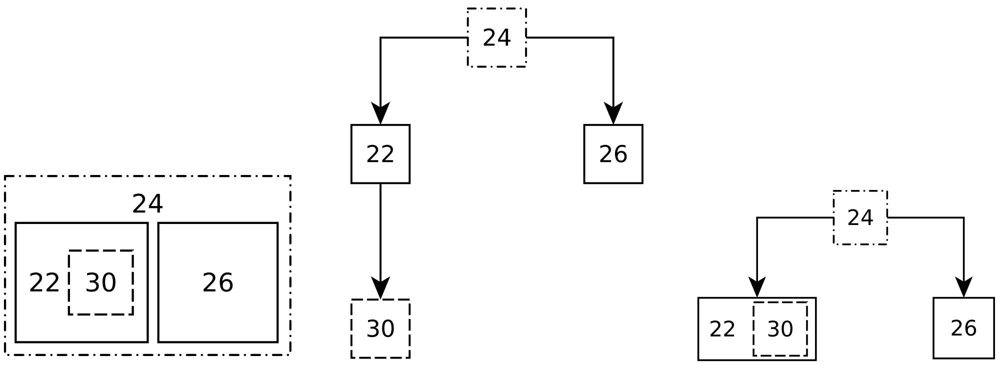
**Figure 6.: Various types of graphs with two nodes $a$ and $b$. (a):** An undirected graph with the nodes $a$ and $b$ with one edge. This graph allows the traversal from $a$ to $b$ and vice versa. **(b):** A directed graph with the nodes $a$ and $b$ with two edges. The edge $a \rightarrow b$ allows for the traversal from $a$ to $b$. The edge $b \rightarrow a$ allows for the traversal from $b$ to $a$. **(c/d):** A directed graph with the nodes $a$ and $b$ with only one edge. Transforming the graph **(b)** into a directed acyclic graph requires that one edge must be removed. This can either be the edge $b \rightarrow a$ resulting in the graph **(c)** or the edge $a \rightarrow b$ resulting in the graph **(d)**.

## 3.5. R-tree

A tree can be seen as a special case of an directed acyclic graph. Depending on the direction of the edges, it holds that $indeg(v) = 1$ or $outdeg(v) = 1$ for any node except the root, where it holds that $indeg(v) = 0$ or $outdeg(v) = 0$ respectively. This ensures that from any child node, only a single path to the root node exists. In the more general directed acyclic graph, multiple paths can exist.

In most trees, the value of each node determines in which subtree smaller or bigger values can be found.

A rectangle tree (R-tree) is a balanced tree for storing spatial data. Rectangles are stored as nodes inside an R-tree, and all children are contained in the rectangle of their parent node. The rectangle of the root node encapsulates all contained data.



**(a)** Areas as grid **(b)** Areas as binary tree **(c)** Areas as R-tree

**Figure 7.: Different representation for spatial areas.** The spatial areas *22*, *24*, *26*, and *30* are represented in different formats. The outline style of the areas is consistent between **(a)**, **(b)**, and **(c)** and depends on the nesting level. **(a):** The areas as found in the world. *30* is inside *22*, *22* and *26* are inside of *24*. **(b):** The nesting levels of the areas as an unbalanced tree. **(c):** The tree from **(b)** transformed into an R-tree, empty slots are omitted. The area *24* is the envelope of the other areas and contains them all.

# 4. Approach

In this section, we present the steps performed by our tool *osm2ttl*.

We start with refining the broad problem into smaller subproblems to solve, and then present how they are solved. Further, we provide some insights into the memory usage, storage and usage of spatial objects, and the use of multithreading.

## 4.1. Problem Definition

In Section 1.1, we introduced the problem of combining OpenStreetMap data with knowledge bases such as Wikidata. To provide a solution to this problem we split it into smaller problems. First, we need to read the OpenStreetMap data. Based on this input, we transform the facts into valid RDF triples. Additionally, we need to store the geometry as text using WKT. The spatial relations must be calculated, and stored as RDF triples.

To calculate the spatial relations, we need access to all relevant spatial objects, and their order and spatial hierarchy.

As the OpenStreetMap contains more than 6.5 billion nodes alone, calculating the relations between each pair of objects would be an enormous task. We only use a subset of objects for these calculations, and we ignore all objects which do not have any tag as introduced in Subsection 3.1.2. We choose this subset since the usage of untagged objects as RDF subjects would yield near to no information, given only spatial coordinates are associated with these.

We further reduce the complexity of calculations by calculating the relations of named areas with other named areas by creating a directed acyclic graph. This allows us to sort all remaining objects into these named areas without the need to calculate all relations. Additionally, this allows us to reduce the number of spatial calculations since the topological order implied in the directed acyclic graph can be used.

## 4.2. Overview

*osm2ttl* is written in *C++* using features from the *C++17 Standard.* We use the *GCC* compile chain in combination with *Make* and *CMake* to generate the executable

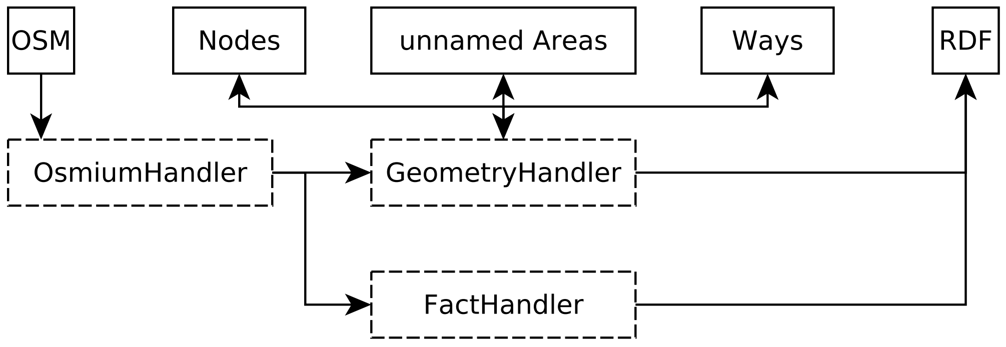


**Figure 8.:** ***osm2ttl*** **dataflow between disk and memory.** Solid boxes represent files on disk, dashed boxes are classes inside the *osm2ttl* program. OpenStreetMap data is read from the *OSM* file by the *OsmiumHandler*. Data is then passed to the *FactHandler* and dumped into the output *RDF* document. Also, the data is passed to the *GeometryHandler* which stores *Nodes*, *Ways*, and *unnamed Areas* on disk. It later retrieves the stored objects and dumps the spatial relations into the *RDF* document. The *GeometryHandler* contains the *R-tree* and *DAG*, as data is not stored on disk we omitted these in the diagram.

file. We parse the provided data from the OpenStreetMap once, and transform the information into our own representation. To reduce the memory load we store spatial objects on disk as shown in Figure 8. Named areas are kept in RAM to improve calculation speed for spatial relations. These calculations are performed multithreaded using the *OpenMP* library which is introduced in Section 4.8.

The incoming data is handled by the *OsmiumHandler* which is responsible for reading OpenStreetMap data, and passing it to the *FactHandler* and *GeometryHandler*. These handlers transform the data either directly into RDF in the case of the *FactHandler*, or into indexing structures.

In the following sections we provide insights into how these actions are performed.

## 4.3. Reading OpenStreetMap (osmium)

The OpenStreetMap project provides data dumps in different formats. We use the *osmium* [Top13] library which provides readers for the various formats, and resolves the referential geometries stored inside the OpenStreetMap into explicit geometries.

To interface with the *osmium* library, we defined a *OsmiumHandler*. The *osmium* library provides the handler interface which defines methods for each OpenStreetMap object type. These members are called for each element according to their type.

**Listing 9:** Basic osm2ttl arguments selecting facts

```
--no-facts                   Do not dump facts
--no-area-facts              Do not dump area facts
--no-node-facts              Do not dump node facts
--no-relation-facts          Do not dump relation facts
--no-way-facts               Do not dump way facts
--add-area-envelope          Add envelope to areas
--add-area-envelope-ratio    Add area/envelope ratio to areas
--add-way-envelope           Add envelope to ways
--add-way-metadata           Add information about the way structure
--add-way-node-order         Add information about the node members in ways
```

Additionally, the area method is defined and called after an area is recognized by *osmium*. The *OsmiumHandler* converts the data provide by the *osmium* library into our own representation. This conversion is needed because the *osmium* library stores data in a reused buffer to limit the memory consumption. The converted objects are dumped as RDF as explained in Section 4.4 using the *FactHandler* and the geometries are stored as explained in Section 4.6 using the *GeometryHandler*.

The *OsmiumHandler* and *FactHandler* contain logic to filter the amount of dumped data, as shown in Listing 9. We added options to add the envelope (containing, axis-aligned rectangle) for each way and area object. The optional way metadata contains information about the number of nodes and unique nodes, as well as the information whether or not the way is closed. Further, we added the possibility to dump the members of ways, including the order information as presented in Section 3.2 and Listing 8.

## 4.4. Writing valid RDF

We need to ensure that all triples written by *osm2ttl* are valid RDF triples. Since we need to handle many of the possible variants of data, we need to implement most of the underlying grammars. We convert the facts contained in OpenStreetMap data into triples: these facts contain different languages, and special characters. We store geometries that are formatted as well-known text which contain only ASCII characters, but need to be annotated with type information. Additionally, we need to store the spatial relations as their own predicates.

### 4.4.1. Implementation

To ensure the correct representation of all data entered by the volunteers of the OpenStreetMap project, we implemented most of the N-Triples and Turtle grammars.

**Listing 10:** Single RDF line implementation

```
template <typename T>
void osm2ttl::ttl::Writer<T>::writeTriple(const std::string& s,
                                           const std::string& p,
                                           const std::string& o) {
  _out->write(s + " " + p + " " + o + " .\n");
}
```

This allows us to dump all data without the need to omit any triple, unlike the Sophox project introduced in Subsection 2.1.1.

We first implemented the overlapping parts of the grammars as explicit classes and interfaces representing each data type and triple position. This approach ensured compile-time correctness, but was very slow as each RDF value was created on the heap. To solve the speed issue, we dropped the compile-time correctness guarantee, and replaced the objects with functions working on `std::string_view` arguments. Using `std::string_view` instead of `std::string` or `const char**`, removed the need to handle *C* or *C++* data types differently. It also sped up most functions using them, since the data is only referenced by address and not copied.

We choose to always return `std::string` instead of writing directly to the output. This decision was made because this way we can ensure that lines are fully assembled before they are written to any output buffer.

To improve writing speeds, `_out->write(...)` as found in Listing 10 redirects the output to a different file for each thread. This eliminates the need for locking the output file, and still ensures correct lines. Writing to `stdout` instead of files disables this optimization, but as `_out->write(...)` is called with a single string correct lines are still guaranteed through the implementation of the `<<` operator of `stdout`.

### 4.4.2. Turtle Prefix

As mentioned in Section 2.1, different projects introduce their own set of Turtle Prefixes. Sophox (Subsection 2.1.1) introduces prefixes based on the URLs used by OpenStreetMap, as shown in Listing 11. In contrast, LinkedGeoData (Subsection 2.1.2) introduces a complete new set of prefixes.

We choose to use prefixes similar to the Sophox project as shown in Listing 13. The only difference between Sophox and our prefixes is that we choose to name the root prefix `osm` instead of `osmroot`. This removes some bytes from the output data.

**Listing 11:** TTL-Prefixes used by Sophox

```
@prefix geo: <http://www.opengis.net/ont/geosparql#>
@prefix schema: <http://schema.org/>
@prefix wd: <http://www.wikidata.org/entity/>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>

@prefix osmm: <https://www.openstreetmap.org/meta/>
@prefix osmnode: <https://www.openstreetmap.org/node/>
@prefix osmrel: <https://www.openstreetmap.org/relation/>
@prefix osmroot: <https://www.openstreetmap.org>
@prefix osmt: <https://wiki.openstreetmap.org/wiki/Key:>
@prefix osmway: <https://www.openstreetmap.org/way/>
```

**Listing 12:** TTL-Prefixes used by LinkedGeoData

```
@prefix dbpedia: <http://dbpedia.org/resource/>
@prefix foa:
    ↪ <http://www.fao.org/countryprofiles/geoinfo/geopolitical/resource/>
@prefix georss: <http://www.georss.org/georss/>
@prefix owl: <http://www.w3.org/2002/07/owl#>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
@prefix wgs84: <http://www.w3.org/2003/01/geo/wgs84_pos#>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>

@prefix lgd: <http://linkedgeodata.org/triplify/>
@prefix lgdo: <http://linkedgeodata.org/ontology/>
```

**Listing 13:** TTL-Prefixes used by osm2ttl

```
@prefix geo: <http://www.opengis.net/ont/geosparql#> .
@prefix ogc: <http://www.opengis.net/rdf#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix wd: <http://www.wikidata.org/entity/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

@prefix osm: <https://www.openstreetmap.org/> .
@prefix osmm: <https://www.openstreetmap.org/meta/> .
@prefix osmnode: <https://www.openstreetmap.org/node/> .
@prefix osmrel: <https://www.openstreetmap.org/relation/> .
@prefix osmt: <https://www.openstreetmap.org/wiki/Key:> .
@prefix osmway: <https://www.openstreetmap.org/way/> .
```

## 4.5. OpenStreetMap as spatial objects

We introduced the storage of the spatial data inside the OpenStreetMap in Subsection 3.1.3. Most geometry libraries require ordered information, as does WKT as introduced in Section 3.3.

We use *Boost.Geometry* for the calculation of spatial relations (Section 4.7), and therefore we need to convert the geometries from the OpenStreetMap representation into the one of *Boost.Geometry.* The *osmium* library handles this conversion and provides its own helpers to store geometries as WKT, but we decided not to use these but the ones provided by *Boost.Geometry.* Using *Boost.Geometry* allows us to stream the WKT result into `std::ostringstream` instances for appending the RDF type information, and setting the number of significant digits using `std::fixed(7)`. We can fix the number of digits after the decimal point to 7 as we will explain in Section 5.1. In Figure 10 we provide an overview over the data flow, going from the OpenStreetMap dump through *Boost.Geometry* to being stored inside an RDF file.

Using *Boost.Geometry* for the representation reduced the amount of work and code needed for the serialization of the spatial data, which we will introduce in Section 4.6.

The aforementioned conversion of the spatial data is required since OpenStreetMap enforces no order of relation members in regard of addition and subtraction of included areas. *Boost.Geometry* and WKT, as presented in Section 3.3, have stronger requirements. Both handle polygons by grouping the *outer* and *inner* members in *rings*, each ring has exactly one *outer* part, adding to the area, and a variable number of *inner* parts subtracting from it, as shown in Figure 9.

## 4.6. Serialization of spatial objects

We use *Boost.Serialize* for serialization of spatial objects. We store the geometry, envelope, and ID of each node. For ways we additionally store the ID of all nodes to speed up some relation calculations. Areas are stored with their area and whether or not they are generated from a closed way.

All nodes, ways, and non-named areas are dumped to disk, as shown in Figure 8, and later retrieved for the spatial lookup as described in Section 4.7. We store named areas inside a vector which is then transformed into an R-tree, as introduced in Section 3.5, and into the directed acyclic graph, introduced in Section 3.4. The delayed insertion into the R-tree improves the runtime, as rebalancing of the tree can be performed after all elements are inserted instead of after each individual insertion.

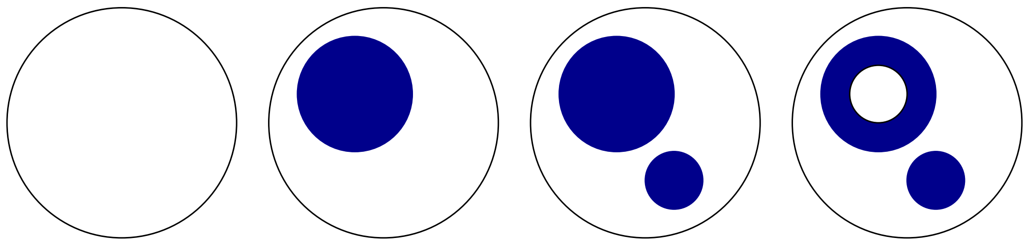

**(a)** Ring with one *outer* member . . . **(b)** . . . and a single *inner* member **(c)** . . . and two *inner* members . . . **(d)** . . . with an additional ring with a single *outer* member

**Figure 9.:** ***Boost.Geometry* ring concept. (a):** A ring with a single geometry. This geometry is used as the *outer* border, everything inside is part of the area of this ring. **(b):** The same ring (a) but with a second (blue) geometry, this blue part is subtracted from the area as it is an *inner* member. Everything inside the blue circle is no longer part of the geometry. **(c):** The same as (b) but with an additional blue geometry. Everything inside the blue circles is no longer part of the geometry. **(d):** Two rings, to add an area inside an *inner* ring, another *outer* ring is required. Everything inside the blue circles is no longer part of the first *outer* ring. Everything marked white is part of a geometry consisting of both rings. If the smaller *outer* ring is not part of the geometry the same result as (c) is obtained.

## 4.7. Geospatial lookup

Calculating all spatial relations using node information only leads to incorrect data. With node information alone containment and intersection of ways can not be solved, as shown in Figure 11.

To ensure that we do not compare every OpenStreetMap entity with every other entity, we introduced some ordering structures.

### 4.7.1. Fixed grid

Our first idea for reducing the number of spatial comparisons was to overlay the planet with a fixed grid, and store for each cell the intersecting and contained entities. A coarse grid with a side length of $> 1.4°$ would keep the memory usage at a manageable level, as for addressing each coordinate a single byte could be used, but did not reduce the number of comparisons sufficiently that we could justify the memory usage. Finer grid sizes were discussed, but larger features like countries would be stored in many cells increasing the memory usage again. It became apparent that storing the explicit

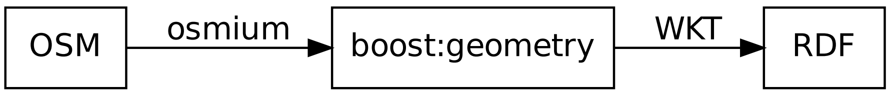


**Figure 10.: Flow of spatial data from OSM to RDF.** We read the spatial data from the OpenStreetMap using *osmium* to convert the data into *boost:geometry* structures. These structures are then converted into RDF using the *WKT* standard.

geometries inside the grid requires too much memory, but storing only the IDs in the grid and the objects in a vector was too memory-intensive as well. This was problematic as the storage structure should be able to handle the whole planet.

With the split of the data into two parts this idea was replaced with an R-tree, and with a directed acyclic graph for named areas.

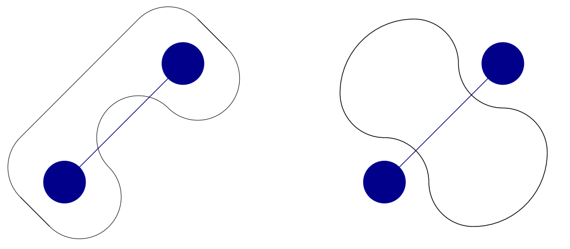
### 4.7.2. R-tree

We store all named areas inside an R-tree, since being contained in an unnamed area does not provide much insight. This R-tree (as introduced in Section 3.5) reduced the number of potential intersection and containment candidates, and is not as memory-intensive as the grid approach at the same time.

As no fixed grid is contained in the R-tree, selections which span multiple grid cells in the fixed grid approach are handled more efficiently, since the computational complexity of combining the results is handled before spatial relations are computed. This was an important step in making the runtimes acceptable as we will discuss in Section 5.1.

### 4.7.3. Directed Acylic Graph

To further improve the containment calculations, we generate a directed acyclic graph. The directed acyclic graph is stored as an adjacency list, and implemented using `std::unordered_map<T, std::vector<T>>`. This directed acyclic graph is generated in a multithreaded manner from the named areas stored in the R-tree. Using multithreading improves generation time significantly, but produces a suboptimal result since unneeded edges may be calculated as shown in Figure 13. The suboptimality is induced by the self-referencing usage of the directed acyclic graph during its creation as described in Algorithm 1. It occurs since areas contributing to


(a) Blue way *only* intersecting the black area

(b) Blue way *intersecting* the black area

**Figure 11.: Node-Way information only. (a):** Both nodes of the blue way are contained in the black area, but the blue line leaves the area. This can only be determined if the exact way geometry is analyzed because the way is only intersecting the black area. **(b):** Both nodes area outside the black area, given only the node information no relation with the black area can be calculated. The blue line is intersecting the black area, this can be found if the explicit way geometry is evaluated.

the set of successors can be examined at the same time, and thus the reducing edge information is not available yet.

The successors of each vertex $v$ can be computed recursively on a given directed acyclic graph. We represent the directed acyclic graph with adjacency lists and therefore the lookup accesses the data in random order as seen in Algorithm 2. To improve lookup speed we precompute the list of successors for each vertex in $G$ in Algorithm 3. This simplifies the lookup to just retrieving a single list of successors as shown in Algorithm 4.

To reduce the number of edges we perform *transitive reduction* on the directed acyclic graph as described in Algorithm 5. We calculate the list of successors for each node and for all of its successors, then we remove those entries in the list of the node successors which already occur in any of the successors' lists

The reduced edge set corresponds to the minimal *contains* relations for the unnamed areas. This allows us to dump the reduced directed acyclic graph as RDF without the need to recheck each explicit geometry again.

### 4.7.4. Spatial lookup of non-named areas

The directed acyclic graph did not only improve the computation of the *ogc:contains_ area* predicate, but is also used for the *ogc:intersects*, and *ogc:contains* predicates which are calculated for all non-named area entities. In Figure 13 we present the possible

**Algorithm 1** Create DAG

```
function CREATEDAG( $areas$ )
    $areas \leftarrow sort(areas, \downarrow)$                    ▷ Sort areas by size from big to small
    for $area \in areas$ do
        $skip \leftarrow \{\}$                                 ▷ Empty set for skipping areas
        $candidates \leftarrow rtreeContains(area)$            ▷ Candidates from rtree
        $candidates \leftarrow sort(candidates, \uparrow)$     ▷ Sort candidates by size from small to big
        for $c \in candidates$ do
            if $c \notin skip$ then
                if $area \subseteq c$ then
                    $addEdge(area, c)$                           ▷ Insert contains information into DAG
                    $skip \leftarrow skip \cup c \cup dagSucc(c)$    ▷ Add successors (using the DAG) to the skipset
                end if
            end if
        end for
    end for
end function
```

**Algorithm 2** Directed Acyclic Graph: FindSuccessors

```
function FINDSUCCESSORS( $src$ )
    $tmp \leftarrow \{\}$                                ▷ Empty list to store parents
    findSuccessorsHelper($src, tmp$)                     ▷ Collect all parents
return unique($tmp$)                                     ▷ Only return unique elements
end function

function FINDSUCCESSORSHELPER($src, tmp$)
    if $src$ found then                                  ▷ Node with id $src$ exists?
        for $p \in P(src)$ do                            ▷ Add direct parents
            $tmp \leftarrow tmp \cup p$
        end for
        for $p \in P(src)$ do                            ▷ Add parents of parents
            findSuccessorsHelper($p, tmp$)
        end for
    end if
end function
```

**Algorithm 3** Directed Acyclic Graph: PrepareFindSuccessorsFast

```
Require: $successors$ is a map $id \rightarrow parents$ denoted as $sucessors_{id}$
  function PREPAREFINDSUCCESSORSFAST
      for  do
          $sucessors_{src} \leftarrow findSuccessors(src)$
      end for
  end function
```

**Algorithm 4** Directed Acyclic Graph: FindSuccessorsFast

```
Require: $successors$ is a precomputed map $id \rightarrow parents$ denoted as $sucessors_{id}$
  function FINDSUCCESSORSFAST( $src$ )
      $result \leftarrow \{\}$                                  ▷ Empty set
      if $sucessors_{src}$ then                                ▷ Successors for $src$ exist?
          $result \leftarrow sucessors_{src}$                  ▷ Return successors
      end if
  return  $result$
  end function
```

**Algorithm 5** Reduce DAG

```
  function REDUCEDAG( $dag$ )
      for $vertex \in dag$ do
          for $p \in P(vertex)$ do                             ▷ For each successor ...
              for $successor \in findSuccessors(p)$ do         ▷ ... get successors
                  if $candidate \in P(vertex)$ then            ▷ If successor of vertex and an-
                                                                 other successor
                      $removeEdge(vertex, successor)$          ▷ Remove edge
                  end if
              end for
          end for
      end for
  end function
```

containment relations of the named areas *Faculty of Engineering (TF)*[1], *Building 51*[2], and the three entrance doors: *southern entrance*[3], *lift entrance*[4] and *northern entrance*[5]. As the *Faculty of Engineering* encapsulates *Building 51* the doors are contained in both, using the directed acyclic graph we can remove all candidates which are higher up in the hierarchy because containment in these areas directly follows from being contained in a lower level. We therefore can skip the spatial comparisons for these areas, resulting in a speedup of the whole process.

We first calculate for each tagged node in which area it is contained. Nodes represent single points, and therefore are always contained given that the border of an area is treated as inside. The information which named area contains the node is stored in a global hash table.

The speedup using the directed acyclic graph can be applied for ways as well. With the usage of the information which tagged nodes are members of the way, and the information where these nodes are contained, we can compute some containment information before querying the R-tree for candidates. Since the directed acyclic graph ensures tagged nodes are contained in the smallest areas possible, we do not lose information through these computations. After all tagged nodes are accounted for, the same candidate retrieval and containment and intersection checks are performed as previously for unnamed areas.

The complete method handling ways is described in Algorithm 6. For the calculation of node and unnamed area relations, some parts are omitted but the basic structure is identical.

## 4.8. OpenMP

We use the *OpenMP* [Boa15] library for iterating over the stored spatial data. This allows us to use the available computing capabilities of the machine running *osm2ttl*.

*OpenMP* offers different settings to distribute the workload. We ran some experiments and provide a synthetic benchmark exploring these options.

Using a task-based distribution was always slower than other settings, since the management overhead is the highest. Using the default settings led to significant waiting times since near the end, only a single thread kept working for multiple hours as all others were finished with their shares.

We found that using `schedule(dynamic)` for most loops optimized the runtime, with the results of our synthetic benchmark shown in Table 1. `schedule(dynamic)` uses

[1]`https://www.openstreetmap.org/way/4498466`
[2]`https://www.openstreetmap.org/way/98284318`
[3]`https://www.openstreetmap.org/node/2110601105`
[4]`https://www.openstreetmap.org/node/5190342871`
[5]`https://www.openstreetmap.org/node/2110601134`

**Algorithm 6** dumpWayRelations

```
function dumpWayRelations( $nodeData$ )
    for $way \in ways$ do
        if $way \in DAG$ then                                    ▷ Skip if named area
            $continue$
        end if
        $skipNodeContained \leftarrow \{\}$
        $skipIntersects \leftarrow \{\}$
        $skipContains \leftarrow \{\}$
        for $node \in way$ do
            if $node \in nodeData$ then
                $skipNodeContained \leftarrow skipNodeContained \cup nodeData[node]$
            end if
        end for

        for $c \in rtree(way)$ do
            $doesIntersect \leftarrow false$
            if $c \in skipIntersects$ then
                $doesIntersect \leftarrow true$
            else
                if $c \in skipNodeContained$ then
                    $doesIntersect \leftarrow true$
                    $skipIntersects \leftarrow skipIntersects \cup successors(c)$
                    $writeIntersectsRelation(way, c)$
                else
                    if $intersects(way, c)$ then
                        $doesIntersect \leftarrow true$
                        $skipIntersects \leftarrow skipIntersects \cup successors(c)$
                        $writeIntersectsRelation(way, c)$
                    end if
                end if
            end if
            if $doesIntersect$ then
                if $c \in skipContains$ then
                    $continue$
                else
                    if $contains(way, c)$ then
                        $skipContains \leftarrow skipContains \cup successors(c)$
                        $writeContainsRelation(way, c)$
                    end if
                end if
            end if
        end for
    end for
end function
```

a fixed size for work packages. The serialized data from Section 4.6 is read one OpenStreetMap element at a time in each thread, this ensures that each thread locks shared resources only for the time needed. Reading one OpenStreetMap at a time also reduces the memory needed to store the other elements of a work package. The fixed size work packages reduce the management overhead which is required for `schedule(guided)`.

`schedule(guided)` starts with large initial chunks, and reduces the size of work packages as time goes on. In theory, this allows for faster running tasks to pick up more work, without the risk of reserving too many work units for any given thread. We observe worse performance than `schedule(static)` in our synthetic benchmark.

One exception from this choice is the dump of the named area relations, as no calculations are necessary. We only convert the edges into RDF triples which is reasonably fast, and `schedule(static)` is more efficient in this case.

| | Items | auto | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|---|---|
| static | 512 | 47.91 | 55.74 | 41.66 | 38.49 | 32.97 | 36.70 | 40.33 | ⋆ |
| | 567 | 55.78 | 55.95 | 61.12 | 46.66 | 38.95 | 40.54 | 45.99 | 56.42 |
| | 934 | 120.86 | 114.86 | 93.63 | 83.96 | 88.74 | 85.09 | 90.85 | 110.74 |
| | 1024 | 151.06 | 135.39 | 106.64 | 96.00 | 96.60 | 100.95 | 111.49 | 122.44 |
| dynamic | 512 | 35.53 | 35.77 | 37.17 | 34.87 | 34.59 | 37.48 | 33.78 | ⋆ |
| | 567 | 39.69 | 41.82 | 42.94 | 42.59 | 38.78 | 38.88 | 41.95 | 46.32 |
| | 934 | 83.44 | 82.29 | 82.76 | 83.35 | 82.13 | 79.56 | 80.05 | 80.20 |
| | 1024 | 96.31 | 94.44 | 96.78 | 93.65 | 97.30 | 94.57 | 96.56 | 93.42 |
| guided | 512 | 47.67 | 47.00 | 46.30 | 45.88 | 45.02 | 46.38 | 48.20 | ⋆ |
| | 567 | 53.41 | 52.77 | 55.83 | 56.33 | 54.17 | 53.03 | 56.49 | 53.85 |
| | 934 | 122.41 | 122.92 | 120.87 | 125.14 | 119.42 | 122.15 | 120.15 | 122.36 |
| | 1024 | 151.01 | 151.78 | 151.14 | 152.60 | 151.52 | 153.13 | 152.80 | 151.15 |

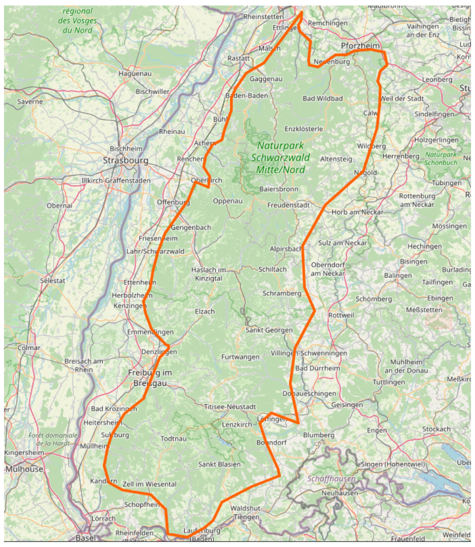
**Table 1.: OpenMP durations for various *schedule* settings.** Number of milliseconds required to consume the given number of items with simulated workload. Depending on the given chunk size different runtimes are achieved for the same number of items. *auto* has a different effects for each of the modi (*static*, *dynamic*, *guided*). More details are available in [Boa15, Table 2.5, Page 60–61]. ⋆: No values calculated as this machine has 8 threads and the work would have been distributed such that each thread would only work on a single (initial) package (512/64 = 8).

We explicitly disabled automatic data sharing for variables using the `default(none)` directive. This reduces and limits undesired variable access or changes inside the multithreaded parts of *osm2ttl*.

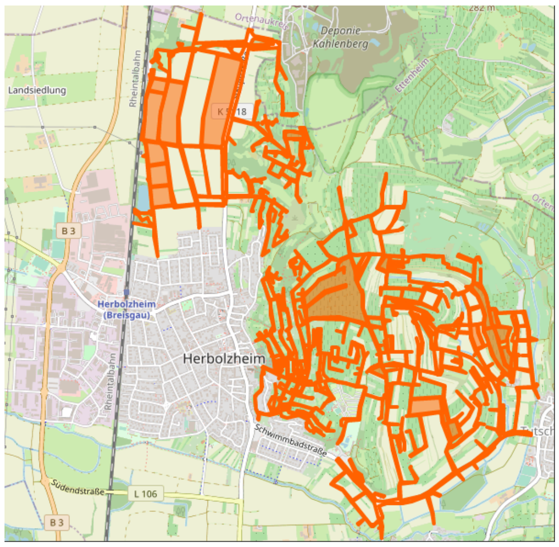
## 4.9. Runtime

We do not provide an asymptotic runtime analysis for the implemented data structures and algorithms. A runtime analysis based on the number of objects alone does not provide good estimates, as many simple objects are handled faster than a few complex ones. Using the area covered by objects as a measurement unit has the same problem, as large but simple objects, e.g. *Black Forest*[6], can be processed faster than smaller complex ones, e.g. *Grass near Herbolzheim*[7] which consists of multiple inner and outer parts, as shown in Figure 12.

An in-depth analysis of the runtime in regard to the possible requirements and difficulties would have exceeded the time frame available for this thesis. We provide an experimental evaluation of the running time in Section 5.2 and rationale in Section 5.4.



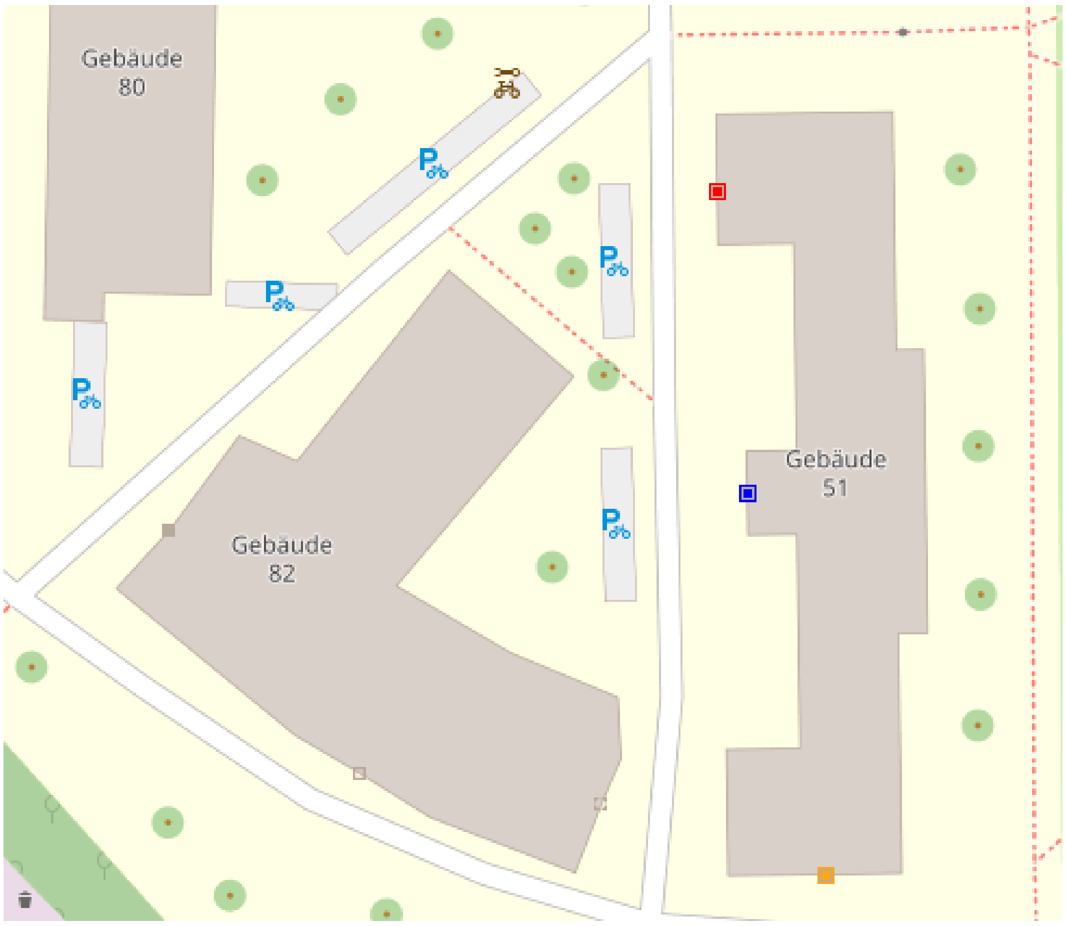
(a) *Black Forest*



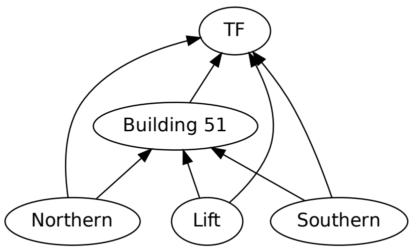
(b) *Grass near Herbolzheim*

**Figure 12.: Complexity of OpenStreetMap objects shown by two examples.** The *Black Forest* **(a)** covers a large area with only a few corners. The *Grass near Herbolzheim* **(b)** covers only a small area but uses different inclusion and exclusion areas and many corners. Provided by: OpenStreetMap | Map data © OpenStreetMap contributors

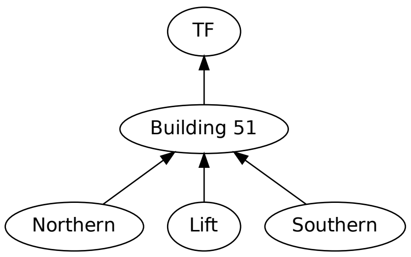
[6]`https://www.openstreetmap.org/relation/3255371`
[7]`https://www.openstreetmap.org/relation/9186582`



**(a)** © OpenStreetMap contributors; Base map and data from OpenStreetMap and OpenStreetMap Foundation; modified



**(b)** Relations without DAG



**(c)** Reduced DAG

**Figure 13.: Non-reduced and reduced directed acyclic graph for Building 51. (a):** Spatial location of the *Building 51* inside the *Faculty of Engineering (TF)* with highlighted entrances: northern (entrance), southern (entrance), and lift (entrance). **(b):** Containment relations between all objects. **(c):** Directed acyclic graph used to reduce relations of all objects. *Building 51* is part of the *Faculty of Engineering (TF)*.

# 5. Experiments

In this section we present our results of using *osm2ttl*. We discuss some reasons why the observed effects occur.

All runs were performed on two machines with *AMD Ryzen 7 3700X 8-Core/16-Threads Processors with* $3.6 - 4.4 GHz$ and *128GB Ram* ($4 \times 32$*GB, 2133MHz)* each. Data was read from and written to a *2TB NVME Samsung 970 Evo+* for each machine, and stored locally.

## 5.1. Real world data

We run *osm2ttl* on different datasets provided by Geofabrik[Kar21]. The datasets in increasing size are *Regierungsbezirk Freiburg*[1], *Baden-Württemberg*[2], *Germany*[3], *Europe*[4], and the whole *Planet*[5]. In Table 2 we provide basic statistics for each dataset. Further, we split the runtime of the geometry calculation into smaller parts in Table 3.

For each dataset we created a compressed output file doubling the size of the input. This effect can be explained by the conversion from a binary format into a text format, and the storing of coordinates multiple times.

The provided OpenStreetMap data stores coordinates only on a per node basis. This allows the construction of the geometry of ways, relations, and areas through referencing the nodes by their ID. To create valid WKT geometries we are required to store the explicit coordinates at each position of each geometry.

Further, we can not store floating point numbers using the original 32-bit representation used by protobuf since we require text to be WKT compliant. Storing the geometries as *well-known binary* (WKB [IEC16] defined alongside WKT) would reduce the output size, but would limit the compatibility with triplestores as capabilities for handling binary input is required. The chosen WKT representation allows for

[1] `https://download.geofabrik.de/europe/germany/baden-wuerttemberg/freiburg-regbez.html`
[2] `https://download.geofabrik.de/europe/germany/baden-wuerttemberg.html`
[3] `https://download.geofabrik.de/europe/germany.html`
[4] `https://download.geofabrik.de/europe.html`
[5] `https://planet.openstreetmap.org/pbf/`

| | freiburg | bawue | germany | europe | planet$^\star$ |
|---|---|---|---|---|---|
| input (.pbf) | 123 M | 482 M | 3.4 G | 23 G | 55 G$^\circ$ |
| output (.bz2) | 257 M | 1.1 G | 7.5 G | 48 G | 100 G$^*$ |
| ram usage | 2.47 G | 2.82 G | 13.51 G | 90.11 G | 108 G$^\diamond$ |
| runtime facts | 2.50 m | 10.07 m | 71.51 m | 7.56 h | 16.68 h |
| runtime geometry | 50.13 s | 44.96 m | 15.30 h | 12.28 d | 47.02 d |
| runtime | 3.34 m | 55.02 m | 16.49 h | 12.59 d | 47.72 d |
| fact triples | 22.05 M | 89.51 M | 598.73 M | 3.38 B | 4.67 B |
| geometry triples$^\odot$ | 11.83 M | 47.99 M | 342.09 M | 2.28 B | 4.88 B |
| nodes (src) | 12.42 M | 46.71 M | 337.22 M | 2.72 B | 6.50 B |
| nodes (fact) | 605.65 K | 2.03 M | 14.05 M | 96.76 M | 164.42 M |
| nodes (geom) | 605.65 K | 2.03 M | 14.05 M | 96.76 M | 164.42 M |
| ways (src) | 1.82 M | 7.72 M | 55.03 M | 326.43 M | 718.51 M |
| ways (fact) | 1.80 M | 7.65 M | 54.61 M | 319.97 M | 706.17 M |
| ways (geom) | 1.80 M | 7.65 M | 54.61 M | 319.97 M | 706.17 M |
| relations (src) | 31.60 K | 101.67 K | 677.84 K | 5.51 M | 8.38 M |
| relations (fact) | 31.58 K | 101.63 K | 677.56 K | 5.51 M | 8.37 M |
| relations (geom) | 0 | 0 | 0 | 0 | 0 |
| areas (src) | 1.24 M | 5.43 M | 39.62 M | 234.80 M | 499.37 M |
| areas (fact) | 1.24 M | 5.43 M | 39.62 M | 234.80 M | 499.37 M |
| areas (geom) | 1.24 M | 5.43 M | 39.62 M | 234.80 M | 499.37 M |

**Table 2.: Results from real world runs.** File sizes as reported by *ls -lh*, RAM usage as reported by *osm2ttl/libosmium*. $^\star$**:**Incomplete/older code used, not enough time to rerun with final code. $^\circ$**:** Input contains metadata. $^*$**:** 12 fraction digits, no additional metadata. $^\diamond$**:** Node locations (during dump) stored on disk and not in RAM. $^\odot$**:** Added relations in both directions, e.g. *ogc:contains* and *ogc:contained_by.*

easy storage and retrieval of spatial data with any triplestore, as long as the WKT strings are not dropped during insertion into the triplestore.

Using text to store coordinates results in the usage of 8 bit per digit. Initially we stored 12 fraction digits, but later we reduced this to 7 as all digits after the $7^{th}$ are always 0. These digits are zero, because the protobuf files store coordinates as integers with a factor of $10,000,000$. Using this factor to divide the integers a floating point value with 7 fraction digits is obtained. This dropped the required space for a single floating point value from 15-17 characters (120-136 bits) to 10-12 characters (80-96 bits) before compression.

| | freiburg | bawue | germany | europe | planet$^\star$ |
|---|---|---|---|---|---|
| R-tree generation | 0.02 s | 0.08 s | 0.60 s | 3.50 s | 7.34 s |
| Sorting named areas | 0.01 s | 0.03 s | 0.29 s | 2.13 s | 1.47 s |
| Generating non-reduced DAG$^\odot$ | 1.95 s | 91.80 s | 57.46 m | 10.67 h | 26.80 h |
| Fast lookup non-reduced DAG | 0.03 s | 0.66 s | 5.43 s | 1.67 m | 2.36 m |
| Reducing non-reduced DAG | 0.05 s | 0.12 s | 0.99 s | 7.47 s | 15.88 s |
| Fast lookup DAG | 0.01 s | 0.11 s | 1.29 s | 13.50 s | 26.48 s |
| Dump DAG and area relations | 0.10 s | 0.36 s | 2.79 s | 18.93 s | 33.36 s |
| Relations for unnamed areas$^\odot$ | 0.72 s | 9.18 s | 3.23 m | 1.79 h | 6.98 h |
| Relations for nodes$^\odot$ | 5.10 s | 4.33 m | 1.70 h | 18.90 h | 60.51 h |
| Relations for ways$^\odot$ | 40.96 s | 38.88 m | 12.58 h | 10.97 d | 43.08 d |

**Table 3.: Geometry parts runtime.** $^\star$**:**Incomplete/older code used, not enough time to rerun with final code. $^\odot$**:** Calculations involving explicit geometries.

## 5.2. Breakdown of *runtime geometry*

In Table 3 we break down the *runtime geometry* times from Table 2 into smaller parts. We denoted parts where operations on the explicit spatial data are performed with $^\odot$. It is apparent that calculations involving many coordinates require more time to be computed. Even for the planet dataset, the runtimes of the non-geometric operations are vanishingly small compared to the geometric parts. Totaling $< 5$ minutes in comparison to 47 days, however, these simpler calculations reduce the amount required in the geometric parts significantly.

The impact of the directed acyclic graph which contributes a huge amount of time to the non-spatial part is further explained in Section 5.3. Irrespective of the use of the directed acyclic graph, the geometric calculations must be carried out, as these determine the relations of the named areas to each other.

To further debug the duration of spatial comparisons, we provide a compile-time option for writing detailed timing information as a `.json` file. This file can be analyzed using the provided `analyse.go` or through other means.

## 5.3. Savings through Directed Acyclic Graph

The values from Table 4 show that for unnamed areas the directed acyclic graph removes 1/4 – 1/3 of the intersection checks and 1/3 – 2/5 of the containment checks. Node containment and way intersection checks are reduced by around 2/5. The DAG has the highest performance impact on way containment where around 4/5 of checks can be skipped for the *europe* dataset. The performance impact of the DAG increases with the depth of the hierarchies inside the dataset.

The amount saved by the directed acyclic graph is restricted by the data in the OpenStreetMap. Named areas are not contained in a single hierarchy, but multiple hierarchies exist, e.g. administrative areas, religious areas, national parks, forests, and many more. As these hierarchies can have overlapping areas, they do not form one single hierarchy.

The directed acyclic graph also contains sparsely filled areas, e.g. relations such as the *Karlsruhe Institute of Technology*[6] shown in Figure 14 which have big envelopes, but the actual area is very small in comparison.

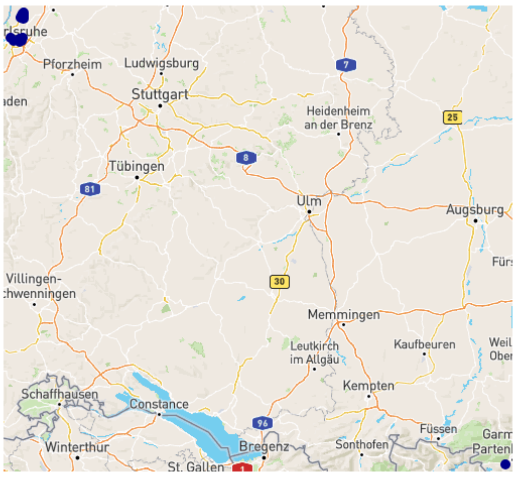


**Figure 14.: Envelope and area of *Karlsruhe Institute of Technology*.** The envelope enclosing the highlighted areas spans Baden-Württemberg and Bavaria and contains multiple bigger cities including Stuttgart, Ulm and Augsburg. Provided by: QLeverUI and Leaflet | Map data © OpenStreetMap contributors, CC-BY-SA, Imagery © Mapbox

To determine the influence of such areas we added the `--minimum-area-envelope-ratio` argument, which removes named areas from the R-tree and the directed acyclic graph if the ratio of *area*/*envelope* is smaller than a given threshold. In Figure 15 Table 5 we show how this reduces the number of areas in which lookups are performed due to their omission in the R-tree. This can speed up the calculations as fewer candidates are returned by the R-tree but can remove small areas too, which in itself speeds up calculations as the directed acyclic graph can be queried if any object is contained in such an area, increasing the runtime again.

[6] `https://www.openstreetmap.org/relation/3350207`

| | | freiburg | bawue | germany | europe |
|---|---|---|---|---|---|
| DAG | vertices | 32.79 K | 146.46 K | 991.31 K | 5.54 M |
| | edges | 40.19 K | 181.20 K | 1.23 M | 7.69 M |
| unnamed area | intersects skipped by DAG | 60.94 K | 272.69 K | 1.61 M | 14.95 M |
| | intersects comparisons | 179.71 K | 678.05 K | 4.44 M | 32.66 M |
| | intersects comparisons yes | 56.37 K | 213.19 K | 1.48 M | 7.61 M |
| | contains skipped by DAG | 46.81 K | 219.58 K | 1.24 M | 12.85 M |
| | contains envelope comparisons | 70.51 K | 266.31 K | 1.85 M | 9.72 M |
| | contains comparisons | 19.84 K | 59.89 K | 398.03 K | 4.76 M |
| | contains comparisons yes | 13.86 K | 37.92 K | 253.33 K | 3.63 M |
| node | contains skipped by DAG | 2.67 M | 15.34 M | 96.58 M | 658.15 M |
| | contains comparisons | 3.54 M | 17.68 M | 118.04 M | 903.54 M |
| | contains comparisons yes | 740.68 K | 2.53 M | 17.62 M | 134.71 M |
| way | in DAG | 31.12 K | 140.18 K | 935.95 K | 4.93 M |
| | intersects skipped by node info | 109.72 K | 535.81 K | 4.32 M | 23.28 M |
| | intersects skipped by DAG | 7.37 M | 55.66 M | 376.04 M | 2.21 B |
| | intersects comparisons | 10.37 M | 66.95 M | 453.93 M | 2.98 B |
| | intersects comparisons yes | 2.06 M | 8.72 M | 62.31 M | 404.87 M |
| | contains skipped by DAG | 7.19 M | 54.80 M | 370.28 M | 2.18 B |
| | contains envelope comparisons | 2.35 M | 10.11 M | 72.40 M | 461.92 M |
| | contains comparisons | 2.27 M | 9.77 M | 69.83 M | 446.00 M |
| | contains comparisons yes | 2.11 M | 9.04 M | 64.95 M | 418.67 M |

**Table 4.: Number of performed and skipped spatial comparisons by entity type.**

| | 0.00 | 0.01 | 0.10 | 0.20 | 0.25 | 0.50 | 0.75 | 0.90 |
|---|---|---|---|---|---|---|---|---|
| freiburg | 33672 | 33667 | 33405 | 33003 | 32674 | 22300 | 4550 | 1191 |
| bawue | 147330 | 147318 | 146322 | 144677 | 143342 | 100414 | 23609 | 5907 |
| germany | 998931 | 998793 | 991235 | 976859 | 968346 | 671116 | 158069 | 39496 |

**Table 5.: Minimum *area/envelop* ratio impact.** Number of named areas stored in the R-tree and DAG with the given minimum *area/envelope* ratio. 50%: around 1/3 of the areas are ignored. 75%: around 1/6 remains. Visualization of the data is available in Figure 15.

## 5.4. Runtime growth with respect to input size

We observe an exponential growth of runtime in the spatial part of the calculations. This exponential growth is partially visible in Figure 15 where we plotted the

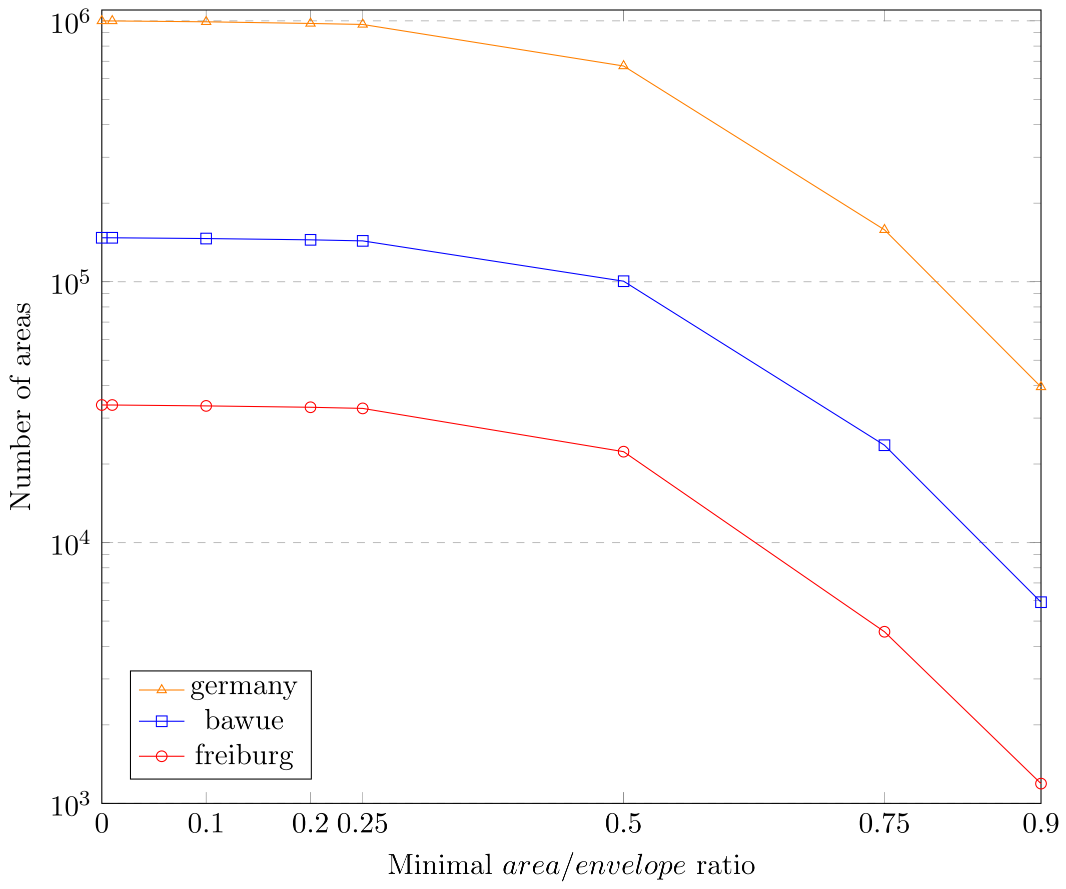


**Figure 15.: Minimum** ***area/envelop*** **ratio impact on the number of areas in R-tree and DAG.** The data for this plot can be found in Table 5.

*area/envelope* ratio effect. *Baden-Württemberg* consists of 4 *Regierungsbezirke*, the first naive assumption would be that *Baden-Württemberg* has roughly four times the amount of data than the *Regierungsbezirk Freiburg*.

This assumption is valid for the number of named areas which are confined to the specific dataset, and which match the used hierarchy which filtered the data, but does not apply to many other relevant data items. The number of comparisons required does not depend on the number of named areas alone. If only the *freiburg* dataset is considered, the areas *Black Forest*[7] and *Protestant Church in Baden*[8] areas are part of this dataset. If we now look at the *bawue* dataset, *Black Forest* intersects both *Regierungsbezirk Freiburg*[9] and *Regierungsbezirk Karlsruhe*[10] The *Protestant Church*

[7] `https://www.openstreetmap.org/relation/3255371`
[8] `https://www.openstreetmap.org/relation/6153362`
[9] `https://www.openstreetmap.org/relation/2106112`
[10] `https://www.openstreetmap.org/relation/22027`

*in Baden* additionally intersects the *Regierungsbezirk Stuttgart*[11] which increases the number of objects possibly contained in it. The growth of such areas does not increase the number of named areas itself, but the number of spatial comparisons.

Moving upwards in the hierarchy of datasets, bigger areas are introduced as well, e.g. the whole German border is not part of the smaller datasets where only partial information can be found. These bigger areas are, in general, more complex geometries and therefore require more time during the directed acyclic graph creation. More comprehensive datasets introduce objects which are not bound to the lower hierarchy levels, e.g. way segments such as *Bundesautobahn 5*[12] (Figure 16) can intersect the highest hierarchy levels of the smaller dataset. This explicit way intersects the *Regierungsbezirk Karlsruhe (Baden-Württemberg)* and *Regierungsbezirk Darmstadt (State of Hessen)*. The first administrative entity which can encapsulate this way is the German state border.

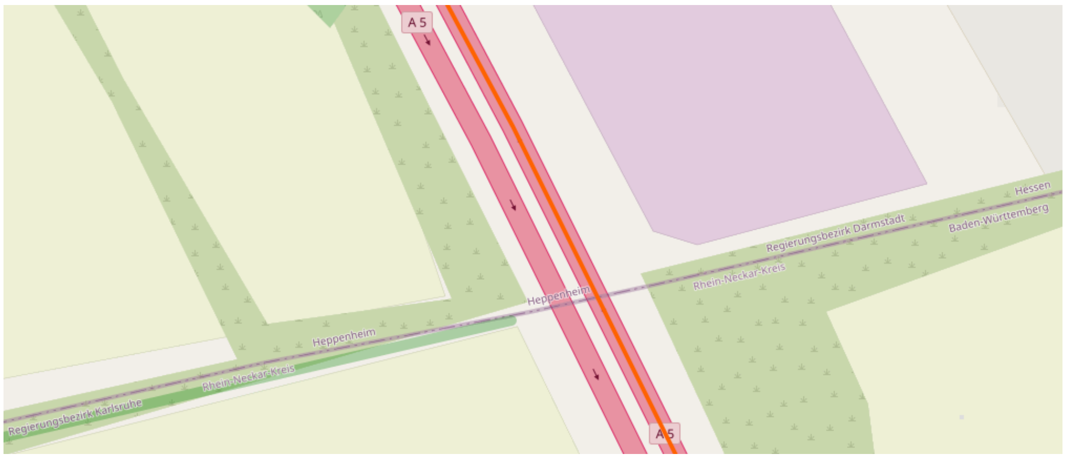


**Figure 16.: Way segment of the *Bundesautobahn 5*.** The highlighted way crosses multiple administrative levels and intersects *Baden-Württemberg* and *Hesse*. It is contained in neither of them but in *Germany*. Provided by: OpenStreetMap | Map data © OpenStreetMap contributors

These problems, combined with the need for explicit geometry comparisons as shown in Section 4.7, are partially responsible for the limited usability of the directed acyclic graph as discussed in Section 5.3.

The R-tree as introduced in Section 3.5 could also perform better if replaced with an index structure which is not axis-aligned, since most real world objects are not aligned with the sides of the axis-aligned envelope boxes. Using a smarter index structure could reduce the number of candidates, and thus reduce the number of geometric calculations.

[11]`https://www.openstreetmap.org/relation/22041`
[12]`https://www.openstreetmap.org/way/143772437`

## 5.5. Exploring data using QLever

Using *osm2ttl* we can explore the data stored in the OpenStreetMap with a triplestore engine, e.g. *QLever*. We performed queries against the transformed OpenStreetMap with, and without linking Wikidata. For all queries the result only contains the subject, which is a valid IRI pointing to the OpenStreetMap site for the given OSM object, and the stored WKT entries. The runtime and result sizes for these queries are presented in Table 6. We provide images of the results for the queries *Q1* and *Q2* only, the graphical representation of the other results does not provide further insights.

| | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 |
|---|---|---|---|---|---|---|---|
| Lines* | 2 | 3909 | 5645 | 358 | 851 | 2130 | 313101 |
| Computation | 184 ms | 4503 ms | 999 ms | 229 ms | 145 ms | 168 ms | 339 ms |

**Table 6.: QLever result statistics for spatial queries.** The *Qx* queries are introduced in Section 5.5. All values are taken from the QLeverUI webpage, the number of displayed items is capped at 100. *: Areas from ways are returned multiple times, once as a *LINESTRING* and once as a *MULTIPOLYGON* geometry.

Q1 ***Places of worship*** **with** ***gothic architecture*** **inside of** ***Altstadt Freiburg***
The previously introduced query in Listing 5 selects all places of worship inside the Altstadt Freiburg, the spatial result is presented in Figure 4.

Q2 ***Buildings*** **inside of** ***Stühlinger***
The query from Listing 14 selects all buildings inside the Stühlinger area, a subset of the spatial result is presented in Figure 17.

Q3 **Elements with tag** ***railway*** **intersecting** ***RVF Zone A*** (Listing 15)

Q4 ***Restaurants*** **in** ***Landkreis Emmendingen*** (Listing 16)

Q5 ***University buildings*** **inside of** ***Berlin*** (Listing 17)

Q6 **Highway parts of the**
***Bundesautobahn 5*** **—** ***Frankfurt - Basel*** (Listing 18)

Q7 **Everything in** ***Freiburg*** (Listing 19)

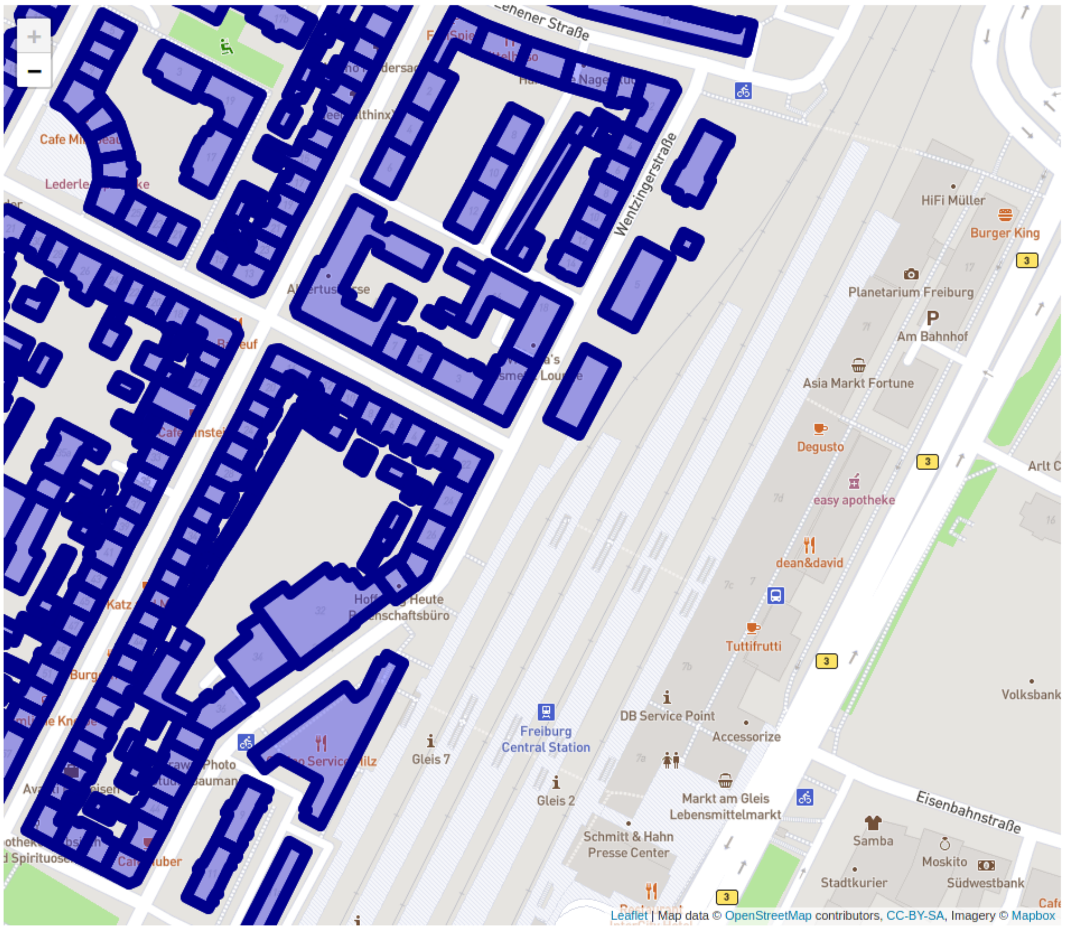


**Figure 17.: Buildings inside *Stühlinger* provided by *QLever* using *osm2ttl*.** This is an excerpt of the result obtained by executing the query from Listing 14. *Stühlinger* is located on the west of the *Mannheim–Karlsruhe–Basel railway (Rheintalbahn)*. The outline and area of buildings inside *Stühlinger* are highlighted with blue color. Provided by: QLeverUI and Leaflet | Map data © OpenStreetMap contributors, CC-BY-SA, Imagery © Mapbox

# 6. Future work

In this section, we present some ideas and thoughts on how to improve *osm2ttl*.

## 6.1. Relations for relations

As shown in Table 2 we do not calculate spatial relations for OpenStreetMap relations which do not form areas, as introduced in Subsection 3.1.4. Calculating these would e.g. allow to state which areas a given bus line intersects. This data could be used for suggestions regarding tickets, especially in more complex situations like traveling beyond fare network boundaries and borders.

One problem with calculating these relations is that the same argument as for the need to calculate the relations for ways explicitly holds. As relations can consist of ways, nodes, and most importantly other relations, we would need to sort the relations to ensure dependencies are resolved first. Further, ways can represent inner and outer borders of area relations. The difference in member semantic through *inner* and *outer* would require the inversion of the contains relation for *inner* ways. Calculating the relations with explicit geometries resolves these issues but increases the computation time as shown in Table 3.

The information which node or way intersects which area can be used to improve the intersection aspect of the relation calculation in the same way as for ways.

## 6.2. Computation on multiple machines

Serializing the named areas and the directed acyclic graph into a file, and adding logic for selecting work-packages could enable computation on multiple machines. The unnamed areas, ways and nodes are already stored on disk while reading the OpenStreetMap data. The calculated node-area must be combined and shared, too. Otherwise, the calculation of way-area intersections would be less efficient.

## 6.3. Better index structure

The current R-tree implementation, introduced in Section 3.5, organizes the contained data using their bounding boxes which are axis-aligned. As many spatial objects in the real world are not perfectly aligned this induces inefficiencies as more candidates are returned than needed. Reducing the number of candidates would speed up the spatial calculations as fewer explicit checks would be performed.

## 6.4. Split sparse/multipart areas

We discussed the problem of sparse and multipart areas, e.g. the *Karlsruhe Institute of Technology* with its *Campus Alpin, Garmisch-Partenkirchen*, in Section 5.3. It could be beneficial to split such areas into multiples. We see the challenges of keeping track of the parts as relations need to reference the whole area. The R-tree and directed acyclic graph require unique identifiers as otherwise the inclusion or exclusion of one part would rule out all others, which could be possible candidates on their own.

## 6.5. Combining intersection and containment checks

We calculate the intersection and containment in separate checks. In the worst case this requires examining the explicit geometry twice. If these calculations can be combined, a speedup could be possible.

## 6.6. More GeoSPARQL predicates

We only implemented *intersects* and *contains* from the GeoSPARQL standard [PH12]. Other GeoSPARQL predicates are *disjoint*, *touches*, *equals*, and more.

## 6.7. additional predicates — OSCAR

Predicates for filters provided by *OSCAR* e.g. *distance* could be implemented between areas. This would allow a coarse estimate for the distance of contained entities. This could enable a triplestore to filter the result set before computing the exact distances using the explicit geometries.

# 7. Conclusion

Our tool *osm2ttl* provides a solid basis for working with OpenStreetMap data and RDF. We implemented functionality to write RDF Triples in both N-Triple and Turtle format according to the respective grammar. We provide a baseline implementation for calculating geometric relations, and some insights into associated problems.

With *osm2ttl* we are able to solve the problem described in Section 1.1. We enabled QLever to solve the problem without changing their implementation. This independence allows for further refinement of the data as proposed in Chapter 6.

We provide a suite of tests to ensure the correctness of *osm2ttl*, and a *Dockerfile* to specify a reproducible environment.

The source code for *osm2ttl* is licensed under the *GNU General Public License v3.0 or later* and available on *GitHub*:

`https://github.com/ad-freiburg/osm2ttl`

# 8. Acknowledgments

I want to thank *Patrick Brosi* for the time he made available for me during the writing of this thesis. He provided deeper insights into the OpenStreetMap, spatial relations, and suggestions regarding details of the implementation. He was always available for questions and provided constructive criticism on the written part of the thesis.

I thank *Prof. Dr. Hannah Bast* for the interesting topic of this thesis, and her interest in the progress, and the results of *osm2ttl*. I am thankful for the provided infrastructure and exclusive access to both machines for the duration of this thesis.

I thank *Prof. Dr. Fabian Kuhn* for agreeing to examine my thesis as well.

I want to thank *all other members of the chair*, which provided pointers on how to write and structure the thesis, and involuntary agreed to hear me ramble about less important details of the implementation not entirely relevant to their work.

Also, I want to thank *Ellie Thiem* and *Johanna Götz* for reading through incoherent sentences and weird words. *Elmar Frerichs* for reviewing the content and improving the clarity of the thesis.

Finally I want to thank my parents, who supported me throughout my studies and especially during times with slow progress.

# 9. References

[Ali+21] Waqas Ali et al. "A Survey of RDF Stores & SPARQL Engines for Querying Knowledge Graphs". In: *CoRR* abs/2102.13027 (2021). arXiv: `2102.13027`. URL: `https://arxiv.org/abs/2102.13027`.

[Alv01] H. Alvestrand. *RFC 3066 — Tags for the Identification of Languages.* 2001. URL: `https://tools.ietf.org/html/rfc3066`.

[Bah20] Daniel Bahrdt. "OSCAR: a textual and spatial exploratory search engine for OpenStreetMap data". PhD thesis. University of Stuttgart, Germany, 2020. URL: `https://nbn-resolving.org/urn:nbn:de:bsz:93-opus-ds-109328`.

[BB17] Hannah Bast and Björn Buchhold. "QLever: A Query Engine for Efficient SPARQL+Text Search". In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017.* Ed. by Ee-Peng Lim et al. ACM, 2017, pp. 647–656. DOI: `10.1145/3132847.3132921`. URL: `https://doi.org/10.1145/3132847.3132921`.

[BBN20] Hannah Bast, Patrick Brosi, and Markus Näther. "staty: Quality Assurance for Public Transit Stations in OpenStreetMap". In: *SIGSPATIAL '20: 28th International Conference on Advances in Geographic Information Systems, Seattle, WA, USA, November 3-6, 2020.* Ed. by Chang-Tien Lu et al. ACM, 2020, pp. 207–210. DOI: `10.1145/3397536.3422342`. URL: `https://doi.org/10.1145/3397536.3422342`.

[Beb21] Brad Bebee. *GeoSpatial · blazegraph/database Wiki · GitHub.* [Online; accessed 16-March-2021]. 2021. URL: `https://github.com/blazegraph/database/wiki/GeoSpatial/94288ff3afdf4fea1dfed84cb3a715ee4773adf8`.

[Boa15] OpenMP Architecture Review Board. *OpenMP Application Programming Interface – Version 4.5.* [Online; accessed 26-April-2021]. Nov. 2015. URL: `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.

[Boo] Boost. *Boost C++ Libraries.* [Online; accessed 1-April-2021]. URL: `https://www.boost.org/`.

[CP14] Gavin Carothers and Eric Prud'hommeaux. *RDF 1.1 Turtle — Terse RDF Triple Language.* Feb. 2014. URL: `https://www.w3.org/TR/2014/REC-turtle-20140225/`.

[Fou21] OpenStreetMap Foundation. *Main Page — OpenStreetMap Foundation,* [Online; accessed 26-April-2021]. 2021. URL: https://wiki.osmfoundation.org/w/index.php?title=Main_Page&oldid=8219.

[HP14] Patrick Hayes and Peter Patel-Schneider. *RDF 1.1 Semantics* (W3C Recommendation). Feb. 2014. URL: https://www.w3.org/TR/2014/REC-rdf11-mt-20140225/.

[IEC16] ISO/ IEC. *Information technology — Database languages — SQL multimedia and application packages — Part 3: Spatial.* ISO/IEC:13249-3:2016. 2016. URL: https://www.iso.org/standard/60343.html.

[JHS21] Milos Jovanovik, Timo Homburg, and Mirko Spasic. "A GeoSPARQL Compliance Benchmark". In: *CoRR* abs/2102.06139 (2021). arXiv: 2102.06139. URL: https://arxiv.org/abs/2102.06139.

[Kar21] Geofabrik GmbH Karlsruhe. *Geofabrik.* [Online; accessed 26-April-2021]. 2021. URL: https://www.geofabrik.de/.

[Lin16] Wen Lin. "OpenStreetMap in GIScience: experiences, research and applications, edited by Jamal Jokar Arsanjani, Alexander Zipf, Peter Mooney and Marco Helbich, Cham, Springer, 2015, 324 pp., US$179.00 (hardcover), ISBN 978-3-319-14279-1". In: *Int. J. Geogr. Inf. Sci.* 30.4 (2016), pp. 823–824. DOI: 10.1080/13658816.2015.1077965. URL: https://doi.org/10.1080/13658816.2015.1077965.

[Ont21] Ontotext. *GeoSPARQL support — GraphDB Free 9.4.0 documentation.* [Online; accessed 16-March-2021]. 2021. URL: https://graphdb.ontotext.com/documentation/9.4/free/geosparql-support.html.

[PH12] Matthew Perry and John Herring. *OGC GeoSPARQL - A Geographic Query Language for RDF Data.* OGC 11-052r4. 2012. URL: https://portal.ogc.org/files/?artifact_id=47664.

[SC14] Andy Seaborne and Gavin Carothers. *RDF 1.1 N-Triples — A line-based syntax for an RDF graph.* Feb. 2014. URL: https://www.w3.org/TR/2014/REC-n-triples-20140225/.

[SH13] Andy Seaborne and Steven Harris. *SPARQL 1.1 Query Language* (W3C Recommendation). Mar. 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[Sta+12] Claus Stadler et al. "LinkedGeoData: A core for a web of spatial open data". In: *Semantic Web* 3.4 (2012), pp. 333–354. DOI: 10.3233/SW-2011-0052. URL: https://doi.org/10.3233/SW-2011-0052.

[TD20] Nicolas Tempelmeier and Elena Demidova. "Linking OpenStreetMap with Knowledge Graphs - Link Discovery for Schema-Agnostic Volunteered Geographic Information". In: *CoRR* abs/2011.05841 (2020). arXiv: 2011.05841. URL: https://arxiv.org/abs/2011.05841.

[Top13] Jochen Topf. *Osmium Library — A fast and flexible C++ library for working with OpenStreetMap data.* 2013. URL: https://osmcode.org/libosmium/.

[Wik20] OpenStreetMap Wiki. *Overpass API — OpenStreetMap Wiki,* [Online; accessed 31-March-2021]. 2020. URL: https://wiki.openstreetmap.org/w/index.php?title=Overpass_API&oldid=2080108.

[Wik21a] OpenStreetMap Wiki. *OpenRailwayMap — OpenStreetMap Wiki,* [Online; accessed 31-March-2021]. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=OpenRailwayMap&oldid=2110469.

[Wik21b] OpenStreetMap Wiki. *Sophox — OpenStreetMap Wiki,* [Online; accessed 15-March-2021]. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=Sophox&oldid=2105584.

# A. Appendix

**Listing 14:** Q2: All *buildings* inside the *Stühlinger*

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osmt: <https://www.openstreetmap.org/wiki/Key:>
PREFIX ogc: <http://www.opengis.net/rdf#>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
SELECT ?osm_id ?hasgeometry WHERE {
  osmrel:1960198
      ↪ (ogc:contains_area+/ogc:contains)|ogc:contains_area+|ogc:contains
      ↪ ?osm_id .
  ?osm_id geo:hasGeometry ?hasgeometry .
  ?osm_id osmt:building ?building .
}
```

**Listing 15:** Q3: Elements with tag *railway* intersecting *RVF Zone A*.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osmt: <https://www.openstreetmap.org/wiki/Key:>
PREFIX ogc: <http://www.opengis.net/rdf#>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
SELECT ?osm_id ?hasgeometry WHERE {
  osmrel:4221993
      ↪ (ogc:contains_area+/ogc:intersects)|ogc:contains_area+|ogc:intersects
      ↪ ?osm_id .
  ?osm_id geo:hasGeometry ?hasgeometry .
  ?osm_id osmt:railway ?railway .
}
```

**Listing 16:** Q4: *Restaurants* in *Landkreis Emmendingen*.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osmt: <https://www.openstreetmap.org/wiki/Key:>
PREFIX ogc: <http://www.opengis.net/rdf#>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
SELECT ?osm_id ?hasgeometry WHERE {
  osmrel:1946117
      ↪ (ogc:contains_area+/ogc:contains)|ogc:contains_area+|ogc:contains
      ↪ ?osm_id .
  ?osm_id geo:hasGeometry ?hasgeometry .
  ?osm_id osmt:amenity "restaurant" .
}
```

**Listing 17:** Q5: *University buildings* inside of *Berlin*.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osmt: <https://www.openstreetmap.org/wiki/Key:>
PREFIX ogc: <http://www.opengis.net/rdf#>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
SELECT ?osm_id ?hasgeometry WHERE {
  osmrel:62422
      ↪ (ogc:contains_area+/ogc:contains)|ogc:contains_area+|ogc:contains
      ↪ ?osm_id .
  ?osm_id geo:hasGeometry ?hasgeometry .
  ?osm_id osmt:building "university" .
}
```

**Listing 18:** Q6: Highway parts of the *Bundesautobahn 5 – Frankfurt — Basel*.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osmt: <https://www.openstreetmap.org/wiki/Key:>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
SELECT ?osm_id ?hasgeometry WHERE {
  ?osm_id geo:hasGeometry ?hasgeometry .
  ?osm_id osmt:ref "A 5" .
  ?osm_id osmt:highway "motorway" .
}
```

**Listing 19:** Q7: Everything in *Freiburg*.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX osmt: <https://www.openstreetmap.org/wiki/Key:>
PREFIX ogc: <http://www.opengis.net/rdf#>
PREFIX osmrel: <https://www.openstreetmap.org/relation/>
SELECT ?osm_id ?hasgeometry WHERE {
  osmrel:62768
      ↪ (ogc:contains_area+/ogc:contains)|ogc:contains_area+|ogc:contains
      ↪ ?osm_id .
  ?osm_id geo:hasGeometry ?hasgeometry .
}
```