Räumliche Partitionierung von OpenStreetMap-Daten mit dem Ziel der Qualitätssicherung

Michael Reichert

Masterarbeit

am

Karlsruher Institut für Technologie (KIT)

Fakultät für Bauingenieur-, Geo- und Umweltwissenschaften
Geodätisches Institut (GIK)

Erstgutachter: Prof. Dr. rer. nat. Martin Breunig
Zweitgutachter: Prof. Dr.-Ing. Stefan Hinz
Betreuer: Dr.-Ing. Norbert Rösch

eingereicht am 23. Januar 2017

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegeben Quellen und Hilfsmittel benutzt habe. Wörtlich oder inhaltlich übernommene Stellen sind als solche kenntlich gemacht. Die Satzung des Karlsruher Instituts für Technologie (KIT) zur Sicherung guter wissenschaftlicher Praxis in der Fassung vom 17. November 2014 wurde beachtet.

Karlsruhe, den 23. Januar 2017

Schreibweisen und Konventionen

Klassennamen sind in dieser Arbeit in serifenloser Schrift gesetzt und werden in allen Programmen, die im Rahmen dieser Arbeit geschrieben wurden, in CamelCase geschrieben.

Methodennamen sind ebenfalls in serifenloser Schrift gesetzt. Meistens werden aus Gründen der Lesbarkeit oft nur die Typen der Argumente, nicht ihre Namen angegeben. Für weitere Details zu den Methoden sei auf die im Anhang befindliche Klassendokumentation der einzelnen Programme verwiesen.

Auch für Klassenattribute wird serifenlose Schrift verwendet.

Die Namen von Datenbanktabellen und ihre Typen sind in serifenloser Schrift gesetzt.

SQL-Befehle und Befehle, die auf der Kommandozeile eingegeben werden müssen, sind in `nichtproportionaler Schrift` gesetzt. Variable Teile dieser Befehle sind *`kursiv gesetzt`*. Kommandozeilenbefehle beginnen mit einem Dollarzeichen.

Wird der Name eines Programms, einer Softwarebibliothek oder ein anderer besonderer Fachbegriff in einem Kapitel erstmals erwähnt, ist er *kursiv* gesetzt. Alle weiteren Verwendungen des Begriffs sind in dem Kapitel nicht mehr kursiv gesetzt, um den Text ruhiger wirken zu lassen.

Für OpenStreetMap wird an vielen Stellen die Abkürzung OSM verwendet.

Lizenz

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1. Technische Grundlagen und Motivation

1.1. Geschichte und Beweggründe von OpenStreetMap

Das OpenStreetMap-Projekt (abgekürzt OSM) wurde von Steve Coast im Sommer 2004 in London gegründet. Ziel war es, Daten für eine freie Karte zu sammeln. Die damals verfügbaren Karten waren entweder teuer oder waren nicht unter einen freien Lizenz zur beliebigen Verwendung verfügbar. Hinter OpenStreetMap steht also eine ähnliche Motivation wie hinter freier Software – der Wunsch, damit tun zu dürfen, was man will [21].

Eine große Zahl Freiwilliger – am 14. November 2016 waren es 3 199 742 Benutzerkonten [84] – trägt die Daten zusammen. Nur ein Teil dieser Benutzer hat überhaupt je ein OSM-Objekt bearbeitet. Im Dezember 2011 hatten von den damals 505 000 Benutzern nur 38 Prozent ein Objekt bearbeitet [76]. Am 31. Juli 2015 waren es ca. 562 000 der 2,2 Millionen Benutzer (26 Prozent), welche jemals ein Objekt editiert hatten [74].

Hinter dem OpenStreetMap-Projekt steht seit 2006 die OpenStreetMap Foundation, welche die Server betreibt und seit dem Wechsel von der Lizenz *Creative Commons Namensnennung Weitergabe unter gleichen Bedingungen 2.0* zur *Open Database License 1.0* (ODbL) im September 2012 auch die Rechte an den Daten hält und sie unter den Bedingungen der ODbL jedem zur Verfügung stellt [12, 21, 121].

OpenStreetMap hat zwar, anders als die Wikipedia, keine Relevanzkriterien, aber erfasst nur Dinge, die vor Ort beobachtbar sind, und keine direkt personenbezogenen Daten wie Klingelschilder. Ausnahmen von dieser Regel existieren nur für die Dinge, die man für eine Karte wichtig erachtet, wie z. B. administrative Grenzen [21, 120, 122].

OpenStreetMap fing vielerorts bei Null an, nur in einigen Gegenden wurde in frühen Jahren durch den Import von Daten aus Fremdquellen eine gleichmäßige Grundabdeckung erreicht. In den anderen Gegenden (v. a. in Europa) hat sich eine schlagkräftige Community gebildet, die die Daten manuell vor Ort erfasst und deren Schwerpunkt im deutschsprachigen Raum liegt (von Projektbeginn bis Dezember 2011 hatten 31 Prozent aller Benutzer überwiegend in Deutschland, Österreich oder der Schweiz Daten erfasst [76]).

Seit November 2010 dürfen die Luftbilder von Bing Maps verwendet werden [22, 76]. In den letzten Jahren wurden in einzelnen Bundesländern Genehmigungen zur Nutzung von WMS-Diensten des Liegenschaftskatasters erteilt.

In England hatte OpenStreetMap im Jahr 2010 in urbanen Gebieten bereits ein längeres Wegenetz als der Datensatz *Meridian 2* der Ordnance Survey (britische Vermessungsverwaltung) [47]. Einem Vergleich von Neis et. al. aus dem Jahr 2011 zufolge hatte OSM zwar ein neun Prozent kleineres Kfz-Straßennetz als der *Multinet*-Datensatz von TomTom, aber das gesamte Netz war 27 Prozent, das für Fußgänger geeignete Wegenetz 31 Prozent

größer. Außerhalb von Ballungsräumen und touristisch stark frequentierten Gebieten war OpenStreetMap doch ziemlich lückenhaft (TomTom hatte dort ein bis zu 80 Prozent längeres Netz) [75].

Das Projekt hat seine Wurzeln in der Freie-Software-Bewegung. Freie Daten zu schaffen, war eine logische Schlussfolgerung. Die Community organisiert sich auf Mailinglisten, in Foren, dem OpenStreetMap-Wiki und weiteren Diskussionskanälen, u. a. diversen IRC-Kanälen[1] [48, 52]. Für Ruhm und Ansehen einer Person in der Community zählen daher nicht seine wissenschaftlichen Veröffentlichungen, sondern was derjenige praktisch als Mapper und/oder Software-Entwickler geleistet hat. Wissenschaftliche Untersuchungen haben sich in der Vergangenheit meist mit der Datenquantität und -qualität sowie der Herkunft und Aktivität der Mitwirkenden beschäftigt. Technische Aspekte werden selten im Detail beleuchtet oder nur in Form studentischer Abschlussarbeiten. Letztere haben kein besonders großes Ansehen, da viele ihrer Verfasser nach Abschluss der Arbeit das Interesse am Thema verlieren.

Viele Quellen dieser Arbeit sind daher nicht wissenschaftliche Veröffentlichungen, sondern meist Beiträge in den Diskussionskanälen der Community sowie die Dokumentation und in einigen Fällen auch der Quellcode veröffentlichter freier Software. Hinzu kommt an einigen Stellen die Erfahrung des Autors als Mitwirkender, welche sich leider schwerlich zitieren lässt.

1.2. OpenStreetMap-Datenmodell

Bei OSM gibt es drei verschiedene Objektarten – Nodes, Ways und Relationen. All diese Objektarten können, müssen aber nicht, Tags haben.

OSM-Daten werden i. d. R. in einem OSM-eigenen XML-Format übertragen, die Dateiendung ist meist `osm` [105]. Zudem ist eine platzsparende Binärvariante gebräuchlich, welche die von Google entwickelten *Protocol Buffers* verwendet [94]. Protocol Buffers sind ein sprach- und plattformunabhängiges Format zur Serialisierung strukturierter Daten, ähnlich wie XML [102]. Bei der Prozessierung von OSM-Daten haben Protocol Buffers in den letzten Jahren das XML-Format verdrängt, da sie schneller zu parsen sind und einen geringeren Overhead haben.

Das Datenmodell folgt dem KISS-Prinzip (Keep it simple, stupid) und ist speziell auf die Bedürfnisse eines Community-Projekts angepasst. Es gibt nur sehr wenige feste Vorgaben, wie bestimmte Dinge zu erfassen sind. Die verschiedenen Objektarten (Nodes, Ways, Relationen) orientieren sich nicht an gebräuchlichen GIS-Standards. Die Einstiegshürde soll für neue Mapper und Entwickler möglichst niedrig sein, denn ohne (neue) Mapper und Entwickler von Drittapplikationen hat das Projekt keine Zukunft [21].

1.2.1. Tags

Mittels *Tags* wird festgelegt, was das jeweilige OSM-Objekt repräsentiert. Bei Tags handelt es sich nicht um Kategorien, wie sie bei anderen Web 2.0-Diensten üblich sind. Tags

[1] Internet Relay Chat

sind bei OpenStreetMap Key-Value-Paare, die frei wählbar sind. Key und Value dürfen jeweils bis zu 256 Zeichen lang sein [105].

Es gibt keine festen, verbindlichen Regeln, welche Tags zu verwenden sind. Da jedoch unzählige Anwendungen OSM-Daten nutzen und eine uneinheitliche Erfassung der weiteren Verarbeitung nicht förderlich ist, haben sich gewisse Regeln herausgebildet. Gebräuchliche Tags sind in der Liste *Map Features* im OSM-Wiki dokumentiert [63]. Dennoch gilt bei OpenStreetMap der Grundsatz *Any tags you like*, d. h. man kann selbst beliebige Tags definieren [16]. Das Datenmodell ist an dieser Stelle absichtlich sehr flexibel gehalten, damit frühe Entwurfsentscheidungen die Entwicklung des Projekts nicht negativ beeinflussen und das Projekt agil bleibt. Beispiele für gebräuchliche Tags sind:

- `highway = residential` für Wohnstraßen
- `shop = supermarket` für einen Supermarkt
- `name = Kaiserstraße` für ein Objekt mit dem Namen „Kaiserstraße“

1.2.2. Nodes

Punkte werden in OSM als *Nodes* bezeichnet. Sie sind die einzigen Objekte in OSM, welche Koordinaten haben. Sie repräsentieren entweder ein punktförmiges Objekt (z. B. einen Baum oder einen Briefkasten) oder dienen als Stützpunkt für einen Way. In letzterem Fall haben sie meist keine Tags. Listing 1.1 zeigt einen als Node erfassten Supermarkt in der OSM-XML-Repräsentation.

Listing 1.1: Beispiel für Node in der OSM-XML-Darstellung [4]

```
<osm version="0.6" generator="CGImap 0.5.8 (26682 thorn-03.openstreetmap.org)"
    ↪copyright="OpenStreetMap and contributors" attribution="http://www.
    ↪openstreetmap.org/copyright" license="http://opendatacommons.org/licenses/
    ↪odbl/1-0/">
  <node id="4400868292" visible="true" version="1" changeset="42144099" timestamp="
      ↪2016-09-14T08:55:21Z" user="wegavision" uid="564585" lat="49.4365025" lon=
      ↪"8.6790851">
    <tag k="name" v="Rewe"/>
    <tag k="opening_hours" v="Mo-Sa 08:00-24:00"/>
    <tag k="shop" v="supermarket"/>
    <tag k="wheelchair" v="yes"/>
  </node>
</osm>
```

1.2.3. Ways

Die Linien in OSM werden *Ways* genannt. Sie können ein oder mehrere Tags haben und referenzieren 1 bis 2000 Nodes[2], die die Geometrie des Ways bilden. Listing 1.2 zeigt eine Straße, die in OSM als Ways modelliert wird, in der OSM-XML-Darstellung.

[2] Die OSM-API lässt es zu, dass Ways mit nur einem Node hochgeladen werden. Sinnvollerweise hat ein Way jedoch mindestens zwei Nodes.

Listing 1.2: Beispiel für einen Way in der OSM-XML-Darstellung (gekürzt) [2]

```
<osm>
  <way id="148588529" visible="true" version="5" changeset="22986547" timestamp="
      ↪2014-06-17T14:57:04Z" user="Uli Maier" uid="729531">
    <nd ref="24956331"/>
    <nd ref="1426415783"/>
    <nd ref="1428967683"/>
    <nd ref="1426415811"/>
    <nd ref="26974332"/>
    <nd ref="1426415812"/>
    <nd ref="26974333"/>
    <tag k="access" v="destination"/>
    <tag k="bicycle" v="yes"/>
    <tag k="foot" v="yes"/>
    <tag k="highway" v="service"/>
    <tag k="name" v="Besoldgasse"/>
    <tag k="width" v="2.4"/>
  </way>
</osm>
```

1.2.4. Flächen

Flächen gibt es im OpenStreetMap mit der aktuellen API 0.6 nicht. Flächen, die nur aus einem äußeren Ring bestehen, der maximal 2000 Punkte hat, werden meist als geschlossene Ways modelliert (erster und letzter Node sind identisch). Alle anderen Flächen werden als Relationen mit dem Tag `type=multipolygon` modelliert. Ob es sich wirklich um eine Fläche und nicht um eine ringförmige Struktur (z. B. Ringstraße) handelt, kann nur anhand der Tags ermittelt werden. Zwar gibt es mit `area=yes` ein Tag zur Kennzeichnung von Flächen, dieses wird aber nur dort verwendet, wo es nicht eindeutig aus anderen Tags hervorgeht, z. B. linienförmige vs. flächenförmige Bahnsteige, Zufahrtsweg vs. Hoffläche.

1.2.5. Relationen

Relationen (engl. *relations*) werden für Objekte und Zusammenhänge verwendet, die sich nicht einfach mit Nodes, Ways und Tags modellieren lassen. Relationen können Tags und eine geordnete Liste an Mitgliedern (engl. *members*) haben. Die Mitglieder haben Rollen (engl. *roles*), welche Strings sind (Strings mit der Länge 0 sind möglich). Das Tag `type` gibt den Relationstyp an. Gebräuchliche Typen sind `multipolygon`, `boundary`, `route` und `turn_restriction`. Diese Aufzählung ist nicht abschließend.

Multipolygone Relationen mit dem Tag `type=multipolygon` für Flächen, die nicht durch einen einzigen Outer-Ring modelliert werden können. Die Mitglieder einer Multipolygon-Relation sind alle äußeren und inneren Ringe, also die Ways, die diese repräsentieren. Äußere Ringe haben die Rolle *outer*, innere Ringe die Rolle *inner*. Ein Ring kann aus einem oder mehreren Ways bestehen. Die Tags, die die Eigenschaften der Fläche

Listing 1.3: Beispiel für eine Relation vom Typ Multipolygon in der OSM-XML-Darstellung (gekürzt) [3]

```
<osm>
  <relation id="5489215" visible="true" version="1" changeset="33838993" timestamp=
      ↪"2015-09-06T18:20:47Z" user="mikecc" uid="14034">
    <member type="way" ref="133745599" role="outer"/>
    <member type="way" ref="369409185" role="inner"/>
    <tag k="landuse" v="forest"/>
    <tag k="leaf_cycle" v="semi_deciduous"/>
    <tag k="leaf_type" v="mixed"/>
    <tag k="name" v="Herrenwald"/>
    <tag k="type" v="multipolygon"/>
  </relation>
</osm>
```

beschreiben, hängen am Relationsobjekt [105, 107]. Insgesamt gibt es etwa 2,5 Millionen Multipolygon-Relationen in OpenStreetMap [118].

Der Begriff *Multipolygon* hat im OpenStreetMap-Umfeld eine andere Bedeutung als im *Simple-Features-Standard* des Open Geospatial Consortium (OGC). Ein OSM-Multipolygon hat meist nur einen äußeren Ring und ist keine Sammelgeometrie. Im OGC-Standard ist ein Multipolygon hingegen eine MultiSurface, deren Elemente vom Typ Polygon sind. Für Flächen, die nur einen äußeren Ring (und beliebig viele innere Ringe) haben, existiert im OGC-Standard der Typ Polygon [51].

Im OSM-Wiki findet sich die Beschreibung, wann ein Multipolygon gültig ist. Im Prinzip entspricht das der *Simple-Features-Spezifikation* der OGC. Darüber hinaus ist es bei OSM jedoch erlaubt, dass zwei innere Ringe sich nicht nur in einem Punkt berühren, sondern eine gemeinsame Kante haben dürfen [107].

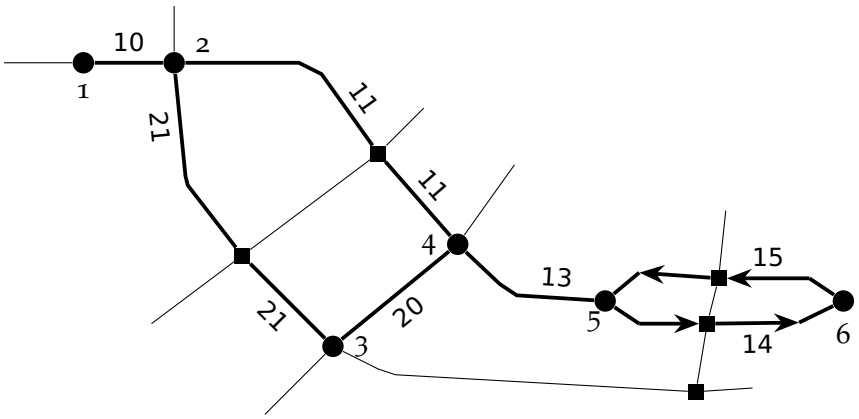
In OpenStreetMap existiert eine Reihe an *Old Style Multipolygons*, welche von der oben wiedergegebenen Spezifikation abweichen. Derzeit existieren noch rund 250 000 Multipolygone, bei denen die Tags am äußeren Ring hängen. Weitere 17 000 Multipolygone folgen einer noch älteren Variante der OSM-Multipolygon-Spezifikation [18, 115]. Bei ihnen finden sich die Tags, welche die Fläche beschreiben, sowohl am äußeren als auch am inneren Ring. Aus dieser Zeit stammen auch noch Multipolygone, deren Mitglieder keine Rolle haben (weder *inner* noch *outer*). Datennutzer sind angehalten, beim Konstruieren von Simple-Feature-Geometrien aus OSM-Daten zuerst die Rollen zu berücksichtigen. Sollte dieser Ansatz nicht zu einer validen Geometrie führen, wird empfohlen, es noch einmal ohne Berücksichtigung der Rollen zu probieren. Listing 1.3 zeigt ein Multipolygon in der OSM-XML-Darstellung.

Grenzen Relationen mit dem Tag `type=boundary` werden für administrative Polygone, Schutz- und Sperrgebiete verwendet. Sie sind wie Multipolygonrelationen aufgebaut. Für Administrative Grenzen gibt es den Key `admin_level` mit den Werten `1` bis `11`. Gebräuchlich sind in Deutschland `2` für die Grenze der Bundesrepublik, `4` für die Länder, `6` für Landkreise und höhere Werte für Gemeinden und deren Ortsteile [112]. Es ist

zwar nicht im OpenStreetMap-Wiki dokumentiert, aber einige Mapper fordern, dass Grenzrelationen streng hierarchisch und möglichst überlappungsfrei (ausgenommen Kondomiate[3]) sein müssen. Das hieße, eine Fläche des Admin-Levels n muss alle Flächen mit Admin-Level $> n$, die mit ihr eine nicht leere Schnittmenge haben, vollständig umgeben. Ein Landkreis muss eine Gemeinde vollständig umgeben, ein Staat muss seine Provinzen/Bundesländer vollständig umgeben. Sie dürfen nicht aus ihm „herausragen“.

Routenrelationen Relationen mit dem Tag `type=route` repräsentieren eine Route, z. B. einen markierten Wanderweg, den Verlauf einer Bundesstraße oder eine Buslinie. Routenrelationen referenzieren alle Ways, die Teil der Route sind. Wenn die Route Verzweigungen oder alternative Wege hat, werden auch diese alternativen Varianten in die Relation mit aufgenommen. Die Mitglieder haben normalerweise keine Rolle (d. h. eine leere Rolle). Wenn ein Way jedoch nur in Richtung des Ways benutzt wird (beispielsweise eine Route über eine Straße mit getrennten Richtungsfahrbahnen), erhält der Way die Rolle *forward*. Wird er ausschließlich entgegen seiner Richtung benutzt,

[3] gemeinsam verwaltete Gebiete




```
<relation>
  <member type="way" ref="10" role=""/>
  <member type="way" ref="11" role=""/>
  <member type="way" ref="14" role="forward"/>
  <member type="way" ref="15" role="forward"/>
  <member type="way" ref="21" role=""/>
  <member type="way" ref="20" role=""/>
  <member type="way" ref="13" role=""/>
  <tag k="ref" v="42"/>
  <tag k="route" v="bus"/>
  <tag k="type" v="route"/>
</relation>
```

Abbildung 1.1.: Beispiel einer Buslinie nach dem „alten“ Public-Transport-Schema. Die Ways mit Pfeilspitzen werden nur in Pfeilrichtung befahren. Haltestellen sind der Übersichtlichkeit wegen nicht in diesem Beispiel enthalten. Neues Schema siehe Anhang A.

erhält er die Rolle *backward*. Die Mitglieder einer Routenrelation sind nicht geordnet, auch wenn auf Seiten der OpenStreetMap-Datenbank Relationen Listen und keine Sets sind [106, 109].

Abbildung 1.1 zeigt ein fiktives Beispiel einer solchen Routenrelation. Die Ways 14 und 15 werden nur in ihrer eigenen Richtung (d. h. vom ersten zum letzten Node, nicht umgekehrt) befahren und haben daher die Rolle *forward*. Der Weg über Node 3 wird von ein oder mehreren Kursen in beide Richtungen bedient. Es ist aber nicht erkennbar, ob diese Kurse nur von Node 1 bis 4 oder die gesamte Strecke fahren.

Linien des öffentlichen Verkehrs wurden in der Vergangenheit auch nach diesem Modell modelliert. Für die Ansprüche der Mapper war das Modell jedoch nicht gut genug. Mittlerweile finden daher ein detailreicheres, oft als schwieriger kritisiertes Taggingschema Anwendung, für das sich später die Bezeichnung *Public Transport Version 2* (kurz PTv2) eingebürgert hat. Alle anderen Routen (für Radfahrer, Wanderer, Reiter, Bundesstraßen usw.) folgen weiterhin dem „alten“ Schema.

Im PTv2-Schema ist die Reihenfolge der Mitglieder relevant. Am Anfang der Mitgliederliste stehen die bedienten Haltestellen. Dies kann entweder die Halteposition (ein Node auf dem Fahrweg) mit der Rolle *stop* oder die Plattform[4] mit der Rolle *platform* oder beides sein. Wenn an einer Haltstelle nur die Halteposition oder nur die Plattform erfasst ist, wird auch nur diese in die Relation aufgenommen. Das Schema verlangt nicht, dass beide Objekte erfasst sein müssen. Existieren sie aber, müssen sie in die Relation aufgenommen werden. Die Liste der Haltestellen muss geordnet sein.

Der Haltestellenliste folgt die Liste der befahrenen Ways, auch diese muss geordnet sein. Pro Variante wird eine Routenrelation erfasst, d. h. die Liste der befahrenen Ways muss einen durchgehenden, lückenlosen Linestring ergeben. Wenn das Fahrzeug Stichfahrten macht (z. B. Fahrten über Kopfbahnhöfe), sind die Ways im Bahnhofsbereich doppelt in der Mitgliederliste vorhanden, weil sie zweimal befahren werden. Im Anhang A ist die PTv2-Modellierung der in Abbildung 1.1 gezeigten Buslinie abgedruckt [34, 101].

Abbiegebeschränkungen Abbiegebeschränkungen werden als Relationen mit dem Tag `type=restriction` modelliert. Das Tag `restricton` gibt an, ob es sich um ein Verbot oder ein Gebot handelt. Beginnt der Wert des Tags mit `no_`, ist es ein Verbot. Beginnt er mit `only_`, ist es ein Gebot. Bei Geboten gibt die Relation an, welches das einzige erlaubte Abbiegemanöver an der Kreuzung ist. Abbiegebeschränkungs-Relationen haben mindestens drei Rollen:

- einen Way mit der Rolle *from*, der den Way repräsentiert, von dem das Fahrzeug kommt
- einen Node mit der Rolle *via*, der den Node der Kreuzung repräsentiert
- einen Way mit der Rolle *to*, der den Way repräsentiert, in den das Fahrzeug nicht hineinfahren darf (Verbot) bzw. hineinfahren muss (Gebot) [108]

[4] bei einfachen Bushaltestellen oft die Position des Haltestellenschilds

1.2.6. Versionsgeschichte

Alle Nodes, Ways und Relationen haben eine Versionsgeschichte (engl. *history*), sodass sich frühere Versionen nachvollziehen lassen und die Objekte, falls erforderlich, auf eine frühere Version zurückgesetzt werden können. Die Versionen eines jeden Objekts sind beginnend mit 1 durchnummeriert. Wenn ein Objekt gelöscht wird, wird eine neue Version angelegt und das Objekt als unsichtbar (im XML `visible="false"` gekennzeichnet) [105]. Muss ein Objekt aus rechtlichen Gründen verborgen werden (z. B. Urheberrechtsverletzung), wird die zu verbergende Version des Objekts von einem Benutzer mit Moderatorprivilegien redigiert (engl. *redactet*). Der history-Call der OSM-API lässt die redigierte Version aus. In der Versionsgeschichte eines Objekts sind dann nicht alle Versionen von 1 bis n enthalten. Ein Beispiel hierfür ist in Anhang B abgedruckt. Streng genommen ist in der API-Dokumentation gar nicht spezifiert, dass Versionsnummern numerisch sind [17].

1.2.7. Änderungssätze

Änderungssätze (engl. *Changesets*) wurden mit der API 0.6 im April 2009 eingeführt und fassen in irgendeiner Art und Weise zusammengehörige Änderungen eines Benutzers zusammen. Bevor ein Benutzer Änderungen in die Datenbank hochlädt, legt er einen Änderungssatz an, lädt darin die Daten hoch und schließt den Änderungssatz danach wieder. Änderungssätze sind jedoch nicht atomar. Ein Objekt kann in einem Änderungssatz mehrfach editiert werden, sodass mehrere Versionen eines Objekts in ein und demselben Änderungssatz hochgeladen wurden. Auch während der Änderungssatz geöffnet ist (bis zu 24 Stunden [17]), können andere Benutzer das Objekt editieren. Änderungssätze haben Tags. Listing 1.4 zeigt die Metadaten eines Änderungssatzes (Tags, Benutzer, Zeitstempel) im OSM-XML-Format. Das Listing in Anhang C zeigt die Änderungen des Änderungssatzes im *OpenStreetMap-Change-XML-Format* (kurz OSC). Weitere Details zum OSC-Format sind im Abschnitt 1.3 zu finden [105].

Listing 1.4: Beispiel für die Metadaten eines Änderungssatzes in OSM-XML-Darstellung [5]

```
<osm version="0.6" generator="CGImap 0.5.8 (1192 thorn-01.openstreetmap.org)"
    ↪copyright="OpenStreetMap and contributors" attribution="http://www.
    ↪openstreetmap.org/copyright" license="http://opendatacommons.org/licenses/
    ↪odbl/1-0/">
  <changeset id="43080643" created_at="2016-10-22T12:52:27Z" closed_at="2016-10-22
      ↪T12:52:28Z" open="false" user="fred_ka" uid="190617" min_lat="49.4726510"
      ↪min_lon="8.4694250" max_lat="49.4729292" max_lon="8.4696162"
      ↪comments_count="0">
    <tag k="comment" v="added store info"/>
    <tag k="created_by" v="JOSM/1.5 (10786 en)"/>
    <tag k="source" v="visit on site"/>
  </changeset>
</osm>
```

Da die Benutzer selbst festlegen, was sie in einem Änderungssatz zusammenfassen und es nur die Beschränkung gibt, maximal 50 000 Objekte in einem Änderungssatz hochzuladen [17], gibt es auch Änderungssätze, die sehr groß sind (die halbe Welt umspannen), keinen Kommentar haben oder deren Änderungen nichts miteinander zu tun haben [105].

1.3. Schnittstellen des OpenStreetMap-Projekts

Planet Der Planet, auch *Planetdump* genannt, ist ein wöchentlich erstelltes Abbild der OSM-Datenbank, das alle nicht gelöschten Objekte in ihrer aktuellen Version enthält. Er ist im OSM-eigenen (BZIP2-komprimierten) XML-Format und als Protocol Buffers erhältlich [40, 95, 105].

Dritte, wie die Geofabrik GmbH oder Mapzen, bieten regionale Extrakte des Planets an, da die Verarbeitung des gesamten Planet mit derzeit 33 GB gewisse Anforderungen an die Hardware stellt [95].

Neben dem Planetdump, der nur die neuste Version aller nicht gelöschten Objekte enthält, stellt die OpenStreetMap Foundation auch noch einen *Full History Planet Dump* bereit, welcher alle Versionen aller Objekte (auch gelöschte Objekte) enthält. Desweiteren existieren noch Dumps aller Änderungsätze im XML-Format (siehe Listing 1.4), aller Änderungssatz-Diskussionen (Änderungssätze können von anderen Benutzern kommentiert werden [27]) und aller Fehlerberichte (*Notes*) [40].

Diffs Zum Planetdump werden minütliche, stündliche und tägliches Updates im OSM-Change-XML-Format (siehe Anhang C) GZIP-komprimiert veröffentlicht. Die Diffs enthalten die neuen Versionen von OSM-Objekten, die seit dem letzten Diff geändert wurden [96].

Das heißt, wenn ein Way geändert wird, sind oft nicht alle von ihm referenzierten Nodes in diesem Diff enthalten, da sie nicht geändert wurden. Wenn z. B. ein Node von Version n eines Ways nicht referenziert wurde, von Version $n + 1$ aber referenziert wird, der Node selbst jedoch nicht gleichzeitig bearbeitet wurde, ist der Node nicht im Diff enthalten. Abbildung 1.2 zeigt einen solchen Fall. Wer die Änderungen der OSM-Daten über die Diffs bezieht verarbeitet und dazu Zugriff auf die Geometrie der Ways benötigt, muss eine Art von Umsetzungstabelle – sei es als Map oder Array im Hauptspeicher oder als PostgreSQL-Datenbank – haben, in der die Koordinaten der Nodes gespeichert sind.

API Die OpenStreetMap-API (oft einfach nur *die API*) ist die älteste und offizielle Schnittstelle von OpenStreetMap. Sie bestimmt das Datenformat und ist heutzutage nur noch für Editoren gedacht, also Anwendungen zum Bearbeiten von OpenStreetMap-Daten. Seit April 2009 ist die Version 0.6 im Einsatz [17]. Es gibt Pläne für eine Version 0.7 und erste Arbeiten daran [15], jedoch befriedigt die API 0.6 die meisten Bedürfnisse und hat anders als ihre Vorgängerversionen eine ausreichende Reife. Die API selbst wird in dieser Arbeit nicht genutzt.

Abbildung 1.2.: Topologische Änderung ohne geometrische Änderung – der Way referenziert nach der Änderung (rechtes Bild) auch den Node 2, welcher selbst aber nicht bearbeitet wurde

Overpass-API Die Overpass-API ist eine zwar nicht offizielle, von der OpenStreetMap Foundation betriebene, aber sehr gebräuchliche Schnittstelle, über die kleine Auszüge aus dem OSM-Daten bezogen werden können. Die Schnittstelle hat eine eigene Abfragesprache, über die thematische und räumliche Abfragen möglich sind. Sie erspart Datennutzern, die eine einfache Spezialkarte im Web betreiben wollen (z. B. eine Karte aller Hotels [73]), den Betrieb einer eigenen PostGIS-Datenbank [93]. Stattdessen stellt der Client die Abfragen an die Overpass-API. Sie ist in dieser Arbeit nicht von weiterer Relevanz.

1.4. Tiles

Tiles (deutsch: Kacheln) sind die Grundlage performanter Kartographie im Web. Während beim Web Map Service (WMS) die Kartengrafik erst bei der Anforderung durch den Client gerendert wird, können Tiles vorberechnet und zwischengespeichert werden. Tiles sind quadratisch und haben klassischerweise eine Kantenlänge von 256 Pixeln. Für hoch aufgelöste Displays („Retina“) werden auch Tiles mit einer Kantenlänge von 512 Pixeln eingesetzt. Tiles sind PNG-Grafiken und werden als Dateien auf dem Tileserver gespeichert.

Da die Tiles vom Server zwischengespeichert werden, sind sie nur in einigen festen Maßstäben verfügbar. Diese Maßstäbe werden meist *Zoomstufen* (engl. *zoom levels*) genannt. Der kleinste Maßstab ist die Zoomstufe 0, auf der die Welt aus einem einzigen Tile besteht. Mit jeder höheren Zoomstufe verdoppelt sich die Anzahl der Kacheln in x- und y-Richtung. Die Zoomstufe n hat also 2^{2n} Tiles [110].

Damit auch an den Rändern der Tiles Beschriftungen gerendert werden können, wird nicht nur der Inhalt des Tiles, sondern auch ein Buffer um das Tile herum abgefragt. Um das Verhältnis von Nutzfläche zur abgefragten Fläche zu verbessern, arbeiten viele Tileserver mit *Meta-Tiles*. Ein Meta-Tile wird gemeinsam gerendert und besteht meistens aus 8×8 Tiles [70].

Tile-basierte Karten nutzen die Web-Mercator-Projektion. Dabei handelt es sich um eine Mercator-Projektion, bei der jedoch die Formeln auf eine Kugel statt einen Ellipsoid angewendet werden [20]. Das beschleunigt die Berechnung. Da bei dieser Projektion die Erde auf einen Zylinder projiziert wird, können polnahe Bereiche nicht abgebildet werden. Die Web-Mercator-Projektion ist nicht für Breitengrade jenseits $\pm 85,07°$ definiert.

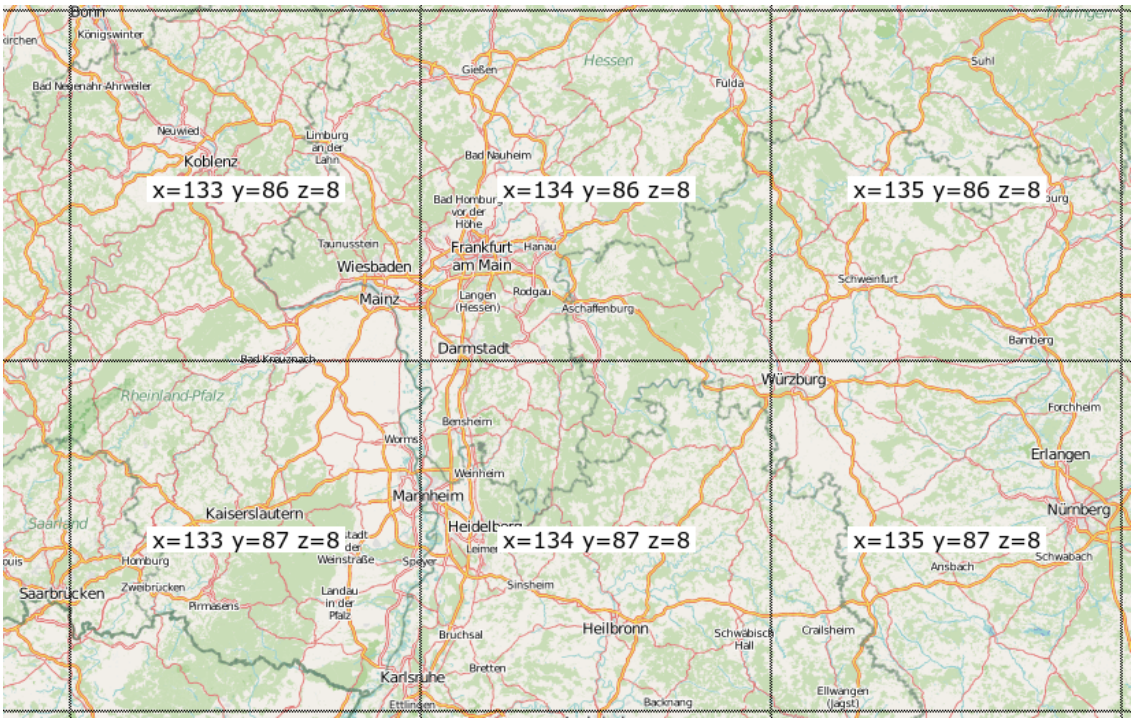


Abbildung 1.3.: Adressen der Tiles auf Zoomstufe 8 [11]

Zur Adressierung (sozusagen als Koordinatensystem) sind verschiedene Verfahren gebräuchlich. Google Maps und die meisten Tileserver auf OSM-Basis identifizieren ein Tile anhand seiner Zoomstufe sowie seines Index in x- und y-Richtung. Der Ursprung dieses Koordinatensystems liegt in der Nordwestecke. Die URL eines Tiles folgt dem Schema `$z/$x/$y.png` (vgl. Abbildung 1.3). Das Tile 5/16/11 wird in der nächsthöheren Zoomstufe 6 also durch die Tiles 6/32/22, 6/32/23, 6/33/22 und 6/33/23 ersetzt [105]. Es wird das Prinzip eines Quadtree angewandt.

Bing Maps verwendet zur Adressierung der Tiles sogenannte *Quadkeys*. Dabei handelt es sich um die Adressen, die bei Quadtrees verwendet werden. Durch jedes Aufteilen des Raumes wird ein Bit hinten an die Adresse angehängt. Der Raum wird abwechselnd in y- und x-Richtung aufgeteilt. Abbildung 1.4 zeigt, wie ein Tile der Zoomstufe 0 in vier Tile der Zoomstufe 1 (00_2, 01_2, 10_2 und 11_2) aufgeteilt wird. Beim erneuten Aufteilen dieser Tiles, verlängert sich die Adresse erneut um zwei Ziffern. Die Zoomstufe eines solchen Quadkeys ist $\frac{l}{2}$ (l sei die Länge des Quadkeys).

Um die Darstellung eines Quadkeys zu verkürzen, ist bei Bing Maps die Darstellung als Zahl zur Basis 4 gebräuchlich, d. h. aus 1110_2 wird 32_4 [110].

1.5. Vektortiles

Bei Vektortiles wird zwischen dem gerasterten Tile und der Datenquelle (meist einer PostGIS-Datenbank) eine Zwischenschicht eingeführt. Anders als der Name es vielleicht vermuten lässt, handelt es sich bei Vektortiles nicht um Vektorgrafiken (z. B. SVG, welches

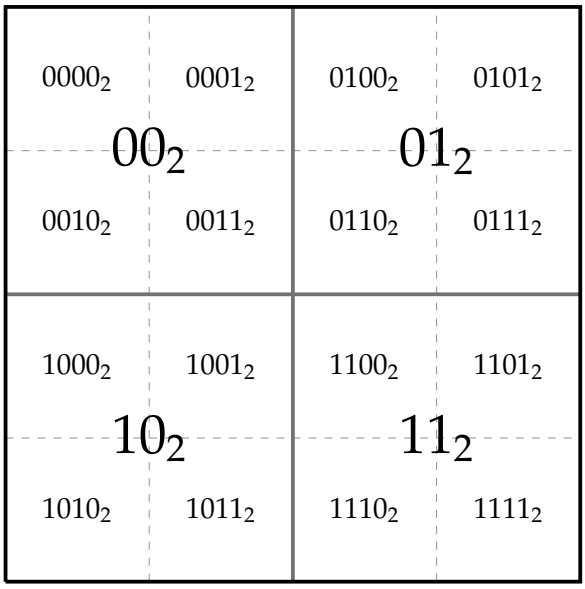


Abbildung 1.4.: Adressierungsschema der Knoten eines Quadtrees

mittlerweile die gebräuchlichen Webbrowser unterstützen), sondern um Vektordaten ohne Angaben, wie diese darzustellen sind.

Die Nutzung von Vektortiles in der Webkartographie kann auf zwei verschiedene Weisen erfolgen. Entweder erhält der Client statt einer Rastergrafik Vektordaten und das Rendering erfolgt clientseitig oder die Vektortiles werden serverseitig abgelegt und on-the-fly gerendert (vgl. Abbildung 1.5). Im ersten Fall kann der Client entscheiden, mit welchem Kartenstil er die Vektordaten rendern möchte und somit direkt Einfluss auf das Erscheinungsbild (Featureselektion, Renderingreihenfolge, Farben, Linienstärken, ...) nehmen. Im zweiten Fall erfolgt das Rendering serverseitig, jedoch wird der Tile-Cache flexibler. Es ist dann möglich, mit einem Server mehrere Kartenstile auszuliefern, da nicht mehr für jeden auszuliefernden Kartenstil die Rastertiles, sondern die rohen Vektordaten vorgehalten werden [36, 64, 69]. Bei serverseitigem Rendering müssen die

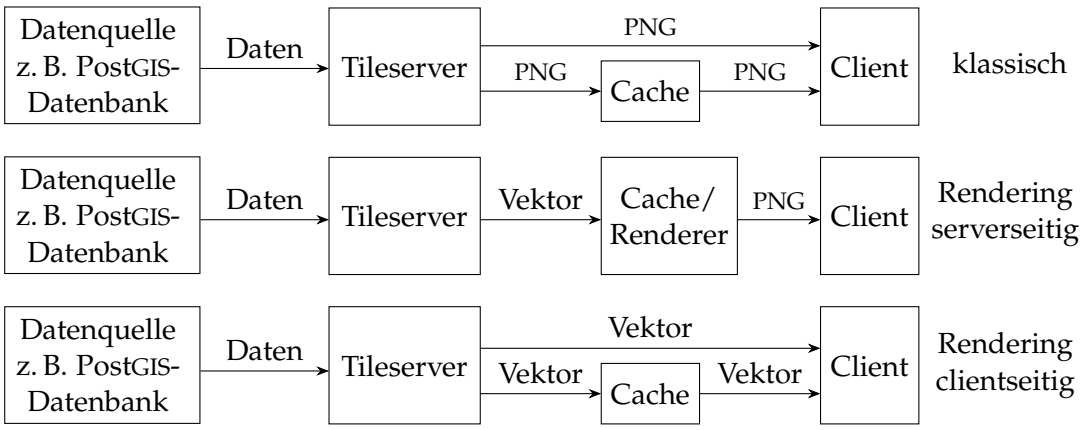


Abbildung 1.5.: Rastertiles und Vektortiles im Vergleich

Clients nicht angepasst werden und die Hardware-Anforderungen an die Clients sind niedriger. Damit das clientseitige Rendering von Vektortiles performant ist, müssen in kleinen Maßstäben die Geometrien vereinfacht werden, damit die Dateigröße der Tiles nicht zu groß und die Anzahl der Stützpunkte nicht zu hoch ist [68].

Für Vektortiles gibt es verschiedene Dateiformate. Neben Vektortiles, die GeoJSON oder TopoJSON enthalten, gibt es das populäre *Mapbox Vector Tile Format* (kurz: MVT), hinter dem die US-amerikanische Firma Mapbox steht [80]. Das Format nutzt Protocol Buffer (vgl. Seite 12) zur Serialisierung der Daten, jedoch mit einer anderen Struktur als die für OpenStreetMap-Rohdaten genutzte Kodierung. Die Koordinaten sind ganzzahlige Bildschirmkoordinaten anstelle von Weltkoordinaten (z. B. WGS84), jedes Tile hat sein eigenes Bildschirmkoordinatensystem [65].

Zur Qualitätssicherung eignen sich diese für die Kartographie gedachten Formate nicht oder nur bedingt. Sie alle haben gemeinsam, dass sie nicht mit den originären OSM-Objekten, sondern davon abgeleiteten Geometrien arbeiten. Die Vektortiles enthalten Punkte, Linestrings und Polygone. Linestrings und Polygone haben Listen von Koordinatenpaaren anstatt Nodes zu referenzieren, was das Rendering vereinfacht. Es kann daher nicht geprüft werden, ob zwei Ways denselben Node referenzieren, da die Linestrings in diesen Formaten für jeden Punkt dessen Koordinaten und nicht die ID des Punkts speichern. Damit zwei Ways in OSM jedoch als verbunden gelten, müssen sie denselben Node referenzieren.

Mapbox bietet zwar mit den *QA-Tiles* spezielle Vektortiles kostenlos für Qualitätssicherungszwecke an, diese sind jedoch für viele Fälle nicht ausreichend. Es handelt sich dabei um GeoJSON, dessen Geometrien nicht vereinfacht wurden und das die unveränderten OSM-Tags enthält. Sie enthalten keine Relationen und statt der Node-Referenzen enthalten die Ways die Koordinaten ihrer Stützpunkte. Die QA-Tiles werden als *mbtile*-Pakete zum Download angeboten. mbtile ist ein von Mapbox entworfenes Austauschformat, welches die einzelnen Tiles als *Binary Large Objects* (Blobs) in einer SQlite-Datenbank enthält. Die Blobs enthalten GeoJSON [46].

1.6. Gebräuchliche Software zur Verarbeitung von OSM-Daten

Alle in dieser Arbeit verwendete Software wurde auf Computern mit dem GNU/Linux-Betriebssystem verwendet. Auch alle in dieser Arbeit geschriebene Software ist für dieses Betriebssystem gedacht.

PostgreSQL ist ein freies Datenbankmanagementsystem, das sich im OpenStreetMap-Umfeld als Datenbankmanagementsystem durchgesetzt hat. Fast alle Software, die Daten in eine Datenbank schreibt oder aus einer Datenbank liest, nutzt PostgreSQL. Grund dafür ist nicht nur PostgreSQL selbst, sondern auch die Verfügbarkeit der mächtigen Erweiterung PostGIS, welche Datentypen und Funktionen für die Verarbeitung räumlicher Informationen bereitstellt. In dieser Arbeit kommt die PostgreSQL-Erweiterung *hstore* zum Einsatz, die den gleichnamigen Datentyp, einen Key-Value-Speicher bereitstellt.

PostgreSQL wird sowohl im Datenbank-Backend der OpenStreetMap-API als auch von Tileservern, Geocoding-Diensten wie *Nominatim* u. v. m. verwendet [53, 105, 113].

osm2pgsql ist ein ursprünglich in C geschriebenes und mittlerweile in C++11 portiertes Programm zum Import von OSM-Daten in eine PostgreSQL-Datenbank mit der PostGIS-Erweiterung. Das Datenbankschema besteht aus vier Tabellen:

- `planet_osm_point` enthält punktförmige Features
- `planet_osm_line` enthält linienförmige Features
- `planet_osm_roads` enthält eine Auswahl an Objekten aus `planet_osm_line`, um das Rendern von Tiles auf niedrigen Zoomstufen zu beschleunigen. Dabei handelt es sich um wichtige Straßen (Landstraßen und höher) sowie Eisenbahngleise. Die Auswahl der Objekte in dieser Tabelle ist hart kodiert und orientiert sich an einem mehrere Jahre alten Zustand des Kartenstils *OSM Mapnik*.
- `planet_osm_polygon` enthält flächenförmige Features.

Die Tabellen 1.1 und 1.2 listen die Spalten dieser Tabellen auf. Eine Konfigurationsdatei legt fest, welche Tags überhaupt in die Datenbank importiert werden. Hat ein Objekt ausschließlich Tags, die nicht im Stylefile aufgeführt sind, wird es nicht importiert. Ob ein geschlossener Way als Linie oder als Fläche importiert wird, hängt von den Tags ab. Dies lässt sich über das Stylefile beeinflussen.

Tabelle 1.1.: gemeinsame Spalten der Geometrie-Tabellen von osm2pgsql-Datenbanken

Feldname	Typ	Beschreibung
osm_id	bigint	ID des OSM-Objekts
access addr:housename addr:housenumber addr:interpolation admin_level ...	text text text text text text/real/int4/...	Wert des Tags des Objekts in OSM (Spaltenname = Key in OSM)
z_order	integer	aus `layer` und anderen Tags berechnete Renderingreihenfolge

Wenn die Datenbank mit Diffs aktuell gehalten werden soll, legt osm2pgsql drei zusätzliche *Slim Tables* an. Sie enthalten alle OSM-Objekte, auch wenn sie kein Tag haben, das in der Stildefinition enthalten ist. Wenn in einem Diff ein Way enthalten ist, bezieht osm2pgsql die Positionen der von ihm referenzierten Nodes aus diesen Tabellen. Ähnliches passiert bei Relationen und den von ihnen referenzierten OSM-Objekten. Die Tabellen 1.3 und 1.4 listen die Spalten dieser Tabellen auf [85].

- `planet_osm_nodes` enthält alle Nodes des importierten OSM-Datensatzes.
- `planet_osm_ways` enthält alle Ways des importierten OSM-Datensatzes.
- `planet_osm_relations` enthält alle Relationen des importieren OSM-Datensatzes.

Tabelle 1.2.: Felder, die bei osm2pgsql nur in einem Teil der Geometrie-Tabellen verfügbar sind

Feldname	Type	Tabellen	Beschreibung
way	geometry(Point)	planet_osm_point	Geometrie
way	geometry(LineString)	planet_osm_line, planet_osm_roads	Geometrie
way	geometry(Geometry)	planet_osm_polygon	Geometrie

Tabelle 1.3.: gemeinsame Spalten der Slim-Tabellen von osm2pgsql-Datenbanken

Feldname	Typ	Beschreibung
id	bigint	ID des OSM-Objekts
tags	text[]	Tags des OSM-Objekts, in folgender Reihenfolge: Key von Tag 1, Value von Tag 1, Key von Tag 2, Value von Tag 2, ...

Tabelle 1.4.: Felder, die bei osm2pgsql nur in einem Teil der Slim-Tabellen verfügbar sind

Feldname	Type	Tabellen	Beschreibung
lat	integer	planet_osm_nodes	geographische Breite in der Integer-Repräsentation
lon	integer	planet_osm_nodes	geographische Länge in der Integer-Repräsentation
nodes	bigint[]	planet_osm_ways	Nodes, die ein Way referenziert
parts	bigint	planet_osm_rels	IDs der referenzierten Objekte
members	text[]	planet_osm_rels	Typen, IDs und Rollen der referenzierten Objekte, z. B. `w293249683,inner,w293249772,outer`

Osmosis ist ein Java-Programm zum Lesen, Schreiben und Verändern (nach Tags filtern, räumliche Extrakte erzeugen, ...) von OSM-Daten. Es ist ein relativ vielseitiges Werkzeug, das auf keine Aufgabe spezialisiert ist. Zu seinen Funktionen gehört auch der Import von OSM-Daten in eine PostgreSQL-Datenbank. Dafür stehen drei Schemata zur Auswahl [90, 91]:

- *apidb*, das Schema der PostgreSQL-Datenbank der OSM-API. Dieses Schema nutzt nicht PostGIS, enthält alle Versionen aller Objekte und hält Metadaten und Tags in separaten Tabellen. Um die Geometrie eines Ways zu erhalten, müssen alle seine Nodes aus den Node-Tabellen abgefragt werden [32, 91]. Für Analysezwecke ist es nicht zu empfehlen.

- Das *simple*-Schema ist eine vereinfachte Version des Schemas *apidb*. Die Tabelle nodes hat eine PostGIS-Geometriespalte. Die Tags sind weiterhin in einer separaten Tabelle (eine Zeile pro Tag) gespeichert. Auch die Node-Referenzen der Ways und die Referenzen der Relationen sind in eigenen Tabellen gespeichert. Pro Referenz haben diese Tabellen einen Eintrag [23, 91].
- Das *pgsnapshot*-Schema ersetzt dieses Schema. Tags werden nicht mehr in einer separaten Tabelle, sondern als hstore-Feld gespeichert [24, 32, 91].

Alle drei Schemata ermöglichen keinen direkten Zugriff auf die Ways anhand eines räumlichen Index. Stattdessen muss zuerst nach den Nodes gesucht werden und anschließend nach den Ways/Relationen, die diese Nodes referenzieren.

Osmium wird ab Version 2 auch *libosmium* genannt und ist eine C++-Bibliothek zum Lesen und Schreiben von OSM-Daten nach dem Streaming-Prinzip. Der Entwickler Jochen Topf legt Wert auf Schnelligkeit und geringen Speicherverbrauch. Bei Osmium werden die OSM-Objekte nicht auf dem Stack, sondern in sogenannten *Buffern* abgelegt, wodurch Platz gespart und der Zugriff beschleunigt wird. Um mit Osmium eine OSM-Datei einzulesen, wird eine Klasse implementiert, welche von der Klasse osmium::handler::Handler abgeleitet ist und deren Callbackmethoden implementiert:

- osmium::handler::Handler::void node(const osmium::Node&) verarbeitet Nodes.
- osmium::handler::Handler::void way(const osmium::Way&) verarbeitet Ways.
- Weitere Methoden existieren für Relationen, Flächen, OSM-Objekte im Allgemeinen (Oberklasse von Node, Way, Relation und Area).

Osmium ruft die jeweils passende Callbackmethode des Handlers auf, sobald es ein neues Objekt aus der einzulesenden OSM-Datei gelesen hat.

Da ein Way nur Referenzen auf die Nodes, nicht aber deren Positionen selbst enthält, muss Osmium eine Map mit den IDs der Nodes als Schlüssel und der Position (Länge, Breite) als Wert im Speicher halten. Dafür bietet Osmium mehrere verschiedene Indexverfahren an. Einige sind für kleinere, andere für größere Extrakte bzw. den gesamten Planetdump geeignet. Die Auswahl kann zur Laufzeit erfolgen, Osmium stellt hierfür eine Factory bereit. [116, 117]

Im Rahmen dieser Arbeit wurden Lücken in der Dokumentation geschlossen und Jochen Topf als Pull-Request zur Übernahme angeboten. Für Details sei auf Anhang G verwiesen.

1.7. Qualitätssicherung bei OpenStreetMap

Änderungen eines Mappers werden sofort in die OSM-Datenbank übernommen, eine Sichtung gibt es bei OpenStreetMap nicht [31, 105]. Vorschläge zur Einführung eines Sichtungsverfahrens konnten sich in der OSM-Community nicht durchsetzen [92, 119]. OSM ist anders als die Wikipedia deutlich weniger von Vandalismus betroffen, da das Projekt weniger bekannt ist und zum Bearbeiten eine gültige E-Mailadresse erforderlich

ist. Außerdem stellt sich das Problem des Zusammenführens konkurrierender Änderungen, wenn ein Neuling eine Änderung hochgeladen hat, vor ihrer Sichtung aber ein anderer Mapper das Objekt erneut bearbeitet hat. Zudem bindet ein Sichtungsverfahren Personal und wirkt sich vermutlich negativ auf die Motivation der Mitwirkenden aus, wenn diese ihre Änderung nicht zeitnah auf der Karte auf openstreetmap.org sehen.

Die OSM-API führt abgesehen von der Validierung des hochgeladenen XML-Dokuments und einfachen Prüfungen wie der maximalen Anzahl an Nodes eines Ways und der Gültigkeit von Koordinaten (keine Längengrade $> \pm 180°$, keine Breitengrade $> \pm 90°$) fast keine Datenprüfungen beim Hochladen von Änderungen durch. Da der Großteil der Informationen über Tags modelliert wird (Multipolygone sind kein eigener Datentyp, sie sind nur Relationen mit speziellen Tags), kann die API solche Prüfungen gar nicht durchführen. Täte sie das, würde sie die weitere Entwicklung von OSM beschränken. Wer weiß, ob wir das, was heute heutzutage als korrekt gilt, in fünf Jahren immer noch als korrekt gilt, geschweige denn Stand der Technik ist? Linien des öffentlichen Verkehrs wurden z. B. 2010 als Routenrelationen modelliert, die alle Ways, welche von den Fahrten dieser Linie benutzt werden, referenzierten. Mittlerweile wird für jeden einzelnen Kurs einer Linie des öffentlichen Verkehrs eine separate Routenrelation angelegt, deren Mitglieder (Haltepositionen und Plattformen ausgenommen) eine durchgehende Strecke vom Start zum Ziel ergeben [33]. Was damals aktuell war, ist heute veraltet.

Dadurch, dass die Prüfung den Anwendungen Dritter überlassen ist, können mehr Entwickler Prüfungen schreiben. Nicht jeder hat ausreichende Kenntnisse in Ruby on Rails, um zur API-Weiterentwicklung beitragen zu können. In der aktuellen Situation kann jedoch jeder seine bevorzugte Programmiersprache wählen und an einem existierenden Validator seiner Wahl mitarbeiten oder seinen eigenen Validator schreiben.

Validierung ist außerdem keine reine Datenprüfung. Manche Qualitätssicherungswerkzeuge können auch als Importwerkzeuge verstanden werden. So gibt es mit `www.regio-osm.de` eine Website, die amtliche Straßen- und Hausnummernlisten mit OSM vergleicht [111]. Solche Prüfungen sind weit entfernt von dem, was die OSM-API tun sollte. Wie soll sie denn reagieren, wenn es eine Straße in den amtlichen Daten nicht gibt? Sie abzulehnen, wäre falsch, denn es könnte eine neue Straße in einem Neubaugebiet sein, das in den amtlichen Daten noch fehlt.

Fehler können also leicht in den OpenStreetMap-Datenbestand gelangen. Die Prüfung der Daten erfolgt daher durch die Community mithilfe von Werkzeugen, die von Dritten entwickelt und betrieben werden. Dabei handelt es sich meist um Webseiten.

Aktive Mapper lassen sich oft mit WhoDidIt [124] und/oder *Latest Changes on OpenStreetMap* [103] die Änderungssätze in ihrer Gegend anzeigen und schauen sich die Änderungssätze genauer an, die in irgendeiner Art und Weise verdächtig aussehen, z. B. weil ein großes Gebiet editiert wurde, kein Änderungssatzkommentar eingegeben wurde oder der Mapper noch unerfahren ist. Selbst wenn sie keine Ortskenntnis haben, können sie damit offensichtliche Fehler wie falsche Tags, invalide Geometrien usw. finden.

Fehler erfahrener Mapper und fehlende Daten bleiben damit aber unentdeckt. Nach solchen Fehlern suchen *Qualitätssicherungswerkzeuge*. Sie sind entweder auf ein Thema spezialisiert oder suchen nach einer Vielzahl an Fehlern. Die entdeckten Fehler sind nur *mögliche* Fehler. Abbildung 1.6 zeigt ein Beispiel. Es kann sein, dass die beiden

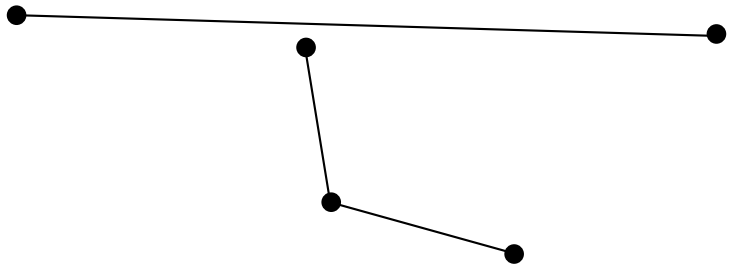


Abbildung 1.6.: Beispiel zweier unverbundener Straßen

Straßen in der Realität wirklich verbunden sind, aber in OSM einen Abstand von nur wenigen Metern haben. Es ist aber auch möglich, dass das kurze Verbindungsstück nur ein Fußweg ist. Eine sichere Entscheidung ist nur mit aktuellen Luftbildern, oft aber auch nur vor Ort möglich. Würden aber stupide/automatisiert diese Straßen verbunden werden, wäre es falsch und würde zu unerwarteten Fehlern bei Routenberechnungen führen.

Erfassungsfehler und ungebräuchliche Tags sind häufig auch ein Indiz für Beiträge, die nicht den Richtlinien der Community entsprechen. Importe von Fremddaten müssen vorher diskutiert und dokumentiert werden, was eine Prüfung der Lizenz einschließt. Geschieht das nicht, fallen sie auf, wenn sie ungebräuchliche oder falsch geschriebene Tags verwenden oder anderweitig Daten anders als gewöhnlich erfassen. Listing 1.5 zeigt einen Node in Kalifornien, welcher durch großgeschriebene Keys (normalerweise bestehen Keys in OSM ausschließlich aus Kleinbuchstaben) auffällt, zudem sind die Koordinaten noch einmal als Tag erfasst, obwohl ein Node bei OSM schon Koordinaten hat. Automatische Reparaturen beseitigen solche Probleme zwar, verwischen die Spuren und potentielle Urheberrechtsverletzungen fallen nicht auf.

Listing 1.5: Beispiel für einen qualitativ schlechten Import [1]

```
<node id="306079948" changeset="10328169" timestamp="2012-01-08T05:29:07Z" version=
    ↪"2" visible="true" user="Bryce C Nesbitt" uid="355242" lat="37.8838199" lon=
    ↪"-122.1422283">
  <tag k="ALAND" v="0"/>
  <tag k="AWATER" v="431202"/>
  <tag k="COUNTYFP" v="013"/>
  <tag k="ele" v="139"/>
  <tag k="landuse" v="reservoir"/>
  <tag k="latitude" v="+37.8812992"/>
  <tag k="longitude" v="-122.1420097"/>
  <tag k="MTFCC" v="H2040"/>
  <tag k="name" v="Lafayette Reservoir"/>
  <tag k="Source" v="Tiger2009"/>
  <tag k="STATEFP" v="06"/>
  <tag k="Tiger:HYDROID" v="110169760987"/>
  <tag k="Tiger:MTFCC" v="H2040"/>
</node>
```

Die Behebung der Fehler bleibt deshalb stets Menschen überlassen, automatische Fehlerkorrekturen finden nur selten und unter strengen Auflagen statt. Viele Qualitätssicherungswerkzeuge stellen die gefundenen Fehler als Overlay auf einer Basiskarte dar.

Während hochspezialisierte Qualitätssicherungswerkzeuge wie die Maxheight Map [72] meist von der Overpass-API die Daten herunterladen, die für ihren Zweck relevant sind (z. B. Straßen unter Brücken, aber keine Landnutzungsflächen, Gewässer usw.) und diese dann clientseitig im Browser ausgewählte Daten (nämlich nur das Straßennetz, nicht auch noch Wasserwege, Gebäude und Landnutzung) prozessieren, arbeiten Qualitätssicherungswerkzeuge, die eine ganze Reihe an Fehlern suchen, serverseitig und prozessieren meist den gesamten Planet bzw. Teile davon. Als Folge dessen werden die Webseiten dieser Tools meist nur alle ein bis vier Tage aktualisiert. Beispiele hierfür sind *Keep Right*, *Osmose* und der *OpenStreetMap Inspector* (kurz: OSMI) [82].

1.7.1. Keep Right

Keep Right wird von OSM-Benutzer mueschel (von 2009 bis 2013 von Harald Klein) [30, 59] betrieben und entwickelt. Es besteht aus 37 *Checks*, welche verschiedene SQL-Abfragen auf eine PostgreSQL-Datenbank durchführen. Der Datenbankimport erfolgt mit Osmosis, als Datenbankschema wird eine modifzierte Version des Simple-Schemas von Osmosis eingesetzt. Um der Größe des OpenStreetMap-Planets Herr zu werden, wird dieser in 85 rechteckige Zonen aufgeteilt, die etwa gleich viele Daten enthalten. Jede Zone hat ihre eigenen Datenbanktabellen für Nodes, Ways usw. Mit dieser Partitionierung ist es noch möglich, die Fehler mittels SQL-Abfragen zu finden. Ansonsten wird im Backend PHP eingesetzt.

Die Visualisierung der gefundenen Fehler erfolgt auf `http://keepright.at` mit der JavaScript-Bibliothek OpenLayers 2. Benutzer können dort einen Fehler als gelöst oder *False Positive* kennzeichnen. Diese Markierungen werden in einer MySQL-Datenbank verwaltet [58, 61]. Abbildung 1.7 zeigt einen Screenshot der Anwendung.

1.7.2. Osmose

Das Qualitätssicherungswerkzeug *Osmose* (OpenStreetMap Oversight Search Engine) wird von OpenStreetMap France entwickelt und betrieben und ist modular aufgebaut. Frontend und Backend können auf getrennten Servern betrieben werden. Dritte können eigene Backends betreiben, von denen der Frontend-Server die Fehlerberichte bezieht und visualisiert. Die OSM-Daten werden im Backend mit Osmosis in eine GIS-Datenbank importiert. Die Analysen werden mit Python durchgeführt. Ein Teil der Fehlerprüfungen wird mit SQL-Abfragen, der andere mit Python-Skripten durchgeführt. Die Python-Skripte verwenden SAX (Simple API for XML)[5] zum Einlesen der OSM-XML-Daten.

Auch Osmose partitioniert die Welt. Die Partitionierung wird jedoch anhand administrativer Grenzen durchgeführt – meist an Staatsgrenzen, Länder mit einer großen

[5] SAX ist ein Verfahren zum Lesen und Verarbeiten von XML-Dateien nach dem Streaming-Prinzip, sodass nicht der gesamte XML-Baum in den Speicher geladen werden muss.

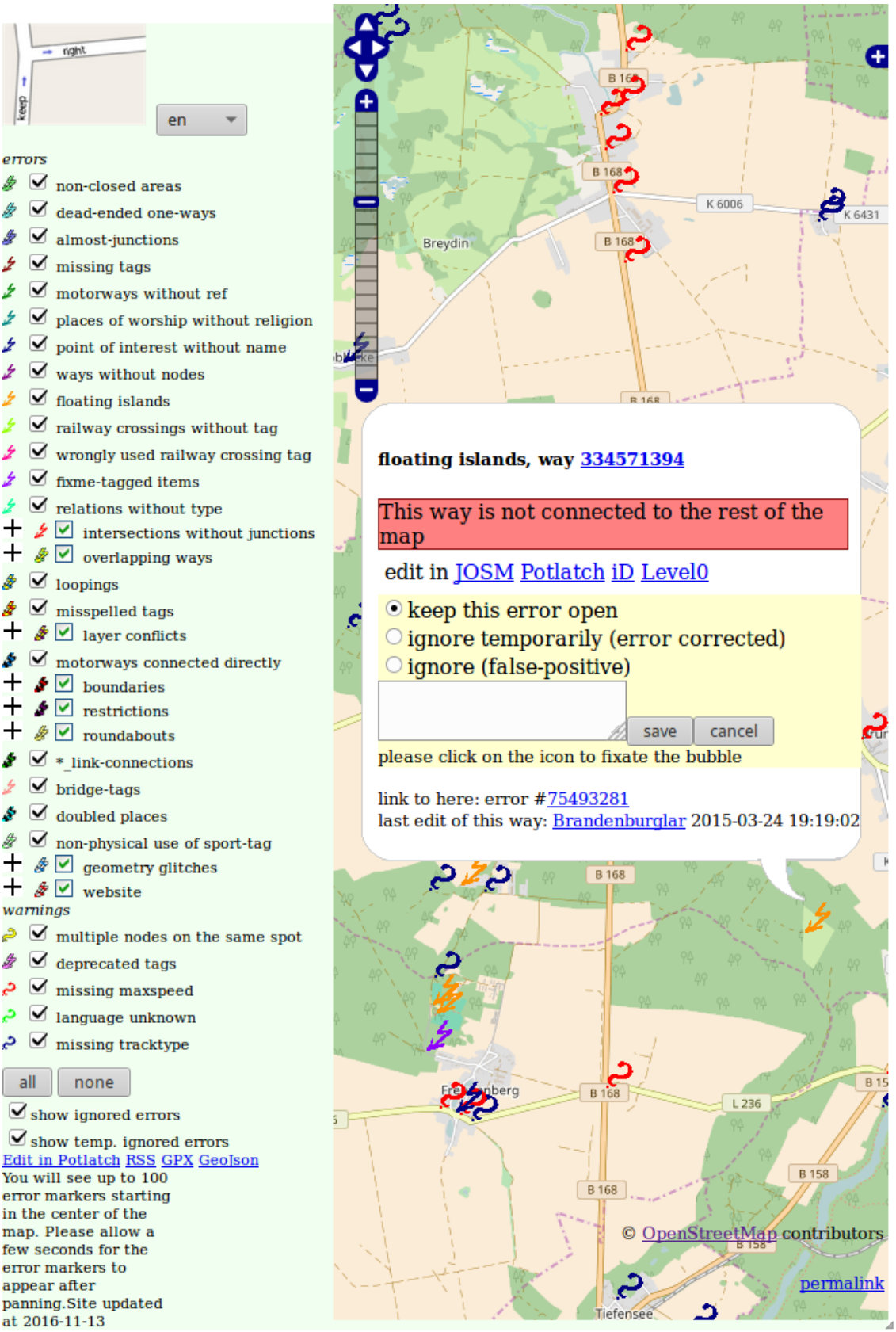


Abbildung 1.7.: Benutzerobfläche von Keep Right [60]

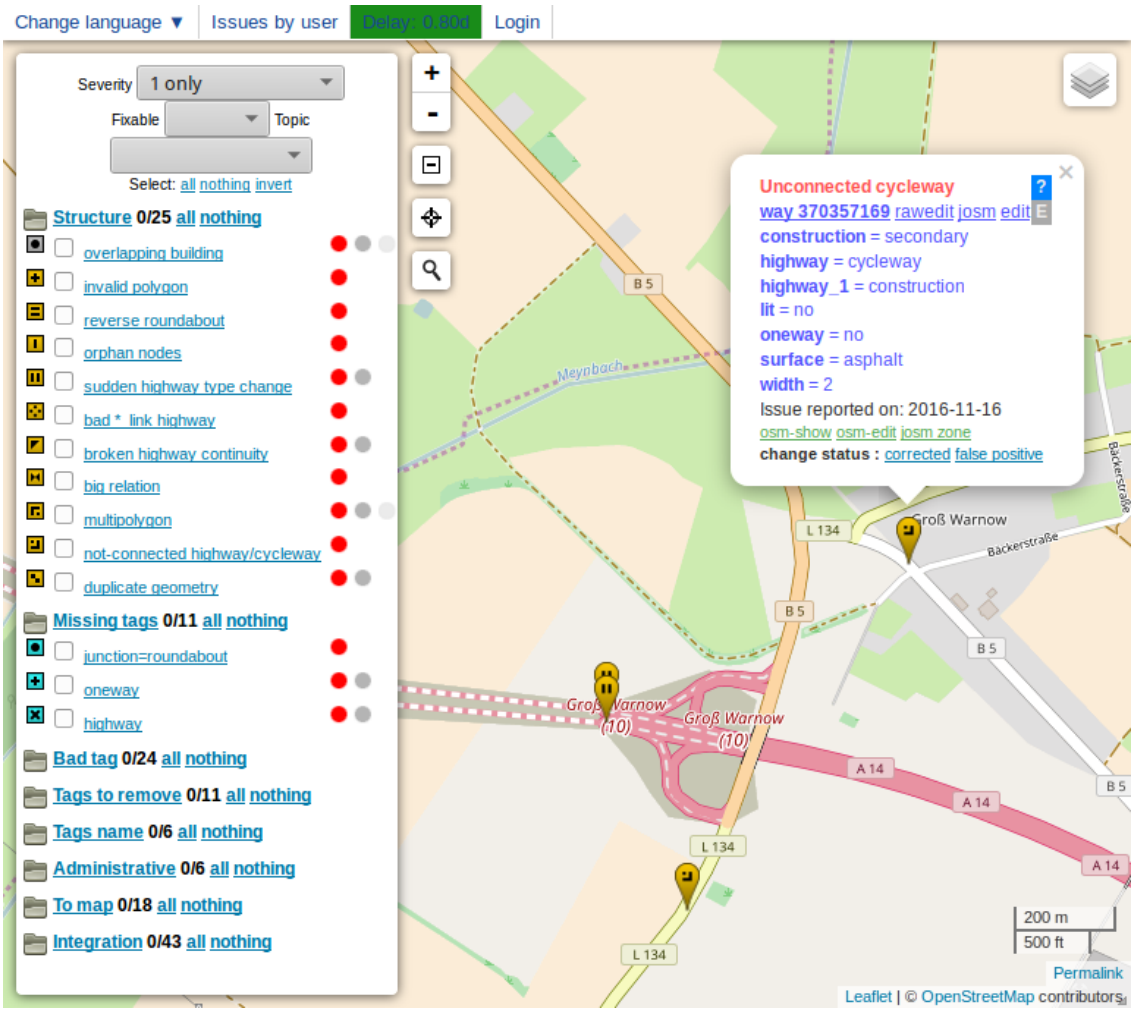


Abbildung 1.8.: Benutzerobfläche von Osmose [88]

Datenmenge werden noch einmal in Bundesländer/Provinzen aufgeteilt. Für manche dieser Gebiete gibt es zusätzliche Prüfroutinen, die etwa an nationale Besonderheiten angepasst sind oder Vergleiche mit Open-Data-Datensätzen durchführen. Statt den Planet herunterzuladen und zu partitionieren, werden von Anbietern, die regionale Planet-Extrakte anbieten, die Extrakte bezogen und in die Datenbank importiert bzw. mittels den Python-Skripten direkt nach Fehlern durchsucht.

Das Frontend (siehe Abbildung 1.8) visualisiert Fehler mittels Markern auf einer Leaflet-Karte. Jedem Fehler wird einer von drei Schweregraden zugeordnet (innerhalb derselben Fehlerart kann es Fehler verschiedener Grade geben). Die Benutzer können nach Schwere und nach Fehlertyp filtern. Bei Osmose ist es ebenso möglich, Fehler über das Frontend als gelöst bzw. False Positive zu melden [86, 87, 89].

1.7.3. OpenStreetMap Inspector

Der OSMI wird von der Geofabrik GmbH entwickelt und betrieben und ist ähnlich modular aufgebaut wie Osmose. Anders als bei Osmose und Keep Right gibt es fast

keine räumliche Partitionierung. Stattdessen wird versucht, mit effizienter Software und Parallelisierung der Datenmenge Herr zu werden. Fehler sind einer Kategorie zugeordnet, die Kategorien heißen im OSMI-Frontend *Views*. Jeder View ist ein WMS-Dienst, der mit Mapserver bereitgestellt wird und die GetFeature-Methode unterstützt. Über letztere können Benutzer zusätzliche Informationen zum Fehler abfragen, z. B. die IDs der beteiligten OSM-Objekte. Datenquelle der WMS-Dienste sind Shapefiles und SQlite-Datenbanken, welche von verschiedenen Prozessen berechnet werden. Während anfangs Shapefiles mit SQL-Abfragen an eine PostgreSQL-Datenbank erstellt wurden, werden mittlerweile aus Performancegründen überwiegend C++-Programme eingesetzt, die die Osmium-Bibliothek benutzen. Jedes Programm rechnet einen View aus. Die einzelnen Programme lesen den OSM-Planetdump ein, der mit den täglichen Diffs aktuell gehalten wird, und erstellen ein Shapefile, welches auf den Frontend-Server kopiert wird (siehe Abbildung 1.9). Da die Programme parallel auf verschiedenen Rechnern laufen, ist eine Turn-Around-Zeit von etwa einem Tag gerade noch erreichbar.

Derzeit enthält der OSMI folgende Views:

- Geometry
- Tagging
- Places
- Highways
- Areas[6] ersetzt den View Multipolygons
- Coastlines

[6] bereitgestellt aus externer Quelle

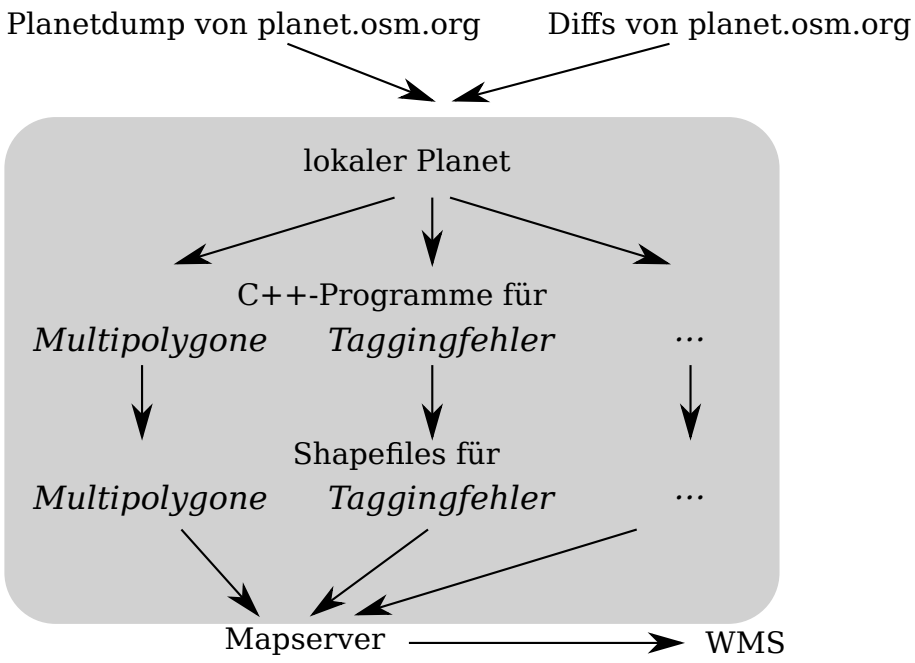


Abbildung 1.9.: Ablaufschema OpenStreetMap Inspector

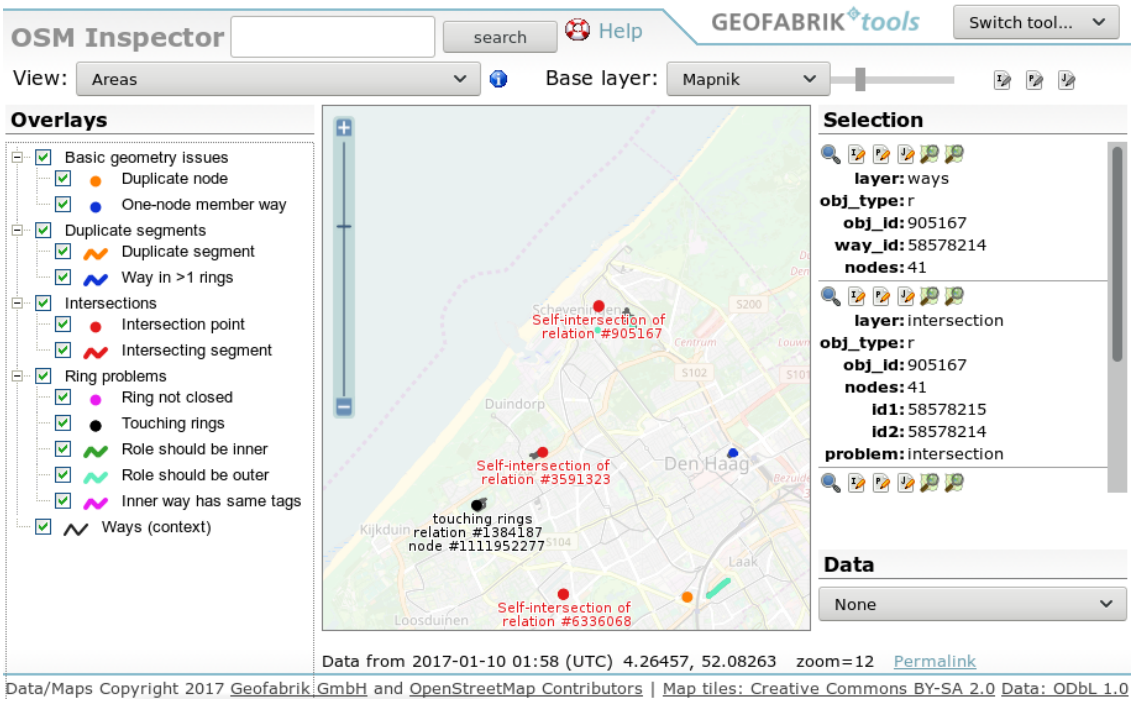


Abbildung 1.10.: Benutzerobfläche des OpenStreetMap Inspectors [83]

- Routing[7]
- Addresses[8]
- Water
- Public Transport (mehrere Views)

Abbildung 1.10 zeigt die Benutzeroberfläche des OpenStreetMap Inspectors.

1.7.4. MapRoulette

MapRoulette (http://maproulette.org) wird von Martijn van Exel und Serge Wroclawski entwickelt [66] und versucht das Korrigieren von Fehlern zum Spiel zu machen [123]. Anders als Osmose und Keep Right sucht MapRoulette nicht gezielt nach Fehlern. Die Fehler werden nicht als Marker auf einer Karte dargestellt, wie es bei Keep Right und Osmose geschieht. Stattdessen kann jeder *Challenges* (Sammlungen von Fehlern einer bestimmten Kategorie) entwickeln und auf MapRoulette zum Korrigieren bereitstellen. Der Nutzer wählt eine Challenge aus, beispielsweise spitze Winkel in Straßen, und bekommt einen zufällig ausgewählten *Task* (ein einzelner Fehler) präsentiert. Diesen kann er dann bearbeiten, übergehen oder als False Positive kennzeichnen. Die Entwickler von MapRoulette schreiben selbst, dass MapRoulette für kleine Aufgaben gedacht ist, die binnen Sekunden gelöst werden können und keine Ortskenntnis benötigen [37]. Das Austauschformat für Challenges ist ein JSON-Format [67].

[7]bereitgestellt aus externer Quelle
[8]Entwicklung extern, Prozessierung durch Geofabrik

1.7.5. To-Fix

To-Fix ist eine „Mikro-Tasking-Plattform“ [62], die starke Ähnlichkeiten zu MapRoulette aufweist [38] und von der Firma Mapbox entwickelt und betrieben wird. Das Austauschformat für die Fehler einer Challenge ist CSV. Die Hauptnutzer sind Mapbox-Angestellte, die damit für das Routing relevante Fehler wie etwa unverbundene Straßen, aber auch andere Fehler korrigieren. Von den 363 799 Änderungssätzen in OSM, welche die Zeichenkette „#to-fix“ (Groß-/Kleinschreibung wurde ignoriert) enthalten, stammen 97,90 Prozent von 31 Benutzerkonten, die von Mapbox als Mitarbeiter benannt werden[9].

1.8. Probleme bestehender Qualitätssicherungswerkzeuge

Keep Right, Osmose und der OSMI haben das Problem gemeinsam, dass sie aufgrund der großen Datenmenge, es nur mit Mühe schaffen, einen akzeptablen Aktualisierungszyklus zu erreichen. Da diese Werkzeuge nur einmal täglich ihre Daten aktualisieren können (das Beziehen der stündlichen Diffs nützt nichts, da die Prozessierung fast einen Tag dauert), kann es im ungünstigsten Fall bis zu zwei Tage dauern, bis eine Änderung eines Benutzers in OpenStreetMap von diesen Werkzeugen als fehlerhaft erkannt wird: An Tag 1 lädt kurz nach Mitternacht der Benutzer seine fehlerhafte Änderung zu OpenStreetMap hoch. Um Mitternacht wurde das tägliche Diff erstellt, das diese Dienste beziehen. Die Änderung des Benutzers wird also erst in dem Diff enthalten sein, das am Ende von Tag 1 erstellt wird. Bis die geänderten Daten dann prozessiert sind, dauert es aber noch fast einen Tag. Der Fehler erscheint also erst am Ende von Tag 2 auf den Websites dieser Dienste.

Viele Datennutzer, die ihren OSM-Datenbestand regelmäßig aktualisieren, beziehen jedoch auch die täglichen Diffs. Bis der Fehler aus ihrem Datenbestand verschwindet, vergeht also noch einmal ein weiterer Tag.

Bis der Fehler wieder aus den Qualitätssicherungswerkzeugen verschwindet, dauert es mindestens einen weiteren Tag nach Upload der Fehlerkorrektur durch einen Mapper. Erst muss die Fehlerkorrektur als Bestandteil des nächsten täglichen Diffs das Qualitätssicherungswerkzeug erreichen (dauert maximal 24 Stunden) und das Qualitätssicherungswerkzeug muss das Diff verarbeiten (noch einmal maximal 24 Stunden). Dadurch vergeht neben der Wartezeit bis zum nächsten täglichen Diff noch ein weiterer Tag. Als Workaround für dieses *Lange-Transaktion-Problem* ermöglichen Keep Right und Osmose es dem Nutzer, Fehler als korrigiert zu markieren. Abbildung 1.11 stellt die Zusammenhänge grafisch dar.

Die räumliche Partitionierung, wie sie bei Osmose und Keep Right verwendet wird, dient allein dazu, den Speicherbedarf beim Import in die SQL-Datenbank zu reduzieren, die zur Qualitätssicherung verwendeten SQL-Abfragen zu beschleunigen und – was aber höchstens bei Osmose erfolgt – die Verarbeitung zu parallelisieren. Dennoch erreichen Keep Right, Osmose und der OpenStreetMap Inspector keine akzeptablen Aktualisierungszyklen, da sie jeden Tag unnötig viele, sich nicht ändernde Daten verarbeiten.

[9] Changeset-Planet vom 22. August 2016 [6],Messfehler 0,03 %

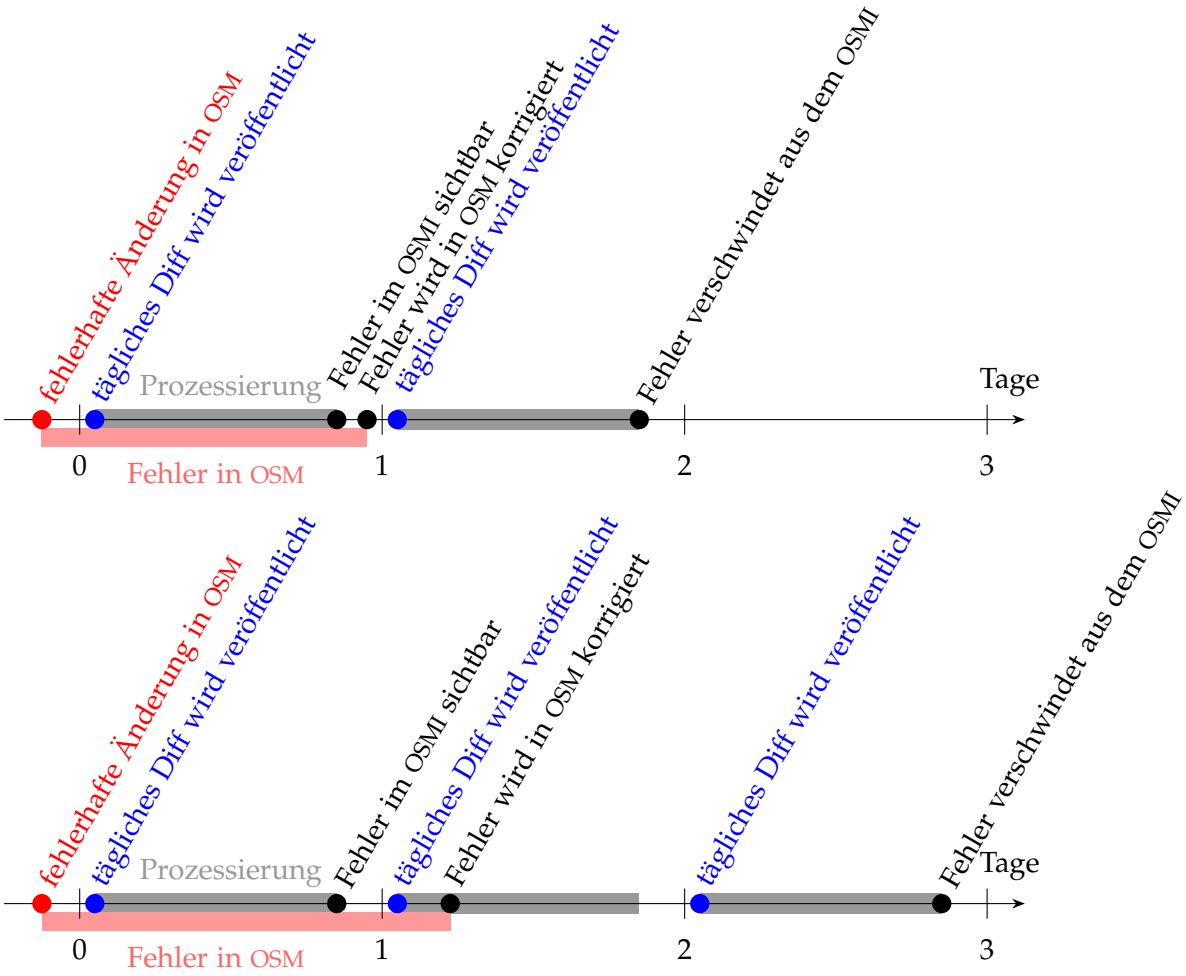


Abbildung 1.11.: Zeiträume der Sichtbarkeit von Fehlern beim OpenStreetMap Inspector

Im Rahmen dieser Masterarbeit soll ein Ansatz entworfen und die zentralen Teile davon implementiert werden, sodass nur noch die Teile des Datenbestandes prozessiert werden müssen, die sich geändert haben.

Während bei Osmose und Keep Right die Partitionierung dazu dient, mit schwächerer Hardware einen großen Planetdump zu verarbeiten, soll die Partitionierung, die in dieser Arbeit entworfen wird, unveränderte von veränderten Daten trennen. Nach der initialen Prozessierung aller Daten im Rahmen des Setups sollen nur noch die Gebiete prozessiert werden, die sich geändert haben. Die Partitionierung ist deutlich kleinteiliger als bei Osmose und Keep Right. Es soll das Prinzip eines Tileservers, der nur die Gebiete neu rendert, deren Daten sich geändert haben, auf ein Qualitätssicherungswerkzeug übertragen.

Da die Partitionierung deutlich kleinteiliger ist, sinken die Anforderungen an die Rechenknoten, die die Datenvalidierung durchführen. Es muss nicht mehr die ganze Welt bzw. ein ganzes Bundesland im Speicher gehalten werden, sondern nur noch ein Tile, das wenige Kilometer oder hundert Meter Kantenlänge hat. Die Partionierung ermöglicht es darüber hinaus, die Verarbeitung zu parallelisieren – sei es in Form mehrerer paralleler Prozesse oder in Form mehrerer Rechenknoten.

2. Architektur

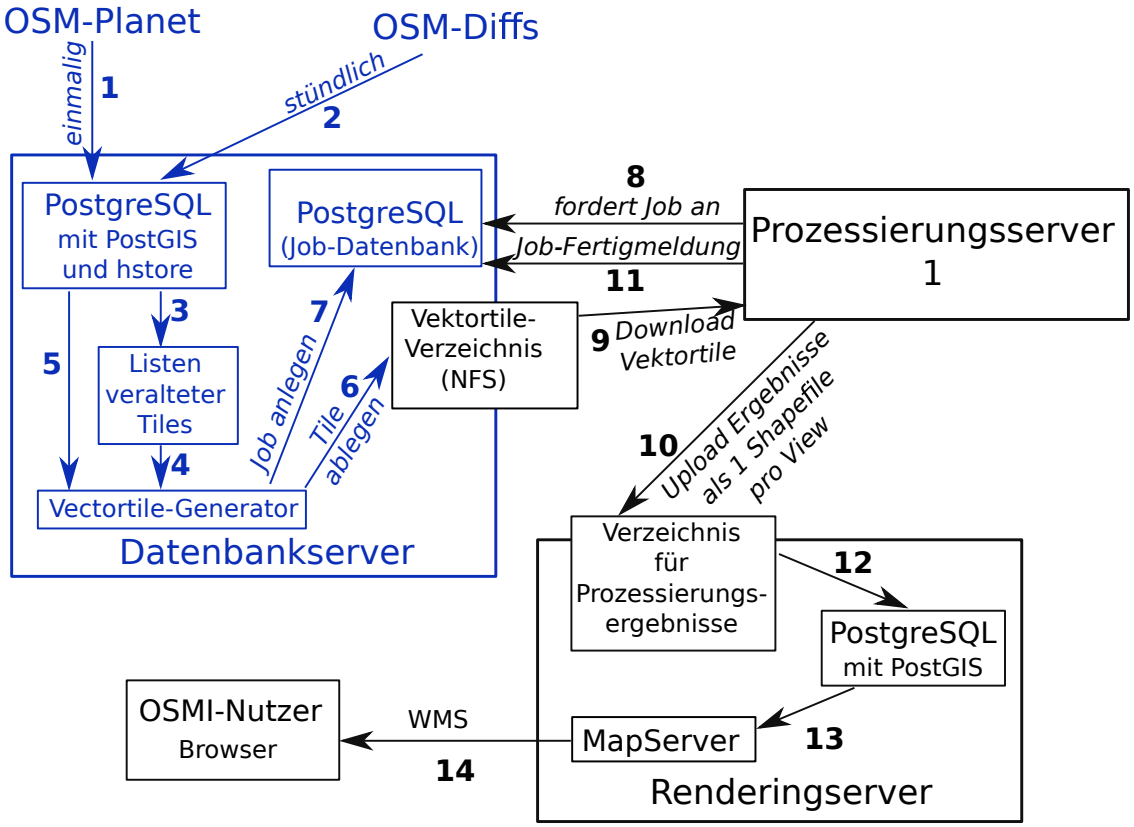


Abbildung 2.1.: Entworfene Architektur für ein Qualitätssicherungswerkzeug mit stündlichen Updates. Die schwarzen Teile wurden nicht im Rahmen dieser Arbeit implementiert bzw. haben keiner Anpassungen bedurft.

Dieses Kapitel gibt einen Überblick über die verwendeten Komponenten. Für Details sei auf die folgenden Kapitel verwiesen, in denen die Komponenten im Detail beschrieben und Gründe für die Designentscheidungen dargelegt werden. Abbildung 2.1 zeigt die entworfene Architektur, die aus mindestens drei Servern besteht.

Der *Datenbankserver* pflegt eine Datenbank mit allen OSM-Daten, erzeugt die Vektortiles und teilt den Prozessierungsservern (anfangs nur einer – dieser Teil der Kette ist skalierbar) Jobs zu. Ein Job entspricht einem neu zu berechnenden Vektortile.

Die *Prozessierungsserver* holen sich ein oder mehrere Jobs und berechnen alle Views für dieses Vektortile. Für jeden View entsteht eine Datei in einem Transportformat, z. B. ein Shapefile, welches auf den Renderingserver übertragen und dort in die Renderingdatenbank (PostgreSQL mit PostGIS) importiert wird.

Der *Renderingserver* existiert schon im bisherigen OSMI ohne Vektortiles. Auf ihm bietet derzeit ein MapServer einen WMS (mit aktiviertem *GetFeature*-Request) an, jedoch sind Shapefiles (eins pro View) derzeit die Datenquelle. In der neuen Version ist eine PostgreSQL-Datenbank (mit PostGIS) die Datenquelle, da laufend Änderungen in die Renderingdatenbank eingefügt werden. Der Renderingserver sucht dazu kontinuierlich auf den Prozessierungsservern nach neuen Shapefiles und importiert diese in seine Renderingdatenbank.

Der Fokus der Arbeit lag auf dem Datenbankserver. Die Komponenten auf dem Prozessierungs- und Renderingserver wurden nicht implementiert bzw. angepasst.

2.1. Datenbankserver

2.1.1. Komponenten und Funktionsweise

Auf dem Datenbankserver läuft eine PostgreSQL-Datenbank mit der PostGIS- und hstore-Erweiterung. In die Datenbank wird initial ein Planetdump importiert (1) und anschließend mit Diffs aktuell gehalten (2). Bei jedem Import eines Diffs wird eine Liste an Tiles erstellt, deren Daten sich durch Import des Diffs geändert haben – das sogenannte *Tile-Expiry-Logfiles* (3). Für den Import und die Pflege der Datenbank war ursprünglich geplant, *osm2pgsql* einzusetzen. Der Umgang von osm2pgsql mit Relationen war jedoch nicht zufriedenstellend und eine Anpassung von osm2pgsql an die Bedürfnisse wurde verworfen. Stattdessen wurde mit *Cerepso* ein Ersatz entwickelt, der die Datenbank mit dem gewünschten Schema importiert und möglichst viele Aufgaben an Bibliotheken abgibt. Zu den Gründen für diese Entscheidung und die Details der Architektur von Cerepso sei auf das Kapitel 3 verwiesen.

Nach dem Import eines Diffs auf dem Datenbankserver wird auf diesem *Cerepso2vt* gestartet. Es liest die Liste an veralteten Tiles ein (4) und erzeugt die nötigen, neu zu berechnenden Vektortiles (5). Die Vektortiles werden auf der Festplatte abgelegt (6).

Der Datenbankserver ist außerdem für das Zuteilen von Prozessierungsjobs für die Prozessierungsserver zuständig. Diese Aufgabe erledigt ein Dienst namens *OSMI-Dispatcher*. Cerepso2vt greift gemeinsam mit dem OSMI-Dispatcher auf eine PostgreSQL-Datenbank zu, in der die Jobs verwaltet werden. Wenn ein Vektortile neu erzeugt wird, trägt der Cerepso2vt es in die Jobdatenbank ein (7). Der OSMI-Dispatcher nimmt Anfragen der Prozessierungsserver für neue Jobs entgegen (8) und teilt ihnen einen Job aus der Jobdatenbank zu. In der Datenbank markiert er diesen Job dann als vergeben, damit jeder Job nur einmal ausgeführt wird. Nach Abschluss der Prozessierung meldet sich der Prozessierungsserver wieder beim OSMI-Dispatcher (11). Der Job wird aus der Datenbank gelöscht und das Vektortile gelöscht.

2.1.2. Warum werden die OSM-Daten in einer Datenbank gespeichert?

Die Entscheidung, ein Datenbankmanagementsystem (für Geoinformationssysteme und im OpenStreetMap-Umfeld ist PostgreSQL gebräuchlich) für die Datenhaltung der OSM-

Daten zu nutzen, fiel relativ früh. Prinzipiell gibt es zwei Möglichkeiten Vektortiles zu erzeugen.

Die OSM-Daten werden in eine Datenbank (i. d. R. PostgreSQL) importiert. Wenn ein Vektortile erzeugt wird, werden alle Objekte, die im Gebiet des Tiles liegen, von der Datenbank abgefragt. Diese Methode benötigt nur beim initialen Import relativ viel Arbeitsspeicher. Im Gegenzug ist das Erzeugen eines Vektortiles wegen der SQL-Abfragen jedoch relativ langsam, selbst wenn Indexe eingesetzt werden. Desweiteren wird mehr Platz auf der Festplatte/SSD benötigt, da die Datenbank die Daten nicht so platzsparend ablegt, wie es ein kompaktes, für den Transport optimiertes Binärformat tut. Mit Metadaten werden etwa 800, ohne etwa 400 Gigabyte benötigt.

Bei der Verwendung einer Datenbank könnten durch entsprechend formulierte SQL-Abfragen auch thematische Tiles erzeugt werden. Wenn die Laufzeit der Abfragen sich dadurch erheblich verlängert, wird nur Prozessierungsaufwand von den einzelnen Rechenknoten (wovon beliebig viele eingesetzt werden können) zur nicht so einfach skalierbaren zentralen Datenbank verlagert. Das ist nicht im Sinne dieser Arbeit.

Ein unwichtiger Vorteil ist, dass die Datenbank auch noch für viele andere Zwecke als die Erzeugung von Vektortiles verwendet werden kann. Wenn ein passendes Datenbankschema verwendet wird, kann die Datenbank auch als Backend für einen Renderingserver verwendet werden, wenn dieser nur schwach ausgelastet ist.

Anstelle alle Daten in einer großen Datenbank zu speichern, können sie auch von Anfang an als Vektortiles gespeichert werden. Beim Einlesen des Planets wird ein OSM-Objekt in die Vektortiles geschrieben, in denen es liegt. Bei Nodes ist das einfach, denn sie haben Koordinaten. Um dasselbe für Ways zu tun, müssen die Koordinaten der Nodes, die der Way referenziert, im Arbeitsspeicher gehalten werden. Bei der Verarbeitung von Relationen müssen zudem auch noch die von diesen Relationen referenzierten Ways im Arbeitsspeicher gehalten werden. Für all das werden derzeit bei Osmium-basierten Programmen etwa 33 Gigabyte Arbeitsspeicher benötigt.

Wenn ein Diff importiert wird, werden dessen Änderungen auf die einzelnen Vektortiles angewendet. Um zu ermitteln, aus welchem Vektortile ein gelöschtes Objekt entfernt werden soll, muss eine Index existieren, der für jedes OSM-Objekt speichert, in welchen Vektortiles es liegt.

Wird ein Node verschoben, der von einem Way referenziert wird, müssen alle Tiles, in denen dieser Way vorher lag bzw. danach liegen wird, aktualisiert werden. Auch alle Relationen, die Ways referenzieren, welche diesen Node referenzieren, müssen aktualisiert werden. Die Listen der Referenzen müssen daher effizient invers durchsuchbar sein (einer ID eines Nodes müssen alle Ways, die ihn referenzieren, zugeordnet sein). Neben einem Index, der einem Objekt ein oder mehrere Tiles zuordnet, muss also noch ein zweiter Index mit der Zuordnung Objekt $\rightarrow$ referenzierende Objekte existieren. Wenn beide Indexe im Arbeitsspeicher gehalten werden, können damit schneller Vektortiles erzeugt werden, als es mit einer Datenbank möglich ist.

Vorteil dieser Lösung ist die schnellere Erzeugung von Vektortiles, Nachteil ist der höhere initiale Entwicklungsaufwand. Bislang existieren zwar schon Programme, wie z. B. *tilemaker* [39], die Vektortiles aus einem Planetdump erzeugen, ohne Datenbank zu verwenden. Diffs können diese aber nicht auf die Vektortiles anwenden.

2.2. Prozessierungsserver

Dieser Teil der Kette ist skalierbar. Die Architektur ist dafür ausgelegt, dass mehrere Prozessierungsserver parallel betrieben werden können. Selbst wenn jedoch nur ein Prozessierungsserver betrieben wird, kann dieser mehrere Jobs parallel ausführen, denn die Tiles sind recht klein und der Speicherbedarf daher gering.

Auf dem Prozessierungsserver läuft ein Python-Skript namens OSMI-Worker, welches in einer Endlosschleife beim OSMI-Dispatcher neue Jobs anfragt (8) und diese ausführt. Wenn es vom OSMI-Dispatcher einen neuen Job erhält, lädt es sich das dazugehörige Vektortile vom Datenbankserver (9). Das Verzeichnis der Vektortiles auf dem Datenbankserver ist per NFS auf allen Prozessierungsservern im Nur-Lese-Modus eingehängt. Nacheinander werden die einzelnen Views berechnet. Dazu werden die bestehenden Osmium-basierenden Programme ausgeführt. Diese Programme geben jeweils ein Shapefile aus. Wenn alle Views berechnet sind, werden die Shapefiles in einem Verzeichnis auf der Festplatte des Renderingservers abgelegt (10), welches per NFS in das Dateisystem des Prozessierungsservers eingehängt ist. Zuletzt wird der Job als fertig gemeldet (11).

2.3. Renderingserver

Der Renderingserver existiert schon im bisherigen OSMI und muss nur an das Vektortile-Backend angepasst werden. Während beim OpenStreetMap Inspector bislang ein Shapefile pro View die Quelle der WMS-Dienste ist, die von MapServer auf diesem Server bereitgestellt werden, werden künftig mehrere PostgreSQL-Datenbanken die Quelle sein (13). Durch die Verwendung eines Datenbankmanagementsystems sind konkurrierende Zugriffe möglich – eine Voraussetzung für kontinuierliche Updates.

Der Renderingserver sucht kontinuierlich auf den Prozessierungsservern nach neuen Shapefiles, die zu importieren sind, und importiert diese in seine Datenbank (12). Er bietet einen WMS an (13), der im OSMI-Webinterface eingebunden ist.

Alternativ kann statt einer PostGIS-Datenbank als Datenquelle auch eine Reihe an Shapefiles als Datenquelle verwendet werden. Mit dem Programm *orgtindex*, das Teil der GDAL-Bibliothek ist, kann ein *Tileindex* erstellt werden, welcher dem MapServer als Datenquelle dient [43, 54]. Jedes einzelne Shapefile kann dann ein prozessiertes Tile repräsentieren und der Import neu prozessierter Tiles würde durch einfaches Überschreiben des alten Shapefiles erfolgen.

3. Cerepso – ein alternativer Importer für PostgreSQL-Datenbanken

Das im Rahmen der Arbeit erstellte PostgreSQL-Importprogramm *Cerepso* (Langname: *Cerepso replaces osm2pgsql*) ist eine der Komponenten, die auf dem Datenbankserver zum Einsatz kommen. Es importiert einen OSM-Planet in die Datenbank und wendet die von OSM bereitgestellten Diffs auf die Datenbank an. Als Datenbankmanagementsystem wird PostgreSQL eingesetzt, weil sich dieses im OpenStreetMap-Umfeld als stabil und zuverlässig erwiesen hat und es mit PostGIS eine gute Erweiterung für räumliche Daten gibt.

3.1. Gründe für die Entwicklung von Cerepso

Anfangs war geplant, für den Import der Datenbank und das Anwenden von Diffs auf *osm2pgsql* zurückzugreifen, welches als Importwerkzeug für die Datenbanken von Renderingservern und zur allgemeinen Datenanalyse verwendet wird. osm2pgsql hat jedoch einige Nachteile, die dazu geführt haben, mit Cerepso einen Ersatz zu entwickeln.

Das Datenbankschema von osm2pgsql ist an den Bedürfnissen des Renderings ausgerichtet. Die Tabellen `planet_osm_point`, `planet_osm_line`, `planet_osm_roads` und `planet_osm_polygon` enthalten neben den Tags und der ID des Objekts noch eine Spalte für die Geometrie. Um jedoch Vektortiles erstellen zu können, welche OSM-Rohdaten enthalten, ist bei einem Way nicht seine Geometrie (Koordinatenpaare der einzelnen Vertexe) relevant, sondern die Nodes, die dieser Way referenziert. Diese Information legt osm2pgsql in den sogenannten *Slim-Tabellen* ab. Allein mit den Slim-Tabellen ist jedoch keine effiziente Erstellung von Vektortiles möglich, da es keinen räumlichen Index für die Slim-Tabellen gibt. Um also die Vektortiles erstellen zu können, sind Joins und somit zu viele (aufwändige) Datenbankzugriffe erforderlich.

Selbst wenn osm2pgsql trotzdem verwendet würde, wäre die Behandlung von Relationen durch osm2pgsql das nächste zu lösende Problem. osm2pgsql betrachtet Relationen als Sets. Die Spalte `members` der Tabelle `planet_osm_rels` enthält zuerst alle Nodes, die von der Relation referenziert werden, dann alle Ways und zuletzt alle referenzierten Relationen. Multipolygon- und Grenzrelationen werden je nach Einstellungen als Polygone oder Multipolygone in die Tabelle `planet_osm_polygon` importiert. Bei Routenrelationen werden alle Mitglieder, welche Ways sind, zu einem MultiLineString zusammengefasst und in die Tabelle `planet_osm_line` geschrieben. Aus Relationen erzeugte Polygone, Multipolygone und MultiLineStrings haben eine negative ID (d. h. der Relations-ID wird ein Minuszeichen vorangestellt). In den Vektortiles sollen jedoch alle Relationen enthalten sein, die mindestens ein Way- oder Node-Mitglied in dem Gebiet haben, welches das

Tile abdeckt. Das ist mit osm2pgsql ohne Modifikationen nicht möglich. Die Tabelle planet_osm_rels hat keinen räumlichen Index und die Feature-Tabellen, welche räumliche Indexe haben, enthalten nicht alle Relationen.

Für die Slim-Tabelle planet_osm_nodes gibt es seit einiger Zeit eine Alternative. Statt die Nodes in einer PostgreSQL-Tabelle zu verwalten, können sie auch in einer Datei auf der Festplatte zwischengespeichert werden, dem sogenannten Flatnodes-File [28]. Da die Slim-Tabellen nicht als öffentliches Interface gedacht sind und schon eine der drei durch eine dateibasierte Lösung ersetzt wurde, kann davon ausgegangen werden, dass die anderen beiden auch eines Tages durch Dateien ersetzt werden. Paul Norman, einer der Maintainer von osm2pgsql, deutet das in einer Diskussionen im Bugtracker von osm2pgsql und auf Mailinglisten so an [77, 79].

Anfangs wurde versucht, die Verarbeitung von Relationen zu verbessern und bei Routenrelationen die Geometrie der Way- und Node-Mitglieder als eine GeometryCollection in die Geometriespalte der Tabelle planet_osm_polygon[1] zu schreiben. Zwar wird zum Einlesen von OpenStreetMap-Daten mittlerweile die Osmium-Bibliothek verwendet, aber sämtliche weiteren OSM-nahen Klassen werden selbst definiert, d. h. die Geometrien konstruiert osm2pgsql selbst aus den OSM-Daten. Für Tags, Mitgliederlisten von Relationen usw. gibt es daher ebenfalls eigene Klassen. Das bläht den Code recht stark auf. An vielen Stellen ist offensichtlich, dass osm2pgsql in C geschrieben und später auf C++11 portiert wurde. Das Alter des Programms (zehn Jahre), die vielen Beitragenden, die nicht vorhandene Klassendokumentation und das nicht auf Erweiterbarkeit ausgelegte Design tun das Übrige dazu.

Das Datenbankschema ist für die Nutzung als Datenbankbackend eines Tileserver entworfen und eignet sich ansonsten nur für wenige andere Anwendungen gut (z. B. GIS-Analysen [78]). Für die hier beabsichtigte Erzeugung von Vektortiles ist das Datenbankschema nur eingeschränkt geeignet, hätte zu unsauberem Code geführt und wäre nur eine Behelfslösung, weshalb von einer Erweiterung von osm2pgsql abgesehen und stattdessen ein eigener Importer für OSM-Daten in PostgreSQL entwickelt wurde. Das Datenbankschema, einige Designentscheidungen und die Optimierungen bezüglich des Datenbankzugriffs sind jedoch von osm2pgsql übernommen worden (siehe dazu Kapitel 3.3).

Da Cerepso eine Zukunft haben soll, wurde an den entscheidenden Stellen auf eine Kompatibilität zu osm2pgsql geachtet, damit Cerepso zu einem vollwertigen Ersatz ausgebaut werden kann. Unterschiede in den Datenbankschemata sollen sich durch den Einsatz von *Views* ausgleichen lassen. Views sind ein Wrapper-artiges Konstrukt der PostgreSQL-Datenbank und können von SQL-Abfragen wie eine Tabelle behandelt werden. Sie führen jedoch kein spezielles Caching ein („are not materialized“) und bestehen aus einer SQL-Abfrage, die bei jedem Aufruf erneut ausgeführt wird. So können Daten aus mehreren Tabellen zusammengeführt oder bestimmte Abfragen in Client-Anwendungen vereinfacht werden [97].

[1] Der Typ der Geometriespalte ist Geometry.

3.2. Datenbankschema und Indexe

Das Datenbankschema ist stark von osm2pgsql inspiriert. Zwar wurde Cerepso nur für den Import der Datenbank des OSMI entwickelt, es wurde jedoch darauf geachtet, dass Cerepso in Zukunft zu einem vollwertigen, aber abwärtskompatiblen Ersatz für osm2pgsql ausgebaut werden kann. Die Differenzen zwischen den Datenbankschemata von Cerepso und osm2pgsql können mit Views überbrückt werden. Die Kompatibilität der Datenbankschemata ist wichtig, denn die meisten OSM-Datenbanken werden zum Rendering verwendet und die verwendeten Kartenstile sind eng an das verwendete Datenbankschema – meist das von osm2pgsql – gebunden.

Das Datenbankschema besteht aus vier Tabellen und Cerepso unterscheidet, wie auch schon osm2pgsql, nicht nach den Tags an den Objekten, d. h. es gibt keine getrennten Tabellen für die verschiedenen Objektklassen (z. B. Landnutzung vs. Gebäude).

- `nodes` für Nodes, die ein oder mehrere Tags haben
- `untagged_nodes` für Nodes ohne Tags
- `ways` für Ways
- `relations` für Relationen

Tabelle 3.1 zeigt die Felder, die in allen vier Tabellen verfügbar sind. Die übrigen Spalten sind nicht in allen Tabellen verfügbar und in Tabelle 3.2 aufgelistet.

Nodes ohne Tags werden in einer separaten Tabelle gespeichert, da für die meisten Anwendungen diese nur zum Anwenden der Diffs benötigt werden. Wer Karten rendern möchte, benötigt nur Nodes mit Tags sowie Ways und Relationen, da diese ein Objekt in der Realität repräsentieren. Auf kleineren Tabellen sind Abfragen und – das ist beim Import wichtig – die Erstellung der Indexe schneller.

Für den effizienten Zugriff werden Indexe auf die Geometriespalte(n) und die Spalte `osm_id` angelegt. Der Geometrie-Index ist für die Erstellung der Tiles erforderlich, da hierbei der Index die `ST_Intersects`-Abfrage beschleunigt. Der Index auf die ID-Spalte ist erforderlich, da beim Einfügen eines Ways alle Nodes des Ways abgefragt werden

Tabelle 3.1.: gemeinsame Felder aller Spalten von Datenbanken, die mit Cerepso importiert wurden

Feldname	Typ	Beschreibung
osm_id	bigint	ID des OSM-Objekts
osm_user	text	Benutzername des letzten Bearbeiters
osm_uid	text	ID des letzten Bearbeiters
osm_version	int	Version des Objekts
osm_lastmodified	char(20)	Zeitstempel der letzten Bearbeitung in OSM
osm_changeset	bigint	ID des Änderungssatzes der letzten Bearbeitung

Tabelle 3.2.: Felder, die nur in einem Teil der Tabellen von mit Cerepso importierten Datenbanken verfügbar sind

Feldname	Type	Tabellen	Beschreibung
geom	Point(2)	nodes, untagged_nodes	Geometrie
geom	LineString(2)	ways	Geometrie
geom_points	MultiPoint(2)	relations	Geometrie aller Mitglieder, die Nodes sind
geom_lines	MultiLineString(2)	relations	Geometrie aller Mitglieder, die Ways sind
way_nodes	bigint[]	ways	IDs der Nodes des Ways
member_ids	bigint[]	relations	IDs der Mitglieder der Relation
member_types	char[]	relations	Typen der Mitglieder der Relation (`n` für Nodes, `w` für Ways, `r` für Relationen)

müssen, um die Geometrie erstellen zu können. Diese Abfrage erfolgt anhand der OSM-ID.

3.3. Optimierung des Datenbankzugriffs

Der naive Ansatz zum Einfügen von Daten in eine Datenbank ist die Verwendung des `INSERT`-Befehls. Um jedoch eine akzeptable Performance beim initialen Import zu haben, wird, wie es auch andere Importwerkzeuge tun und von der PostgreSQL-Dokumentation empfohlen wird [98], der `COPY`-Befehl verwendet.

Ein kleiner Versuch bestätigt den Performanceunterschied. Ein Import von 39709 Nodes per `INSERT` dauert abhängig von dem verwendeten Verfahren zur Erzeugung des Well-Known-Binary-Strings für die Geometrie-Spalte 5,3 bis 6,4 Sekunden. Wird `COPY` verwendet, dauert es nur noch 0,25 Sekunden.[2]

Weil zum Import der `COPY`-Befehl verwendet werden muss, muss die C-Bibliothek *libpq* zum Zugriff auf die PostgreSQL-Datenbank verwendet werden. In der C++-Bibliothek *libpqxx* sind die dafür notwendigen Klassen als *deprecated* gekennzeichnet [100], außerdem ist die Entwicklung auch mehr oder weniger eingeschlafen [29], was gegen eine Verwendung von libpqxx spricht.

Wie auch osm2pgsql verwendet der Cerepso `COPY FROM STDIN ...`, d.h. nach Senden dieses Befehls muss pro einzufügendem Objekt eine Zeile als CSV an die Datenbank gesendet werden. Als Trennzeichen wird ein Tabulator verwendet.

Zeitweise wurde versucht, gepuffert zu senden (eine Zeile wird mit einem Newline beendet). osm2pgsql verwendet als Puffer einen String und prüft, nachdem eine neue Zeile in den Pufferstring eingefügt wurde, ob der Puffer größer als 1024 Byte ist. Ist der Schwellwert überschritten, wird der Inhalt des Puffers an die Datenbank gesendet und

[2] Intel Core i5-750, 16 GB RAM, Festplatte Seagate ST1000VX000 mit 7200 rpm SATA-600

der Puffer geleert. Woher die Größe von 1024 Byte stammt ist leider nicht nachvollziehbar, weder über Kommentare im Quellcode noch über Commit-Nachrichten in der Git-Versionsgeschichte. Im Rahmen der Entwicklung von Cerepso wurde gepuffertes und ungepuffertes Schreiben verglichen, die Puffer waren 1024, 2048 und 4096 Byte groß. Das Schreiben mit 1024 und 2048 Byte großen Puffern ist ähnlich schnell wie ungepuffertes Schreiben bzw. die Geschwindigkeitsunterschiede liegen innerhalb der Messgenauigkeit. Der Speicherverbrauch verändert sich ebenfalls nur innerhalb der Schwankungsbreite. Erst bei 4096 Byte großen Puffern ist eine leichte Verlangsamung messbar.

Eine weitere Optimierungsmöglichkeit ist, die Protokollierung der Tabellen auszuschalten (`CREATE UNLOGGED TABLE ...` anstelle von `CREATE TABLE ...`). Das bringt zwar das Risiko mit sich, dass die Datenbank in einen inkonsistenten Zustand gerät, wenn während des Imports das System abstürzt (z. B. Stromausfall), in diesem Fall muss aber der Import ohnehin erneut durchgeführt werden. Die Verwendung einer Tabelle ohne Protokollierung beschleunigt das Programm um knapp die Hälfte; der Import von Baden-Württemberg dauert mit Protokollierung etwa 7:30 Minuten, ohne nur noch 4:10 Minuten.[3] Die Protokollierung kann ab PostgreSQL 9.5 nach dem initialen Import mittels `ALTER TABLE name SET LOGGED` eingeschaltet werden.

Durch die Verwendung von *Prepared Statements* kann die Laufzeit von Cerepso weiter verkürzt werden. Prepared Statements sind vordefinierte SQL-Befehle, die zuerst bei der Datenbank registriert werden müssen und anschließend anhand ihres Namens aufgerufen werden. Sie werden vom SQL-Interpreter nur einmal geparst. Daher rührt der Geschwindigkeitsvorteil.

3.4. Funktionsweise

Cerepso nutzt intensiv die Osmium-Bibliothek. Dadurch müssen keine eigenen Klassen für OSM-Objekte wie Nodes, Ways, Tags usw. definiert werden; es werden die von Osmium bereitgestellten Typen verwendet. Auch das Einlesen und das Konstruieren der Geometrien wird Osmium überlassen.

Abbildung 3.1 zeigt ein vereinfachtes UML-Klassendiagramm von Cerepso. Klassen mit Bezug zur Tile Expiry wurden weggelassen, sofern sie nicht direkt mit einer anderen Klasse assoziiert sind. Für eine vollständige Darstellung sei auf Anhang D verwiesen.

Die Architektur ist erweiterbar gehalten. Alle Datenbankzugriffe (d. h. Verwendung von Funktionen der Bibliothek libpq) sind in der Klasse `PostgresTable` zusammengefasst; Teile davon sind sogar in die Klasse, `postgres_drivers::Table` im der Bibliothek *Cerepso-Postgres-Backend* ausgelagert, da sie auch von Cerepso2vt (siehe Kapitel 5) genutzt werden. `PostgresTable` erbt von `postgres_drivers::Table`. Es handelt sich dabei um den Einsatz des *Wrapper*-Patterns [42].

Die Klasse `postgres_drivers::Table` besitzt eine Referenz auf eine Instanz der Klasse `postgres_drivers::Columns`. `postgres_drivers::Columns` enthält die Spaltendefinitionen der Tabelle (Spaltennamen und Typen) und stellt einen Iterator [42] über die einzelnen Spalten bereit. Die Spaltenfunktionen sind in `postgres_drivers::Columns` derzeit noch im

[3] Intel Core i5-750, 16 GB RAM, Festplatte Seagate ST1000VX000 mit 7200 rpm SATA-600

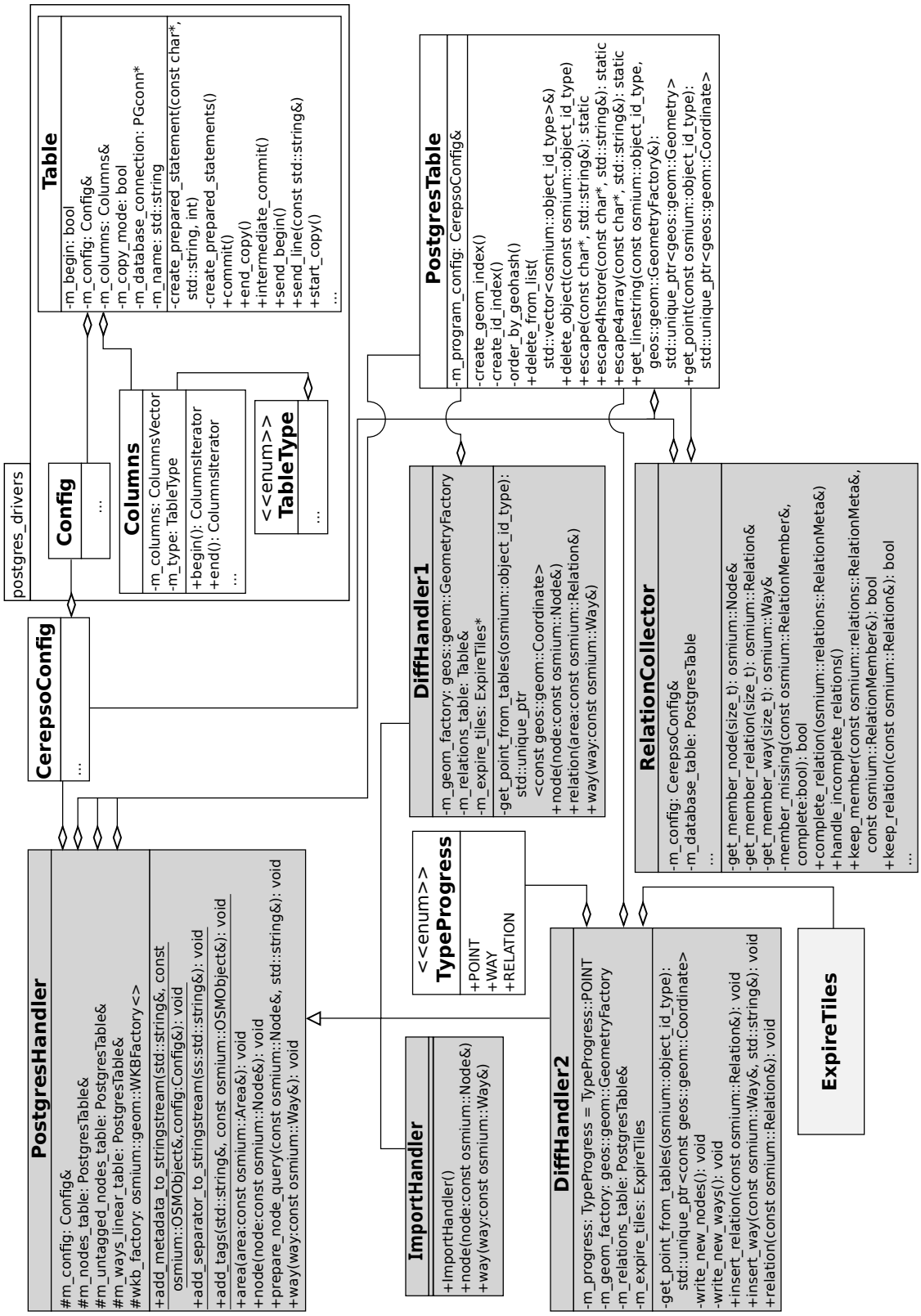


Abbildung 3.1.: UML-Klassendiagramm von Cerepso. Klassen mit Bezug zur Tile Expiry sind nicht enthalten. Für ein vollständiges Klassendiagramm sei auf Anhang D verwiesen.

Konstruktor hart kodiert. Wenn Cerepso in Zukunft flexibler gestaltet sein soll, müssen die Spaltendefinitionen in eine Konfigurationsdatei ausgelagert und stattdessen aus dieser Datei eingelesen werden.

Die vom Benutzer über Kommandozeilenparameter gesetzten Einstellungen werden von der Klasse `CerepsoConfig` verwaltet, die von den meisten Klassen referenziert wird. Nach dem Einlesen der Kommandozeilenparameter wird für jede der Tabellen je ein Objekt der Klasse `CerepsoColumns` instanziiert. Dem Konstruktor wird der Geometrietyp der Tabelle übergeben. Hart kodierte Regeln im Konstruktor dieser Klasse legen dann die Spalten der Tabelle fest.

Der Instanziierung von `postgres_drivers::Columns` folgt die Instanziierung von `PostgresTable`. Der Konstruktor der Klasse `PostgresTable` baut eine Verbindung zur Datenbank auf, registriert die Prepared Statements und löscht, falls die Tabelle schon existiert und ein Neuimport erfolgen soll, die existierende Tabelle.

Durch Scoping wird der Destruktor des Location-Handlers vor dem Destruktor der `PostgresTable`-Instanzen aufgerufen. Das sorgt dafür, dass der vom Location-Handler beanspruchte Speicher freigegeben wird, bevor im Destruktor der `PostgresTable`-Instanzen die Indexe erstellt und ggf. die Tabellen nach `ST_GeoHash` sortiert werden. Abbildung 3.2 zeigt den Speicherverbrauch Cerepsos und aller PostgreSQL-Daemons zusammen. Deshalb geht nach etwa vier Stunden der Speicherverbrauch stark zurück. Zu diesem Zeitpunkt wurde der vom Location-Handler reservierte Speicher freigegeben. Die Daten wurden mit dem Bash-Skript `pgimporter/util/get-ram-usage.sh` sekündlich ermittelt.

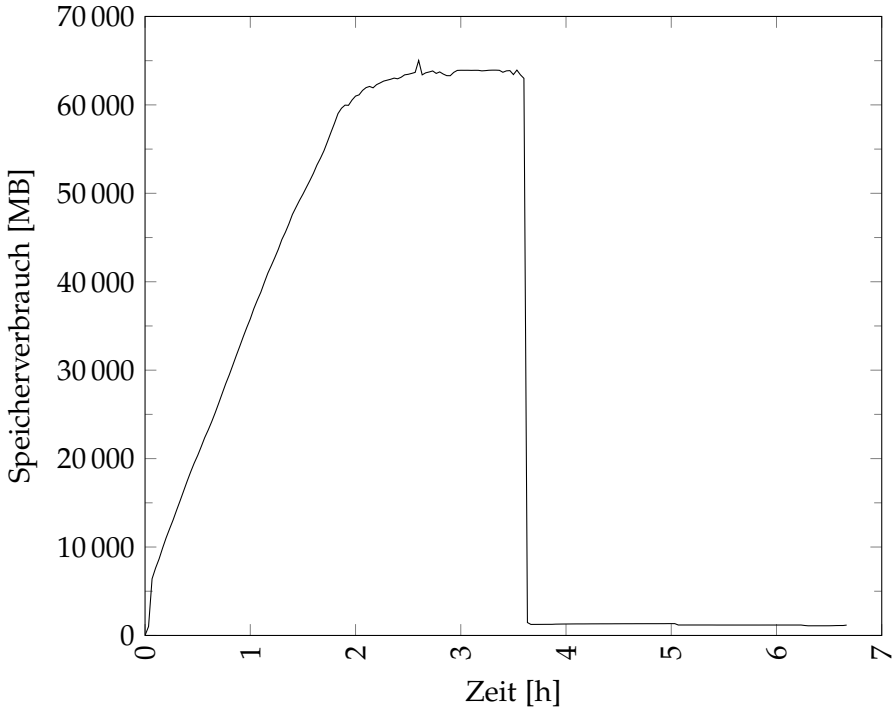


Abbildung 3.2.: Speicherverbrauch von Cerepso und aller PostgreSQL-Daemons beim Import eines Planetdumps (Details zur Hardware und den Einstellungen der PostgreSQL-Datenbank siehe Anhang H)

Der weitere Ablauf unterscheidet sich zwischen den beiden Modi – dem *initialen Import* und dem *Diff-Import*. Der Standardmodus ist der initiale Import, der Diff-Import wird mit `--append` aktiviert.

3.4.1. Initialer Import

Bei einem *initialen Import* (bei osm2pgsql wird das *Planet-Import* genannt) werden die nötigen Datenbanktabellen angelegt und die Daten initial importiert. Zuerst initialisiert das Hauptprogramm den von Osmium bereitgestellten `NodeLocationsForWays`-Handler, der einen zur Laufzeit ausgewählten Index-Typ verwendet. Der `NodeLocationsForWays`-Handler ermöglicht beim Einlesen den Zugriff auf die Positionen der Nodes anhand ihrer ID, welche von einem Way referenziert werden. Die Auswahl des Indexverfahrens erfolgt zur Laufzeit, der Benutzer legt über den Kommandozeilenparameter `--location-handler=`*`index_type`* fest, welches Indexverfahren verwendet werden soll.

In diesem Modus wird die zu importierende OSM-Datei zweimal eingelesen. Beim ersten Einlesen werden nur die Relationen gelesen und vom `RelationCollector` gespeichert. Dieser Durchlauf wird in der Osmium-Terminologie *Pass 1* genannt. Im darauf folgenden *Pass 2* werden Nodes und Ways aus der OSM-Datei gelesen und den Callback-Funktionen des `ImportHandler` und des `RelationCollector` übergeben. Deren im Folgenden beschriebene Callback-Methoden werden von Osmium aufgerufen und fügen per `COPY` die Objekte in die Datenbank ein.

`ImportHandler::node (const osmium::Node&)` prüft erst, ob der Node eine gültige Position hat. Aus der Frühzeit von OpenStreetMap gibt es noch Nodes im Planet, die ungültige Koordinaten haben [116]. Anschließend wird der Query-String mit der Methode `prepare_node_query(const osmium::Node&, std::string&)` der Klasse `PostgresHandler` erstellt. Die Geometrie wird als WKB-HEX-String, der ASCII-Repräsentation des Well-Known-Binary-Formats, eingefügt; das erspart den Aufruf von PostGIS-Funktionen wie `ST_GeometryFromText`. Der WKB-HEX-String wird von der Methode `postgres_drivers::Table::send_line(std::string&)` an die Datenbank gesendet (die Verbindung ist zu diesem Zeitpunkt im `COPY`-Modus).

`ImportHandler::way (const osmium::Way&)` führt ähnliche Prüfungen auch zu Beginn durch. Wenn der Way weniger als zwei Nodes referenziert oder einer der von ihm referenzierten Nodes invalide Koordinaten hat, wird er nicht in die Datenbank geschrieben. Das ist keine optimale Lösung. Würde der Node jedoch ignoriert und der Way dennoch in die Datenbank importiert werden, würde die Datenbank eine falsche Geometrie des Ways enthalten.

Der weitere Ablauf ist ähnlich zu dem in der Methode `PostgresHandler::prepare_node_query(osmium::Node&, std::string&)`. Weil anders als bei der `node`-Methode der Code zur Erzeugung des Query-Strings beim Diff-Import nicht auch noch einmal aufgerufen wird, ist er nicht in die Oberklasse `PostgresHandler` ausgelagert worden. Sollte beim Erzeugen WKB-HEX-Strings eine Exception geworfen werden, wird ein leerer LineString in die Geometrie-Spalte der Tabelle `ways` geschrieben.

RelationCollector::complete_relation(const osmium::RelationMeta&) stellt wie das way-Callback des ImportHandler einen String zusammen, der an die Datenbank gesendet wird. Da die Geometrie einer Relation als GeometryCollection in die Datenbank geschrieben wird, wird für jedes Mitglied der Relation, das ein Way oder Node ist, ein GEOS-Objekt mit der GEOSFactory von Osmium erzeugt. Über die GEOS-Bibliothek wird dann eine GeometryCollection gebildet und als WKB kodiert.

3.4.2. Diff-Import

Beim *Diff-Import* werden die in einer OSC-Datei enthaltenen Änderungen im OSM-Datenbestand auf die Datenbank angewendet. Auch der Diff-Import erfolgt in zwei Durchläufen. In Pass 1 werden alle Objekte, die in dem Diff enthalten sind und deren Version größer als 1 ist, gelöscht. In Pass 2 werden alle Objekte, die als neu oder geändert gekennzeichnet sind, in die Datenbank eingefügt. Das Einfügen erfolgt aus Performancegründen mittels `COPY` (vgl. Kapitel 3.3).

Anders als beim initialen Import sind beim Diff-Import während der Anwendung des Diffs auch noch lesende Zugriffe auf die Datenbank erforderlich. Wenn in OpenStreetMap die Tags eines Ways geändert wurden, seine Nodes aber nicht, ist nur der Way im Diff enthalten, nicht aber die von ihm referenzierte Nodes. Wenn der Way in die Datenbank eingefügt werden soll, müssen die Koordinaten jedes Nodes nachgeschlagen werden, um die WKB-Geometrie für die Geometrie-Spalte der Tabelle `ways` erzeugen zu können. Da die Ways erst eingefügt werden, nachdem alle Nodes eingefügt worden sind, hat die Datenbankverbindung für die Node-Tabellen den `COPY`-Modus wieder verlassen und die Änderungen wurden committet. Deshalb können im zweiten Durchlauf die Nodes der Ways per `SELECT` abgefragt werden. Für den Import geänderter Relationen gilt dasselbe. Die Geometrie-Spalte der Tabelle `relations` enthält die WKB-Geometrie aller Ways und Nodes, die Mitglied der Relation sind.

Dieses Vorgehen ist möglich, da die OSC-Dateien nach Typen geordnet sind – erst alle Änderungen an Nodes, dann die Änderungen der Ways und zuletzt die Relationen.

Relationen, die von dieser Relation referenziert werden, sind nicht enthalten. Da derzeit kein Bedarf besteht, Relationen, die Relationen referenzieren, zu unterstützen, wurde auf die Implementierung einer Rekursion verzichtet.

Sämtliche Änderungen, die im Rahmen eines Diff-Imports erfolgen, werden in einem Transaktionsblock zusammengefasst, d. h. vor dem Initialisieren der Diff-Handler wird für sämtliche Instanzen der Klasse PostgresTable die Methode PostgresTable::send_begin() aufgerufen, welches ein `BEGIN` an die Datenbank sendet. Im Destruktor der Klasse PostgresTable wird der Transaktionsblock mit `COMMIT` abgeschlossen. Ein Dirty Read durch eine andere Applikation, die gleichzeitig auf die Datenbank lesend zugreift (z. B. Cerepso2vt), wird dadurch vermieden.

3.5. Tile Expiry

3.5.1. Prinzip der Tile Expiry und derzeit vorhandene Software

Um feststellen zu können, welche Tiles nach dem Import eines Diffs neu gerendert werden müssen, werden auf vielen Tileservern Tile Expiry Logs erstellt. Nach diesem Prinzip arbeitet auch der in dieser Arbeit entwickelte Datenbankserver. Wenn ein Node, ein Way oder eine Relation geändert wird, werden die Tiles als veraltet markiert, auf denen sich das Objekt in der alten Version befand und in der neuen Version befindet, denn diese Tiles müssen neu prozessiert werden. Diese Logs sind Textdateien, die pro neu zu berechnendem Tile eine Zeile in der Form `Z/X/Y` enthalten, z. B. `12/564/125` für das Tile $x = 564, y = 125, z = 12$.

Es gibt bislang zwei verschiedene Wege, wie Tile Expiry Logs erstellt werden können [114]. osm2pgsql erstellt sie im Rahmen des Diff-Imports, jedoch ist die Art und Weise, wie es geschieht, ineffizient. Die Koordinaten werden unnötig oft zwischen verschiedenen Koordinatensystemen transformiert und anstatt frühzeitiger im Programmablauf direkt die Koordinaten als Gleitkommazahl (bzw. Array aus Gleitkommazahlen) abzugreifen, wird das Well Known Binary geparst, welches von osm2pgsql gleich danach in die PostgreSQL-Datenbank geschrieben wird. Der Code selbst ist auch nicht besonders gut leserlich und könnte auf absehbare Zeit im Rahmen des noch immer andauernden Refactorings entfallen oder durch eine bessere Alternative ersetzt werden.

Die andere Methode ist ein Ruby-Skript von Matt Amos. Dieses Skript hat jedoch Schwächen. Es berücksichtigt keine Änderungen an Relationen. Änderungen an Ways, die über ein Meta-Tile hinwegreichen, ohne einen Node in diesem Meta-Tile zu haben, lösen keine Neuberechnung des Meta-Tiles aus [14].

3.5.2. Implementierung in Cerepso und Herausforderungen

Cerepso verwendet eine verbesserte Version der Erstellung der Tile Expiry Logs. Das Interface der Klasse `ExpireTiles` ist von osm2pgsql inspiriert. `ExpireTiles` ist eine abstrakte Klasse, zu der drei Implementierungen existieren. Die Klasse `ExpireTilesFactory` stellt eine Factory bereit, mit der die Implementierung zur Laufzeit gewählt werden kann (vgl. Abbildung 3.3); auf der Kommandozeile mit dem Parameter `--expire-tiles=`*`METHODE`*.

- `ExpireTilesDummy` implementiert das Null Object Pattern. Ihre Methoden haben leere Implementierungen [50]. Sie wird verwendet, wenn beim Import von Diffs keine Tile-Expiry-Logs erstellt werden sollen. (`--expire-tiles=`*`dummy`*)
- `ExpireTilesClassic` ist die von osm2pgsql zu Cerepso portierte Implementierung der Tile Expiry. Bei der Portierung wurde nur das Nötigste angepasst. Daher ist dieser Teil des Quellcodes schlecht dokumentiert und der Programmierstil ist sehr C-artig. Die Portierung ist unvollständig und dient nur Vergleichszwecken. Anders als bei osm2pgsql wird jedoch nicht das WKB eingelesen. (`--expire-tiles=`*`classic`*)
- `ExpireTilesQuadtree` ist die neue Implementierung, die im Folgenden genauer erläutert wird. (`--expire-tiles=`*`quadtree`*)

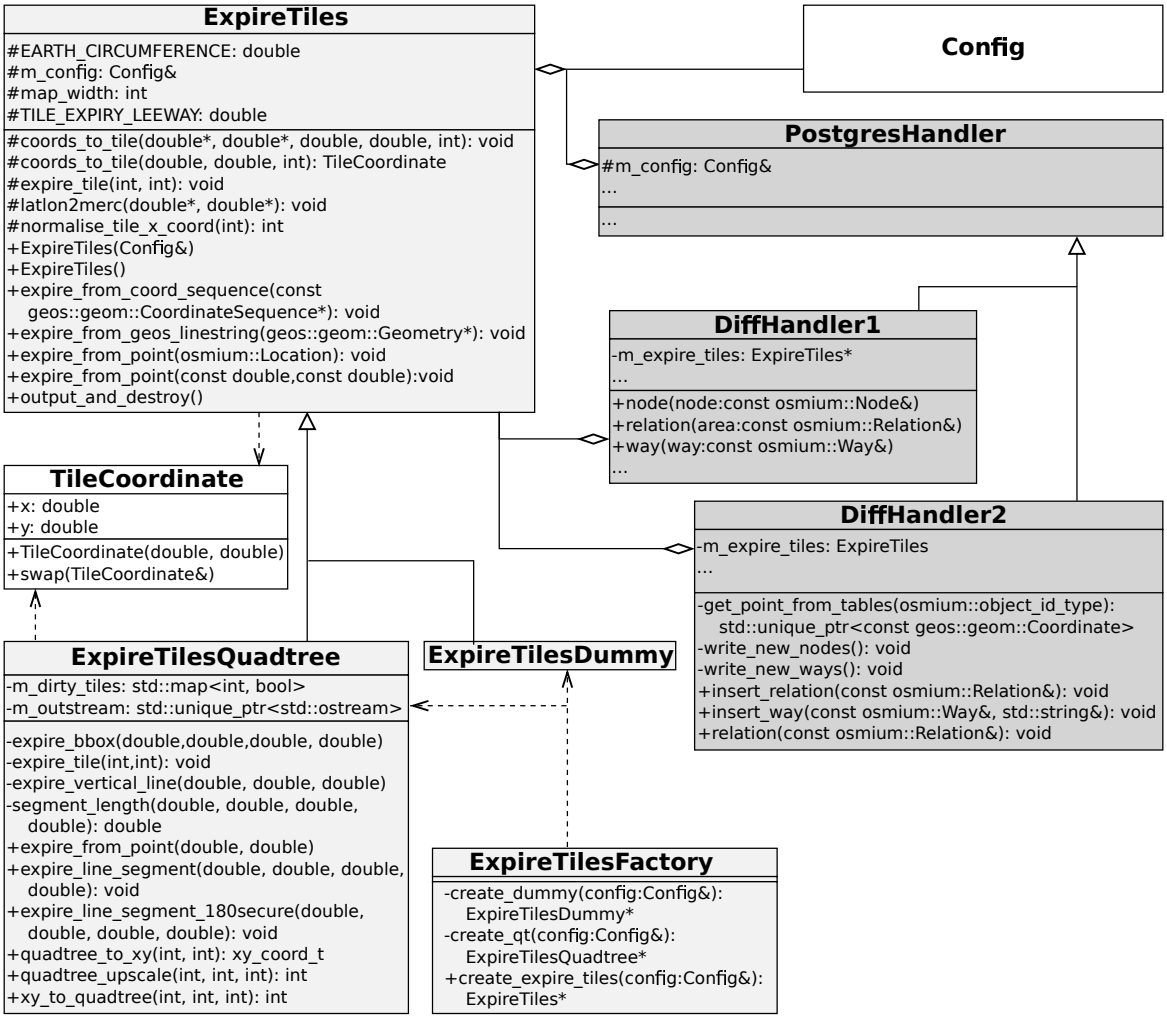


Abbildung 3.3.: UML-Klassendiagramm von Cerepso mit Fokus auf die Tile Expiry, die Klassen Config, PostgresHandler, DiffHandler1 und DiffHandler2 sind nur auszugsweise wiedergegeben

Mit `--min-zoom=`*ZOOMSTUFE* und `--max-zoom=`*ZOOMSTUFE* kann der Nutzer angeben, die Tiles welcher Zoomstufen in den Tile Expiry Logs enthalten sein sollen. Standardwert ist 9 als minimale und 15 als maximale Zoomstufe.

Beim Löschen aus und beim Einfügen eines Objekts in die Datenbank wird jeweils die für den Objekttyp passende Methode der Klasse `ExpireTiles` aufgerufen. Bei Nodes ist es die Methode `ExpireTiles::expire_from_point(osmium::Location&)`, bei Ways ist es die Methode `ExpireTiles::expire_from_coord_sequence(geos::geom::CoordinateSequence*)`. Bei Relationen werden von all deren Mitgliedern ebenfalls diese beiden Methoden aufgerufen. Um Cerepso einfach zu halten, werden Relationen, welche andere Relationen referenzieren, nicht unterstützt. Daher werden keine Tiles als veraltet markiert, wenn eine Relation, die nur andere Relationen referenziert, geändert wird. Die jeweilige Implementierung der Klasse `ExpireTiles` transformiert die geographischen Koordinaten in Web Mercator und wandelt sie in Tile-IDs um.

Im Folgenden wird die neue Quadtree-basierte Implementierung (die Klasse ExpireTilesQuadtree) genauer beschrieben. Kern der Quadtree-Implementierung ist ein Set (std::set), welches für jedes Tile, das als veraltet markiert werden soll, einen Eintrag enthält. Die Einträge sind vom Typ Integer, es handelt sich dabei um die Quadtree-Repräsentation der Tile-Adressen, die bei Bing-Maps verwendet wird (siehe Seite 21). Das Set enthält nur die Adressen in der höchsten Zoomstufe, für die die Tile Expiry berechnet werden soll. Die Tile-IDs der niedrigeren Zoomstufen werden erst beim Schreiben der Tile-Expiry-Datei berechnet. Um eine Quadtree-Adresse in der Zoomstufe n in eine Quadtree-Adresse der Zoomstufe $n - d$ zu transformieren, ist ein Bitshift um $2 \cdot d$ nach rechts erforderlich. (vgl. Abschnitt 1.4 und insbesondere Abbildung 1.4)

Da ein Set jedes Element höchstens einmal enthält, wird kein Tile doppelt ausgegeben, auch wenn sich mehrere Geometrien in diesem Tile geändert haben und für jede geänderte Geometrie einer der Methoden der Klasse ExpireTiles aufgerufen wurde.

Das Tile, in dem ein veränderter Node liegt, als veraltet zu markieren, ist relativ einfach. Die Koordinaten des Nodes werden in Tile-IDs konvertiert und in das Set eingefügt, falls das Tile nicht schon wegen einer anderen Änderung als veraltet markiert wurde.

Bei Ways ist es nicht so einfach. Es genügt nicht, nur die Tiles, in denen ein Way Nodes hat, als veraltet zu markieren. Stattdessen müssen alle Tiles, in denen der Way liegt, als veraltet markiert werden. Abbildung 3.4 zeigt auf der linken Seite, wie ein Node eines Ways verschoben wurde. Wenn nur die Tiles neu gerendert werden, in denen die Nodes liegen, weil nur diese Tiles als veraltet markiert wurden, führt das zu unerwünschten Ergebnissen. Bei gerenderten Karten hängen z. B. Straßen in der Luft (rechte Seite). Durch das Verschieben des Nodes könnten auch Multipolygone invalide werden (z. B. könnten sich jetzt zwei Ringe kreuzen). Wenn die Tiles nicht als veraltet markiert werden, werden sie nicht prozessiert und der Fehler wird nicht gefunden.

Erst ab einem gewissen, hart kodierten, Schwellwert (derzeit 20 km) werden nur noch die Tiles als veraltet markiert, in denen der LineString Stützpunkte hat. Damit soll vermieden werden, dass bei sehr langen Segmenten zu viele Tiles als veraltet markiert werden. Die meisten langen Linien haben viel häufiger einen Stützpunkt – Hochspannungsleitungen z. B. alle paar hundert Meter einen Masten. Etwa ein bis fünf Mal im Jahr erzeugen Mapper versehentlich Polygone, die über einen ganzen Kontinent

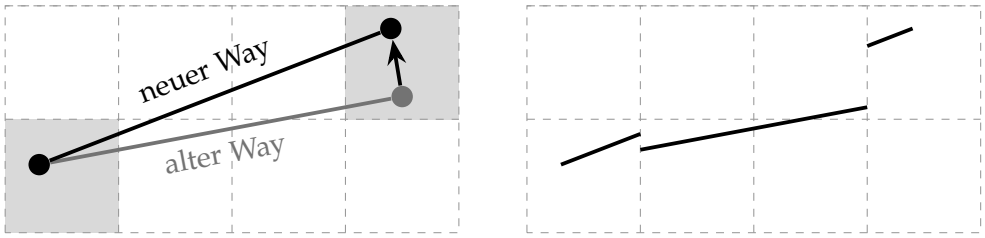


Abbildung 3.4.: Wenn ein Node eines Ways verschoben wird, aber nur die Tiles als veraltet (grauer Hintergrund) markiert werden, in denen die Nodes liegen (links), führt das zu unerwünschten Ergebnissen, die gerade beim Rendering von Karten besonders auffällig sind (rechts).

reichen. Sei es, dass sie die Skalierungsfunktion ihres Editors falsch bedienen oder einen Node von Nordamerika nach Europa verschieben. Eine solche Veschiebung würde ohne den Schwellwert das Neuprozessieren unzähliger Tiles veranlassen und ein manueller Eingriff wäre notwendig, der die Warteschlange leert (auch wenn dadurch einige korrekte Tiles nicht prozessiert werden würden).

Durch Unit-Tests wurde sichergestellt, dass die Transformationen korrekt sind und auch bei Geometrien, die den 180. Längengrad kreuzen, die Ergebnisse korrekt sind. Solche Geometrien können prinzipiell in OSM existieren, auch wenn sie meistens recht schnell durch an diesem Meridian aufgeteilte Geometrien ersetzt werden. Viele Anwendungsprogramme können mit diesen Geometrien nicht umgehen.

4. Partitionierung

4.1. Das Partitionierungsverfahren

In der Online-Kartographie hat sich in den vergangenen zehn Jahren das Prinzip der *Map Tiles* (siehe Kapitel 1.4) durchgesetzt. Dieses Partitionierungsverfahren ist nicht datengetrieben, sondern teilt die Karte in gleich große quadratische Kacheln (Tiles) ein. Es ist für Entwickler recht einfach zu implementieren und bringt dennoch spürbare Performance-Verbesserungen, da sich die Daten bzw. die daraus erstellten Produkte (z. B. Kartentiles) leicht cachen lassen.

Die alleinige Verwendung von Tiles als Partitionierungsschema ist für den Zweck der Datenprozessierung nicht geeignet. Wenn man auf einer zu niedrigen Zoomstufe partitioniert, sind in Gebieten mit hoher Datendichte die Tiles zu groß und ihre Prozessierung dauert zu lange. In Gebieten mit niedriger Datendichte (dünn besiedelte und unbesiedelte Gebiete) sind die Tiles zu klein. Die Änderung eines Ways in solchen Gebieten in OpenStreetMap sorgt dafür, dass unzählige Tiles neu prozessiert werden müssen, da die Ways in diesen Gebieten relativ lang sind.

Ein rein datengetriebenes Partitionierungsverfahren, beispielsweise die Verwendung eines Quadtrees, hat ebenfalls Nachteile. Die Implementierung ist aufwendiger, da nicht allein anhand der Koordinaten eines Objekts entschieden werden kann, zu welchem Blattknoten es gehört. Erst wenn ein Knoten zu „voll“ ist, wird er aufgeteilt. Weder der Planetdump noch die Diffs sind räumlich sortiert, stattdessen sind sie aufsteigend nach ID sortiert. Es wären daher während des Einlesens der OSM-Daten andauernd Neuaufteilungen zu voller Knoten erforderlich, was der Performance nicht zuträglich wäre.

Die Verwendung eines Tile-basierten Partitionierungsverfahrens ermöglicht es außerdem, Software, die im Rahmen dieser Arbeit entwickelt wurde, auch anderweitig einzusetzen. *Cerepso* und *Cerepso2vt* sind Beispiele hierfür. Sowohl die Tile-Expiry von Cerepso als auch die Erzeugung von Vektortiles durch den Cerepso2vt setzen auf dem Prinzip der Tiles auf.

Daher fiel die Entscheidung, ein hybrides Verfahren zu verwenden, das zwar ein Tile-basiertes Verfahren ist, aber auch datengetriebene Elemente hat. Gegenüber der meisten eingesetzten Software verhält es sich wie ein reines Tile-basiertes Verfahren.

Grundsätzlich erfolgt die Partitionierung auf einer Zoomstufe z_0. Diese ist so gewählt, dass in Gebieten mit geringer Datendichte und einer großen projektionsbedingten Verzerrung keine zu kleinen Tiles entstehen. Gerade Kanada, Alaska und weite Teile Russlands haben sehr wenig Daten pro Quadratkilometer; ein Tile auf Zoomstufe z_0 deckt dort aber ein kleineres Gebiet als weitere südlich ab. Die Größe (Datenmenge) eines Tiles wird nur näherungsweise anhand der Anzahl der Nodes in diesem Tile bestimmt, anstatt

alle Objekte zu zählen. Hat ein Tile mehr als s Nodes, wird es in seine vier Untertiles aufgeteilt. Haben die vier Untertiles immer noch mehr als n Nodes, werden sie erneut aufgeteilt. Das Aufteilen endet an einer festgelegten Zoomstufe z_n, die so gewählt ist, dass die entstehenden Tiles keine zu kleine Fläche abdecken. An dieser Stelle hat das Partitionierungsschema die Charakteristik eines *linearen Quadtree* [25]. Abbildung 4.1 zeigt die resultierende Partitionierung im Bereich der Elbe- und Wesermündung, sie wurde mit *expiries2shp* (siehe Anhang F) erstellt. Für die Details zur Wahl von z_0, z_n und s sei auf Kapitel 4.3 verwiesen.

Abbildung 4.1.: Resultierende Partitionierung im Bereich der Elbe- und Wesermündung mit $z_0 = 9$, $z_n = 16$, $s = 5000$; Datenbasis ist der Planetdump vom 29. August 2016

Diese Aufteilung ist statisch und passt sich nicht dynamisch an die Änderungen im Datenbestand an. Dadurch kann das Verfahren nach außen als (statisches) Tile-basiertes Verfahren auftreten, sich intern aber dennoch an die ungleiche räumliche Verteilung der Daten anpassen, was eine der Forderungen an räumliche Indexe ist [25]. In größeren zeitlichen Abschnitten muss mit *dense_tiles* von Frederik Ramm [104] für alle Zoomstufen von z_0 bis z_n berechnet werden, wie viele Nodes die einzelnen Tiles enthalten:

1. Für die Zoomstufe z_0:
   ```
   $ ./dense_tiles -sac -z ZOOM_LEVEL PLANETFILE | sort > tile-sizes-z9.txt
   ```
   Die Ausgabe von dense_tiles erfolgt auf der Standardausgabe und dient als Eingabe für *sort* [45], das die Ausgabe alphabetisch sortiert. Die endgültige Ausgabe wird in eine Datei umgeleitet.
   `ZOOM_LEVEL` ist die Zoomstufe. `PLANETFILE` ist der Pfad zum Planetdump.
   Listing 4.1 zeigt einen Auszug aus der Ausgabe von dense-tiles für Zoomstufe 9.

2. Für jede weitere Zoomstufe von z_1 bis z_n:
   ```
   $ ./dense_tiles -sac -M MAX_NODES -z ZOOM_LEVEL PLANETFILE | sort
     ↪ > tile-sizes-zZOOM_LEVEL.txt
   ```
   `MAX_NODES` ist die maximale Kapazität s eines Tiles. Für die Zoomstufen $> z_0$ müssen nur die Tiles ermittelt werden, die mehr als s Nodes haben, da nur sie weiter aufgeteilt werden müssen. Für jede Zoomstufe werden etwa 10 bis 15 Minuten[1] benötigt, wovon etwa die Hälfte der Zeit allein für das Einlesen des Planetdumps benötigt wird und sich durch einen schnelleren Prozessor ein wenig, aber eher durch einen Prozessor mit mehreren Kernen um zwei bis vier Minuten senken ließe. Listing 4.2 zeigt die alphabetisch sortierte Ausgabe für Zoomstufe 10.

Listing 4.1: alphabetisch sortierte Ausgabe von *dense_tiles* für die Zoomstufe 9

```
9/0/0 1
...
9/100/14 0
9/100/140 64367
9/100/141 5933
9/100/142 2441
...
9/99/99 73
```

Listing 4.2: sortierte Ausgabe von *dense_tiles* für die Zoomstufe 10

```
...
10/200/267 9530
10/200/279 6331
10/200/281 29166
10/200/326 11702
...
10/201/267 15827
10/201/280 7154
10/201/281 23868
10/201/282 5080
...
```

3. Das im Rahmen dieser Arbeit geschriebenen Python-Skript *split-inital-tiles.py* ausführen:
   ```
   $ python3 split-inital-tiles.py tile-sizes-z9.txt MAX_NODES | sort > z9.txt
   ```
   Es liest die in Schritt 1 erstellte Datei ein, welche für jedes Tile auf Zoomstufe z angibt, wie viele Nodes es enthält. Enthält das Tile weniger als s Nodes, wird es unverändert (jedoch ohne die Anzahl der Nodes) auf der Standardausgabe ausgegeben. Enthält es mehr als s Nodes, werden seine vier Untertiles auf der Zoomstufe z_1 ausgegeben. Die Ausgabe muss mit sort alphabetisch sortiert werden. Sie hat dasselbe Format wie Tile-Expiry-Listen.

[1] Intel Core i5-750, 16 GB RAM, Festplatte Seagate ST1000VX000 mit 7200 rpm SATA-600

4. Für jede weitere Zoomstufe von z_1 bis z_n das im Rahmen dieser Arbeit geschriebene Python-Skript *next-level.py* ausführen:
   `$ python3 next-level.py z`*`ZOOM_THIS`*`.txt tile-sizes-z`*`ZOOM_NEXT`*`.txt `*`ZOOM_NEXT`*
   `↪ `*`MAX_NODES`*` | sort > z`*`ZOOM_NEXT`*`.txt`
   ZOOM_THIS ist dabei z_i, *ZOOM_NEXT* ist dabei z_{i+1}. Auch die Ausgabe dieser Schritte wird jedes Mal alphabetisch sortiert.
   Für jedes in der Datei z*ZOOM_THIS*.txt aufgeführte Tile wird unverändert ausgegeben, wenn es ein Tile der Zoomstufe z_i oder kleiner ist, da dann schon beim vorherigen Durchlauf geprüft wurde, dass das Tile weniger als s Nodes enthält. Für die in der Datei z*ZOOM_THIS*.txt aufgeführten Tiles der Zoomstufe z_{i+1} wird in der Datei tile-sizes-z*ZOOM_NEXT*.txt geprüft, ob das Tile zu groß ist. Wenn es zu groß ist, werden seine vier Untertiles auf Zoomstufe z_{i+2} ausgegeben. Andernfalls wird es unverändert ausgegeben.

Ein Durchlauf von split-inital-tiles.py bzw. next-level.py dauert je nach Zoomstufe 5 bis 15 Sekunden, der Speicherverbrauch ist sehr gering[2]. Da die Ausgabe jeweils alphabetisch sortiert wird, können die Dateien nach dem Prinzip des Streamings zeilenweise eingelesen werden und keine Daten müssen gepuffert werden.

Da die Expiry-Listen, die von *Cerepso* bei jedem Diff-Import erzeugt werden, dasselbe Format haben wie die Ausgabe von next-level.py, können sie mit dem Diff-Werkzeug *comm* vereinigt werden. comm dient dem zeilenweisen Vergleich zweier Textdateien und zeigt standardmäßig drei Spalten an. In der ersten Spalte stehen die Zeilen, die nur in der ersten Datei enthalten sind, in der zweiten Spalte nur die Zeilen, die nur in der zweiten Datei enthalten sind. Die dritte Spalte enthält die Schnittmenge beider Dateien [44]. Listing 4.3, 4.4 und 4.5 zeigen das anschaulich. Um die beiden Listen mit comm vereinigen zu können, müssen sie alphabetisch sortiert sein (erfolgt mit sort).

Listing 4.3: Inhalt von Beispieldatei 1 (Eingabe für *comm* in Listing 4.5)

```
$cat datei1.txt
line only in 1
line in both
line in both, too
last line in 1
```

Listing 4.4: Inhalt von Beispieldatei 2 (Eingabe für *comm* in Listing 4.5)

```
$cat datei2.txt
line only in 2
line in both
line in both, too
last line in 2
```

Listing 4.5: Beispielhafte Ausgabe von comm

```
$comm datei1.txt datei2.txt
line only in 1
                    line only in 2
                                        line in both
                                        line in both, too
last line in 1
                    last line in 2
```

[2] Intel Core i5-750, 16 GB RAM, Festplatte Seagate ST1000VX000 mit 7200 rpm SATA-600

Da in den Expiry-Listen ein Tile auf allen Zoomstufen von z_0 bis z_n enthalten ist, wird mit comm das Tile herausgefiltert, das von Cerepso2vt erzeugt werden soll:

```
$ comm EXPIRY_LOGFILE TILE_LIST
```

EXPIRY_LOGFILE ist die Expiry-Liste, die Cerepso ausgegeben hat. *TILE_LIST* ist das Ergebnis der oben erwähnten Schritte 1 bis 4.

4.2. Wechselwirkungen zwischen Inhalt und Häufigkeit

Wie häufig ein Tile bei einem Diff-Import mit *Cerepso* als veraltet markiert wird, ist nicht allein davon abhängig, ob es im Gebiet des Tiles Änderungen gegeben hat. Die ursprüngliche Implementierung in Cerepso markierte Tiles als veraltet, wenn eine große Geometrie (z. B. ein langer Way) die Tiles durchquerte, auch wenn sich in den Tiles selbst nichts an dem Way geändert hatte, sondern die Änderungen alle außerhalb des Tiles lagen (z. B. einen Node einige Tiles entfernt verschoben wurde). Prinzipiell könnte die Tile-Expiry das berücksichtigen, auf die Implementierung wurde jedoch aus Zeitgründen verzichtet. Tileserver würden partielle Tile Expiry bei rein geometrischen Änderungen nicht benötigen. Ändert sich die Geometrie eines Ways, so ändert sich fast immer auch dessen Länge. Beschriftungen entlang des Ways (z. B. Straßen- und Flussnamen) würden bei einer veränderten Länge an etwas anderen Positionen gerendert werden. Da aber nur ein Teil der Tiles, in denen Way liegt, neu gerendert werden, liefert der Tileserver weiterhin für Teile des Ways alte Tiles aus.

Um einen Eindruck zu erhalten, wie häufig die Tiles neu prozessiert werden müssten, wurden der Planetdump vom 29. August 2016 und anschließend die stündlichen Diffs vom 29. August 2016 02:00 UTC (Diff-ID 000/034/723) bis zum 5. Oktober 2016 08:00 UTC (000/035/617) importiert und für jedes Diff ein separates Tile-Expiry-Logfile erstellt. Die Berechnung der veralteten Tiles erfolgte für die Zoomstufen 9 bis 15. Zoomstufen 9 wurden von Anfang an als zu kleinmaßstäblich betrachtet, da die Tiles in diesen Zoomstufen viel zu groß wären. Zoomstufen größer als 15 haben den Nachteil, dass die Tiles dort sehr klein sind und zu wenig Umgebung enthalten wäre. Das im Rahmen dieser Masterarbeit geschriebene C++-Hilfsprogramm *expiries2shp* erstellte aus den Logfiles ein Shapefile. Das Shapefile enthielt für jeden Eintrag im Logfile das Gebiet des Tiles als Polygon und zusätzliche Spalten für den x- und y-Index des Tiles, dessen Zoomstufe und die Sequenznummer des Diffs. Alle Shapefiles wurden mit *shp2pgsql* in eine PostGIS-Datenbank importiert. In der PostGIS-Datenbank wurden mit dem in Listing 4.6 abgedruckten SQL-Befehl die Tiles zusammengefasst, sodass es keine flächengleichen Polygone mehr gab. Jedes Polygon erhielt dabei ein zusätzliches Attribut *count*, das beschrieb, wie oft das Tile als veraltet markiert wurde. Für Details zu expiries2shp sei auf Anhang F verwiesen. Dies ermöglichte eine Visualisierung der betroffenen Tiles (mit QGIS erfolgt) und diente als Grundlage für weitere Analysen. Der Zeitraum von etwas mehr als 38 Tagen dient für alle weiteren Analysen als Basis.

Abbildung 4.2 zeigt, wie oft Tiles der Zoomstufe 12 in diesem 38 Tage langen Zeitraum als veraltet markiert worden wären. Es fällt auf, dass vor allem die Tiles in großen Städten als veraltet markiert worden wären. Vor allem das Tile 12/2200/1343, welches Berlin-

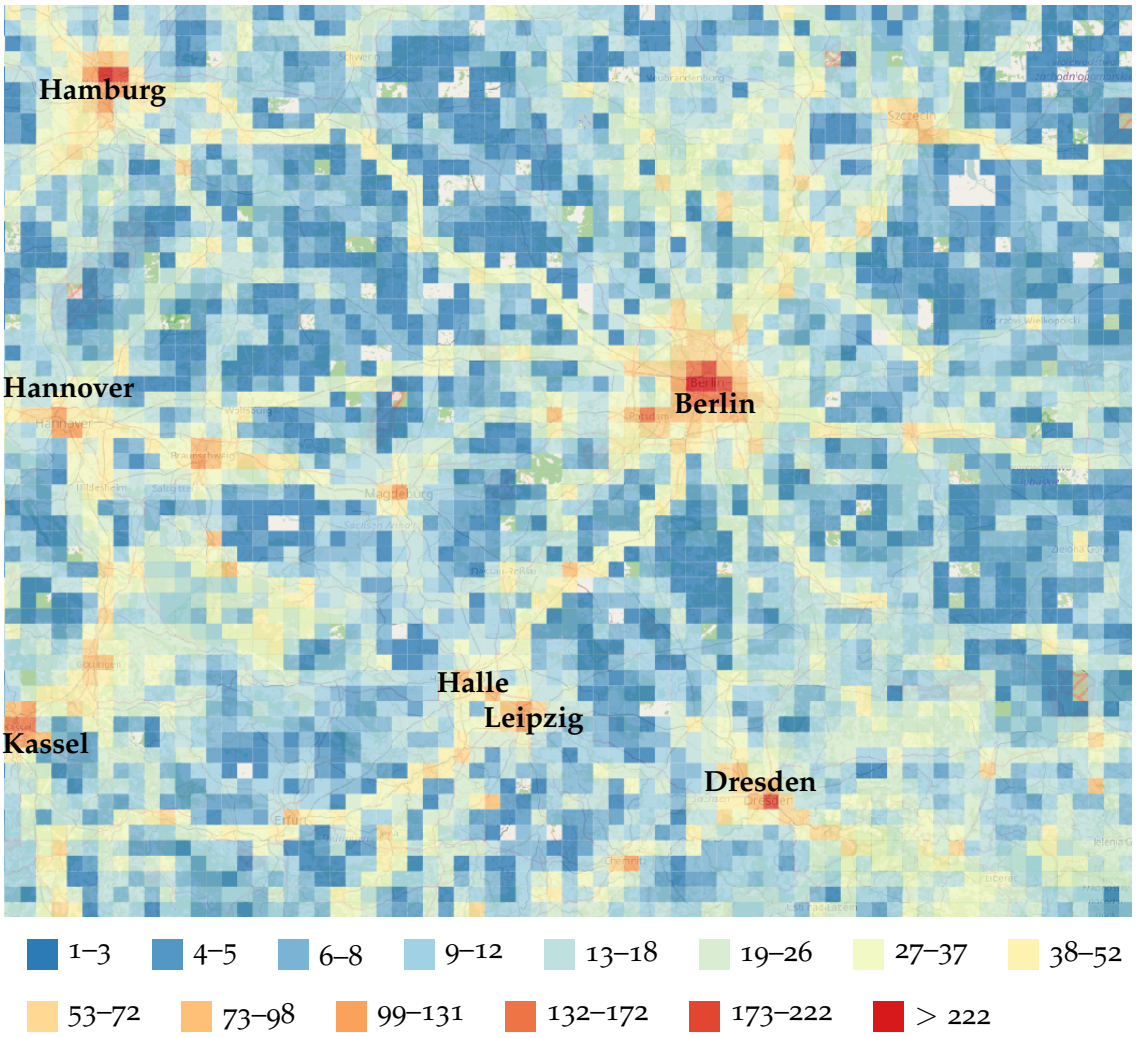


Abbildung 4.2.: Anzahl der Markierungen von Tiles auf Zoomstufe 12 als veraltet im Nordosten Deutschlands beim Import der stündlichen Diffs, Zeitraum 29. 8.–5. 10.2016

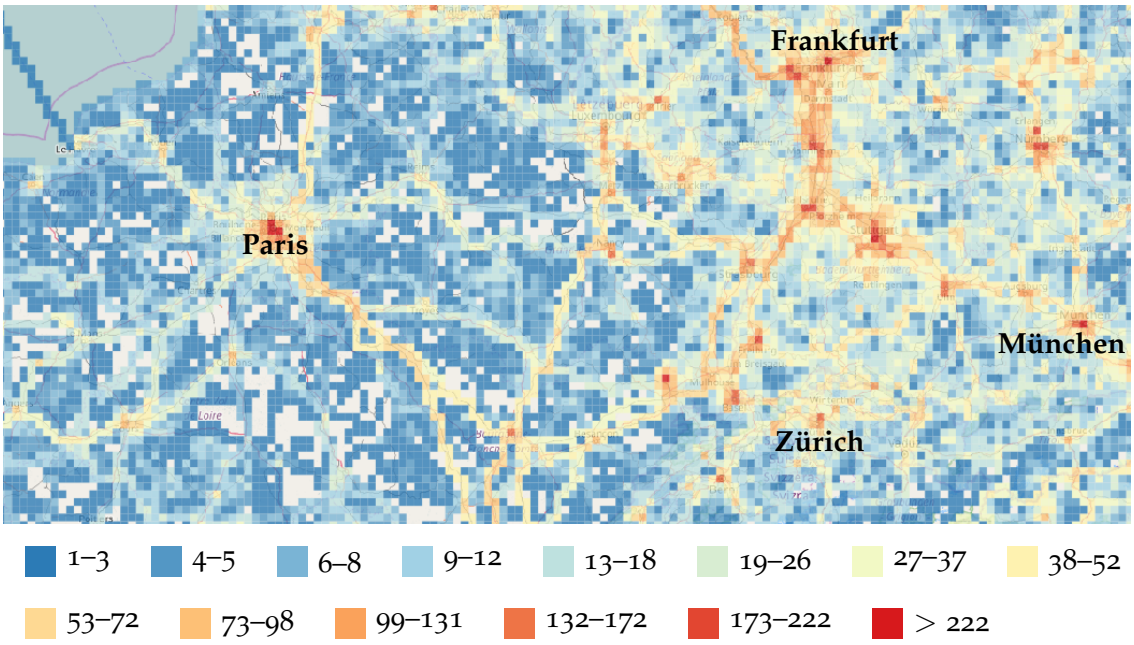


Abbildung 4.3.: Anzahl der Markierungen von Tiles auf Zoomstufe 12 als veraltet beim Import der stündlichen Diffs im Nordosten Frankreichs und Südwestdeutschland, Zeitraum 29. 8.–5. 10.2016

Mitte, Kreuzberg und Friedrichshain enthält, sticht heraus. Es wurde 382 Mal (d. h. durchschnittlich alle 2 bis 3 Stunden) markiert und gehört zu Top 5 Tiles weltweit, was die Anzahl der Markierungen als veraltet betrifft. Insgesamt wurden in diesem Zeitraum 3,464 Millionen Tiles auf Zoomstufe 12 als veraltet markiert. Wenn ein Tile mehrfach als veraltet markiert wurde, wird es mehrfach gezählt, weil jede Markierung eine Neuprozessierung auslöst. Wenn in dem Tile jedoch mehrere Objekte innerhalb einer Stunde editiert wurden, erfolgt nur eine Markierung.

Diese Beobachtungen decken sich mit der Erfahrung, dass die Daten vor allem dort detailreich und gut gepflegt sind, wo die Mapper wohnen. Aber nicht nur Ballungsräume werden oft als veraltet markiert. Bei genauerem Hinsehen fällt auf, dass die meisten Bahnstrecken, die von Fernverkehrszügen befahren werden, recht oft als veraltet markiert wurden. Die Tiles entlang der von mehreren ICE-Linien befahrenen Bahnstrecke Berlin–Leipzig/Halle wurden, auch wenn sie im ländlichen Brandenburg und Sachsen-Anhalt liegen, alle mindestens 35 Mal als veraltet markiert; benachbarte Tiles i. d. R. nur ein- bis sechsmal. Das Muster tritt in Deutschland intensiv, dazu noch an entlang einiger TGV-Strecken in Frankreich auf (vgl. Abbildung 4.3).

Listing 4.6: SQL-Abfrage zum Zusammenfassen von Polygonen, die mit *expiries2shp* aus den Tile-Expiry-Listen erzeugt wurden

```
SELECT count(geom), x, y, zoom, geom FROM expiry WHERE zoom = 12 GROUP BY x, y,
    ↪zoom, geom;
```

Es scheint, als wären Änderungen an den Routenrelationen hunderte Kilometer entfernt dafür verantwortlich. Mit den Standardeinstellungen der Tile-Expiry von Cerepso (bis zu dieser Analyse waren diese Parameter hart kodiert) werden bei einer Änderung einer Relation alle Tiles als veraltet markiert, in denen ein Node oder einen Way liegt, der Mitglied der Relation ist. Teilt ein Mapper in Frankfurt am Main einen Way, welcher ein Eisenbahngleis repräsentiert, in zwei Ways auf, werden auch zwangsläufig alle Relationen, die diesen Way referenzieren geändert, denn der zweite Teil des Ways muss auch von den Relationen referenziert werden. So genügt eine Änderung eine Gleises im Frankfurter Hauptbahnhof, um alle Tiles von Frankfurt über Halle bis Berlin als veraltet zu markieren.

Betrachtungen der Daten außerhalb Deutschlands bestätigen diesen Eindruck. Die meisten Tiles, durch die eine Grenze eines souveränen Staats verläuft, haben sich geändert. Das fällt dort auf, wo die Staatsgrenze als seeseitiger Rand der 12-Meilen-Zone entlang der Küste verläuft. Im Meer gibt es meist wenig Daten, Änderungen sind dort selten.

Änderungen an Relationen sind daher vermutlich für den größten Teil der Markierungen als veraltet verantwortlich. Es gibt mehrere Möglichkeiten, dieses Problem zu lösen und dadurch die Anzahl der veralteten Tiles zu reduzieren:

- Durch Ignorieren der Relationen reduziert sich die Anzahl der Markierungen erheblich. Dieser Ansatz hat jedoch Nachteile. Wenn die Tags, die Rollen der Mitglieder oder die Mitgliederliste an sich geändert werden, ohne dass eine geometrische Änderung beteiligt ist, werden die Tiles der Relation nicht als veraltet markiert und die durch die Änderung eingeführten Fehler werden nicht entdeckt. Das Ergänzen eines Ways zu einer Multipolygon-Relation kann diese jedoch invalidieren, wenn sich dadurch z. B. zwei Ringe kreuzen oder nicht-tangential berühren. Zu den möglichen Fehlern einer Multipolygon-Relation sei auf Abschnitt 4.6.3 verwiesen.

  Auf den Tileservern der OpenStreetMap Foundation wird das Tile-Expiry-Skript von Matt Amos [14] (vgl. Seite 49) eingesetzt [7, 8]. Das zeigt, dass sich ein Betreiber eines Tileservers sich das Ignorieren von Relationen leisten kann, ohne dass es den Nutzern auffällt. Für die Fehlersuche ist diese Lösung jedoch zu simpel und mit zu großen Nachteilen behaftet.

- Der ideale Weg wäre, durch Vergleichen beider Relationen (sowohl die Tags als auch die Mitgliederliste und die resultierende Geometrie) die Tiles zu finden, die sich wirklich geändert haben. Es handelt sich jedoch um eine sehr geringe Zahl an zusätzlichen Fehlern, die damit zusätzlich entdeckt werden. Wenn eine Relation durch eine Bearbeitung eines Mappers invalide wird, ist es meist auch mit einer Änderung eines der Mitglieder verbunden. Eine perfekte Lösung brächte daher einen unnötig hohen Entwicklungsaufwand mit sich. Sie würde die Tile-Expiry-Implementierung komplexer machen und ist für die meisten Anwender nicht von Interesse. Deren Bedürfnisse sind mit einer 80-20-Lösung hinreichend befriedigt. Performance (schneller und speichersparsamer Diff-Import) ist ihnen wichtiger.

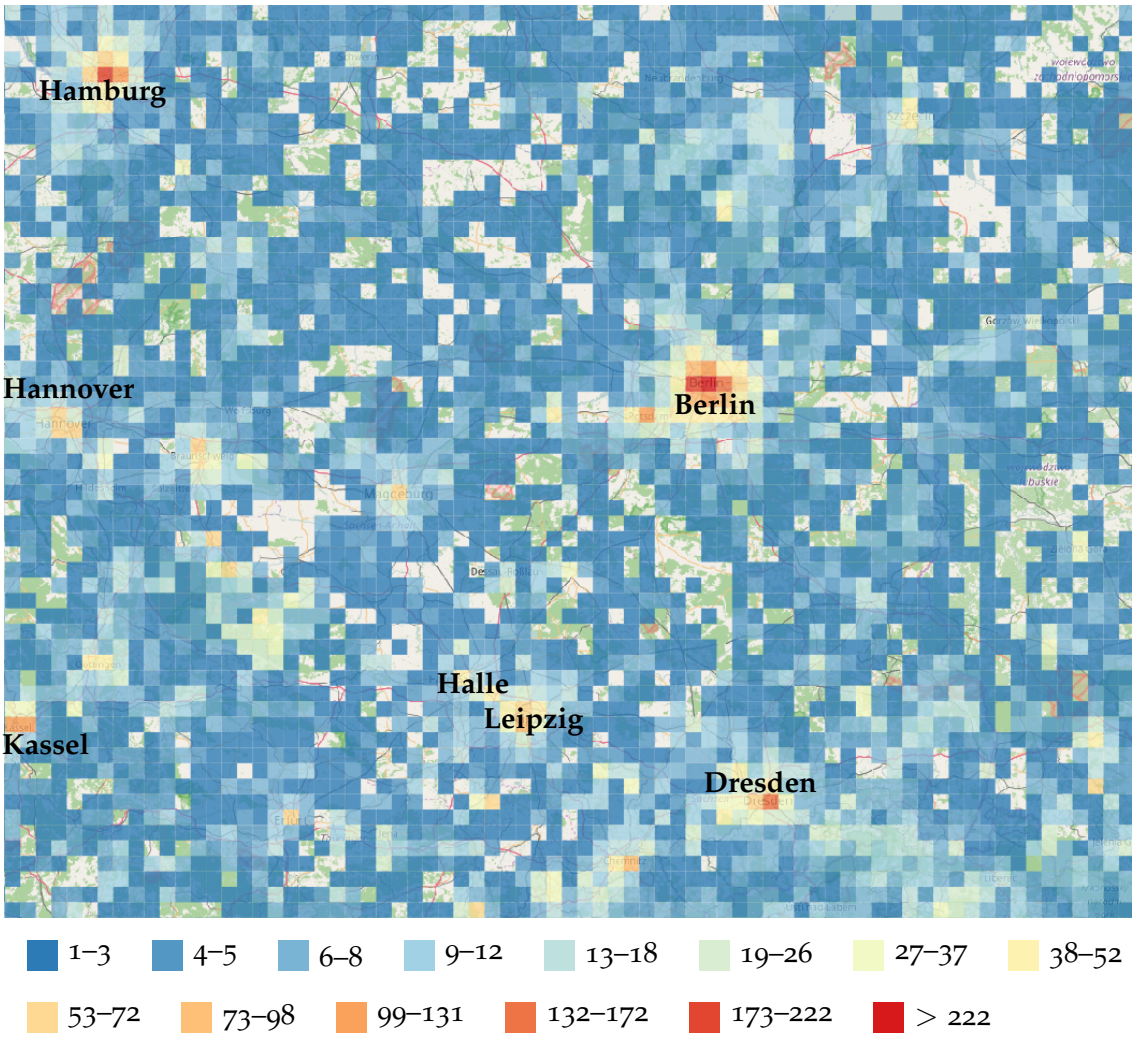


Abbildung 4.4.: Anzahl der Markierungen von Tiles auf Zoomstufe 12 als veraltet beim Import der stündlichen Diffs, Zeitraum 29. 8.–5. 10.2016, wenn keinerlei Relationen berücksichtigt werden

- Staatsgrenzen ändern sich seltener als Routenrelationen. Es liegt also nahe, Relationen prinzipiell zu berücksichtigen, aber Routenrelationen zu ignorieren.

Abbildung 4.4 zeigt dasselbe Gebiet wie Abbildung 4.2 mit derselben Farbskala, jedoch ohne Relationen zu berücksichtigen. Änderungen an Relationen lösten keine Markierung eines Tiles aus. Erst, wenn ein Way oder Node geändert wurde, wurden alle Tiles, in denen das Objekt liegt, als veraltet markiert. Auf Zoomstufe 12 gab es nur noch knapp 981 000 Markierungen als veraltet (statt 3,464 Millionen). Eisenbahnlinien treten nicht mehr hervor. Änderungen an administrativen Grenzen haben keinen Einfluss mehr, wenn nur die Tags oder Mitgliederliste der Relation bearbeitet wurde. Stattdessen treten Gebiete mit höherer Bevölkerungsdichte stärker hervor und in ländlichen Gebieten tun sich teils große Löcher auf.

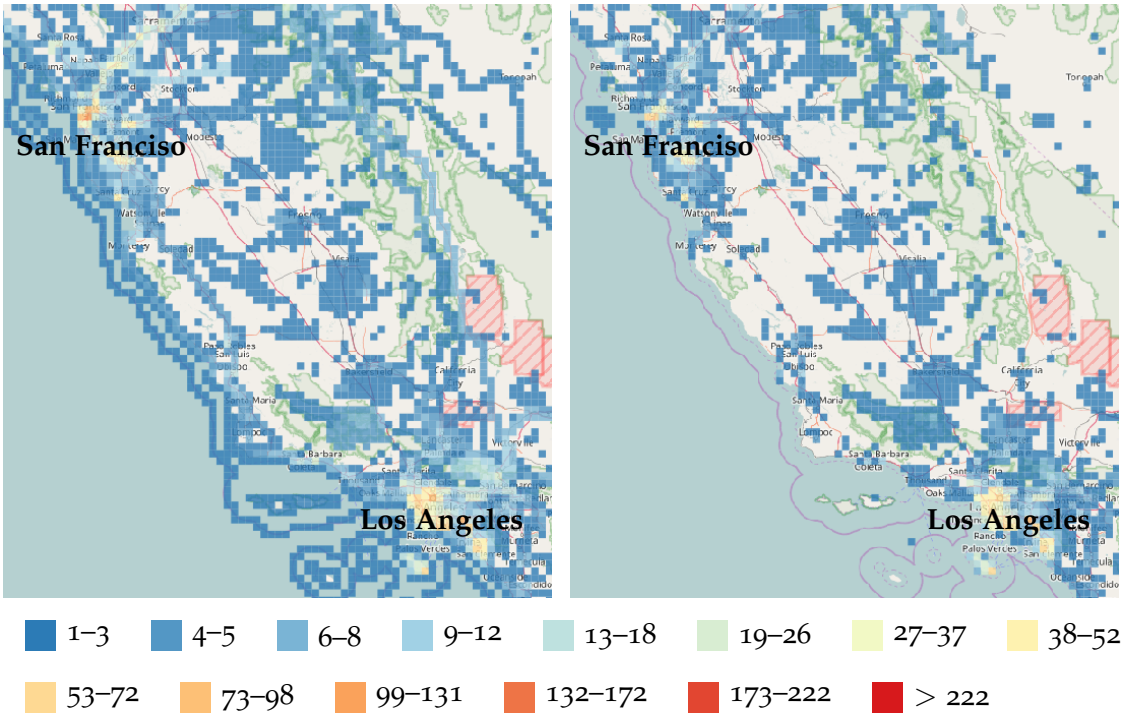


Abbildung 4.5.: Anzahl der Markierungen von Tiles auf Zoomstufe 12 als veraltet beim Import der stündlichen Diffs im mittleren Kalifornien, Zeitraum 29. 8.–5. 10.2016; links: mit Relationen, rechts: ohne Relationen

Abbildung 4.6.: Anzahl der Markierungen von Tiles auf Zoomstufe 12 als veraltet beim Import der stündlichen Diffs im Norden Südamerikas und Süden Mittelamerikas, Zeitraum 29. 8.–5. 10.2016; Legende siehe Abbildung 4.5

Die Abbildungen 4.2 und 4.3 ermöglichen auch eine Analyse der Mappingaktivität in den einzelnen Regionen der Erde und bestätigen Statistiken zur Mappingaktivität, die sich meist nur auf der Ebene eines Landes auflösen. Frankreich ist offensichtlich inaktiver als Deutschland. In ländlichen Regionen gibt es dort zusammenhängende Blöcke mehrerer Zoom-12-Tiles, in denen es in diesem Zeitraum keine Änderungen gab. Die Vereinigten Staaten zeigten ein ähnliches Editiermuster wie Frankreich. Jedoch fehlen in den Vereinigten Staaten die Effekte langer Routenrelationen mangels eines gut ausgebauten Schienenfernverkehrs. Abbildung 4.5 zeigt beispielhaft Kalifornien zwischen San Francisco und Los Angeles.

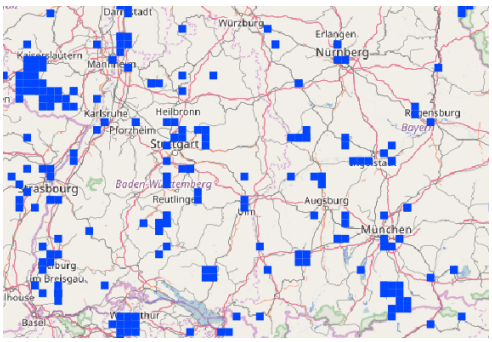
Mit einer Visualisierung der Tile-Expiry lassen sich jedoch auch schädliche Änderungen entdecken und schneller finden, die vielen Mappern und Datennutzern auffallen. Abbildung 4.6 zeigt die als veraltet markierten Tiles im Norden Südamerikas und dem südlichen Mittelamerika vom 29. August bis 5. Oktober 2016. Dabei springt das dunkelblaue Rechteck über den halben Kontinent ins Auge. Es handelte sich dabei um ein Gebäude, das durch eine Fehlbedienung der Skalierungsfunktion eines Editors vergrößert wurde. Eine Fehlerüberprüfung führt der verwendete Editor nicht durch. Solche Fehler fallen den Nutzern des auf openstreetmap.org verwendeten Kartenstils *OSM Carto* auf, weil anschließend alle Tiles in dem Gebiet, sobald sie aus einem anderen Grund als veraltet markiert wurden, beim Rendern das für Gebäude verwendete Braun als Hintergrundfarbe haben. Die Ursache zu finden, ist nicht ganz trivial, herkömmliche Werkzeuge können das nicht leisten, sodass es gern eine Weile dauert, bis das Problem gefunden ist.

4.3. Größe der Tiles

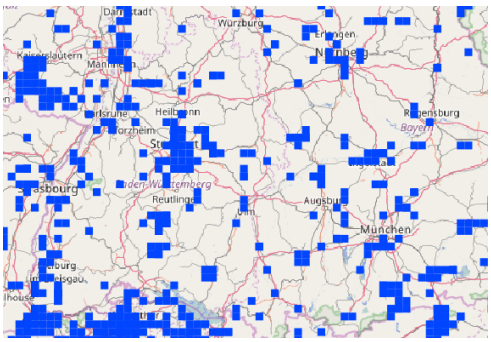
Die Größe der Tiles muss so klein gewählt werden, dass ein Tile innerhalb einer akzeptablen Zeit verarbeitet werden kann. Andererseits muss das Tile so groß sein, dass es eine ausreichend große Umgebung enthält, denn bei einigen Abfragen ist die Umgebung des Objekts relevant. Wenn unverbundene, aber nahe bei einander liegende Straßenenden gesucht werden sollen, sollte zwischen zwei unverbundenen Enden möglichst keine Kante eines Tiles liegen. Daher müssen die Tiles sich ein wenig überlappen.

Abbildung 4.7 zeigt die Tiles, die zwischen dem 4. September 2016 05:00:01 UTC und dem 5. September 2016 05:00:00 UTC in Süddeutschland als veraltet markiert wurden. Wird eine zu niedrige Zoomstufe für die Partitionierung gewählt, werden in Gebieten mit hoher Datendichte, wie beispielsweise Mitteleuropa, unnötig viele Daten verarbeitet. Die Gesamtfläche aller zu prozessierenden Tiles ist bei einer Partitionierung auf Zoomstufe 14 um 75,4 Prozent geringer als bei einer Partitionierung auf Zoomstufe 12. Diese Ersparnis lässt sich dadurch erklären, dass die als veraltet markierten Rechtecke ein an die Geometrie angepasstes Raster darstellen. Je feiner aufgelöst das Raster ist (d. h. je höher die Zoomstufe), desto weniger Tiles werden als veraltet markiert.

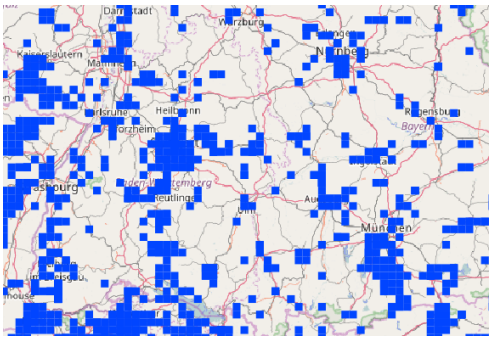
Dennoch ist es keine gute Idee, einfach auf einer sehr hohen Zoomstufe zu partitionieren. In unbesiedelten Gebieten befindet sich oft nur ein Way (z. B. ein Fluss) in einem Tile der Zoomstufe 12. Wenn die Partitionierung auf Zoomstufe 12 oder gar einer



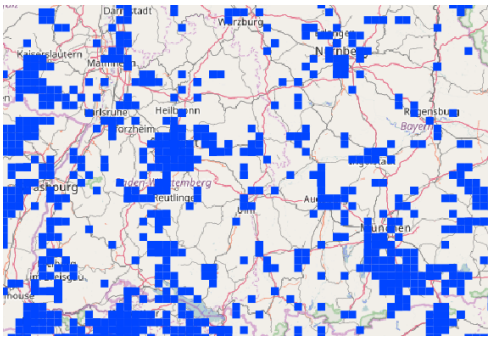
(a) Zoomstufe 12, 05:00–11:00



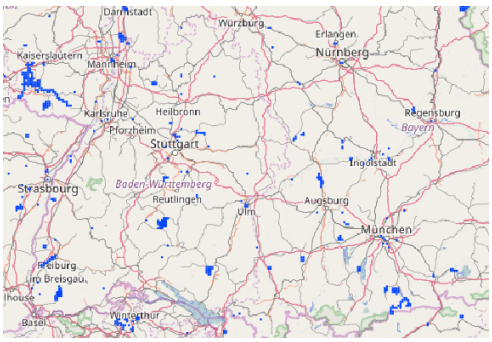
(b) Zoomstufe 12, 05:00–17:00



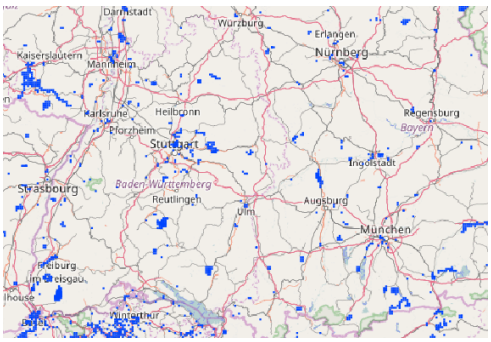
(c) Zoomstufe 12, 05:00–23:00



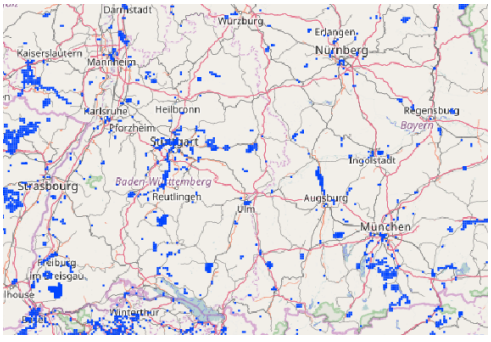
(d) Zoomstufe 12, 05:00–05:00



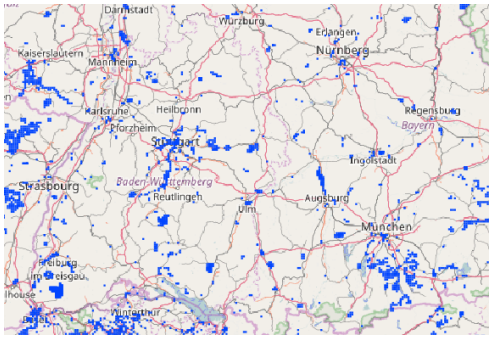
(e) Zoomstufe 14, 05:00–11:00



(f) Zoomstufe 14, 05:00–17:00



(g) Zoomstufe 14, 05:00–23:00



(h) Zoomstufe 14, 05:00–05:00

Abbildung 4.7.: zeitliche Entwicklung der Tile-Expiry (Relationen nicht berücksichtigt) zwischem dem 5. 9. 2016 05:00:01 und 6. 9. 2016 05:00:00 UTC

noch höheren Zoomstufe erfolgt, wird derselbe Way mehrfach prozessiert, da die Tiles Ways stets vollständig enthalten (Begründung siehe Seite 66). Neben der mehrfachen Prozessierung geht noch ein wenig Zeit für das Starten des Jobs verloren. Je weniger Daten in einem Tile enthalten sind, desto schneller ist seine Prozessierung und desto höher ist der Anteil des Overheads (Abfragen eines neuen Jobs, Schreiben der Ergebnisse in die Renderingdatenbank).

Aufgrund der auf den vorhergehenden Seiten dokumentierten Erkenntnissen fiel die Entscheidung, die Zoomstufe 9 als Baiss zu verwenden. Wählt man eine zu hohe Zoomstufe, sind die kleinsten Tiles zu klein. Als Schwellenwert wurde 5000 gewählt. In typischen mitteleuropäischen Städten haben bei geschlossener Bebauung, wenn alle Gebäude gemappt sind, Tiles auf Zoomstufe 15 etwa 10 000 Nodes. Solche Tiles als Vektortile zu erzeugen, kann wegen der vielen Daten zwei bis fünf Sekunden dauern. Tiles mit weniger Nodes beschleunigen daher nicht nur die Prozessierung, sondern auch die Erstellung der Vektortiles. Wenn ein Tile mehr als 5000 Nodes hat, wird es, wie in Kapitel 4.1 beschrieben, so oft aufgeteilt, bis die Anzahl der Nodes unter dem Schwellenwert ist. Für den Schwellenwert wurde 5000 gewählt.

4.4. Erzeugung eines Vektortiles

Ein Vektortile muss alle Objekte enthalten, die sich im Gebiet des Tiles befinden. Bei der Erzeugung wird dies durch eine `ST_Intersects`-Bedingung sichergestellt. Es wären also die in Listing 4.7 aufgelisteten SQL-Abfragen erforderlich. min_lon, min_lat usw. sind hierbei Platzhalter für die zwei Koordinatenpaare der Bounding-Box.

Listing 4.7: SQL-Abfragen, die prinzipiell zur Erstellung eines Vektortiles notwendig wären

```
SELECT tags, osm_id, osm_uid, osm_version, osm_lastmodified, osm_changeset, ST_X(
    ↪geom), ST_Y(geom)
  FROM nodes
  WHERE ST_Intersects(geom, ST_MakeEnvelope(min_lon, min_lat, max_lon, max_lat,
      ↪4326));
SELECT tags, osm_id, osm_uid, osm_version, osm_lastmodified, osm_changeset,
    ↪way_nodes
  FROM ways
  WHERE ST_Intersects(geom, ST_MakeEnvelope(min_lon, min_lat, max_lon, max_lat,
      ↪4326));
SELECT tags, osm_id, osm_uid, osm_version, osm_lastmodified, osm_changeset,
    ↪member_ids, member_types, member_roles
  FROM relations
  WHERE ST_Intersects(geom_points, ST_MakeEnvelope(min_lon, min_lat, max_lon,
      ↪max_lat, 4326)) OR ST_Intersects(geom_lines, ST_MakeEnvelope(min_lon,
      ↪min_lat, max_lon, max_lat, 4326));
```

4.5. Herausforderungen bei der Partitionierung in kleine Teile

Die in diesem Abschnitt aufgezeigten Probleme und Lösungen sind unabhängig vom Partitionierungsverfahren, solange die einzelnen kleinen Teile, die aus der Partitionierung hervorgehen, klein sind (wenige hundert Meter oder wenige Kilometer Länge und Breite). Sie treten auch auf, wenn statt Tiles bzw. einem Quadtree-Schema in Dreiecksmaschen oder Sechsecke partitioniert werden würde, denn jede Partitionierung führt zu mehr oder minder kleinen Teilen, die Kanten haben.

Die SQL-Abfragen aus Listing 4.7 lösen das Problem nicht. Stattdessen tun sich neue Herausforderungen auf:

1. Eine räumliche Abfrage benötigt einen räumlichen Index. Andernfalls führt die Datenbank einen *sequential Scan* durch (auch als *Full Table Scan* bekannt) und liest die gesamten Inhalte der Datenbank ein. Mit allen Metadaten (ausgenommen Benutzernamen) und Indexen belegt die Datenbank der OSM-Daten derzeit etwa 800 GB. Für sequentielles Einlesen ist kein Datenbankmanagementsystem erforderlich, diesen Overhead (Import, Indexe, Platzverbrauch) könnte dann eingespart werden.

2. Ein räumlicher Index auf die Geometriespalte der Tabelle `untagged_nodes` ist nicht zu empfehlen. Seine Erstellung benötigt mehrere Stunden und zahlreiche Gigabyte Platz. Dass das Ziel eigentlich auch ohne einen räumlichen Index auf diese Tabelle erreicht werden kann, wird im weiteren Verlauf dieser Arbeit gezeigt.

3. Würden die Vektortiles ausschließlich mit den obigen Abfragen erzeugt werden, wären sie unvollständig. Das Vektortile soll jedoch alle Ways enthalten, welche die `ST_Intersects`-Bedingung erfüllen, d. h. die das Gebiet des Tiles schneiden oder vollständig in ihm liegen. Zum Verarbeiten eines Ways mit anderen Anwendungen müssen alle Nodes, die ein Way referenziert, verfügbar sein. Ways, deren Enden aus dem Tile herausragen, sind hier nicht das Problem. Der Way könnte jedoch auch in der Mitte aus dem Tile herausragen. Das hieße, Anfang und Ende des Ways befänden sich im Vektortile und diese beiden Nodes wären somit in der Ergebnismenge enthalten. Die Nodes dazwischen lägen jedoch außerhalb des Gebiets des Vektortiles und wären nicht im Vektortile (der Datei) enthalten. Würde ein Programm diesen Mangel einfach ignorieren und Anfang und Ende direkt verbinden, entstünde eine falsche Geometrie. Gerenderte Kartenkacheln würden falsche Straßen enthalten, Analysen wären genauso verfälscht (vgl. Abbildung 4.8). Deshalb müssen in den weiteren Schritten der Vektortileerzeugung die noch fehlenden Nodes abgefragt werden.

4. Das Problem der unerfüllten Referenzen besteht auch bei Relationen. Relationsmitglieder, die außerhalb des Tiles liegen, sind bei Verwendung der obigen SQL-Abfragen nicht im Tile enthalten. Bestimmte Analysen können damit nicht durchgeführt werden.

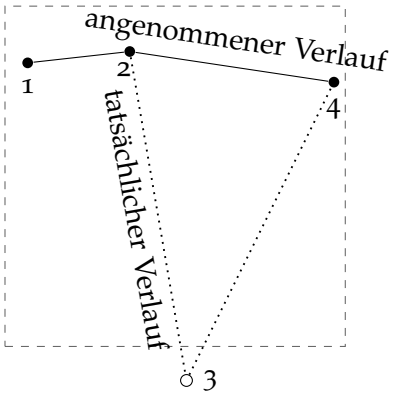


Abbildung 4.8.: Beispiel für die Bildung einer falschen Geometrie eines Ways, dessen Nodes nicht vollständig vorhanden sind. Node 3 liegt außerhalb des Tiles. Wird er verworfen, ist die resultierende Geometrie (durchgezogene Linie) falsch.

4.6. Kompromisse und Lösungen

4.6.1. Unvollständige Referenzen

Um das in Abschnitt 4.5, Punkt 3 beschriebene Problem der unerfüllten Referenzen zu lösen, müssen Listen (oder vergleichbare Datenstrukturen) gepflegt werden, mit denen verwaltet wird, welche Objekte in der Menge der Ergebnisse enthalten sind. Bei jedem Objekt, das Referenzen auf andere Objekte haben kann (das sind Ways und Relationen), muss geprüft werden, ob diese Objekte schon in der Menge der Ergebnisse enthalten sind oder ob sie noch abgefragt werden müssen. Die fehlenden Objekte werden anschließend anhand ihrer ID abgefragt. Deshalb muss es für jede Datenbanktabelle einen Index auf die Spalte `osm_id` geben.

4.6.2. Nodes ohne Tags

Für das Problem der verwaisten Nodes (Abschnitt 4.5, Punkt 1 und 2) existiert zwar keine perfekte Lösung, aber eine praktikable. Wenn keine räumliche Abfrage auf die Tabelle `untagged_nodes` ausgeführt werden, werden die Nodes, die von Ways referenziert werden, aber keine Tags haben, als „fehlend“ erkannt, weil sie fast alle von Ways referenziert werden. Bei der räumlichen Abfrage der Tabelle `ways` fällt das auf, sodass sie später einzeln anhand ihrer ID abgefragt werden. Das resultierende Vektortile ist genauso gut mit Drittanwendungen verarbeitbar.

Es existiert ein Nachteil. Nodes, die keine Tags haben und weder von einer Relation noch von einem Way referenziert werden, sind nicht im Vektortile enthalten. In den Vektortiles sind damit keine Suchen nach verwaisten Nodes möglich. Verwaiste Nodes richten keinen Schaden an. Sie sind einfach da, benötigen ein paar Byte Speicherplatz und stören höchstens Mapper, die gerade dieses Gebiet in ihrem Editor geladen haben. Da sie anders als etwa ein invalides Multipolygon keinen „Schaden“ anrichten, ist beim Beheben dieses „Fehlers“ keine Eile geboten. Wenn alle paar Tage auf die klassische

Art und Weise (ein Programm, das das gesamte Planetfile oder Extrakte davon einliest), nach verwaisten Nodes gesucht wird, genügt das.

Verwaiste Nodes können ein Indiz für einen Änderungssatz sein, der fragwürdig ist und genauer untersucht werden sollte. Das können Importe sein, die gegen die einschlägigen Community-Richtlinien verstoßen [19, 55], oder der Beitrag eines unerfahrenen Benutzers. Solche Änderungssätze fallen, der Erfahrung des Autors zufolge, meist auch anderweitig auf. Seien es Fehler anderer Art, wie invalide Multipolygone, Routingfehler usw., oder durch die Beobachtung der Editieraktivität durch andere Mapper.

4.6.3. Performanceeinbußen durch vollständige Relationen

Der in Punkt 4 geschilderte Mangel ist je nach Sichtweise kein Mangel, sondern ein Vorteil. Wenn ein Vektortile alle Mitglieder der Relationen enthält, die mindestens einen Node oder Way referenzieren, der im Gebiet des Tiles liegt, führt das zu einer fragwürdigen Datenduplizierung. Ein Vektortile am Frankfurter Hauptbahnhof enthielte dann alle Mitglieder aller Routenrelationen aller Linien des Fernverkehrs, die über Frankfurt Hauptbahnhof führen. Es wären somit alle Gleise von Zürich bis Berlin und Zagreb bis Amsterdam enthalten. Bei den Vektortiles entlang der Strecken wäre es ähnlich. Bei Tiles an den Grenzen von Bundesländern und Staaten bestünde ein ähnliches Problem. Tiles an der deutsch-französischen Grenze enthielten die Grenze von Deutschland und Frankreich (inklusive Übersee-Departements) und bei jeder Änderung in diesem grenznahen Tile würden ganz Frankreich und Deutschland geprüft werden.

Viele, wenn auch nicht alle Analysen an Relationen können auch durchgeführt werden, ohne dass alle Referenzen befriedigt sind. Das wird in Kapitel 4.7 gezeigt.

4.7. Validierung unvollständiger Relationen

4.7.1. Multipolygone

Prinzipiell lassen sich auch mit insgesamt unvollständigen, aber lokal (im Gebiet des Tiles) vollständigen Relationen vom Typ Multipolygon fast alle nötigen Qualitätssicherungsanalysen durchführen:

- Wenn sich zwei Ways kreuzen und der Schnittpunkt im Gebiet des Tiles liegt, sind beide Ways im Vektortile vollständig enthalten, da die `ST_Intersects`-Funktion true zurückgibt (vgl. Abbildung 4.9).
- Wenn an einer Position zwei Nodes sind (Koordinaten identisch, aber ID verschieden) und die Position im Gebiet des Tiles liegt, sind beide Nodes im Vektortile enthalten.
- Wenn ein äußerer und ein innerer Ring sich berühren und die Berührung nicht tangential ist, ist es ein Fehler. Wenn der Abschnitt, auf dem der sowohl zum inneren als auch zum äußeren Ring gehört, ganz oder teilweise im Gebiet des Tiles liegt, sind alle beteiligen Ways vollständig im Tile enthalten (vgl. Abbildung 4.10).

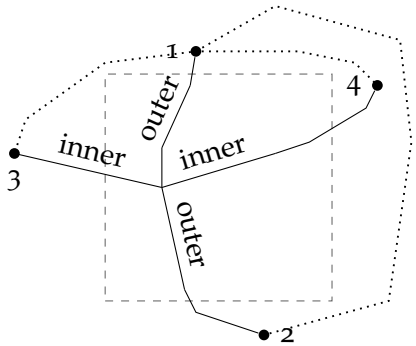


Abbildung 4.9.: Sich überkreuzende Ringe eines Multipolygons, deren Schnittpunkt im Vektortile liegt, weshalb sowohl Way 1–2 als auch Way 3–4 im Vektortile enthalten sind. Gepunktete Ways sind nicht im Vektortile enthalten, die Grenze des Tiles inkl. Buffer ist gestrichelt dargestellt.

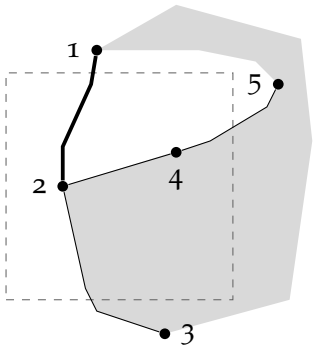


Abbildung 4.10.: Überlappung eines inneren und äußeren Rings eines Multipolygons am Rande eines Tiles. Wenn der Way 1–2 zweimal existiert oder zweimal Mitglied der Relation ist, wird der Fehler erkannt. Wenn im Abschnitt 1–2 nur ein Way gemappt ist und dieser nur einmal Mitglied der Relation ist, ist einer der Ringe nicht geschlossen, was ebenfalls ein Fehler ist. Die Grenze des Tiles inkl. Buffer ist gestrichelt dargestellt.

- Wenn ein innerer oder äußerer Ring eine Lücke hat, kann das auch erkannt werden. Dazu müssen aus allen im Vektortile enthaltenen Ways der Relation möglichst lange Linienzüge gebildet werden. Wenn die Lücke eines Rings in dem Gebiet des Tiles liegt, liegt ein Ende eines der gebildeten Linienzüge im Gebiet des Tiles. Enden außerhalb des Gebiets des Tiles sind normal und nicht zwangsläufig ein Fehler. Ways, die vollständig außerhalb des Gebiets des Tiles liegen, sind nämlich aufgrund der Abfragebedingung nicht im Vektortile enthalten. (vgl. Abbildung 4.11)

4.7.2. Administrative Grenzen

Da administrative Grenzen denselben Regeln wie Multipolygone folgen, lassen sich die meisten Analysen auch für sie durchführen. Eine Prüfung ist jedoch nicht durchführbar: Ein administratives Polygon A mit Admin-Level n muss alle Gebiete mit Admin-Level

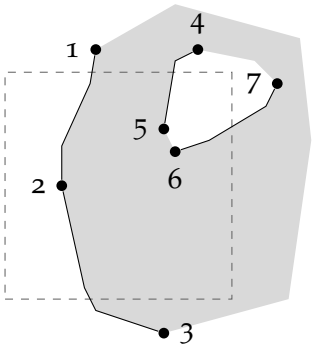


Abbildung 4.11.: Nicht geschlossener innerer Ring eines Multipolygons. Der innere Ring ist zwischen den Nodes 5 und 6 nicht geschlossen (der Way zwischen 4 und 7 liegt außerhalb des Tiles und ist daher nicht in dem Datensatz enthalten). Die Grenze des Tiles inkl. Buffer ist gestrichelt dargestellt.

$> n$, die mit A eine gemeinsame Fläche haben, vollständig umgeben.[3] Das heißt, ein Landkreis muss vollständig innerhalb eines Bundeslandes liegen. Diese Bedingung kann mit Vektortiles, in denen die Relationen unvollständig enthalten sind, nicht überprüft werden. Selbst wenn die Mitglieder der Grenzrelation Rollen (*outer* oder *inner*) hätten, kann nicht ermittelt werden, ob eine Seite eines Rings innen oder außen ist, solange der Ring nicht vollständig im Multipolygon enthalten ist.

Abbildung 4.12 zeigt das anhand eines Beispiels. Seien A und B zwei administrative Grenzen und das Admin-Level von B sei größer als das Admin-Level von A, dann muss B vollständig in A liegen. Vektortiles müssen als atomare Einheit prozessierbar sein, ihre Prozessierung darf folglicherweise keine zusätzlichen Informationen über die Umgebung erfordern.

Die Bedingung der vollständigen Bedeckung ist immer dann verletzt, wenn sich zwei Ringe (eigentlich zwei Ringstücke, da die Relationen nicht vollständig im Tile enthalten sind) A und B kreuzen und die Schnittmenge der beiden Ringstücke vom Typ Point oder MultiPoint ist. Dies ist mit einem Vektortile-basiertem Ansatz detektierbar. Für die Begründung sei auf den vorhergehenden Abschnitt zu Multipolygonen verwiesen.

Weitaus häufiger ist jedoch der Fall, dass die Schnittmenge zweier Ringstücke verschiedener administrativer Grenzrelationen vom Typ Linestring ist. In den allermeisten Fällen ist es kein Fehler, sondern die valide Realität. Das tritt bei administrativen Polygonen auf, die am Rand des Polygons eines höheren Admin-Levels liegen (z.B. Rheinland-Pfalz in Deutschland). Hier muss der Verlauf der beiden Ringe zueinander vor und nach dem gemeinsamen Abschnitt (der Schnittmenge vom Type LineString) untersucht werden. Abbildung 4.12 zeigt links den Fall, dass der gemeinsame Abschnitt 2–3 vollständig in einem einzigen Vektortile liegt. Wenn der zu A gehörige Abschnitt 1–2 links[4] des zu B gehörigen Abschnitt 4–2 liegt, aber der zu A gehörige Abschnitt 3–6 *rechts* des zu B gehörenden Abschnitts 3–5 liegt, ist die Bedingung der vollständigen Überdeckung

[3] Es sei darauf hingewiesen, dass das keine offizielle und dokumentierte Anforderung ist (vgl. Seite 15).

[4] In einem Vorverarbeitungsschritt müssen bei der Prozessierung des Tiles alle Ringstücke aller administrativen Grenzen gerichtet werden.

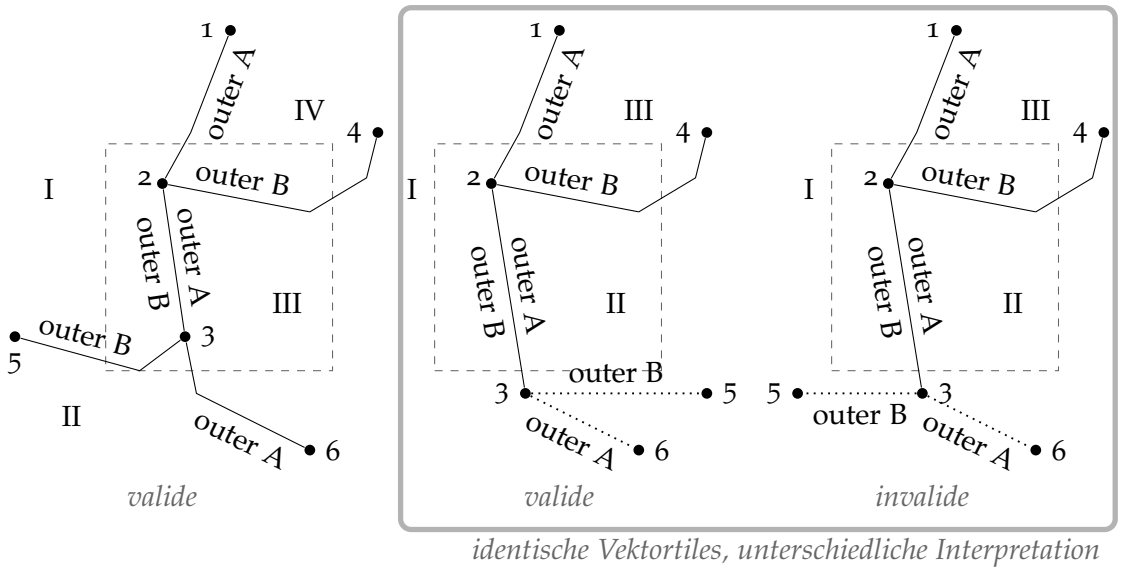


Abbildung 4.12.: Unmöglichkeit der Suche von Verstößen gegen die Hierarchie-Regel zweier adminstrativer Grenzen, wenn die Schnittmenge zweier ihrer Ringe ein LineString ist

verletzt. Multipolygon- und Grenzrelationen haben in OpenStreetMap keine vorgeschriebene Drehrichtung. Das Innere von A kann also entweder die Sektoren I und II oder die Sektoren III und IV sein. Auf welcher der beiden Seiten das Innere von A liegt, ist egal. In beiden Fällen umgibt das Innere von A nicht das Innere von B. Dieses ist nämlich entweder I+IV oder II+III.

Die Welt ist leider nicht so einfach. In vielen Fällen wird der gemeinsame Abschnitt aus dem Vektortile herausragen (Abbildung 4.12 Mitte und rechts). Die Koordinaten von Node 3 werden zwar bekannt sein, da die Vektortiles alle Ways vollständig enthalten, die das Tile schneiden. Es ist beim Prozessieren des Tiles jedoch unbekannt, ob 3–5 rechts oder links von 3–6 liegt. Läge Node 2 auch außerhalb des Tiles, wäre zwar die gemeinsame Strecke von A und B bekannt, aber nicht deren relativer Verlauf zueinander außerhalb der gemeinsamen Strecke. Es kann nicht einmal angenommen werden, dass 2 das Ende der gemeinsamen Strecke ist!

Die vollständige Aufnahme von Grenzrelationen in die Vektortiles ist keine Lösung. Sowohl die Erstellung der Vektortiles als auch die Prozessierung der Vektortiles, die entlang von Grenzen liegen, würde erheblich länger dauern. Jedes Tile an der Grenze Russlands müsste die gesamte russische Grenze verarbeiten.

4.7.3. Routenrelationen

Routenrelationen (vgl. Seite 16) lassen sich mit atomaren Vektortiles ebenfalls auf Validität prüfen. Routenrelationen, die dem alten Schema folgen, d. h. Routenrelationen von Wander- und Radwegen, sowie Fernstraßen, lassen sich kaum mit sich selbst validieren, da es kaum Anforderungen an ihre Gültigkeit gibt. Einzig und allein die Prüfung, ob ein Way zweimal in der Relation enthalten ist, ist möglich.

Routenrelationen, die dem PTv2-Schema (vgl. Kapitel 1.2.5 und Anhang A) folgen, lassen sich nur sehr eingeschränkt validieren. Nach Lücken kann gesucht werden, denn für die Suche nach Lücken gilt dasselbe wie für Multipolygone.

Ob die Referenzen auf die Ways des Fahrwegs in der richtigen Reihenfolge aufgeführt sind, kann jedoch nur in einem Teil der Fälle geprüft werden. Wenn zwei Haltestellen und die gesamte Strecke dazwischen in einem Vektortile liegen, ist diese Prüfung möglich. Beide Haltestellen und die Strecke dazwischen sind dann vollständig im Tile enthalten. Liegt ein Teil der Strecke oder – was viel häufiger auftritt – mindestens eine Haltestelle außerhalb des Tiles, ist diese Prüfung nicht mehr möglich. Der Prüfalgorithmus kann dann nicht mehr ermitteln, in welcher Reihenfolge die Ways befahren werden sollten, da Start- oder Endpunkt unbekannt sind.

Neben der Lückendetektion bleibt nur eine einfache Prüfung der Mitgliederliste möglich, für die die referenzierten Objekte nicht im Tile enthalten sein müssen. Es kann geprüft werden, ob alle Haltstellen am Anfang der Mitgliederliste aufgeführt sind und ob danach die Ways folgen.

Wenn der Inhalt eines Vektortiles jedoch ein wenig erweitert ist, ist eine Prüfung der Reihenfolge eingeschränkt möglich. In vielen Fällen reicht das aus. Dazu müssen alle referenzierten Ways im Vektortile enthalten sein, nicht aber ihre Nodes. Wenn von den Haltestellen auch die Haltepositionen erfasst sind und von der Routenrelation referenziert werden, kann auch geprüft werden, ob der Fahrweg in der richtigen Richtung befahren wird. Haltepositionen werden per Definition vom Way des Fahrwegs referenziert. Die referenzierten Nodes des Ways, nicht aber ihre Koordinaten, sind bekannt.

4.7.4. Abbiegebeschränkungen

Für Abbiegebeschränkungen gibt es, wie auf Seite 17 beschrieben, Gültigkeitsvorschriften, denen eine Abbiegebeschränkung in OpenStreetMap entsprechen muss. Auch bei Vektortiles ohne vollständige Relationen lassen sich Abbiegebeschränkungen prüfen, wenn sie nur einen Via-Node haben. Wenn dieser Via-Node im Vektortile oder auf dessen Rand liegt, sind sowohl der From- als auch der To-Way im Tile enthalten, denn der Rand ist Teil des Tiles. Dann kann geprüft werden, ob die drei Mitglieder mit einander verbunden sind. Nur wenn der From- und To-Way in verschiedenen Tiles (inkl. Buffer) liegen, ist diese Prüfung nicht möglich. Dazu muss der Via-Way jedoch länger als die doppelte Breite eines Buffer sein. Solche Fälle sind äußerst selten und rechtfertigen keine Maßnahmen mit negativen Auswirkungen auf die Performance. Selbst wenn die Mitglieder nicht alle vollständig im Tile enthalten sind, lässt sich prüfen, ob die Anzahl, der From-, Via- und To-Mitglieder korrekt ist.

5. Cerepso2vt

Cerepso2vt ist die Komponente, welche aus der PostGIS-Datenbank mit den gesamten Daten eines OSM-Planetdumps die Vektortiles erzeugt und Prozessierungsjobs in die Datenbank einfügt. Das Programm hat zwei Betriebsmodi – entweder wird die ID des zu produzierenden Tiles als Kommandozeilenargument übergeben oder es wird Tile-Expiry-Logfile (siehe Abschnitts 3.5) eingelesen und die darin angegebenen Tiles erzeugt. Letzteres wird im Folgenden *Batch-Modus* genannt.

5.1. Programmablauf und Architektur

Abbildung 5.1 zeigt ein vereinfachtes UML-Klassendiagramm von Cerepso2vt. Für ein vollständiges UML-Diagramm sei auf Anhang E verwiesen.

Programmkonfiguration Nach dem Start des Programms werden zuerst die vom Benutzer übergebenen Kommandozeilenparameter eingelesen und in Form von je einer Instanz der Klassen `postgres_drivers::Config` und `VectortileGeneratorConfig` gespeichert. Die Informationen, die für die Verbindung mit der von Cerepso importierten Datenbank erforderlich sind, werden von `postgres_drivers::Config` verwaltet, weil diese Klasse von der Bibliothek *Cerepso-Postgres-Backend* bereitgestellt wird und dem Konstruktor ihrer Klasse `postgres_drivers::Table` übergeben wird.

Jedes Tile hat eine Bounding Box Noch während des Einlesens wird im Batch-Modus das Tile-Expiry-Logfile eingelesen. Dies geschieht mit der statischen Fabrikmethode [42] `BoundingBox::read_tiles_list(const char*)`, welche einen Vektor mit Instanzen der Klasse `BoundingBox` zurückgibt.

Die Klasse `BoundingBox` speichert Informationen über das geographische Gebiet, das ein einzelnes Vektortile abdeckt. Zwecks schnellen Zugriffs speichert sie sowohl die geographische Länge und Breite (WGS84) des Tiles als auch dessen ID. Andere Klassen besitzen Attribute, die eine Referenz auf `BoundingBox` sind.

Verbindung zur Datenbank der OpenStreetMap-Daten Anschließend werden die Spaltendefinitionen initialisiert, wozu die Klasse `postgres_drivers::Columns` verwendet wird. Um die Verbindung mit der Datenbank, das Senden der Abfragen und das Auswerten der Antworten der PostgreSQL-Datenbank auf die Abfragen kümmert sich die Klasse `OSMDataTable`. Von ihr werden vier Instanzen (für die vier Datenbanktabellen) instanziiert. Diese Klasse ist von `postgres_drivers::Table` der Bibliothek *Cerepso-Postgres-Backend* abgeleitet und ergänzt sie um Attribute für die Koordinaten der Bounding

Box, die das Gebiet des Tiles beschreibt, und um eine Methode zur Ausführung von Prepared Statements, die alle Objekte eines Typs im Gebiet des Vektortiles abfragen. Die Koordinaten werden als ein Array von C-Strings gespeichert, da die Bibliothek *libpq* sie bei der Ausführung der Datenbankabfragen als ein Array an C-Strings erwartet. Für jede Datenbanktabelle existiert eine Instanz von `OSMDataTable`. Im Batch-Modus werden die Instanzen für alle Tiles wiederverwendet und besitzen deshalb eine Methode `set_bbox(BoundingBox&)`, mit der für jedes zu produzierende Vektortile die Bounding-Box geändert werden kann.

Die Erzeugung des Vektortiles als eine Strategie Nun kann eine Instanz der Klasse `OSMVectorTileImpl` erzeugt werden. Um Cerepso2vt an dieser Stelle sehr flexibel zu halten, wird das *Strategy*-Pattern [42] eingesetzt. Die Implementierung eines Ausgabeformats ist die Strategie. Es soll in Zukunft möglich sein, mit überschaubarem Aufwand weitere Ausgabeformate zu implementieren, die nichts direkt mit OSM als Ausgabeformat zu tun haben und/oder nicht von *Osmium* unterstützt werden. Es gibt zwar schon eine Reihe an Programmen zur Erzeugung von Vektortiles, aber alle haben ihre Vor- und Nachteile [80].

Falls Prozessierungs-Jobs in die Datenbank geschrieben werden sollen, wird eine Instanz der Klasse `JobsDatabase` erzeugt, welche Methoden zum Anlegen und Löschen von Jobs bereitstellt und, wie die Bibliothek Cerepso-Postgres-Backend, als Wrapper um die C-Bibliothek *libpq* fungiert. Da sich die Aufgaben von `OSMDataTable` und `JobsDatabase` erheblich unterscheiden, erben sie nicht beide von einer Klasse. Die Code-Duplizierung beschränkt sich nämlich auf die üblichen libpq-Aufrufe, wie etwa dem Herstellen der Verbindung zur Datenbank mit `PQconnectdb(const char*)`, dem Anlegen einiger Prepared Statements, dem Ausführen dieser Prepared Statements und dem Schließen der Verbindung mit den entsprechenden libpq-Funktionen.

Nachdem all diese Vorbereitungen abgeschlossen sind, iteriert der Cerepso2vt über den Vektor, der die Instanzen der Klasse `BoundingBox` enthält. Für jedes Element instanziiert er ein Objekt der Klasse `VectorTile`. `VectorTile` ist ein Klassentemplate, dessen Argument (im Folgenden `TVectorTileImpl` die zu verwendende Strategie (Implementierung eines Ausgabeformats) ist. `VectorTile` hat ein Attribut `m_implementation`, welches eine Referenz auf eine Instanz von `TVectorTileImpl` ist. `VectorTile` besitzt nur die öffentliche Methode `generate_vectortile()`, welche die gleichnamige Methode der Implementierung aufruft. Alle Implementierungen müssen diese Methode bereitstellen und in dieser Methode sämtliche nötigen Schritte durchführen, die zur Erzeugung eines Vektortiles im gewünschten Gebiet erforderlich sind (Datenbankabfragen, ggf. Vereinfachung, Objekte erzeugen, Datei schreiben).

Das Strategy-Pattern kommt hier nicht in der klassischen, von der *Gang of Four* beschriebenen Variante (siehe [42]) zum Einsatz. Stattdessen wird auf die Möglichkeiten der generischen Programmierung zurückgegriffen. Beim Kompilieren wird sichergestellt, dass die Strategie die erforderliche Methode `generate_vectortile()` bereitstellt. Andrei Alexandrescu beschreibt in seinem Buch *Modern C++ Design* ein Strategy-Pattern unter der Verwendung von Templates und nennt es *Policy-based Design*. Beide Patterns lassen

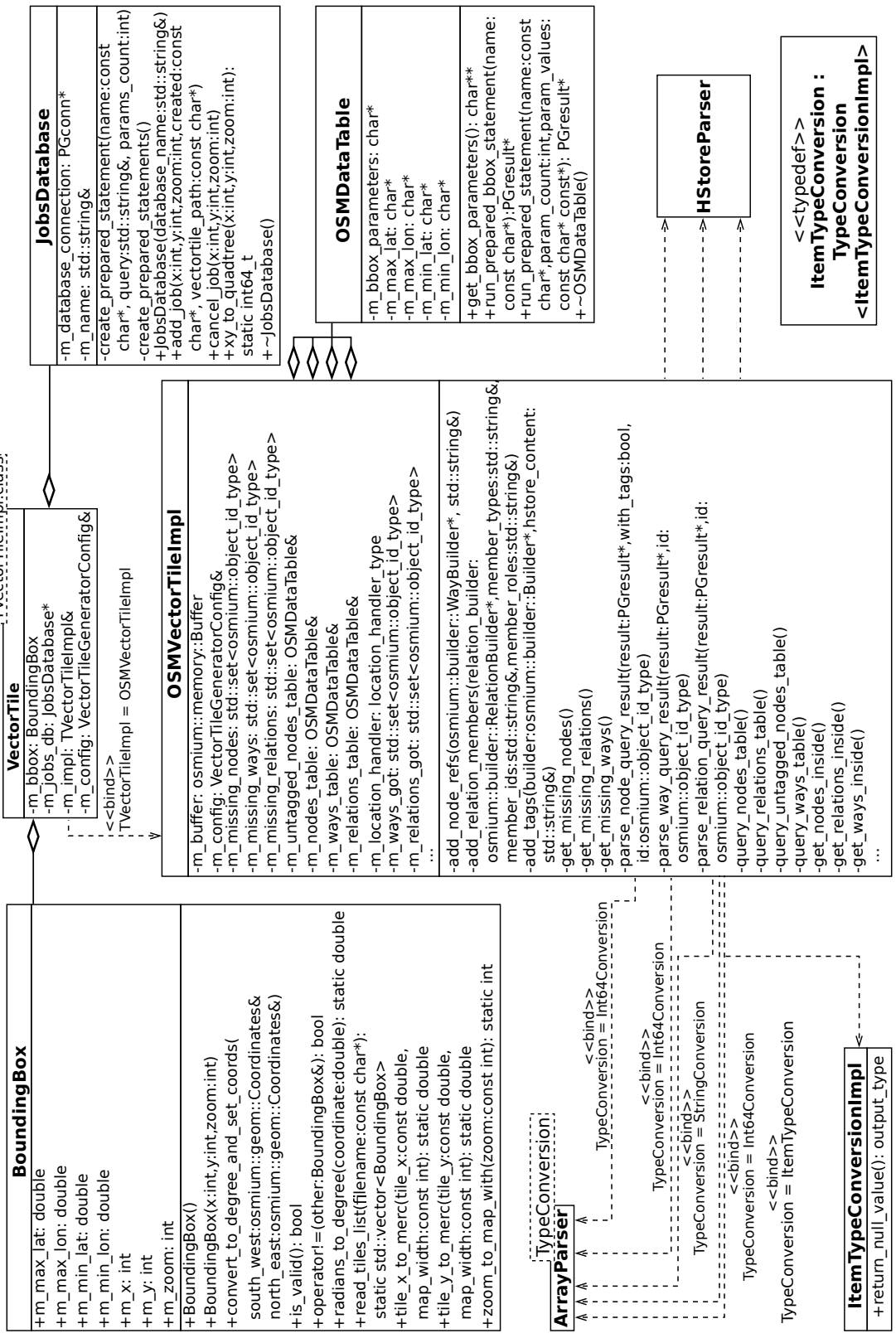


Abbildung 5.1.: UML-Klassendiagramm von Cerepso2vt. Klassen der Bibliothek Cerepso-Postgres-Backend sind nicht enthalten. Klassen der Bibliothek pg-array-hstore-parser sind nur ansatzweise enthalten. Für ein vollständiges UML-Klassendiagramm sei auf Anhang E verwiesen.

sich nicht klar trennen. Das hier verwendete Pattern ist aber keine artreine Anwendung des Policy-based Design, da die Templateklasse nicht von den Templateargumenten abgeleitet ist. Das ist nämlich nicht erforderlich. Würde die Strategie (Alexandrescu nennt sie *Policy*) ihre öffentlichen Methoden nicht bereitstellen und implementieren, würde der Code nicht kompilieren [13].

Für jede Instanz der Klasse VectorTile wird deren Methode generate_vectortile() aufgerufen, welche die entsprechenden Datenbankabfragen veranlasst und das Vektortile schreibt.

5.2. Die Erzeugung eines Vektortiles

Die Klasse VectorTile repräsentiert ein zu erzeugendes Vektortile. Neben dem Konstruktor gibt es nur eine öffentliche Methode, generate_vectortile(). Diese ruft nacheinander diverse private Methoden auf, welche Nodes, Ways und Relationen von der Datenbank abfragen und daraus mit der Osmium-Bibliothek OSM-Objekte erzeugen. Da bei Osmium alle Objekte in sogenannten *Buffern* gespeichert werden und nicht auf dem Stack erzeugt werden können, hat die Klasse VectorTile eine Instanz der Klasse osmium::memory::Buffer. Bevor VectorTile::generate_vectortile() aufgerufen wird, wird durch einen Aufruf von OSMVectorTileImpl::clear() sichergestellt, dass die Buffer leer sind und bei den vier Instanzen der Klasse OSMDataTable die richtige Bounding-Box gesetzt ist.

Die Erzeugung des Vektortiles in der Klasse OSMVectorTileImpl::generate_vectortile() läuft in folgenden Schritten ab:

1. Nodes, die im Gebiet des Vektortiles einschließlich eines Puffers um das Tile liegen, mit der SQL-Abfrage 2 aus Listing 4.7 abfragen. Alle Nodes, die von dieser Abfrage zurückgegeben werden, werden einem LocationHandler übergeben.

2. Selbiges für Ways (Abfrage 3 aus Listing 4.7). Für jeden Way, den diese Abfrage zurückgibt, wird geprüft, ob jeder Node, den die Ways referenzieren, schon von der Abfrage 1 zurückgegeben wurde und seine Position also bekannt ist. Wenn er im LocationHandler fehlt, wird er in ein Set m_missing_nodes eingefügt. Die IDs aller Ways, die von dieser Abfrage zurückgegeben werden, werden in ein Set m_ways_got eingefügt.

3. Selbiges für Relationen (Abfrage 4 aus Listing 4.7). Es wird anschließend geprüft, ob die Mitglieder dieser Relationen schon von den ersten beiden Abfragen zurückgegeben wurden. Dazu wird nach der ID im Set m_ways_got bzw. im LocationHandler gesucht. Falls die ID dort nicht gefunden wurde, wird die ID entweder in m_missing_relations, m_missing_ways oder m_missing_nodes eingefügt.

Die Abfragen werden als Prepared Statements ausgeführt. Bei diesen drei Abfragen ist der Performancevorteil minimal, aber es nützt der allgemeinen Struktur und Übersichtlichkeit des Programms, festgelegte Abfragen statt frei formulierter SQL-Abfragen zu verwenden. Für die Gründe, warum die unvollständigen Referenzen verwaltet werden müssen, sei auf Abschnitt 4.4 verwiesen.

Bei der Auswertung der oben genannten drei Abfragen werden die IDs aller Objekte, die in der Ergebnismenge enthalten sind, in ein Set des jeweiligen Objekttyps eingetragen. Folgende Sets werden dafür gepflegt:

- Nodes, die schon in der Ergebnismenge enthalten sind, werden vom Location-Handler verwaltet.
- IDs der Nodes, die noch fehlen, sind im Set m_missing_nodes enthalten.
- IDs der Ways, die in der Ergebnismenge enthalten sind, sind im Set m_ways_got enthalten.
- IDs der Ways, die noch fehlen, sind im Set m_missing_ways enthalten.
- IDs der Relationen, die in der Ergebnismenge enthalten sind, sind im Set m_relations _got enthalten.
- IDs der Relationen, die noch fehlen, sind im Set m_missing_relations enthalten.

Somit ergeben sich nacheinander folgende Schritte, um die Ergebnismenge zu vervollständigen:

1. Fehlende Relationen abfragen (d. h. solche, die von einer Relation aus der Ergebnismenge referenziert werden). In diesem Schritt kommen weitere fehlende Ways und Nodes in den Sets hinzu, die von den jetzt abgefragten Relationen direkt referenziert werden.
2. Fehlende Ways abfragen. In diesem Schritt kommen weitere fehlende Nodes im Set m_missing_nodes hinzu.
3. Fehlende Nodes abfragen.

Alle Objekte werden im selben Buffer angelegt. Die Antworten der Datenbank auf die räumlichen Abfragen sind jedoch nicht aufsteigend nach ID sortiert. Die Abfragen anhand der ID zur Vervollständigung der unvollständigen Referenzen sind genauso wenig nach ID sortiert. Deshalb muss der Buffer sortiert werden.

Osmium versucht möglichst selten Daten im Speicher zu verschieben oder zu kopieren. Deshalb wird eine osmium::ObjectPointerCollection angelegt (die Klasse wird von Osmium bereitgestellt). Eine osmium::ObjectPointerCollection ist eine Sammlung an Pointern auf Objekte in einem Buffer, welche von der Klasse osmium::handler::Handler abgeleitet ist. Das Sortieren von Pointern ist schneller als das Sortieren ganzer Objekte, da die Pointer weniger Speicher benötigen. Für das Sortieren wird ebenfalls eine von Osmium bereitgestellte Sortierfunktion verwendet. Zuletzt wird die osmium::Object PointerCollection mit std::copy(InputIterator, InputIterator, OutputIterator)[1] einem osmium::OutputIterator übergeben, welcher sie einer Instanz der Klasse osmium::io::Writer übergibt. std::copy(InputIterator, InputIterator, OutputIterator) bekommt zwei Iteratoren übergeben – einer ist der Anfang der ObjectPointerCollection, der andere das Ende, bis zum dem kopiert werden soll. Hier kommt das Iterator-Pattern [42] zum Einsatz, ein in der C++-Standardbibliothek und Osmium häufig anzutreffendes Pattern.

[1] InputIterator und OutputIterator sind Templateparameter und müssen diverse Methode, die Iterator haben, bereitstellen. Diese Funktion ist, wie viele Funktionen der C++-Standardbibliothek generisch.

5.3. Benutzung

Ein einzelnes Vektortile wird erzeugt, wenn Cerepso2vt wie folgt aufgerufen wird:
`$vectortile-generator [Optionen] X Y Z AUSGABEDATEI`
`X`, `Y` und `Z` sind dabei die drei Bestandteile der ID eines Tiles.
Wenn mehrere Vektortiles auf einmal erzeugt werden sollen, muss Cerepso2vt wie folgt aufgerufen werden:
`$vectortile-generator [Optionen] Tile-Expiry-Liste Ausgabeformat Ausgabeverzeichnis`

Dann werden aus der Datei *Tile-Expiry-Liste* die IDs der zu erzeugenden Tiles eingelesen und die Tiles im Format *Ausgabeformat* in das Verzeichnis *Ausgabeverzeichnis* geschrieben. Die Dateinamen sind hart kodiert und folgen dem Schema `Z_X_Y.suffix`. Dieser Modus wird im Folgenden *Batch-Modus* genannt.

Cerepso2vt hat folgende Optionen:

- `--help` gibt eine Hilfe aus und beendet das Programm.
- `--verbose` gibt die ID jedes Tiles aus, wenn es geschrieben wird
- `--database-name=DATABASE` sorgt dafür, dass die OSM-Daten aus der Datenbank DATENBANK gelesen werden
- `--jobs-database=DATABASE` aktiviert das Anlegen von Prozessierungsjobs in der Datenbank DATENBANK. Ohne dieses Argument werden keine Jobs angelegt.
- `--orphaned-nodes` Ohne dieses Argument wird das erzeugte Vektortile keine Nodes enthalten, die keine Tags haben und nicht von einem Way oder einer Relation referenziert werden. Wenn solche Nodes in den Vektortiles enthalten sein sollen, ist ein räumlicher Index auf die Datenbanktabelle `untagged_nodes` erforderlich. Für weitere Details dazu sei auf die Kapitel 4.5 und 4.6.2 verwiesen.
- `--recurse-relations` Wenn dieser Parameter übergeben wird, werden auch alle Relationen (ohne Mitglieder) in das Vektortile geschrieben, die von einer Relation referenziert werden, welche ein Way- oder Node-Mitglied im Gebiet des zu exportierenden Tiles hat.
- `--recurse-ways` Wenn dieser Parameter übergeben wird, werden auch alle Ways in das Vektortile geschrieben, welche außerhalb des Tiles liegen, aber von einer Relation referenziert werden, die ein Way- oder Node-Mitglied in dem Tile hat.
- `--recurse-nodes` Wenn dieser Parameter übergeben wird, werden auch alle Nodes in das Vektortile geschrieben, welche außerhalb des Tiles liegen aber von einer Relation referenziert werden, die ein Way- oder Node-Mitglied in dem Tile hat.
- `--force-overwrite` Wenn das Programm nicht im Batch Modus läuft, erlaubt dieser Parameter das Überschreiben der Ausgabedatei, falls sie schon existiert.

Als Ausgabeformate stehen OSM-XML, PBF und die anderen von Osmium unterstützen Formate zur Verfügung. Das Ausgabeformat wird im Nicht-Batch-Modus automatisch durch Osmium anhand der Dateiendung erkannt. Im Batch-Modus muss die zu verwendende Dateiendung (ohne Punkt davor) übergeben werden, Osmium wählt anhand dieser Endung dann das Ausgabeformat aus.

5.4. Hstore- und Array-Parser

Die Datenbankfelder, die vom Typ Hstore oder Array (irgendeines Datentyps) sind, werden von der Bibliothek libpq als C-Strings zurückgegeben. Die Bibliothek libpq bietet selbst keine Funktionen, um sie zu parsen und daraus entsprechende C++-Objekte zu erzeugen. Es gibt für andere Programmiersprachen zwar Bibliotheken, meist jedoch für Skriptsprachen wie Javascript [35], Ruby, PHP und Python. Die Mehrzahl davon verwendet reguläre Ausdrücke [26, 41, 49], was für die Geschwindigkeit nicht vorteilhaft ist. Die Python-Bibliothek pghstore von Hong Minhee [71] implementiert den Parser in C und ist somit nur ein Python-Binding für den C-Code. Es gäbe also eine in C geschriebene (und damit in C++ direkt nutzbare) Bibliothek, um Arrays zu parsen, aber kein Äquivalent zum Parsen von Hstore-Feldern. Um pghstore sinnvoll nutzen zu können, hätten die Python-spezifischen Code-Bestandteile entfernt werden müssen. Dennoch wäre das Ergebnis nicht besonders flexibel gewesen.

Die Implementierung eines Parsers für Hstore und Arrays wurde in eine separate Bibliothek ausgelagert, da es noch keine gibt, die den Vorstellungen des Autors bezüglich der Erweiterbarkeit (siehe folgende Absätze) entspricht.

5.4.1. PostgresParser als Basis des Array- und Hstore-Parsers

Der Array-Parser wird durch die Klasse `ArrayParser`, der Hstore-Parser durch die Klasse `HStoreParser` implementiert. Beide sind von `PostgresParser<typename TSingleElementType>` abgeleitet, weil sie Teile des öffentlichen Interfaces gemeinsam haben. Die Klasse `PostgresParser` hat mehrere virtuelle Methoden, welche von den beiden Klassen, die von ihr abgeleitet sind, erben. `PostgresParser` ist ein Klassentemplate, dessen Template-Argument der Ausgabetyp ist (bei Hstore String, bei Arrays der Datentyp des Arrays). Durch die Verwendung von Templates (auch als *Generische Programmierung* bekannt) wird die unnötige Duplizierung von Quellcode vermieden und die Klassen können in weiten Teilen implementiert werden, ohne ihre Typen zu kennen. Welche Typen verwendet werden, wird erst beim Kompilieren festgelegt.

Mit der Methoden `has_next()` kann abgefragt werden, ob es noch ein weiteres Element in der zu parsenden Struktur (Array bzw. Hstore) gibt. Ein Aufruf von `get_next()` gibt ein Element aus der zu parsenden Struktur (d. h. ein Element des Arrays bzw. ein Schlüssel-Wert-Paar) zurück. Der Rückgabewert der Methode `get_next()` ist vom Typ `TSingleElementType` – das ist das Templateargument der Klasse `PostgresParser<typename TSingleElementType>`.

Das Attribut `m_string_repr` ist eine Referenz auf den String, der die String-Repräsentation der zu parsenden Struktur enthält. Diese wird beim Aufruf des Konstruktors übergeben. `m_current_position` ist vom Typ `size_t` und speichert, bis zu welcher Position der String schon geparst ist.

Beide Parser gehen von links nach rechts durch den String. Beim Aufruf von `has_next()` prüfen sie, ob sie am Ende der String-Repräsentation angekommen sind. Falls ja, wird false zurückgegeben. Beim Aufruf von `get_next()` wird das nächste Element eingelesen – Zeichen für Zeichen, bis der nächste Elementtrenner erreicht wird.

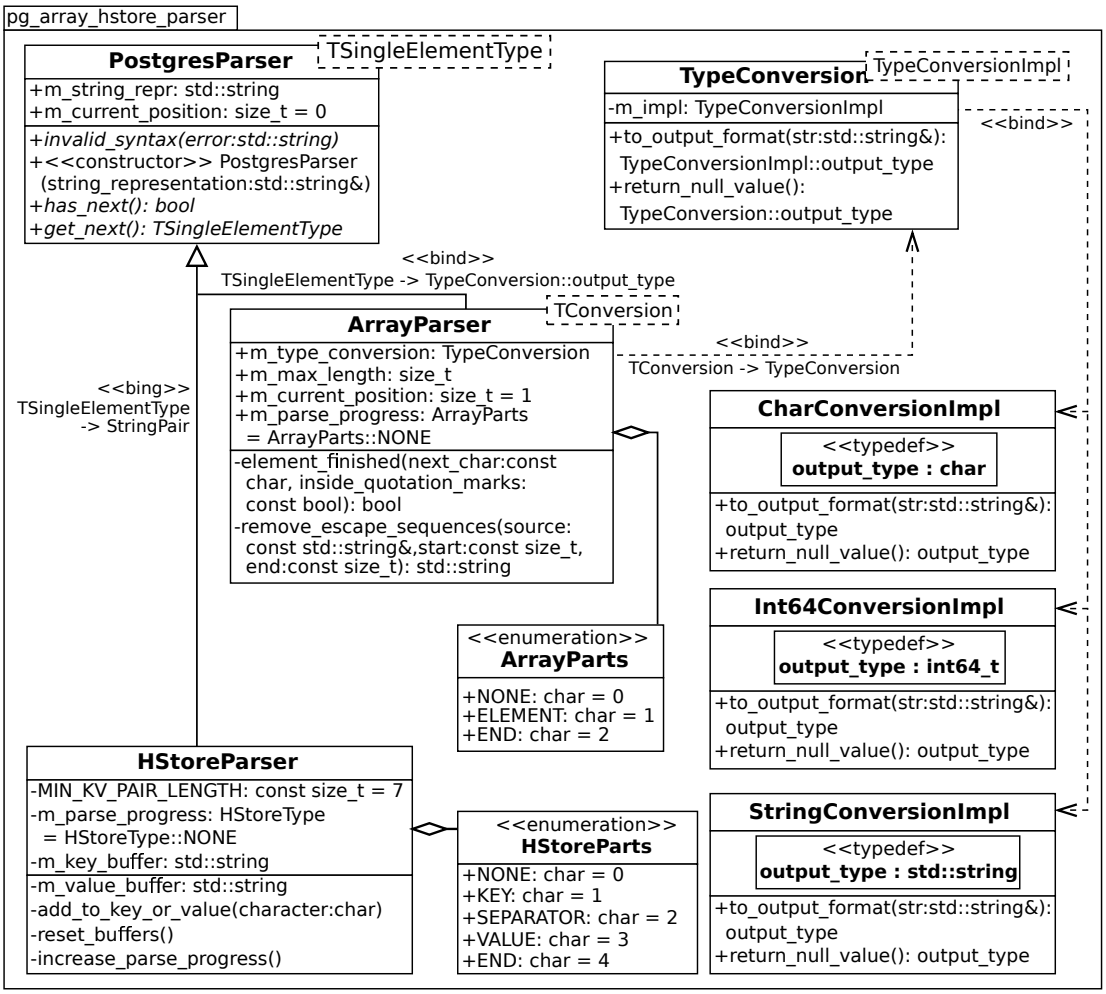


Abbildung 5.2.: UML-Klassendiagramm der Bibliothek pg-array-hstore-parser

5.4.2. Hstore-Parser

Die Klasse HStoreParser übergibt beim Erben von der Klasse PostgresParser die Klasse StringPair als Templateargument. StringPair ist ein Alias für std::pair<std::string, std::string>.

Das Attribut m_parse_progress vom Typ HStoreParts speichert, in welchem Teil der Hstore-Struktur sich der Parser gerade befindet. HStoreParts ist eine *Enumeration* mit fünf verschiedenen Werten:

- None (Parser befindet sich vor dem Beginn eines Schlüssels)
- Key (Parser befindet sich derzeit in einem Schlüssel)
- Separator (Parser befindet sich derzeit zwischen Key und Value)
- Value (Parser befindet sich derzeit in einem Wert)
- End (Parser befindet sich derzeit einem Wert und dem Ende bzw. dem nächsten Schlüssel)

Es genügt nicht, einfach nur nach dem Trennzeichen => zu suchen, denn Schlüssel und Werte können von Anführungszeichen umgeben sein, Anführungszeichen und Backslashes wird ein Backslash vorangestellt [99].

Zum Zwischenspeichern des Schlüssels und Werts, der beim Aufruf von get_next() zurückgegeben wird, hat die Klasse get_next() zwei private Attribute: m_key_buffer und m_value_buffer, beide vom Typ String.

5.4.3. Array-Parser

Der ArrayParser ist generisch implementiert (er ist ein Klassentemplate) und verwendet, wie auch schon die Klasse OSMVectorTileImpl das Policy-based Design, eine Variation des *Strategy-Pattern* unter Verwendung der Möglichkeiten der generischen Programmierung. Beim Strategy-Pattern wird einer Klasse eine Instanz einer Strategie übergeben. Das kann entweder im Konstruktor oder über eine separate Methode im Stil set_strategy(Abstract Strategy) geschehen. Alle Strategien müssen von einer abstrakten Strategie abgeleitet sein. Die Strategie ist in diesem Falle das Templateargument TConversion, das Konvertierungsverfahren, welches das Array-Element in den gewünschten Ausgabetyp (z. B. 64-Bit-Ganzzahl, String, Char, ...) umwandelt. TConversion selbst ist wiederum ein Klassentemplate, welches TypeConversionImpl als Templateparameter hat. TypeConversionImpl dient als Platzhalter für die konkrete Implementierung eines Konvertierungsverfahrens.

TypeConversion stellt eine mit to_output_format(std::string&) eine Methode zur Konvertierung von String in das gewünschte Ausgabeformat und ruft dazu die gleichnamige Methode der der konkreten Konvertierungsimplementierung (alias TypeConversionImpl) auf. Der Rückgabetyp von to_output_format(std::string&) ist TypeConversionImpl::output_type und wird somit von der verwendeten Implementierung festgelegt, was z. B. mit

```
using output_type = int64_t;
```

im Falle von int64_t geschieht.

Mit mehreren Typedefs wird diese geschachtelte Template-Struktur für den Nutzer einfacher gehalten, damit der Quellcode von Anwendungen, die den ArrayParser nutzen, nicht mit unnötigen vielen Templateargument unleserlich gemacht wird:

```
using StringConversion = TypeConversion<StringConversionImpl>;
using CharConversion = TypeConversion<CharConversionImpl>;
using Int64Conversion = TypeConversion<Int64ConversionImpl>;
```

Dadurch genügt ein

```
pg_array_hstore_parser::ArrayParser<Int64Conversion> my_array_parser (
    ↪array_string_repr);
```

zur Instanziierung eines Array-Parsers, der einen Array vom PostgreSQL-Typ Bigint parst.

Listing 5.1 enthält die für das Verständnis der Architektur wichtigsten Teile der Klasse ArrayParser. Listing 5.2 enthält die relevanten Teile der Klasse TypeConversion. Aus Platz-

Listing 5.1: Die wichtigsten Teile der Klasse ArrayParser

```
template <typename TConversion>
class ArrayParser : public PostgresParser<typename TConversion::output_type> {
protected:
  TConversion m_type_conversion;
  size_t m_max_length;
public:
  ArrayParser(std::string& string_repr) : PostgresParser<typename TypeConversion::
      ↪output_type>(string_repr) {
    m_max_length = string_repr.length();
  };

  // Has the parser reached the end of the hstore?
  bool has_next();

  // Return the next key value pair as a pair of strings.
  typename TConversion::output_type get_next() {
    std::string to_convert;
    // vollständige Implementierung siehe Anlage
    return m_type_conversion.to_output_format(to_convert);
  }
};
```

gründen sind die Dokumentation und weite Teile der Implementierung (teilweise sogar einige Attribute) nicht enthalten. Hierfür sei auf den beigelegten Quellcode verwiesen.

Mit Strategien, welche von einer abstrakten Strategie abgleitet sind und deren Methoden implementieren, kann hier nicht gearbeitet werden. Die abstrakte Strategie würde den Rückgabetyp der Methode to_output_format(std::string&) festlegen und der Bibliothek die gesamte Flexibilität entreißen. Deshalb wird auf Vererbung verzichtet, zur Kompilierzeit wird jedoch sichergestellt, dass die verwendeten Konvertierungsstrategien das erforderliche Interface haben.

Listing 5.2: Die Klasse TypeConversion

```
template <typename TypeConversionImpl>
class TypeConversion {
  TypeConversionImpl m_impl;

public:
  using output_type = typename TypeConversionImpl::output_type;

  output_type to_output_format(std::string& str) {
      return m_impl.to_output_format(str);
  }

  output_type return_null_value() {
          return m_impl.return_null_value();
  }
};
```

Derzeit sind folgende Konvertierungsimplementierungen verfügbar:

- TypeConversion<CharConversionImpl> alias CharConversion interpretiert das Array-Element als String und gibt das erste Zeichen als char zurück.
- TypeConversion<Int64ConversionImpl> alias Int64Conversion konvertiert mit std ::stoll den String in eine vorzeichenbehaftete 64-Bit Ganzzahl.
- TypeConversion<StringConversionImpl> alias StringConversion implementiert das Null Object Pattern und gibt den String unverändert als String zurück.

In Cerepso2vt selbst (nicht in der Bibliothek pg-array-hstore-parser) findet sich noch eine weitere Konvertierungsimplementierung, die TypeConversion<ItemTypeConversion-Impl> alias ItemTypeConversion, welche osmium::item_type zurückgibt. Sie betrachtet nur das erste Zeichen des Array-Elements. Ist es ein n, wird osmium::item_type::node zurückgegeben. Ist es ein w, wird osmium::item_type::way und, wenn es ein r ist, wird osmium::item_type::relation zurückgegeben. Da diese Implementierung Osmium als Abhängigkeit hat und nur für Cerepso2vt benötigt wird, ist sie nicht in der Bibliothek enthalten. Dieses Pattern wird auch bei der GeometryFactory von Osmium verwendet und ist ihr nachempfunden.

6. Ausblick

6.1. Cerepso

Cerepso kann in Zukunft zu einem vollwertigen Ersatz von *osm2pgsql* ausgebaut werden. Dafür fehlen ihm noch u. a. folgende Funktionen:

- Import in Datenbanken in anderen Projektionen als WGS84 (EPSG:4326). Für Renderingdatenbanken wird meist Web-Mercator (EPSG:3857) verwendet, da die gerenderten Tiles die Web-Mercator-Projektion nutzen und ansonsten unnötige Transformationen beim Rendern jedes Tiles erforderlich wären.
- Flächen werden derzeit noch nicht unterstützt. Als geschlossene Ways erfasste Flächen sind derzeit LineStrings, Multipolygon- und Grenzrelationen sind MultiLineStrings.
- Derzeit werden alle Tags in einer hstore-Spalte gehalten. osm2pgsql kann jedoch auch einzelne Spalten für einzelne Tags anlegen (die hstore-Unterstützung in osm2pgsql wurde erst später ergänzt).
- osm2pgsql kann bestimmte Tags verwerfen, sodass sie gar nicht erst in der Datenbank gehalten werden und Speicherplatz belegen. Dies gilt insbesondere für Fremdreferenzen, die bei Datenimporten aus Fremdquellen in OpenStreetMap als Tags eingefügt werden, z.B. `tiger:* = *`.
- Bei osm2pgsql kann der Nutzer darauf Einfluss nehmen, ob ein geschlossener Way als Fläche oder als Linie importiert wird. Dies sowie die Modifikation von Tags beim Import (TagFilter) fehlen in Cerepso.
- osm2pgsql hat neben der Klasse `output_pgsql_t` noch weitere Ausgabemöglichkeiten (Datenbankschemata), z. B. für den Geocoding-Dienst Nominatim.
- Statt die Nodes ohne Tags in einer PostgreSQL-Tabelle zu speichern, könnte man sie auch in einer Datei speichern. Bei einem Array-artigen Datenformat könnten dann auch große Extrakte und sogar der Planet mit nicht allzu viel Hauptspeicherbedarf importiert werden.

Es bleibt zu hoffen, dass sich die Community auch für den Importer, wenn sein Funktionsumfang zugenommen hat, zu interessieren beginnt. Neben Cerepso gibt es schon einen osm2pgsql-Konkurrenten – *imposm3*. imposm3 ist in der Programmiersprache *Go* geschrieben und speichert die für die Anwendung von Diffs notwendigen Informationen nicht in Datenbanktabellen, sondern, ähnlich wie die Flatnodes-Files von osm2pgsql

in Dateien. imposm3 hat ein zu osm2pgsql inkompatibles Datenbankschema, weshalb es kein Plug-in-Ersatz für osm2pgsql ist [56, 57]. Diesem Hindernis wurde bei der Entwicklung von Cerepso aus dem Weg gegangen; das Datenbankschema von Cerepso versucht möglichst kompatibel zu osm2pgsql zu sein. Schema-Kompatibilität ist nur eine Seite der Medaille. osm2pgsql wird auch zum Import der Datenbank von *Nominatim* verwendet. Selbst wenn Cerepso sich nicht durchsetzt, so werden sicherlich Teile seines Quellcodes und die ein oder andere Idee den Weg zu osm2pgsql finden.

6.2. Partitionierung

Das in dieser Arbeit entwickelte und implementierte Partitionierungsverfahren ist nur ein Anfang. Die verwendeten Parameter für die minimale und maximale Zoomstufe, auf denen partitioniert wird, sowie der Schwellenwert für die maximale Anzahl an Nodes pro Tile sind auf Basis von Gefühl und Erfahrungswerten geschätzt. Es bleibt zukünftigen Arbeiten überlassen, diese Parameter zu untersuchen und zu ermitteln, ob sie geeignet sind und welche Parameter optimal sind.

Das für das Partitionierungsverfahren verwendete Programm *dense_tiles* ist derzeit noch nicht so effizient, wie es sein könnte. Ein Durchlauf dauert etwa 15 Minuten, für jede Zoomstufe muss es neu gestartet werden. Könnte es alle Zoomstufen auf einmal ausrechnen, könnte man damit sechs Durchläufe (d. h. etwa 90 Minuten) sparen. Es wäre dann sogar denkbar, die Partitionierung alle ein bis zwei Tage neu auszurechnen, da die Neuberechnung nur noch 15 Minuten dauert.

6.3. Cerepso2vt

An den entscheidenden Stellen ist, wie im Kapitel 5.1 dokumentiert ist, *Cerepso2vt* dafür vorbereitet, weitere Ausgabeformate zu unterstützen. Das wohl interessanteste Format dürfte das *Mapbox-Vector-Tile*-Format sein, welches das populärste Format für Vektortiles im OpenStreetMap-Umfeld geworden ist.

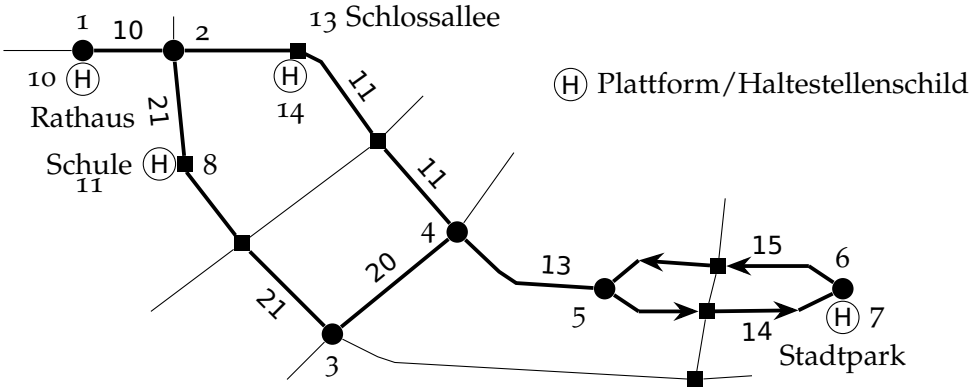
6.4. Prozessierungs- und Renderingserver

Die Komponenten für den Prozessierungs- und Renderingserver wurden im Rahmen dieser Arbeit nicht implementiert. Es handelt sich überwiegend nur um einige einfache Skripte, die in Python und/oder Bash implementiert worden wären. Die Arbeit hat sich stattdessen auf den Datenbankimporter Cerepso und den Cerepso2vt konzentriert.

In das Anpassen der Programme, die die Datenvalidierung durchführen, kann jedoch noch etwas „Gehirnschmalz“ fließen. So müssen beispielsweise Programme, die bislang Multipolygone prüfen, neu geschrieben werden. Schließlich stehen zu dem Zeitpunkt, zu dem ein Multipolygon geprüft wird, nur die Mitglieder zur Verfügung, die im Gebiet des Vektortiles liegen.

A. Beispiel einer Buslinie nach dem Public-Transport-Schema 2

Die Ways mit Pfeilspitzen werden nur in Pfeilrichtung befahren. Aus Platzgründen sind nur die Kurse vom Rathaus zum Stadtpark abgedruckt.




```
<relation id="1">
  <member type="node" ref="1" role="stop"/>
  <member type="node" ref="10" role="platform"/>
  <member type="node" ref="13" role="stop"/>
  <member type="node" ref="14" role="platform"/>
  <member type="node" ref="6" role="stop"/>
  <member type="node" ref="7" role="platform"/>
  <member type="way" ref="10" role=""/>
  <member type="way" ref="11" role=""/>
  <member type="way" ref="13" role=""/>
  <member type="way" ref="14" role=""/>
  <tag k="from" v="Rathaus"/>
  <tag k="ref" v="42"/>
  <tag k="route" v="bus"/>
  <tag k="to" v="Stadtpark"/>
  <tag k="type" v="route"/>
  <tag k="via" v="Schlossalee"/>
</relation>
<relation id="2">
  <member type="node" ref="1" role="stop"/>
  <member type="node" ref="10" role="platform"/>
  <member type="node" ref="8" role="stop"/>
  <member type="node" ref="11" role="platform"/>
  <member type="node" ref="6" role="stop"/>
  <member type="node" ref="7" role="platform"/>
```

```
  <member type="way" ref="10" role=""/>
  <member type="way" ref="21" role=""/>
  <member type="way" ref="20" role=""/>
  <member type="way" ref="13" role=""/>
  <member type="way" ref="14" role=""/>
  <tag k="from" v="Rathaus"/>
  <tag k="ref" v="42"/>
  <tag k="route" v="bus"/>
  <tag k="to" v="Stadtpark"/>
  <tag k="type" v="route"/>
  <tag k="via" v="Schule"/>
</relation>
```

B. Versionsgeschichte eines Ways, die verborgene Version enthält

Versionsgeschichte des Way 224314208 in de OSM-XML-Darstellung. Version 2 ist verborgen, weil sie urheberrechtlich geschützte Daten enthält, für deren Verbreitung keine Erlaubnis vorliegt. Die Version 3 wurde hochgeladen, um einen Ersatz für die Version 2 zu haben. Andernfalls würden Datennutzer, welche über die von OSM bereitgestellten Diffs ihre Datenbestände aktuell halten, weiterhin die unfreie Version 2 verwenden [9].

```
<osm version="0.6" generator="OpenStreetMap server" copyright="OpenStreetMap and
    ↪contributors" attribution="http://www.openstreetmap.org/copyright" license="
    ↪http://opendatacommons.org/licenses/odbl/1-0/">
  <way id="224314208" changeset="16414343" timestamp="2013-06-04T00:59:32Z" version
      ↪="1" visible="true" user="MrJott" uid="12165">
    <nd ref="2331336866"/>
    <nd ref="2331336852"/>
    <nd ref="2331336854"/>
    <nd ref="2331336870"/>
    <nd ref="2331336866"/>
    <tag k="building" v="yes"/>
  </way>
  <way id="224314208" changeset="22096782" timestamp="2014-05-03T03:38:28Z" version
      ↪="3" visible="true" user="pnorman redaction revert" uid="760215">
    <nd ref="2331336866"/>
    <nd ref="2331336852"/>
    <nd ref="2331336854"/>
    <nd ref="2331336870"/>
    <nd ref="2331336866"/>
    <tag k="building" v="yes"/>
  </way>
  <way id="224314208" changeset="38457115" timestamp="2016-04-10T18:00:35Z" version
      ↪="4" visible="true" user="catweazle67" uid="1976209">
    <nd ref="2331336866"/>
    <nd ref="2331336852"/>
    <nd ref="2331336854"/>
    <nd ref="2331336870"/>
    <nd ref="4112166152"/>
    <nd ref="4112166147"/>
    <nd ref="2331336866"/>
    <tag k="building" v="yes"/>
  </way>
</osm>
```

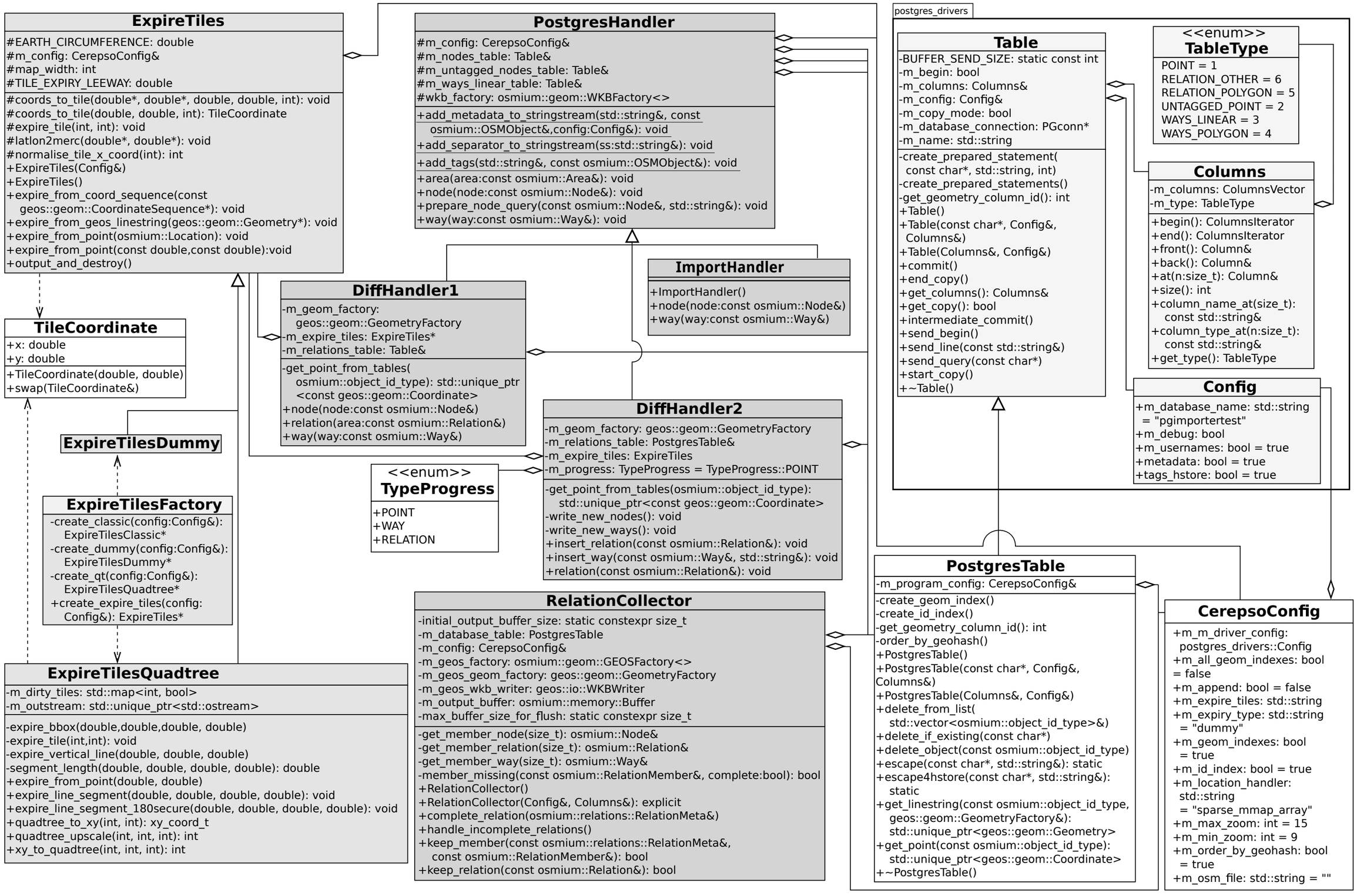
C. OpenStreetMap Change XML

OpenStreetMap Change XML am Beispiel des Änderungssatzes 43080643 [10] (gekürzt)

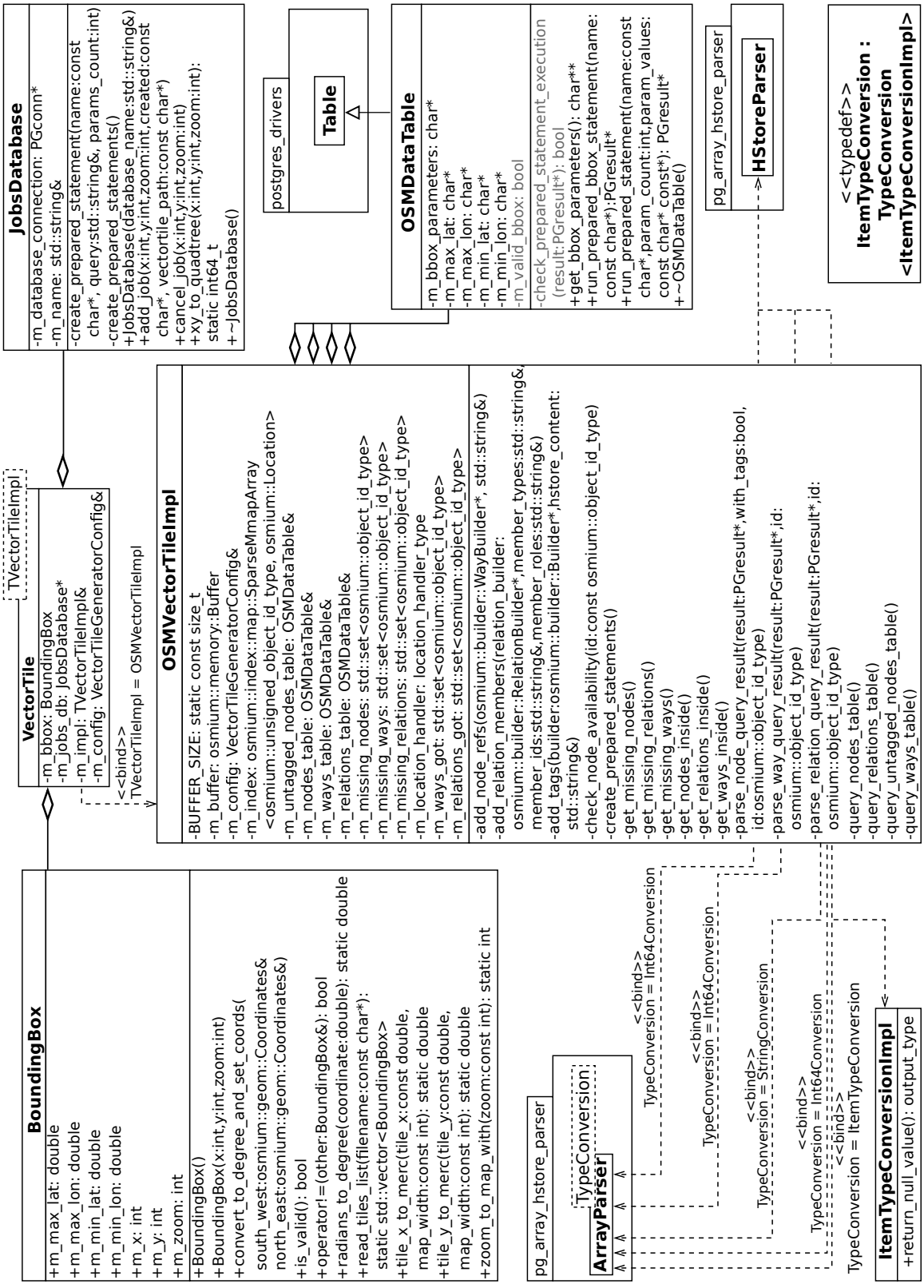
```
<osmChange version="0.6" generator="OpenStreetMap server" copyright="OpenStreetMap
    ↪and contributors" attribution="http://www.openstreetmap.org/copyright"
    ↪license="http://opendatacommons.org/licenses/odbl/1-0/">
  <create>
    <node id="4459088862" changeset="43080643" timestamp="2016-10-22T12:52:28Z"
        ↪version="1" visible="true" user="fred_ka" uid="190617" lat="49.4727659"
        ↪lon="8.4696162">
      <tag k="fax" v="0621 4406694"/>
      <tag k="health_facility:type" v="laboratory"/>
      <tag k="name" v="Julius Gehl Kieferorthopädisches Fachlabor"/>
      <tag k="phone" v="0621 4406687"/>
      <tag k="website" v="www.kfo-labor.com"/>
    </node>
  </create>
  <create>
    <node id="4459088863" changeset="43080643" timestamp="2016-10-22T12:52:28Z"
        ↪version="1" visible="true" user="fred_ka" uid="190617" lat="49.4728862"
        ↪lon="8.4694746">
      <tag k="amenity" v="cafe"/>
      <tag k="name" v="Perle"/>
      <tag k="opening_hours" v="Mo-Fr 10:30-18:30; Sa 10:00-19:00; Su 11:00-19:00"/
          ↪>
      <tag k="wheelchair" v="no"/>
    </node>
  </create>
  <modify>
    <node id="2527712159" changeset="43080643" timestamp="2016-10-22T12:52:28Z"
        ↪version="3" visible="true" user="fred_ka" uid="190617" lat="49.4726892"
        ↪lon="8.4695871">
      <tag k="amenity" v="dentist"/>
      <tag k="name" v="Dr. Carola Wißmeier"/>
      <tag k="opening_hours" v="Mo 08:15-18:00; Tu 08:15-18:00; We 08:15-13:00; Th
          ↪08:15-19:00; Fr 08:15-13:00"/>
      <tag k="phone" v="0621 - 87 20 792"/>
      <tag k="website" v="www.zahnarzt-mannheim.com"/>
      <tag k="wheelchair" v="yes"/>
    </node>
  </modify>
  <modify>
```

```
    <node id="2169148192" changeset="43080643" timestamp="2016-10-22T12:52:28Z"
        ↪version="4" visible="true" user="fred_ka" uid="190617" lat="49.4727536"
        ↪lon="8.4694667"/>
  </modify>
  <delete>
    <node id="1559413846" changeset="43080643" timestamp="2016-10-22T12:52:28Z"
        ↪version="7" visible="false" user="fred_ka" uid="190617"/>
  </delete>
  <delete>
    <node id="1559413844" changeset="43080643" timestamp="2016-10-22T12:52:28Z"
        ↪version="5" visible="false" user="fred_ka" uid="190617"/>
  </delete>
</osmChange>
```

D. Vollständiges UML-Diagramm von Cerepso und Cerepso-Postgres-Backend



E. Vollständiges UML-Diagramm von Cerepso2vt



F. expiries2shp

Das Hilfsprogramm expiries2shp ist ein relative einfaches, kleines C++-Programm, um Tile-Expiry-Logs zu visualisieren. Es erstellt Shapefiles, welche Polygone enthalten. Jedes Polygon ist ein Eintrag aus dem Tile-Expiry-Log.

Das Programm wurde unter https://github.com/Nakaner/expiries2shp unter den Bedingungen der GNU General Public License 3 oder neuer veröffentlicht.

F.1. Benutzung

`expiries2shp [`*`Optionen`*`] `*`Eingabedateien Augabedatei`*

`Eingabedateien` ist ein Wildcard-Suchmuster, das die einzulesenden Tile-Expiry-Listen umfasst. Das Programm kann mehrere Listen nach einander in einem Durchlauf einlesen und daraus ein gemeinsame Shapefile erstellen. Das ist deutlich schneller, als die mehrere Shapefiles zu erstellen und diese nacheinandern mit *ogr2ogr* [81] zu vereinigen, da das Öffnen eines Shapefiles mit der GDAL-Bibliothek einen Moment in Anspruch nimmt.

`Ausgabedatei` ist der Dateiname des zu erstellenden Shapefiles

Weitere Optionen:

- `--format=`*`FORMAT`* ändert das Ausgabeformat (standardmäßig „ESRI Shapefile“). Es muss die von GDAL verwendete Bezeichnung des Ausgabeformats verwendet werden.
- `--projection=`*`PROJEKTION`* verwendet *PROJEKTION* als räumliches Bezugssystem für die Ausgabe. *PROJEKTION* muss ein EPSG-Code sein. Standard ist 3857 (Web-Mercator).
- `--sequence=`*`NUMMER`* schreibt stets die Nummer *NUMMER* in die Spalte *sequence* des Shapefiles.
- `--ids` schreibt die x- und y-ID des Tiles in das Shapefile. Diese ergeben sich zwar aus den Koordinaten des Polygons, als separate Spalten ist der Zugriff darauf jedoch einfacher.
- `--verbose` gibt mehr Details zum Ablauf auf der Standardausgabe aus.
- `--min-zoom=`*`ZOOM`* legt die minimale Zoomstufe fest. Tiles mit einer niedrigeren Zoomstufe werden nicht in das Shapefile übernommen.
- `--max-zoom=`*`ZOOM`* dto., aber die maximale Zoomstufe

F.2. Anmerkungen zur Implementierung

Inklusive Kommentar- und Leerzeilen hat es nur 514 Zeilen. Es besitzt zwei Klassen, `OutputLayer` und `Tile`.

Die Klasse OutputLayer dient als *Wrapper* um die GDAL-Bibliothek und stellt Methoden zum Anlegen eines Layers undSchreiben von Daten in diesen bereit. Bei Shapefiles achtet sich darauf, dass diese nicht größer als zwei Gigabyte werden. Viele Programme können nicht mit größeren Shapefiles umgehen.

Die Klasse Tile stellt Methoden zur Konvertierung der Tile-ID (x, y und Zoom) in Web-Mercator-Koordinaten bereit. Die Methode get_square() ist neben dem Konstruktur die einzige öffentliche Methode. Sie gibt einen *Unique Pointer* auf eine Instanz von OGRPolygon zurück.

Alle übrige Funktionalität ist in Funktionen implementiert, die zu keiner Klasse gehören:

- Parsen der Tile-Expiry-Dateien mit split()
- Auflösen des Wildcard-Patterns in eine Liste von Dateinamen (Vektor vom Typ String)
- Kommandozeilenargumente einlesen
- Verzeichnis und Dateiname aus einem Pfad extrahieren

G. Beiträge zur Dokumentation von Osmium

Im Rahmen dieser Arbeiten wurden Lücken in der Dokumentation der Osmium-Bibliothek geschlossen und diese Lücken dem Maintainer Jochen Topf als Pull-Requests auf der Github-Plattform zur Übernahme angeboten. Es handelte sich um folgende Pull-Requests:

- *Explain how writing files works* (`https://github.com/osmcode/libosmium-manual/pull/2`), mit der Commit-ID: 5a820e1
- *Document how to write to a buffer* (`https://github.com/osmcode/libosmium-manual/pull/3`), mit der Commit-ID: 2749bed
- *Add documentation of handlers*, (`https://github.com/osmcode/libosmium-manual/pull/4`), mit der Commit-ID: 89c7ec4
- *Fix markup* (`https://github.com/osmcode/libosmium-manual/pull/5`), mit der Commit-ID: e9e3626
- *Document Collector and MultipolygonCollector* `https://github.com/osmcode/libosmium-manual/pull/7` mit den Commit-IDs f56347d, c7ecbd1 und 01c39b2
- *Explain MapFactory* (`https://github.com/osmcode/libosmium-manual/pull/8`), mit der Commit-ID: 085dca1. Dieses Pull-Request wurde nicht gemergt, sondern die Änderungen manuell in das Nachfolge-Repository (`https://github.com/osmcode/osmcode.github.io`) übernommen.

H. Technische Details zum Performance-Test von Cerepso

Für die Performancetests von Cerepso (Abbildung 3.2) wurde ein Server mit Ubuntu 16.04 und zwei Prozessoren vom Typ Intel Xeon E5-2623 v3 verwendet. Der Server verfügte über 256 GB RAM, der PostgreSQL-Tablespace befand sich auf SSDs.

Im Folgenden ist die Konfiguration (`/etc/postgresql/9.5/main/postgresql.conf`) von PostgreSQL 9.5, das beim Import verwendet wurde, abgedruckt. Alle nicht aufgelisteten Parameter haben die Standardeinstellungen. Die Konfiguration der Unix-Sockets, SSL-Zertifikate und des Datenverzeichnisses sind in diesem Auszug aus der Datenbankkonfiguration nicht enthalten.

```
shared_buffers = 10GB
temp_buffers = 100MB
work_mem = 1GB
maintenance_work_mem = 4GB
dynamic_shared_memory_type = posix
effective_io_concurrency = 6
fsync = off
checkpoint_timeout = 1h
log_min_duration_statement = 0
log_line_prefix = '%t [%p-%l] %q%u@%d '
log_timezone = 'localtime'
stats_temp_directory = '/var/run/postgresql/9.5-main.pg_stat_tmp'
autovacuum = off
datestyle = 'iso, mdy'
timezone = 'localtime'
lc_messages = 'en_US.UTF-8'
lc_monetary = 'en_US.UTF-8'
lc_numeric = 'en_US.UTF-8'
lc_time = 'en_US.UTF-8'
default_text_search_config = 'pg_catalog.english'
```

Literatur

[1] 8. Jan. 2012. URL: https://www.openstreetmap.org/api/0.6/node/306079948/history (besucht am 07. 11. 2016).

[2] 17. Juni 2014. URL: https://www.openstreetmap.org/api/0.6/way/148588529 (besucht am 14. 11. 2016).

[3] 9. Juni 2015. URL: https://www.openstreetmap.org/api/0.6/relation/5489215 (besucht am 14. 11. 2016).

[4] 14. Sep. 2016. URL: https://www.openstreetmap.org/api/0.6/node/4400868292 (besucht am 14. 11. 2016).

[5] 22. Okt. 2016. URL: https://www.openstreetmap.org/api/0.6/changeset/43080643 (besucht am 14. 11. 2016).

[6] 22. Aug. 2016. URL: http://planet.openstreetmap.org/planet/2016/changesets-160822.osm.bz2 (besucht am 25. 08. 2016).

[7] OpenStreetMap Foundation Operations Working Group. 20. Okt. 2016. URL: https://git.openstreetmap.org/chef.git/blob/HEAD:/cookbooks/tile/files/default/ruby/expire.rb (besucht am 15. 12. 2016).

[8] 28. Juli 2016. URL: https://git.openstreetmap.org/chef.git/blob/HEAD:/cookbooks/tile/README.md (besucht am 15. 12. 2016).

[9] 10. Apr. 2016. URL: https://www.openstreetmap.org/api/0.6/way/224314208/history (besucht am 16. 11. 2016).

[10] 22. Okt. 2016. URL: https://www.openstreetmap.org/api/0.6/changeset/43080643/download (besucht am 14. 11. 2016).

[11] URL: http://tools.geofabrik.de/calc/#type=geofabrik_german&grid=1 (besucht am 05. 10. 2016).

[12] *About The License Change*. OpenStreetMap Foundation. 2012. URL: https://wiki.osmfoundation.org/wiki/Licence/About_The_Licence_Change (besucht am 15. 11. 2016).

[13] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Boston: Addison-Wesley, 2001.

[14] Matt Amos. *Diff-Based Tile Expiry Script*. 28. März 2010. URL: http://svn.openstreetmap.org/applications/utils/export/tile_expiry/README (besucht am 10. 11. 2016).

[15] Matt Amos. *The Road to API 0.7*. 18. Nov. 2015. URL: http://www.asklater.com/matt/blog/2015/11/18/the-road-to-api-07/ (besucht am 16. 11. 2016).

[16] *Any tags you like*. 2008. URL: https://wiki.openstreetmap.org/wiki/Any_tags_you_like (besucht am 15. 11. 2016).

[17] *API v0.6*. URL: https://wiki.openstreetmap.org/wiki/API_v0.6 (besucht am 16. 11. 2016).

[18] *Area Workshop*. Vortrag auf der State of the Map 2016. Brüssel, 25. Sep. 2016. URL: https://media.jochentopf.com/media/2016-09-25-talk-sotm2016-area-workshop-en-slides.pdf (besucht am 14. 12. 2016).

[19] *Automatische Edits und Importe in OpenStreetMap. Das Gegenteil von gut ist gut gemeint*. Vortrag auf der FOSSGIS 2016. Salzburg, 4. Juli 2016. URL: http://ftp5.gwdg.de/pub/misc/openstreetmap/FOSSGIS2016/FOSSGIS2016-5096-automatische_edits_und_importe_in_openstreetmap-hd.webm (besucht am 07. 11. 2016).

[20] Sarah E Battersby u. a. »Implications of web Mercator and its use in online mapping«. In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 49.2 (2014), S. 85–101.

[21] Jonathan Bennett. *OpenStreetMap. Be your own Cartographer*. Birmingham: Packt Publishing, 2010.

[22] *Bing Maps*. URL: https://wiki.openstreetmap.org/wiki/Bing_Maps (besucht am 15. 11. 2016).

[23] brettch. 20. Nov. 2010. URL: https://github.com/openstreetmap/osmosis/blob/master/package/script/pgsimple_schema_0.6.sql (besucht am 19. 12. 2016).

[24] brettch und Paul Norman. 3. Juli 2013. URL: https://github.com/openstreetmap/osmosis/commits/master/package/script/pgsnapshot_schema_0.6.sql (besucht am 19. 12. 2016).

[25] Thomas Brinkhoff. *Geodatenbanksysteme in Theorie und Praxis. Einführung in objektrelationale Geodatenbanken unter besonderer Berücksichtigung von Oracle Spatial*. 2. Aufl. Heidelberg: Wichmann, 2008.

[26] chandeeland. *PHP-postgresql-array-parser*. URL: https://github.com/chandeeland/PHP-postgresql-array-parser (besucht am 05. 12. 2016).

[27] *Changeset*. 20. Sep. 2016. URL: https://wiki.openstreetmap.org/wiki/Changeset (besucht am 14. 11. 2016).

[28] *Command-line usage*. URL: https://github.com/openstreetmap/osm2pgsql/blob/master/docs/usage.md (besucht am 15. 08. 2016).

[29] *Commits jtv/libpqxx*. URL: https://github.com/jtv/libpqxx/commits/master (besucht am 30. 10. 2016).

[30] *Contributors to keepright/keepright*. Github. URL: https://github.com/keepright/keepright/graphs/contributors (besucht am 24. 08. 2016).

[31] *Data Verification Process*. 26. Feb. 2013. URL: https://help.openstreetmap.org/questions/20312/data-verification-process?page=1%5C&focusedAnswerId=20323#20323 (besucht am 01. 09. 2016).

[32] *Databases and data access APIs*. URL: https://wiki.openstreetmap.org/wiki/Databases_and_data_access_APIs (besucht am 19. 12. 2016).

[33] *DE:Public Transport*. 4. Sep. 2016. URL: https://wiki.openstreetmap.org/wiki/DE:Public_transport (besucht am 16. 11. 2016).

[34] *DE:Public Transport*. URL: http://wiki.openstreetmap.org/wiki/DE:Public_transport (besucht am 16. 01. 2017).

[35] Ben Drucker. *postgres-array*. URL: https://github.com/bendrucker/postgres-array (besucht am 05. 12. 2016).

[36] Oskar Eriksson und Emil Rydkvist. »An in-depth analysis of dynamically rendered vector-based maps with WebGL using Mapbox GL JS«. Masterthesis. 26. Aug. 2015. URL: http://www.diva-portal.org/smash/get/diva2:851452/FULLTEXT02.pdf (besucht am 19. 12. 2016).

[37] Martijn van Exel. *MapRoulette Next Generation. The Most Fun Way To Fix The Map Has Just Become A Lot Better*. 15. Juni 2016. URL: http://www.sotm-eu.org/slides/65.pdf (besucht am 29. 08. 2016).

[38] Martijn van Exel, Aaron Lidman u. a. *What is this for?* 2014. URL: https://github.com/osmlab/to-fix/issues/1 (besucht am 29. 08. 2016).

[39] Richard Fairhurst. *tilemaker*. URL: https://github.com/systemed/tilemaker (besucht am 10. 01. 2017).

[40] OpenStreetMap Foundation, Hrsg. URL: http://planet.openstreetmap.org/ (besucht am 20. 10. 2016).

[41] Matt Freeman. *node-postgres-hstore*. URL: https://github.com/sitepodmatt/node-postgres-hstore (besucht am 05. 12. 2016).

[42] Erich Gamma u. a. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Boston: Addison-Wesley, 2009.

[43] *GDAL: orgtindex*. URL: http://www.gdal.org/ogrtindex.html (besucht am 19. 12. 2016).

[44] *GNU coreutils: comm invocation*. URL: https://www.gnu.org/software/coreutils/manual/html_node/comm-invocation.html (besucht am 17. 01. 2017).

[45] *GNU coreutils: sort invocation*. URL: https://www.gnu.org/software/coreutils/manual/html_node/sort-invocation.html (besucht am 17. 01. 2017).

[46] Matt Greene. *Quality analysis for OpenStreetMap*. Mapbox Inc. 3. Sep. 2015. URL: https://www.mapbox.com/blog/osm-qa-tiles/ (besucht am 19. 12. 2016).

[47] Mordechai Haklay. »How good is volunteered geographical information? A comparative study of OpenStreetMap and Ordnance Survey datasets«. In: *Environment and planning B: Planning and design* 37.4 (2010), S. 682–703.

[48] Mordechai Haklay und Patrick Weber. »Openstreetmap: User-generated street maps«. In: *IEEE Pervasive Computing* 7.4 (2008), S. 12–18.

[49] Peter van Hardenberg und Seamus Abshere. *pg_hstore*. URL: https://github.com/seamusabshere/pg-hstore (besucht am 05. 12. 2016).

[50] Kevlin Henney. »Null object«. In: *Proceedings of the Seventh European Conference on Pattern Languages of Programming, EuroPLoP*. 2002. URL: http://hillside.net/europlop/HillsideEurope/Papers/EuroPLoP2002/2002_Henney_NullObject.pdf.

[51] John R. Herring, Hrsg. *OpenGIS Implementation Standard for Geographic information – Simple feature access*. Open Geospatial Consortium Inc. 28. Mai 2011. URL: http://portal.opengeospatial.org/files/?artifact_id=25355 (besucht am 14. 12. 2016).

[52] *Hilfe erhalten*. URL: https://www.openstreetmap.org/help (besucht am 20. 12. 2016).

[53] Sarah Hofmann. *Nominatim installation*. OpenStreetMap Foundation. URL: https://github.com/openstreetmap/Nominatim/blob/master/docs/Installation.md (besucht am 20. 08. 2016).

[54] HostGIS. *Tile Indexes – MapServer 7.0.3 Documentation*. URL: http://mapserver.org/optimization/tileindex.html (besucht am 19. 12. 2016).

[55] *Import/Guidelines*. URL: https://wiki.openstreetmap.org/wiki/Import/Guidelines (besucht am 18. 01. 2017).

[56] *imposm3*. Github-Repository. Omniscale. URL: https://github.com/omniscale/imposm3 (besucht am 17. 01. 2017).

[57] *Imposm3 3.0.0a documentation*. URL: https://imposm.org/docs/imposm3/latest/ (besucht am 19. 01. 2017).

[58] *Keep Right*. URL: https://wiki.openstreetmap.org/wiki/Keep_Right (besucht am 24. 08. 2016).

[59] *Keepright sucht neuen Maintainer*. 2. Okt. 2013. URL: http://forum.openstreetmap.org/viewtopic.php?id=22784 (besucht am 24. 08. 2016).

[60] Harald Klein und mueschel. *Keep Right*. 13. Nov. 2016. URL: http://keepright.at/report_map.php?zoom=13&lat=52.73012&lon=13.84159&layers=B0T&ch=0%5C%2C30%5C%2C40%5C%2C50%5C%2C70%5C%2C90%5C%2C100%5C%2C110%5C%2C120%5C%2C130%5C%2C150%5C%2C160%5C%2C170%5C%2C180%5C%2C191%5C%2C192%5C%2C193%5C%2C194%5C%2C195%5C%2C196%5C%2C197%5C%2C198%5C%2C201%5C%2C202%5C%2C203%5C%2C204%5C%2C205%5C%2C206%5C%2C207%5C%2C208%5C%2C210%5C%2C220%5C%2C231%5C%2C232%5C%2C270%5C%2C281%5C%2C282%5C%2C283%5C%2C284%5C%2C285%5C%2C291%5C%2C292%5C%2C293%5C%2C294%5C%2C295%5C%2C296%5C%2C297%5C%2C298%5C%2C311%5C%2C312%5C%2C313%5C%2C320%5C%2C350%5C%2C370%5C%2C380%5C%2C401%5C%2C402%5C%2C411%5C%2C412%5C%2C413%5C%2C20%5C%2C60%5C%2C300%5C%2C360%5C%2C390&show_ign=1&show_tmpign=1 (besucht am 16. 11. 2016).

[61] Harald Klein, mueschel u. a. *Keep Right*. URL: https://github.com/keepright/keepright (besucht am 24. 08. 2016).

[62] Aaron Lidman. *To-fix: improving OpenStreetMap through micro tasking*. 20. Jan. 2015. URL: https://www.mapbox.com/blog/introducing-to-fix/ (besucht am 29. 08. 2016).

[63] *Map Features*. URL: https://wiki.openstreetmap.org/wiki/Map_Features (besucht am 15. 11. 2016).

[64] *Mapbox GL JS fundamentals*. Mapbox Inc. URL: https://www.mapbox.com/help/mapbox-gl-js-fundamentals/ (besucht am 20. 12. 2016).

[65] *Mapbox Vector Tile Specification*. Mapbox Inc. 19. Jan. 2016. URL: https://www.mapbox.com/vector-tiles/specification/ (besucht am 19. 12. 2016).

[66] *Maproulette*. URL: https://wiki.openstreetmap.org/wiki/MapRoulette (besucht am 29. 08. 2016).

[67] *MapRoulette API Doc*. URL: https://github.com/maproulette/maproulette2/blob/master/docs/api.md (besucht am 29. 08. 2016).

[68] Lukas Martinelli und Manuel Roth. »Updatable Vector Tile from OpenStreetMap«. Bachelorthesis. Hochschule für Technik Rapperswil, 2016. URL: https://cdn.rawgit.com/osm2vectortiles/bachelor-thesis/882a46977f7b984fc59be188335127d62921c thesis.pdf (besucht am 19. 12. 2016).

[69] Rory McCann. *Converting a regular carto project to vector tiles: OSM-carto case study*. Vortrag auf der State of the Map 2016. Brüssel, 23. Sep. 2016. URL: https://www.youtube.com/watch?v=wQc492mtK-c (besucht am 20. 12. 2016).

[70] *Meta Tiles*. 4. Juni 2015. URL: https://wiki.openstreetmap.org/wiki/Meta_tiles (besucht am 10. 11. 2016).

[71] Hong Minhee. *pghstore*. URL: https://github.com/dahlia/pghstore (besucht am 05. 12. 2016).

[72] mmd. *Maxheight Map*. URL: http://maxheight.bplaced.net/overpass/map.html (besucht am 09. 08. 2016).

[73] Constantin Müller. *Über die Unterkunftskarte*. URL: http://unterkunftskarte.de/about.html (besucht am 15. 11. 2016).

[74] Pascal Neis. *OpenStreetMap Crowd Report – Season 2015*. 10. Aug. 2015. URL: http://neis-one.org/2015/08/osm-report-2015/.

[75] Pascal Neis, Dennis Zielstra und Alexander Zipf. »The street network evolution of crowdsourced maps: OpenStreetMap in Germany 2007–2011«. In: *Future Internet* 4.1 (2011), S. 1–21.

[76] Pascal Neis und Alexander Zipf. »Analyzing the Contributor Activity of a Volunteered Geographic Information Project – The Case of OpenStreetMap«. In: *ISPRS International Journal of Geo-Information* 1.3 (Juli 2012), S. 146–165. ISSN: 2220-9964. DOI: 10.3390/ijgi1020146. URL: http://dx.doi.org/10.3390/ijgi1020146.

[77] Paul Norman. *Enhance with a datamodel option, using JSONB*. 25. Dez. 2016. URL: https://github.com/openstreetmap/osm2pgsql/issues/533#issuecomment-188786948 (besucht am 11. 02. 2016).

[78] Paul Norman. *Geospatial analysis with osm2pgsql*. 4. Sep. 2014. URL: https://github.com/openstreetmap/osm2pgsql/blob/master/docs/analysis.md (besucht am 11. 12. 2016).

[79] Paul Norman. *Incoming osm2pgsql change without migrations*. 6. Jan. 2017. URL: https://lists.openstreetmap.org/pipermail/dev/2017-January/029650.html (besucht am 14. 01. 2017).

[80] Paul Norman. *Serving Vector Tiles*. 19. Nov. 2016. URL: http://www.paulnorman.ca/blog/2016/11/serving-vector-tiles/ (besucht am 05. 12. 2016).

[81] *ogr2ogr*. OSGeo Foundation. URL: http://www.gdal.org/ogr2ogr.html (besucht am 16. 01. 2017).

[82] *OpenStreetMap Inspector*. Geofabrik GmbH. URL: http://tools.geofabrik.de/osmi/ (besucht am 09. 08. 2016).

[83] *OpenStreetMap Inspector*. Geofabrik GmbH. URL: http://tools.geofabrik.de/osmi/?view=areas%5C&lon=4.27247%5C&lat=52.09613%5C&zoom=12 (besucht am 10. 01. 2017).

[84] *OpenStreetMap Statistics*. OpenStreetMap Foundation. 15. Nov. 2016. URL: https://www.openstreetmap.org/stats/data_stats.html (besucht am 15. 11. 2016).

[85] *osm2pgsql*. Github-Repository. URL: https://github.com/openstreetmap/osm2pgsql (besucht am 16. 01. 2017).

[86] *Osmose*. URL: https://wiki.openstreetmap.org/wiki/Osmose (besucht am 25. 08. 2016).

[87] *Osmose-Backend*. OpenStreetMap France, 2011–2016. URL: https://github.com/osm-fr/osmose-backend (besucht am 24. 08. 2016).

[88] *Osmose-Karte*. OpenStreetMap France. 15. Nov. 2016. URL: http://osmose.openstreetmap.fr/en/map/#zoom=15&lat=53.22567&lon=11.61782&layer=Mapnik&overlays=FFFFFFFFFFFFFFFFFFFT&item=&level=1&tags=&fixable= (besucht am 16. 11. 2016).

[89] *Osmose-Karte*. OpenStreetMap France. URL: http://osmose.openstreetmap.fr/de/map/ (besucht am 25. 08. 2016).

[90] *Osmosis*. URL: https://wiki.openstreetmap.org/wiki/Osmosis (besucht am 24. 08. 2016).

[91] *Osmosis/Detailed Usage 0.45*. 27. Mai 2016. URL: https://wiki.openstreetmap.org/wiki/Osmosis/Detailed_Usage_0.45 (besucht am 19. 12. 2016).

[92] *[OSM-talk] OSM TrustPoints*. 10. Juli 2009. URL: https://lists.openstreetmap.org/pipermail/talk/2009-July/038385.html (besucht am 01. 09. 2016).

[93] *Overpass API*. URL: https://wiki.openstreetmap.org/wiki/Overpass_API (besucht am 16. 01. 2017).

[94] *PBF Format*. URL: https://wiki.openstreetmap.org/wiki/PBF_Format (besucht am 16. 11. 2016).

[95] *Planet.osm*. URL: https://wiki.openstreetmap.org/wiki/Planet.osm (besucht am 16. 01. 2017).

[96] *Planet.osm/diffs*. URL: https://wiki.openstreetmap.org/wiki/Planet.osm/diffs (besucht am 16. 01. 2017).

[97] *PostgreSQL 9.6 Documentation*. The PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs/9.6/static/sql-createview.html (besucht am 10. 12. 2016).

[98] *PostgreSQL 9.6 Documentation*. The PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs/9.6/static/populate.html (besucht am 30. 10. 2016).

[99] *PostgreSQL: Documentation: 9.5: hstore*. URL: https://www.postgresql.org/docs/9.5/static/hstore.html (besucht am 06. 12. 2016).

[100] *pqxx::tablestream Class Reference*. 20. Juli 2013. URL: http://pqxx.org/devprojects/libpqxx/doc/4.0/html/Reference/a00099.html (besucht am 30. 10. 2016).

[101] *Proposed features/Public Transport*. 26. Sep. 2011. URL: http://wiki.openstreetmap.org/w/index.php?title=Proposed_features/Public_Transport%5C&oldid=687630 (besucht am 16. 01. 2017).

[102] *Protocol Buffers*. Google Inc. URL: https://developers.google.com/protocol-buffers/ (besucht am 16. 11. 2016).

[103] Martin Raifer. *Latest Changes on OpenStreetMap*. URL: https://tyrasd.github.io/latest-changes/ (besucht am 09. 08. 2016).

[104] Frederik Ramm. *dense_tiles*. 23. Nov. 2016. URL: https://github.com/osmcode/osmium-contrib/tree/master/dense_tiles (besucht am 10. 01. 2017).

[105] Frederik Ramm und Jochen Topf. *OpenStreetMap. Die freie Weltkarte nutzen und mitgestalten*. 3. Aufl. Berlin: Lehmanns Media, 2010.

[106] *Relation*. URL: http://wiki.openstreetmap.org/wiki/Relation (besucht am 16. 01. 2017).

[107] *Relation:multipolygon*. URL: https://wiki.openstreetmap.org/wiki/Relation:multipolygon (besucht am 14. 12. 2016).

[108] *Relation:restriction*. URL: http://wiki.openstreetmap.org/wiki/Relation:restriction (besucht am 16. 01. 2017).

[109] *Relation:route*. URL: http://wiki.openstreetmap.org/wiki/Relation:route (besucht am 16. 01. 2017).

[110] Joe Schwartz. *Bing Maps Tile System*. Microsoft. URL: https://msdn.microsoft.com/en-us/library/bb259689.aspx (besucht am 05. 10. 2016).

[111] Dietmar Seifert. *Regionale OpenStreetMap-Dienste*. URL: http://www.regio-osm.de/ (besucht am 16. 11. 2016).

[112] *Tag:boundary=administrative*. URL: http://wiki.openstreetmap.org/wiki/Tag:boundary%5C%3Dadministrative (besucht am 16. 01. 2017).

[113] *The Rails Port*. OpenStreetMap Foundation. URL: https://github.com/openstreetmap/openstreetmap-website (besucht am 20. 08. 2016).

[114] *Tile Expiry Methods*. 14. Okt. 2015. URL: https://wiki.openstreetmap.org/wiki/Tile_expire_methods (besucht am 10. 11. 2016).

[115] Jochen Topf. *OSM Polygon Statistics*. 2016. URL: http://area.jochentopf.com/stats/ (besucht am 14. 12. 2016).

[116] Jochen Topf, Hrsg. *Osmium Concepts Manual*. URL: http://docs.osmcode.org/osmium-concepts-manual/ (besucht am 02. 11. 2016).

[117] Jochen Topf, Hrsg. *Osmium Library Manual*. URL: http://docs.osmcode.org/libosmium-manual/ (besucht am 02. 11. 2016).

[118] Jochen Topf. *type=multipolygon | Tags | OpenStreetMap Taginfo*. URL: https://taginfo.openstreetmap.org/tags/type=multipolygon (besucht am 14. 12. 2016).

[119] *Vandalismus vorbeugen*. 12. Sep. 2010. URL: http://forum.openstreetmap.org/viewtopic.php?id=9209 (besucht am 31. 08. 2016).

[120] *Verifiability*. 2009. URL: https://wiki.openstreetmap.org/wiki/Verifiability (besucht am 15. 11. 2016).

[121] *We Are Changing The License*. OpenStreetMap Foundation. 2010. URL: https://wiki.osmfoundation.org/wiki/Licence/Historic/We_Are_Changing_The_License (besucht am 15. 11. 2016).

[122] *Welcome to Wikipedia users*. 2016. URL: https://wiki.openstreetmap.org/wiki/Welcome_to_Wikipedia_users#We_don.27t_have_a_Notability_rule (besucht am 15. 11. 2016).

[123] Serge Wroclawski. *MapRoulette 2: Electric Boogaloo*. 13. Juni 2014. URL: http://ftp5.gwdg.de/pub/misc/openstreetmap/SOTMEU2014/13.mp4 (besucht am 29. 08. 2016).

[124] Ilya Zverv. *WhoDidIt*. URL: http://zverik.osm.rambler.ru/whodidit/ (besucht am 06. 08. 2016).

Klassen- und Methodenindex

Stichwort- und Softwareindex

Datenbanktabellen-, Datenbankfelder- und SQL-Befehlindex

Patternindex

I. Inhalt der beiliegenden CD

Alle Pfade beziehen sich auf das Root-Verzeichnis der CD.

forked-git-repositories/ enthält im Rahmen dieser Arbeit geforkte Git-Repositories.

forked-git-repositories/libosmium-manual/ enthält Beiträge zum *libosmium-manual*, der Dokumentation und Einführung der Osmium-Bibliothek. Das Repository, von dem geforkt wurde, ist mittlerweile im Github-Repository https://github.com/osmcode/osmcode.github.io aufgegangen.

forked-git-repositories/osmium-contrib/ enthält im Unterverzeichnis *dense_tiles* die erweiterte Version des gleichnamigen Programms.

my-git-repositories/ enthält die im Rahmen dieser Arbeit erstellten Git-Repositories mit Programmquellcode.

my-git-repositories/Cerepso/ enthält den Datenbankimporter Cerepso.

my-git-repositories/Cerepso2vt/ enthält den Vektortile-Generator Cerepso2vt.

my-git-repositories/Cerepso-Postgres-Backend/ enthält die sowohl von Cerepso als auch von Cerepso2vt benötigten Wrapper für die Bibliothek libpq.

my-git-repositories/expiries2shp/ enthält das Hilfsprogramm expiries2shp.

my-git-repositories/expiry_list_filter/ enthält die beiden Python-Skript zum Filtern der Ausgabe von *dense_tiles*.

my-git-repositories/pg-array-hstore-parser/ enthält die gleichnamige Bibliothek für das Parsen von Hstore- und Array-Feldern der PostgreSQL-Datenbank.

thesis/ enthält den Quellcode dieser Arbeit und ein PDF.

data/ enthält Datensätze, die in dieser Arbeit analysisiert wurden, ausgenommen OSM-Rohdaten. Letztere sind aus Platzgründen nicht enthalten, aber auf https://planet.openstreetmap.org/ verfügbar.

data/one-day-edits/ enthält die Expiry-Logfiles (Relationen nicht berücksichtigt), die durch den Import der stündlichen Diffs vom 5. 9. 2016 05:00:01 bis 6. 9. 2016 05:00:00 UTC erzeugt wurden. Außerdem enthält das Verzeichnis die daraus erstellten Shapefiles, welche Basis der Abbildung 4.7 sind.

data/final-tiling/ enthält die Ergebnisse, wenn man die in Abschnitt 4.1 beschriebenen Schritte mit dem Planetdump vom 29. August 2016 durchführt.
`ausgabe-z*-sorted.txt` enthält die alphabetisch sortierte Ausgabe von *dense_tiles*.
`z*.txt` enthält die Ergebnisse nach jedem Durchlauf von *split-initial-tiles.py* und *next-level.py*. `qt-z15.*` sind die einzelnen Dateien des Shapefiles, welches aus `z15.txt` erstellt wurde und in Abbildung 4.1 abgebildet ist.

`data/expiry-lists/` enthält die Tile-Expiry-Listen für den Zeitraum 29. 8.–5. 10.2016. Das Unterverzeichnis `all` enthält die, die entstehen, wenn Änderungen an Relationen berücksichtigt werden. Das Unterverzeichnis `without-relations` enthält die, die entstehen, wenn Änderungen Relationen ignoriert werden.

`data/expiry-shapes/` enthält die aus den Tile-Expiry-Listen des Zeitraums 29. 8.–5. 10.2016 erstellten Shapefiles (nur die aggregierten, d. h. es gibt keine sich überlappenden Polygone). Das Unterverzeichnis `all` enthält die, die entstehen, wenn Änderungen an Relationen berücksichtigt werden. Das Unterverzeichnis `without-relations` enthält die, die entstehen, wenn Änderungen Relationen ignoriert werden. Die Shapefiles in diesen beiden Unterverzeichnissen dienten als Datengrundlage für die Abbildungen 4.2 bis 4.6.