# Berechnung topologisch korrekter Nachbarflächen

Erstellt von: Dr. Peter Korduan, GDI-Service

**Änderungen:**

| Datum | Änderung |
|---|---|
| 27.07.2017 | 1. Version des Dokumentes erstellt |
| 03.03.2017 | Beschreibung der Berechnung des Außenpolygon |
| 03.05.2018 | Beschreibung des Standes der Berechnung |
| 16.07.2018 | Topologie bereinigen, Finale Version und Zusammenfassung |
| 15.08.2018 | Logging-Tabellen |
| 18.08.2018 | Überarbeitung von CloseTopoGaps, Logging und Test gemeinden_mv |
| 19.08.2018 | Korrekturen |
| 04.09.2018 | Reihenfolge toTopoGeom removeOverlap und closeGap geändert |
| | |
| | |
| | |
| | |

# Inhaltsverzeichnis

# 1 Ansätze zur Lösung

## 1.1 Fremdlösungen

- FME
- mapshaper.org

## 1.2 Lösungen mit Reparatur von Geometrien

### 1.2.1 Funktion zum Suchen und Beheben von Polygon Overlaps

Quelle: https://trac.osgeo.org/postgis/wiki/UsersWikiExamplesPolygonOverlaps

### 1.2.2 Simplify a map layer using PostGIS topology

Quelle: http://strk.kbt.io/blog/2012/04/13/simplifying-a-map-layer-using-postgis-topology/

### 1.2.3 Topology cleaning with PostGIS

Quelle: https://strk.kbt.io/blog/2011/11/21/topology-cleaning-with-postgis/
Topology cleaning with PostGIS
An early tester of the new PostGIS Topology submitted an interesting dataset which kept me busy for a couple of weeks fixing a bunch of bugs related to numerical stability/robustness.
Finally, the ST_CreateTopoGeo function succeeded and imported the dataset as a proper topological schema. Here's what it looks like:

Edges of the built topology

At a first glance it doesn't seem to be particularly problematic. Here's the composition summary:

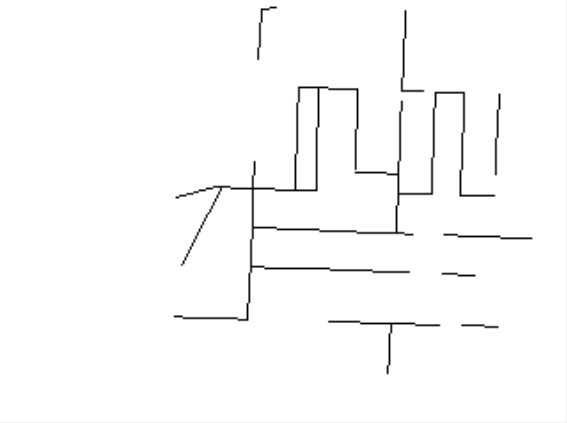
```
=# select topologysummary('small_sample_topo');
                          topologysummary
-------------------------------------------------------------
  Topology small_sample_topo (2042), SRID 0, precision 0
  83 nodes, 156 edges, 74 faces, 0 topogeoms in 0 layers
```

But the devil hides at high zoom levels. Where to zoom ? What are we looking for ?

We are guaranteed none of the constructed edges cross so the only leftover problem we might encounter is very small faces constructed wherever the original input had small overlaps or underlaps (gaps). We can have a visual signal of those faces by creating a view showing faces with an area below a given threshold. Let's do that:

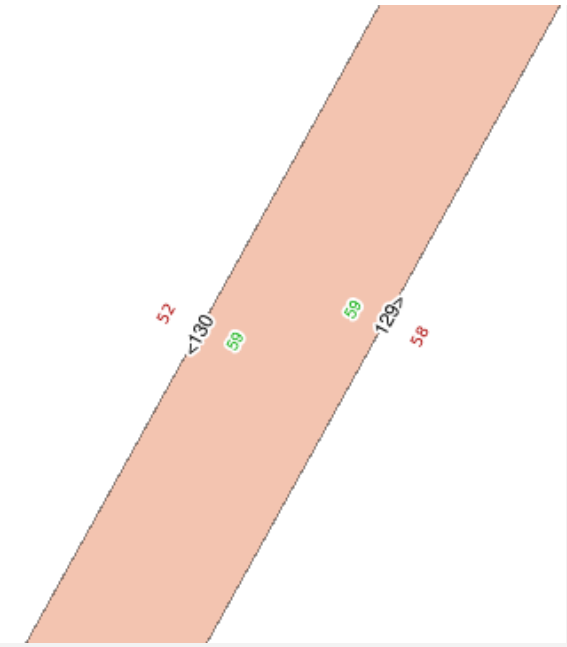
```
CREATE VIEW small_sample_topo.small_areas AS
SELECT face_id, st_getfacegeometry('small_sample_topo', face_id)
FROM small_sample_topo.face
WHERE face_id > 0
AND st_area(st_getfacegeometry('small_sample_topo', face_id)) < 0.1;
```

That query would let us see where to find faces with area < 0.1 units. And here's qgis showing it:



Areas smaller than 0.1 square units

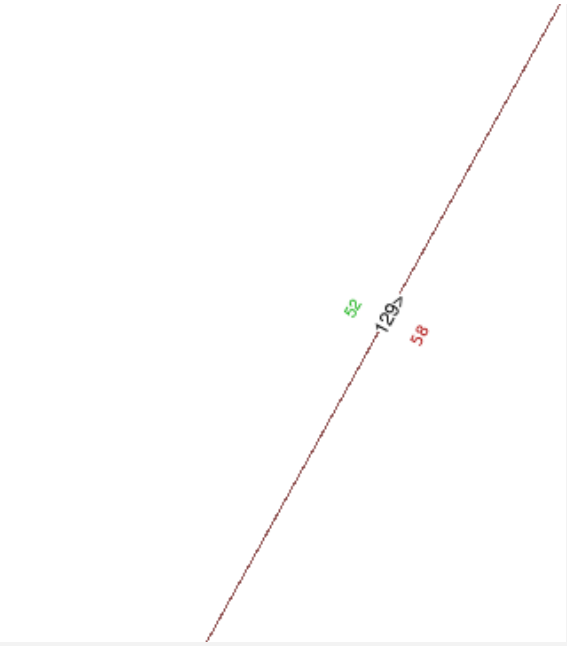
Now we know where to zoom, and also the ID of the offending faces.
Let's zoom in and show some labels:




Detail of small area

You can now see that face 59 is bound by (among others) edges 130 and 129. Just get rid of one of them to assign the area to an adjacent face.

We drop edge 130 using ST_RemEdgeModFace, assigning the area to face 52. Here's the result:




Area after cleanup

Cleaning further would require removing further edges and thus getting rid of all the small faces. There's a lot of room for automating such processes. The good new is you can now build your own automation around your specific use cases while still using robust and standard foundations.
It is to be noted that the whole process I described here only involved the geometrical/topological level and didn't affect at all the semantic/feature level. If we had TopoGeometry objects defined by the faces we'd also know which small faces were part of overlaps or underlaps and could then act consequently by adding or removing faces to the definition of the appropriate TopoGeometry object. Such step would have been required for the overlap situations as the ST_RemEdgeModFace function doesn't let you change the shape of a defined TopoGeometry.
Unfortunately the semantic level is lost when using the ISO functions, as the whole ISO topology model doesn't deal with features at all. This is why I think PostGIS would benefit from having a function that converts your simple features into topologically-defined features by adding any missing primitive to a topology schema and constructing the feature for you. Such function, is only waiting for sponsorship to become a reality of PostGIS 2.0. If you like what we're building for your data integrity, please consider supporting the effort!

share this

Tags: postgis, qgis, topology

This entry was posted on Monday, November 21st, 2011 at 5:51 pm and is filed under Free Software, GIS. You can follow any responses to this entry through the RSS 2.0feed. You can leave a response, or trackback from your own site.

## 1.2.4 Cannot delete slivers (or gaps)

Quelle: http://lists.osgeo.org/pipermail/postgis-users/2014-November/039755.html

**Sandro Santilli** strk at keybit.net
*Wed Nov 5 03:03:25 PST 2014*

- Previous message: [postgis-users] Topology: cannot delete slivers (or gaps)
- Next message: [postgis-users] Topology: cannot delete slivers (or gaps)
- **Messages sorted by:** [ date ] [ thread ] [ subject ] [ author ]

On Tue, Nov 04, 2014 at 03:08:15PM -0800, Guillaume Drolet wrote:

> *But because I'm stubborn and never give up, I still want to succeed on*
> *fixing TopoGeometry objects in PostGIS (bear with me, what follows is a bit*
> *long):*

I like you ! :)

> *First, I looked at faces NOT taking part in the composition of any*
> *TopoGeometry object:*
>
> */SELECT face_id FROM de_20k_topo.face WHERE face_id NOT IN (SELECT*
> *element_id*
>
> *FROM de_20k_topo.relation);/*
> *face_id*
> *0*
> *2418*
> *2650*
> *2663*
> *2766*
> *3077*
> *3118*
> *3146*
> *3267*
> *3270*
>
> *None of these 10 faces are involved with my three remaining slivers... And*
> *when I look at them, they ARE related to edges, as a next_face or a*
> *right_face. Is there any useful information in that for my purpose?*

Except face 0, which is the "universe" face, all the others
should be "gaps" in your topology. You can zoom to the face's
bounding box ("mbr" field) to take a closer look (do you have qgis
set up by now?).

What you could do programmatically is for each face to fetch the list
of edges bounding it and pick the face on the other side as a candidate;
then for each candidate face you find TopoGeometry objects containing
it in their definition and pick one to assign that orphaned face
(you could use smaller TopoGeometry, or larger, or whatever).

It might be better first to deal with the faces that are associated
with multiple TopoGeometry, just to avoid assigning an orphaned face
to a TopoGeometry based on it's containing a face which will later
be removed from the definition because duplicated.

> *Second, I tried to work along the lines suggested by Sandro, using a new*
> *topology (tolerance 2.0 m). As in one of my previous attempts, I*
> *successfully fixed 20 of the 23 slivers using ST_RemEdgeModFace. For the*
> *remaining three slivers, however, I tried modyfying TopoGeometry objects but*
> *with no success. I must admit I'm a bit lost and not too sure if I really*
> *understand what TopoGeometry objects are:*

Eh, that part is a still a bit weak in documantation, maybe.
A TopoGeometry is a Geometry defined by its components.

In your case (areal TopoGeometry objects) it is a (multi)Polygon
defined by a set of faces. Note that QGIS is also able to
show you the TopoGeometry objects, and lets you edit them "spatially"
(less robust than editing them by adding/removing components but with
some care easier to use).

> *For example, let's look at problematic face_id: 2227*
> *(https://dl.dropboxusercontent.com/u/5196336/example_2227.bmp). In this*
> *case, the light grey line is where a single edge should be. Blue edges 6381*
> *and 6341 should not be there. The light grey line should start from the node*
> *at the bottom right and connect at its other end somewhere in the middle of*
> *edge 6340:*

Uhm, edge identifiers are missing from that dump.
I take it 6381 is the lower blue and 6341 the upper blue.

> *I thought, let's first remove edge 6346, which should not exist in the first*
> *place, and then split edge 6340 with a node and connect it to node 4753 on*
> *the lower right corner of the image:*
>
> *SELECT ST_RemEdgeModFace('de_20k_topo', 6346);*
>
> *As in my previous post, this yields:*
>
> *ERROR:  TopoGeom 1546 in layer 1 (syshiera.de_20k.topogeom) cannot be*
> *represented healing faces 2225 and 2227*

I take it that edge 6346 is the one on the right of the yellow face.
The message is telling you that a TopoGeometry object exists (id=1546)
that is composed only by one of the two faces 2225 and 2227.

I suggest you load the TopoGeometry layer with qgis and see for yourself.
Unfortunately (that'd be useful to add in qgis) it won't be easy to
find TopoGeometry 1546 as extracting the id of a TopoGeometry requires
running an expression on the database: id(topogeom).

You could create a view selecting just that topoGeometry, or a subset
of them, or you could just load them all, hoping it won't be too expensive
(another spot where qgis support for postgis topology could be improved).

Chances are it would be enough to assign face 2227 to the TopoGeometry
with id 1546 (likely the one not-assigned). But as face 2227 was not
in your initial set (unassigned faces) I guess there's another TopoGeometry
which does have 2227 in his definition, but not the one on the right.
In that case you have to pick which of the two contending TopoGeometry
objects should get face 2227 assigned, before you drop it. Basically, which
TopoGeometry will get assigned the space occupied by that face.

Given QGIS editing support for TopoGeometry objects you should be able
to see all of this by loading the layer and setting 50% transparency.
Enabling topological editing and editing those two TopoGeometry objects
should help you with manual resolution of this issue.

> *So I looked at TopoGeom 1546:*
>
> */SELECT * FROM de_20k_topo.relation WHERE topogeo_id = 1546;/*
>

> *topogeo_id;layer_id;element_id;element_type*
> *1546;1;2225;3*
> *1546;1;2233;3*
>
> *Edge 2227 is missing from TopoGeom 1546: Where is it?*

You meant _face_ 2227.
Right, it's not assigned to TopoGeometry 1546,
but to another one.

> */SELECT * FROM de_20k_topo.relation WHERE element_id = 2227;/*
>
> *topogeo_id;layer_id;element_id;element_type*
> *1551;1;2227;3*
>
> *There it is, in TopoGeom 1551. What else is there:*
>
> */SELECT * FROM de_20k_topo.relation WHERE topogeo_id = 1551;/*
>
> *topogeo_id;layer_id;element_id;element_type*
> *1551;1;2225;3*
> *1551;1;2234;3*
> *1551;1;2227;3*
>
> *So edges 2225 and 2227 are associated with TopoGeom 1551. Will change that:*

Again, you mean faces (last number is element_type, 3==face).
So TopoGeometry objects 1546 and 1551 overlap, and their intersection
is the union of faces 2225 and 2227.

> *UPDATE de_20k_topo.relation SET topogeo_id = 1546*
> *WHERE topogeo_id = 1551 AND element_id = 2227;/*

There's already a record associating face 2227 with TopoGeometry 1546
so you don't need another record, just drop the one associating it
with TopoGeometry 1551:

```
  DELETE FROM de_20k_topo.relation
  WHERE topogeo_id = 1551 AND element_id = 2227;
```

> *And then again:*
>
> */SELECT ST_RemEdgeModFace('de_20k_topo', 6346); /*
>
> *ERROR:  TopoGeom 1551 in layer 1 (syshiera.de_20k.topogeom) cannot be*
> *represented healing faces 2225 and 2227*

Right, you only resolved the overlap of face 2227, but there's still
the overlap of face 2225 :)

```
  DELETE FROM de_20k_topo.relation
  WHERE topogeo_id = 1551 AND element_id = 2225;
```

> *Obviously, as mentionned above, I don't quite understand how TopoGeometry*
> *objects work and the link between TopoGeoms, geometries and primitives. I*
> *need a bit of education here.*

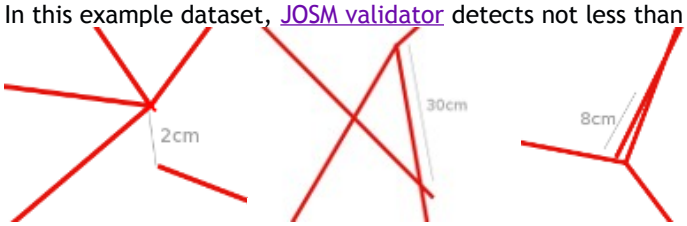
```
It looks like you're on the right track so far.

> Thanks a lot for your patience and your help.

Thank you for your stubbornnes in using PostGIS Topology !

--strk;

  ()   Free GIS & Flash consultant/developer
  /\   http://strk.keybit.net/services.html
```

## 1.2.5 Use PostGIS topologies to clean-up road networks

Quelle: http://blog.mathieu-leplatre.info/use-postgis-topologies-to-clean-up-road-networks.html
Use PostGIS topologies to clean-up road networks

By Mathieu Leplatre, Frédéric Bonifas
In Dev.
tags: postgissqltopologyopendatatoulouse
This article gives a few basics to get started with using the PostGIS topology extension.
We will take avandtage of topologies to clean-up a real topological road network, coming from Toulouse OpenData files.

Topological
A topology is a general concept, where objects are defined by their relationships instead of their geometries.
Instead of lines, we manipulate edges, vertices and faces : might remind you the core concepts of graph theory.
A topological road network is supposed to have their lines (edges) connected at single points (nodes).
In this example dataset, JOSM validator detects not less than 1643 errors :) Broken connections, crossing lines ...




**Let's clean this up !**
Installation
On Ubuntu 12.04, you just have to install PostGIS :

```
sudo apt-add-repository -y ppa:ubuntugis/ppa

sudo apt-get update

sudo apt-get install -y postgresql postgis
```

The topology extension is installed by default. Just activate it in your database:

```
CREATE DATABASE "roadsdb";
CREATE EXTENSION postgis;
CREATE EXTENSION postgis_topology;
SET search_path = topology,public;
```

## Data Import

Load your shapefile (using command-line) like usual :

```
schema="public."
db="roadsdb"
user="postgres"
password="postgres"
host="localhost"
ogr2ogr -f "PostgreSQL" PG:"host=${host} user=${user} dbname=${db} password=${password}" -a_srs "EPSG:2154" -nln ${schema}roads -nlt
MULTILINESTRING ROAD_SHAPEFILE.SHP
```

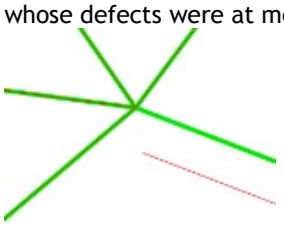
Create and associate the PostGIS topology:

```
SELECT topology.CreateTopology('roads_topo', 2154);
SELECT topology.AddTopoGeometryColumn('roads_topo', 'public', 'roads', 'topo_geom', 'LINESTRING');
```

Convert linestrings to vertices and edges within the topology :

```
-- Layer 1, with 1.0 meter tolerance
UPDATE roads SET topo_geom = topology.toTopoGeom(wkb_geometry, 'roads_topo', 1, 1.0);
```

From now on, we have a topology, whose imperfections were corrected. It smoothly merged all *dirty* junctions, whose defects were at most 1.0 meter wide.



You may enconter insertion problems : the tool fails [1] and aborts the whole transaction. Use this snippet to skip errors and go on with the next records:

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT * FROM roads LOOP
        BEGIN
            UPDATE roads SET topo_geom = topology.toTopoGeom(wkb_geometry, 'roads_topo', 1, 1.0)
            WHERE ogc_fid = r.ogc_fid;
        EXCEPTION
            WHEN OTHERS THEN
                RAISE WARNING 'Loading of record % failed: %', r.ogc_fid, SQLERRM;
        END;
    END LOOP;
END$$;
```

This is rather frustrating to face topological errors at insertion ! You can try with a lower tolerance, or check that your records have at least valid geometries. *Any clarification or help on this would be welcome* :)

## Visualize and export

In order to visualize your topology vertices in QGIS, browse your database tables, and add the following layers: roads_topo.edge_data and roads_topo.node.

You can also export the resulting geometries into a new table :

```
CREATE TABLE roads_clean AS (
        SELECT ogc_fid, topo_geom::geometry
        FROM roads
);
```

Or obtain your lovable Shapefile in return :

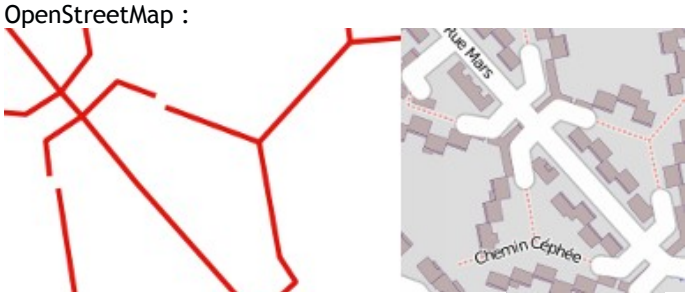
```
ogr2ogr -f "ESRI Shapefile" ROAD_CLEAN.SHP PG:"host=${host} user=${user} dbname=${db} password=${password}" -sql "SELECT
topo_geom::geometry FROM roads"
```

If, like Amit you want to split the lines at intersections and assign original attributes, just join `roads_topo.edge_data`and on the `roads` table :

```
SELECT r.lib_off, r.ogc_fid, e.geom
FROM roads_topo.edge e,
          roads_topo.relation rel,
          roads r
WHERE e.edge_id = rel.element_id
    AND rel.topogeo_id = (r.topo_geom).id
```

Going further...

We could collapse crossing lines and disconnected junctions into a nice and clean network.
Yes ahem, we weren't able to repair *every* topological error of this dataset using this automatic method. Some inconsistencies, like the following one, are like 6 meters wide ! They are, by the way, perfectly described in OpenStreetMap :




We could also play with simplifications using Sandro Santilli's `SimplifyEdgeGeom` [2] function, it will collapse edges with a higher tolerance ...

```
SELECT SimplifyEdgeGeom('roads_topo', edge_id, 1.0) FROM roads_topo.edge;
```

Don't hesitate to share your thoughts and feedback. Concrete use cases and examples are rare about this! And as usual, drop a comment if anything is wrong or not clear :)

[1] `SQL/MM Spatial exception, geometry intersects edge, side location conflict, ...`

[2] Just execute the function SQL code. It's just an elegant wrapper around `ST_ChangeEdgeGeom` and `ST_Simplify`.

## 1.2.6 Clean Polygons

Quelle: https://trac.osgeo.org/postgis/wiki/UsersWikiCleanPolygons
Clean Polygons

It often happens that geometry data has invalid topology according to the OGC model used by PostGIS. This situation causes many operations to fail or give incorrect results.

To correct this, it is necessary to correct the invalid polygonal topology by cleaning the polygon. One the basic approach to do this is:

- extract the linework from the polygon to a lineal geometry
- union the lineal geometry together along with a single point from the geometry (this effectively forces the the linework to be noded & dissolved)
- polygonize the resulting linework

Caution: this may not create the expected topology if the polygon contains holes.

Alternatively you can download a plpgsql function from http://www.kappasys.ch/pgtools/cleangeometry/cleanGeometry.sql This function works with polygons containig holes as well as with linestrings.

Example:

```
SELECT cleanGeometry(the_geom) FROM foo;
```

Example:

```
-- Polygonize the noded linework
SELECT ST_BuildArea(geom) AS geom
FROM
(
 -- Node & dissolve the linear ring
 SELECT ST_Union(geom, ST_Startpoint(geom)) AS geom
 FROM
 (
    -- Extract the exterior ring of the polygon
    SELECT ST_ExteriorRing(
       'POLYGON((
            9.50351715087891 47.3943328857422,
            9.50386047363281 47.3943328857422,
            9.50351715087891 47.3943328857422,
            9.50248718261719 47.3943328857422,
            9.50214385986328 47.3939895629883,
            9.50180053710938 47.3943328857422,
            9.50145721435547 47.3939895629883,
            9.50111389160156 47.3936462402344,
            9.50145721435547 47.3936462402344,
            9.50145721435547 47.3939895629883,
            9.50214385986328 47.3939895629883,
            9.50248718261719 47.3939895629883,
            9.50386047363281 47.3943328857422,
            9.50351715087891 47.3943328857422))'::geometry) AS geom
 ) AS ring
) AS fixed_ring
```

which produces:

```
MULTIPOLYGON (((
            9.50351715087891 47.3943328857422,
```

```
            9.50386047363281 47.3943328857422,
            9.50248718261719 47.3939895629883,
            9.50214385986328 47.3939895629883,
            9.50248718261719 47.3943328857422,
            9.50351715087891 47.3943328857422
        )), ((
            9.50145721435547 47.3939895629883,
            9.50180053710938 47.3943328857422,
            9.50214385986328 47.3939895629883,
            9.50145721435547 47.3939895629883
        )), ((
            9.50145721435547 47.3939895629883,
            9.50145721435547 47.3936462402344,
            9.50111389160156 47.3936462402344,
            9.50145721435547 47.3939895629883
        )))
```

## 1.2.7 Introduction to PostGIS, Validiy

Quelle: http://workshops.boundlessgeo.com/postgis-intro/validity.html
Validity

In 90% of the cases the answer to the question, “why is my query giving me a ‘TopologyException’ error” is “one or more of the inputs are invalid”. Which begs the question: what does it mean to be invalid, and why should we care?

### 21.1. What is Validity

Validity is most important for polygons, which define bounded areas and require a good deal of structure. Lines are very simple and cannot be invalid, nor can points.

Some of the rules of polygon validity feel obvious, and others feel arbitrary (and in fact, are arbitrary).

- Polygon rings must close.
- Rings that define holes should be inside rings that define exterior boundaries.
- Rings may not self-intersect (they may neither touch nor cross one another).
- Rings may not touch other rings, except at a point.

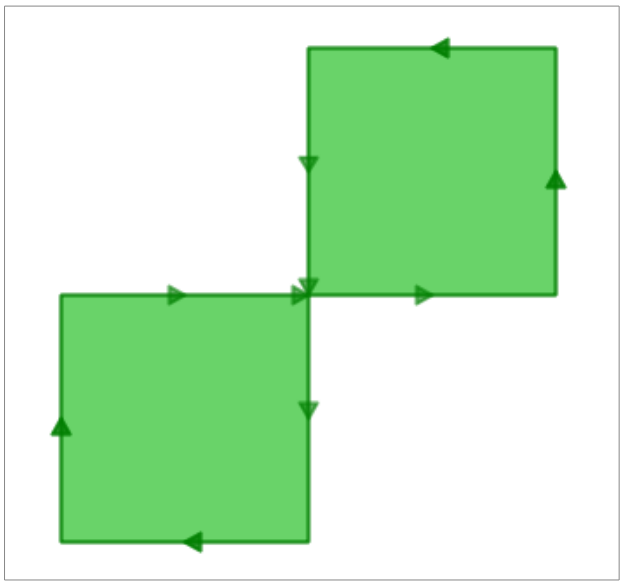
The last two rules are in the arbitrary category. There are other ways to define polygons that are equally self-consistent but the rules above are the ones used by the *OGC SFSQL* standard that PostGIS conforms to.

The reason the rules are important is because algorithms for geometry calculations depend on consistent structure in the inputs. It is possible to build algorithms that have no structural assumptions, but those routines tend to be very slow, because the first step in any structure-free routine is to *analyze the inputs and build structure into them*.

Here’s an example of why structure matters. This polygon is invalid:

```
POLYGON((0 0, 0 1, 2 1, 2 2, 1 2, 1 0, 0 0));
```

You can see the invalidity a little more clearly in this diagram:



The outer ring is actually a figure-eight, with a self-intersection in the middle. Note that the graphic routines successfully render the polygon fill, so that visually it is appears to be an “area”: two one-unit squares, so a total area of two units of area.

Let’s see what the database thinks the area of our polygon is:

```
SELECT ST_Area(ST_GeometryFromText(
           'POLYGON((0 0, 0 1, 1 1, 2 1, 2 2, 1 2, 1 1, 1 0, 0 0))'
        ));
 st_area
---------
         0
```

What’s going on here? The algorithm that calculates area assumes that rings to not self-intersect. A well-behaved ring will always have the area that is bounded (the interior) on one side of the bounding line (it doesn’t matter which side, just that it is on *one* side). However, in our (poorly behaved) figure-eight, the bounded area is to the right of the line for one lobe and to the left for the other. This causes the areas calculated for each lobe to cancel out (one comes out as 1, the other as -1) hence the “zero area” result.

## 21.2. Detecting Validity

In the previous example we had one polygon that we **knew** was invalid. How do we detect invalidity in a table with millions of geometries? With the **ST_IsValid(geometry)** function. Used against our figure-eight, we get a quick answer:

```
SELECT ST_IsValid(ST_GeometryFromText(
           'POLYGON((0 0, 0 1, 1 1, 2 1, 2 2, 1 2, 1 1, 1 0, 0 0))'
        ));
f
```

Now we know that the feature is invalid, but we don’t know why. We can use the **ST_IsValidReason(geometry)** function to find out the source of the invalidity:

```
SELECT ST_IsValidReason(ST_GeometryFromText(
           'POLYGON((0 0, 0 1, 1 1, 2 1, 2 2, 1 2, 1 1, 1 0, 0 0))'
        ));
Self-intersection[1 1]
```

Note that in addition to the reason (self-intersection) the location of the invalidity (coordinate (1 1)) is also returned.

We can use the **ST_IsValid(geometry)** function to test our tables too:

```
-- Find all the invalid polygons and what their problem is
SELECT name, boroname, ST_IsValidReason(geom)
FROM nyc_neighborhoods
WHERE NOT ST_IsValid(geom);
             name           |   boroname    |                    st_isvalidreason
------------------------+---------------
+-----------------------------------------------------------
 Howard Beach            | Queens        | Self-intersection[597264.083368305 4499924.54228856]
 Corona                  | Queens        | Self-intersection[595483.058764138 4513817.95350787]
 Steinway                | Queens        | Self-intersection[593545.572199759 4514735.20870587]
 Red Hook                | Brooklyn      | Self-intersection[584306.820375986 4502360.51774956]
```

## 21.3. Repairing Invalidity

First the bad news: there is no 100% guaranteed way to fix invalid geometries. The worst case scenario is identifying them with the **ST_IsValid(geometry)** function, moving them to a side table, exporting that table, and repairing them externally.

Here's an example of SQL to move invalid geometries out of the main table into a side table suitable for dumping to an external cleaning process.

```
-- Side table of invalids
CREATE TABLE nyc_neighborhoods_invalid AS
SELECT * FROM nyc_neighborhoods
WHERE NOT ST_IsValid(geom);

-- Remove them from the main table
DELETE FROM nyc_neighborhoods
WHERE NOT ST_IsValid(geom);
```

A good tool for visually repairing invalid geometry is OpenJump (http://openjump.org) which includes a validation routine under **Tools->QA->Validate Selected Layers**.

Now the good news: a large proportion of invalidities **can be fixed inside the database**using either:

- the **ST_MakeValid** function or,
- the **ST_Buffer** function.

### 21.3.1. ST_MakeValid

**ST_MakeValid** attempts to repair invalidities without only minimal alterations to the input geometries. No vertices are dropped or moved, the structure of the object is simply re-arranged. This is a good thing for clean, but invalid data, and a bad thing for messy and invalid data.

```
-- Fix the invalid figure-8 polygon
SELECT ST_AsText(ST_MakeValid(
            'POLYGON((0 0, 0 1, 1 1, 2 1, 2 2, 1 2, 1 1, 1 0, 0 0))'
        ));
MULTIPOLYGON(
   ((0 0,0 1,1 1,1 0,0 0)),
   ((1 1,1 2,2 2,2 1,1 1))
)
```

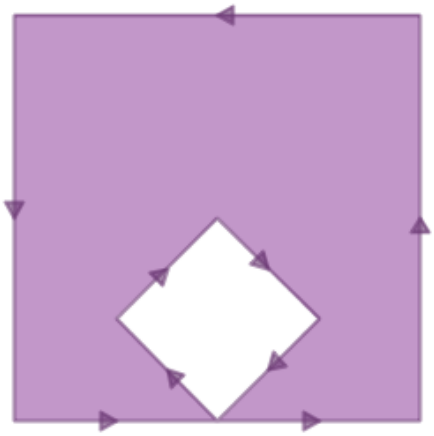
**ST_MakeValid** successfully converts the figure-8 into a multi-polygon that represents the same area.

### 21.3.2. ST_Buffer

Cleaning using the buffer trick takes advantage of the way buffers are built: a buffered geometry is a brand new geometry, constructed by offsetting lines from the original geometry. If you offset the original lines by **nothing** (zero) then the new geometry will be structurally identical to the original one, but because it is built using the *OGC* topology rules, it will be valid.

For example, here's a classic invalidity – the "banana polygon" – a single ring that encloses an area but bends around to touch itself, leaving a "hole" which is not actually a hole.

```
POLYGON((0 0, 2 0, 1 1, 2 2, 3 1, 2 0, 4 0, 4 4, 0 4, 0 0))
```



Running the zero-offset buffer on the polygon returns a valid *OGC* polygon, consisting of an outer and inner ring that touch at one point.

```
SELECT ST_AsText(
            ST_Buffer(
               ST_GeometryFromText('POLYGON((0 0, 2 0, 1 1, 2 2, 3 1, 2 0, 4 0, 4 4, 0 4, 0 0))'),
               0.0
            )
         );
POLYGON((0 0,0 4,4 4,4 0,2 0,0 0),(2 0,3 1,2 2,1 1,2 0))
```

**Note**

The “banana polygon” (or “inverted shell”) is a case where the *OGC* topology model for valid geometry and the model used internally by ESRI differ. The ESRI model considers rings that touch to be invalid, and prefers the banana form for this kind of shape. The OGC model is the reverse. Neither is “correct”, they are just different ways to model the same situation.

## 1.2.8 Erkennen und Eliminieren von Sliver Polygonen und Spikes

https://giswiki.hsr.ch/PostGIS_-_Tipps_und_Tricks#Erkennen_und_Eliminieren_von_Sliver_Polygonen_und_Spikes

Implementiert als sog. 'Stored Procedures'.

Siehe auch PostGIS-Beispiele.

Eliminate sliver polygons

Sources:

- "Spike analyzer and remover"
- PostGIS NormalizeGeometry: https://gasparesganga.com/labs/postgis-normalize-geometry/
- Schmidt, Andreas; Krüger, Nils: spike_analyzer.sql 2009-12-01, Version 1.0, Informatikzentrum Landesverwaltung Baden-Württemberg (IZLBW), url: http://trac.osgeo.org/postgis/wiki/UsersWikiExamplesSpikeAnalyzer.
- Schmidt, Andreas; Krüger, Nils: spikeremover und spikeRemoverCore.sql 2009-10-01, Version 1.0, Informatikzentrum Landesverwaltung Baden-Württemberg (IZLBW), url: http://trac.osgeo.org/postgis/wiki/UsersWikiExamplesSpikeRemover.
- Birgit Laggner und Helge Meyer-Borstel, „Geoprocessing von Massendaten in PostGIS - Probleme und Lösungsansätze“, FOSSGIS 2010, [7]
- Komplettverschneidung: http://trac.osgeo.org/postgis/wiki/UsersWikiExamplesOverlayTables2
- Überlappungsbereinigung: http://trac.osgeo.org/postgis/wiki/UsersWikiExamplesPolygonOverlaps

Given a polygon table that has many small areas and holes. How to remove "small" areas and holes (smaller than a given area in m2)?

Remarks:

- Similar like the ELIMINATE command in ArcGIS.
- See also section Clean topology below

```
CREATE OR REPLACE FUNCTION Filter_Rings(geometry,float)
RETURNS geometry AS
$$
SELECT ST_MakePolygon(c.outer_ring, d.inner_rings) as final_geom
 FROM (/* Get outer ring of polygon */
       SELECT ST_ExteriorRing(b.the_geom) as outer_ring
          FROM (SELECT (ST_DumpRings($1)).geom As the_geom, path(ST_DumpRings($1)) as
path) b
          WHERE b.path[1] = 0 /* ie the outer ring */
       ) c,
      (/* Get all inner rings > a particular area */
       SELECT ST_Accum(ST_ExteriorRing(b.the_geom)) as inner_rings
          FROM (SELECT (ST_DumpRings($1)).geom As the_geom, path(ST_DumpRings($1)) as
path) b
          WHERE b.path[1] > 0 /* ie not the outer ring */
            AND ST_Area(b.the_geom) > $2
```

```
        ) d
 $$
 LANGUAGE 'sql' IMMUTABLE;

 CREATE OR REPLACE FUNCTION Filter_Rings(geometry GEOMETRY, param FLOAT)
 RETURNS geometry AS
 $$
 SELECT ST_MakePolygon(c.outer_ring, d.inner_rings) as final_geom
   SELECT * FROM foo WHERE attr > param; -- statt $2 steht hier ein benanntes Fn.-
Argument
   ...
 $$
 LANGUAGE 'sql';
```

Usage example:

```
 % SELECT ST_AsText(
   Filter_Rings(
     ST_PolyFromText(
        'POLYGON((10 10,10 20,20 20,20 10,10 10),(0 0,0 1,1 1,1 0,0 0),(5 5,5 7,7 7,7
5,5 5))'
     ) ,1::float
   )
 );
 % "POLYGON((10 10,10 20,20 20,20 10,10 10),(5 5,5 7,7 7,7 5,5 5))"
```

Shorter alternative to Filter_Rings:

```
 CREATE OR REPLACE FUNCTION Filter_Rings2(geometry,float) RETURNS geometry AS
 $$ SELECT ST_BuildArea(ST_Collect(a.geom)) as final_geom
         FROM ST_DumpRings($1) AS a
           WHERE a.path[1] = 0 OR
                 (a.path[1] > 0 AND ST_Area(a.geom) > $2)
 $$
 LANGUAGE 'sql' IMMUTABLE;
```

With following restrictions: Possibly slower and squashes 3D geometries to 2D. (Source: http://postgis.refractions.net/pipermail/postgis-users/2009-January/022325.html)

### 1.2.9 Clean topology

Remarks: Similar to ArcGIS' CLEAN. See also Eliminate sliver polygons.

See:

- Function cleanGeometry(): http://trac.osgeo.org/postgis/wiki/UsersWikiCleanPolygons
- Topology function: http://trac.osgeo.org/postgis/wiki/UsersWikiPostgisTopology

### 1.2.10 Tidying feature geometries with sf

Quelle: https://www.r-spatial.org/r/2017/03/19/invalid.html

## 1.3 Lösungen mit Triangulation

### 1.3.1 Processing real-world datasets into clean geometric models

Quelle: https://3d.bk.tudelft.nl/ken/en/thesis/cleaning.html

10 Processing real-world datasets into clean geometric models

The representations described in **Part I** and the operations described in **Part II** work well on perfectly valid data. Objects are assumed not to overlap each other, to be properly closed with no degenerate geometries, and to have perfectly planar faces which are consistently oriented, among many other validity criteria. Unfortunately, as §10.1 explains, GIS processes often fail to clearly specify which criteria are expected and real-world data often fails to meet them, with consequences ranging from the innocuous to a complete inability to use a desired tool, including instances where software gives erroneous results unbeknownst to the user. As GIS datasets can be rather intricate and expensive to acquire, they are not easily replaceable and must therefore be repaired, i.e. they must be processed into geometric models that conform to certain validity specifications, so as to make them fit for use.

In the context of this thesis, clean geometric models are important as they are the base for the higher-dimensional models using the representations and operations described in previous chapters. The following sections thus describe the background and a particular solution used in this thesis to obtain valid polygons and planar partitions in §10.2, and valid polyhedra and 3D space partitions in §10.3. The chapter concludes with a generalisation of the definition of validity in arbitrary dimensions in §10.4, which can be used for both higher-dimensional objects and space partitions.

§10.2 is largely based on the papers:

**Validation and automatic repair of planar partitions using a constrained triangulation**. Ken Arroyo Ohori, Hugo Ledoux and Martijn Meijers. Photogrammetrie, Fernerkundung, Geoinformation 5, October 2012, pp. 613–630. ISSN: 1432–8364.
PDF DOI BibTeX

**Constructing an n-dimensional cell complex from a soup of (n-1)-dimensional faces**. Ken Arroyo Ohori, Guillaume Damiand and Hugo Ledoux. In Prosenjit Gupta and Christos Zaroliagis (eds.), Applied Algorithms. First International Conference, ICAA 2014, Kolkata, India, January 13-15, 2014. Proceedings, Lecture Notes in Computer Science 8321, Springer International Publishing Switzerland, Kolkata, India, January 2014, pp. 37–48. ISBN: 978–3–319–04125–4 (Print) 978–3–319–04126–1 (Online) ISSN: 0302–9743 (Print) 1611–3349 (Online).
PDF Slides DOI BibTeX

10.1 Motivation

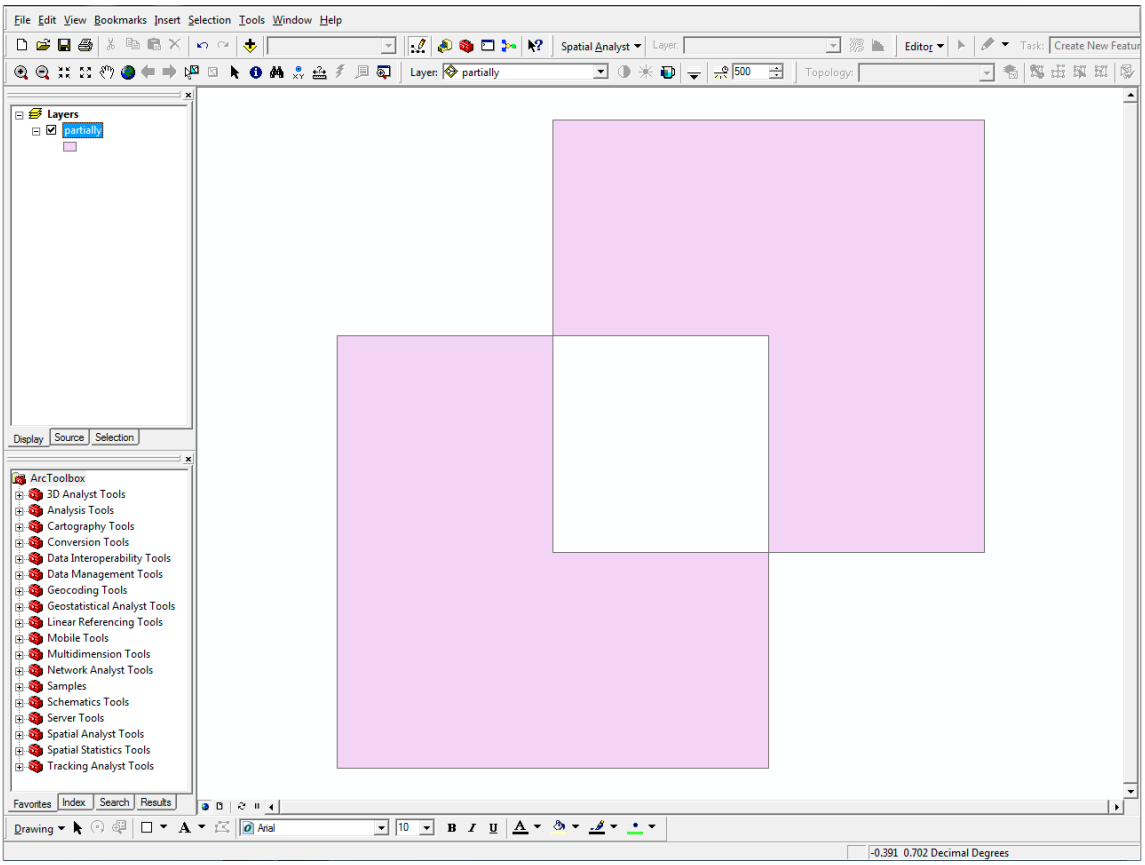
The representations described in **Part I** are each able to effectively represent a particular class of objects. For example, most data structures are intended for 3D space partitions whose 3D objects have surfaces that form 2-manifolds, and nD combinatorial maps are capable of representing subdivisions of orientable quasi-manifolds, which within this thesis are generally further limited to having only linear geometries.
Similarly, the operations described in **Part II** can only return good results when the input fulfils certain requirements. For instance, the extrusion operation from Chapter 6 requires the input data to form a space partition, the incremental construction algorithm from Chapter 7 requires the ridges of the facets of an object to form matching pairs (i.e. a quasi-manifold), and the linking approach from Chapter 8 will only form a valid 4D cell complex with 4-cells if a matching scheme that preserves all topological relationships between cells can be found or is provided. All these operations are also based on the assumption that the input cells themselves are valid, and are being completely bounded by valid lower-dimensional cells (i.e. facets, ridges, etc.) so as to form a valid cell complex.

The representations and operations presented in this thesis are not special in this sense. Validity assumptions are widely used in all software, especially when complex data is used as input, and are used to make many tasks more manageable, such as to interpret a dataset and load it into a particular data structure for internal usage, as well as for further operations that may be performed using this structure. However, GIS datasets are more complex than most other data[130], and GIS software thus tends to make more assumptions than most other software.

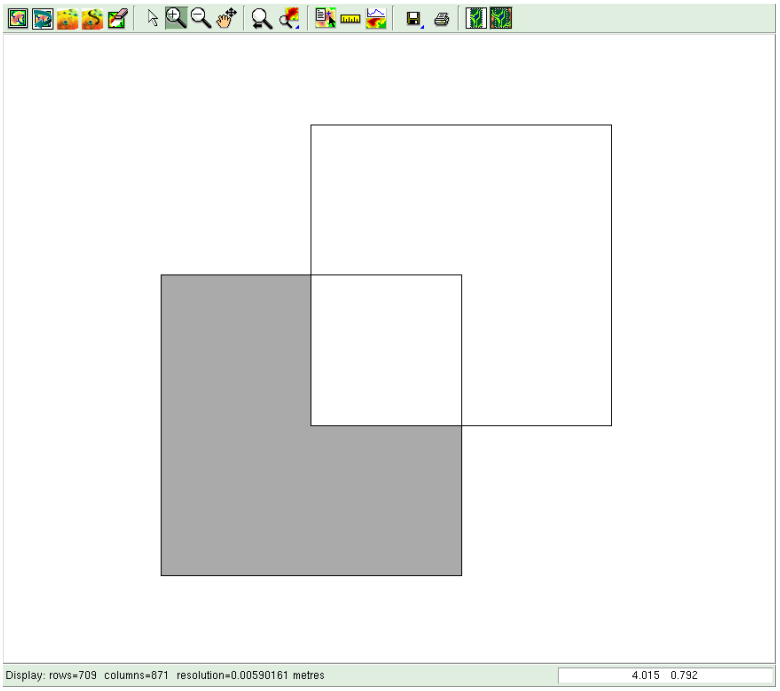
Moreover, though making sure that these assumptions are true is highly desirable, testing for every possible invalid configuration in a spatial dataset is cumbersome and often unnecessary for a particular task at hand, while testing for only some invalid configurations depending on what needs to be done can easily become intractable and can result in a large number of redundant tests. Running these validation tests can also be difficult and computationally expensive, as many tests take longer to execute than some of the common tasks that a GIS is used for (e.g. visualisation of a dataset, simple statistical analyses, or checking the attributes of some objects).

At the same time, the datasets found in the GIS world are very diverse and their properties vary significantly. They can be generated using a large variety of GIS, CAD and 3D modelling software based on different processes, complying to different specifications and stored in different formats, each of which follows its own internal logic and structure. Most importantly, GIS datasets are created for different purposes or meant for general-purpose applications (e.g. CityGML [Gröger et al., 2012]). As such, a dataset's specifications are often only vaguely defined, are defined only with regard for a particular application, or are not conformed with in practice.

It is thus perhaps unsurprising that invalid GIS datasets are prevalent [Panigrahi, 2014, Ch. 7] and a major source of problems for those who work with them. As shown in Figure 10.1, invalid datasets can be interpreted inconsistently in different software, leading to inconsistent or erroneous results. They can also make it impossible to perform a certain operation, either due to a failing precondition check or due to software crashes. A partial solution to this issue lies in validating these datasets, i.e. identifying the problematic objects in the data so that they can be discarded or (manually) fixed. There are a variety of checklists and (semi) automatic tools for this purpose, such as those provided by SAFE[131] and ESRI[132].




(a)




(b)

Figure 10.1: Different interpretations of a polygon with a hole that is partly outside its outer boundary (p3 in Figure 10.2). (a) ArcGIS133 considers the overlapping region as a hole, but the non-overlapping part of the hole as a new polygon (QGIS134 and FME135 do this

as well). (b) GRASS136 removes the overlapping part from the polygon, becoming a new polygon with a different shape. No warning is shown in any software.↩
For example, it is possible to incorporate a set of formal preconditions for every operation, as is done in the design by contract software engineering pattern [Meyer, 1986] or the Eiffel programming language [ISO/IEC, 2007]. However, simply discarding problematic (subsets of) datasets is not always feasible, as GIS datasets can be expensive to acquire and thus irreplaceable in practice, and manually fixing errors can be an extremely time-consuming process. In fact, according to McKenney [1998], users of 3D CAD models for finite element analysis—which has similar requirements as certain computations in GIS, such as well-shaped and non-overlapping mesh elements—spend up to 70% of their time fixing the input CAD models. While similar figures for GIS are to the best of my knowledge not available, it is worth noting that CAD software tends to produce better quality models than GIS software133.

A more complete solution therefore lies in using methods to automatically repair a dataset, i.e. to make it conform to a particular set of validity criteria, enabling the full use of many more datasets than would otherwise be possible. As this requires a rather complex defensive programming approach, testing for various types of partially overlapping cascading errors and fixing them accordingly, it is best performed separately and called as needed rather than integrated into every operation of a GIS. The following sections describe such independent repair methods as were used in this thesis, which involve the creation of valid (multi)polygons and planar partitions from 2D GIS datasets (§10.2), and the creation of valid polyhedra and 3D space partitions from 3D BIM datasets (§10.3). These were then used as input for the different experiments described in **Part II**.

## 10.2 Creating valid (multi)polygons and planar partitions

### 10.2.1 What is a valid (multi)polygon or planar partition?

In most GIS file formats and the software that reads and writes them, polygons and multipolygons are defined in a manner that is consistent with the definitions in the Simple Features Specification [OGC, 2011; ISO, 2006]—an implementation of the ISO 19107 standard [ISO, 2005a]. The specification states that: 'A `Polygon` is a planar `Surface` defined by 1 exterior boundary and 0 or more interior boundaries. Each interior boundary defines a hole in the `Polygon`'. Each of these boundaries is described as a `LinearRing` (cf. Figure 3.17). According to the specification, an outer ring should be oriented anticlockwise when viewed from a predefined top direction, which is generally (but not necessarily) the viewing direction in 2D or outwardswhen the polygon specifies part of the boundary of a polyhedron. Inner rings should be oppositely oriented, i.e. generally clockwise when viewed from the top direction.

The Simple Features Specification provides several **validity rules for polygons**, the most relevant of which are described below with examples of invalid polygons provided in Figure 10.2. The rules can be summarised as follows:

- each ring defining the exterior and interior boundaries is simple, i.e. non-self-intersecting ($p_1$ and $p_{10}$);
- each ring is closed ($p_{11}$), i.e. its first and its last points should be the same;
- the rings of a polygon do not cross ($p_3$, $p_7$, $p_8$ and $p_{12}$), but they may intersect at one tangent point;
- a polygon does not have cut lines, spikes or punctures ($p_5$ and $p_6$);
- the interior of every polygon is a connected point set ($p_4$);
- each interior ring creates a new area that is disconnected from the exterior ($p_2$ and $p_9$).

Similarly, the specification provides a definition and some **validity rules for multipolygons**. A `MultiPolygon` is defined as a `MultiSurface` forming an aggregation of `Polygons`, which also follows certain validity criteria, which can be summarised as follows:

- the interiors of its polygons do not overlap, i.e. their point set intersection should be empty;
- the boundaries of its polygons may only touch at a finite number of points;
- a multipolygon does not have cut lines, spikes or punctures;
- the interior of a multipolygon with more than one polygon is not a connected point set.

Intuitively, a planar partition is a set of polygons that form a subdivision of a region of the plane. Planar partitions are thus commonly used to model concepts where objects are expected not to overlap, such as land cover, cadastral parcels, or the administrative boundaries of a given country. Despite being a very frequently used representation in GIS, planar partitions are not explicitly defined in the main GIS standards.

Within the classes in the ISO 19107 standard [ISO, 2005a, §6.6], a planar partition could be considered as a `GM_CompositeSurface`, defined in the standard as 'a collection of oriented surfaces that join in pairs on common boundary curves and which, when considered as a whole, form a single surface'. By following this definition, overlaps between polygons are explicitly forbidden, as a `GM_Complex` (a parent of `GM_CompositeSurface`) is defined as 'a set of primitive geometric objects (in a common coordinate system) whose interiors are disjoint'. However, a `GM_CompositeSurface` explicitly allows gaps between the surfaces, as these would simply result in inner rings within the overarching single surface.

An alternative definition could be created based on the ISO 19123 standard [ISO, 2007, §6.8]—a standard focusing on coverages of various types. According to the standard, a planar partition can be considered as a type of `CV_DiscreteSurfaceCoverage` where 'the surfaces that constitute the domain of a coverage are mutually exclusive and exhaustively partition the extent of the coverage'. Overlapping polygons are disallowed by them being 'mutually exclusive' and gaps are disallowed by the surfaces 'exhaustively partitioning' the extent. However, the standard states these conditions as something that occurs 'in most cases', whereas in a planar partition it should be considered as a strict prerequisite.

In a **valid planar partition**, there should thus be no overlapping polygons, and no gaps between them either unless these gaps are considered to be outside of the region. These two conditions are covered by the ISO 19107 standard in a different context, when it lists some possible inconsistencies of 'spaghetti' datasets represented as a `GM_Complex`, stating that 'slivers and gaps are multiple lines that should represent the same geometry, but do not coincide, leaving areas of overlap between two surface boundaries (slivers), and gaps between them' [ISO, 2005a, §6.2.2.6].

### 10.2.2 Commonly used validation and repair methods

Starting from an arrangement of line segments that are meant to define the boundary of a (multi)polygon, there are various rules that can be used to define its interior and exterior. Foley et al. [1995] discusses two commonly used sets of rules in vector-based graphic software, which are shown in Figure 10.3[134].

In practice, GIS users often repair invalid polygons manually. Among the few documented automatic solutions, it is possible to use a 'buffer-by-0' operation [Ramsey, 2010] or PostGIS 2.0's `ST_MakeValid` function[139].

The validation of planar partitions is usually performed using a checklist of individual tests that together ensure its validity. For instance, Plümer and Gröger [1997] specify that a valid planar partition consists of: no dangling edges, no zero-length edges, planarity, no holes, no self-intersections, no overlaps, and having a connected graph. However, it is worth noting that without the use of a supporting structure, some of these tests can be problematic or computationally expensive. For instance, checking whether any possible pair of polygons overlap can have quadratic behaviour even when heuristics to speed up the process are used [Badawy and Aref, 1999; Kirkpatrick et al., 2000], and robustness issues are significant in polygon intersection tests [Hoffmann, 1988]. Finding the potential

gaps in a planar partition is also a problem, as it can require computing the union of the entire set of polygons [Margalit and Knott, 1989; Rivero and Feito, 2000].

The most common method used to repair a planar partition is based on the assumption that polygons approximately match each other at their common boundaries. If the adjacent polygons are within a certain distance threshold of each other along their common boundaries (Figure 10.4a), and all parts further apart than this threshold are known not to be common boundaries (Figure 10.4b), it is possible to snap together the polygons using this threshold, while in theory keeping the rest untouched. This method of planar partition repair is available in many GIS packages, including ArcGIS, FME, GRASS and Radius Topology[140].

The threshold value for certain input dataset(s) is then usually manually determined, either by trial and error, or by analysing certain properties of the datasets involved (e.g. point spacing, precision, or map scale). However, it is often hard to find an optimal threshold for certain datasets, and sometimes impossible as such a threshold does not even exist (e.g. because point spacing in some places might be smaller than the width of the gaps and overlaps present).

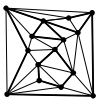
### 10.2.3 Repair using a constrained triangulation

The method developed to repair polygons and planar partitions uses a constrained triangulation of the input polygons as a base structure. Constrained triangulations have distinct properties that make them useful as a base for a repair algorithm. The can be built efficiently with a variety of approaches [Guibas and Stolfi, 1985; Clarkson et al., 1992], can easily be made numerically robust[141], can be used for quick traversal and point location [Mücke et al., 1999], and have fast and robust implementations in several libraries, such as CGAL [Boissonnat et al., 2002], Triangle[142] [Shewchuk, 1997] and GTS[143].

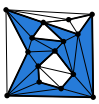
The method used to **repair individual (multi)polygons** exploits these properties and consists of three broad steps, which are shown in Figure 10.5 and described as follows:

1. construction of the constrained triangulation of the line segments in the input, processing outer and inner rings identically and including an extra edge that connects the first and last vertices of a ring when these are not the same (Figure 10.5b);
2. labelling of each triangle as either outside or inside, which is based on an extension of the odd-even rule that supports overlapping lines by adding or removing (parts of) constraints in the triangulation (Figure 10.5c), taking only edges that are constraints into account;
3. reconstruction of the interior areas as a repaired multipolygon[140] (Figure 10.5d).

(a)



(b)

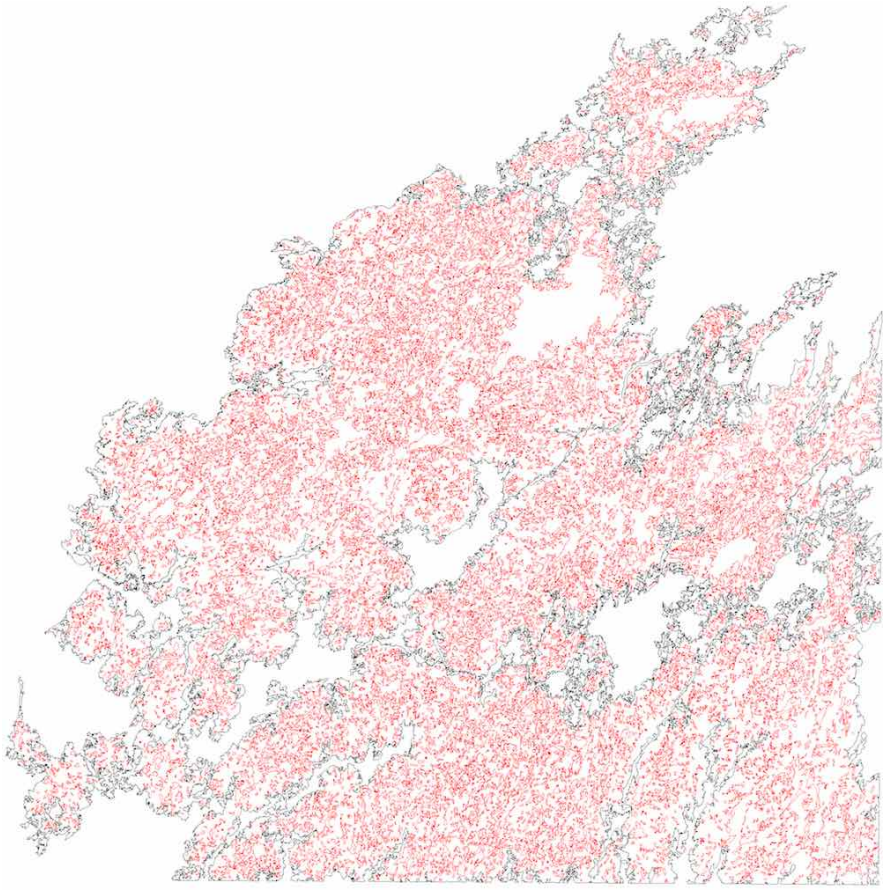


(c)

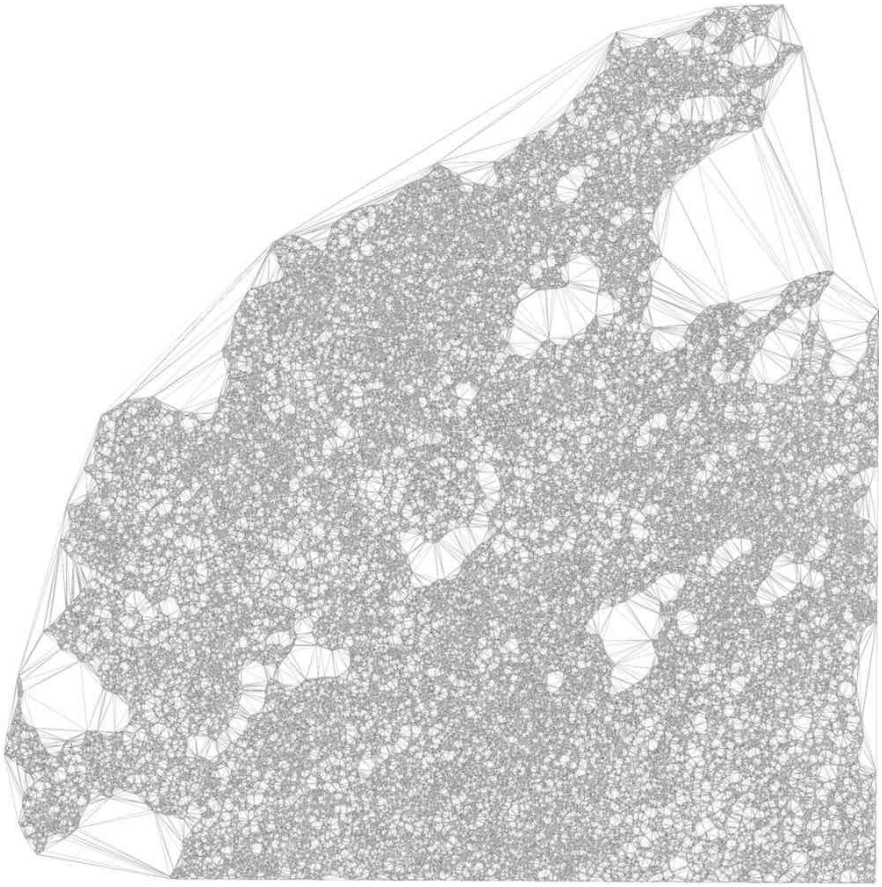
(d)

Figure 10.5: Steps to repair a (multi)polygon using a constrained triangulation: (a) input data, (b) triangulation, (c) labelling, and (d) reconstruction.↩

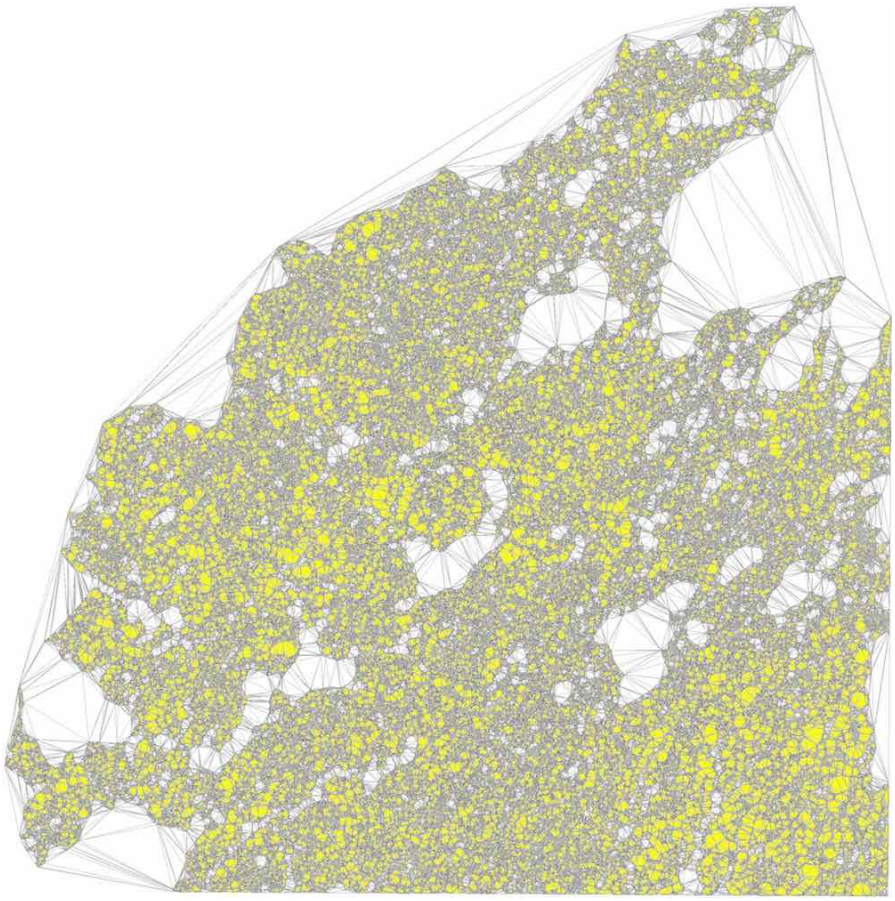
This method is remarkably efficient, and its implementation based on CGAL classes is able to process large polygons quickly. As an example, Figure 10.6 shows the process on the largest polygon in the CORINE land cover dataset[145], which consists of almost 1 189 903 vertices and 7 672 holes, which is processed in under a second.



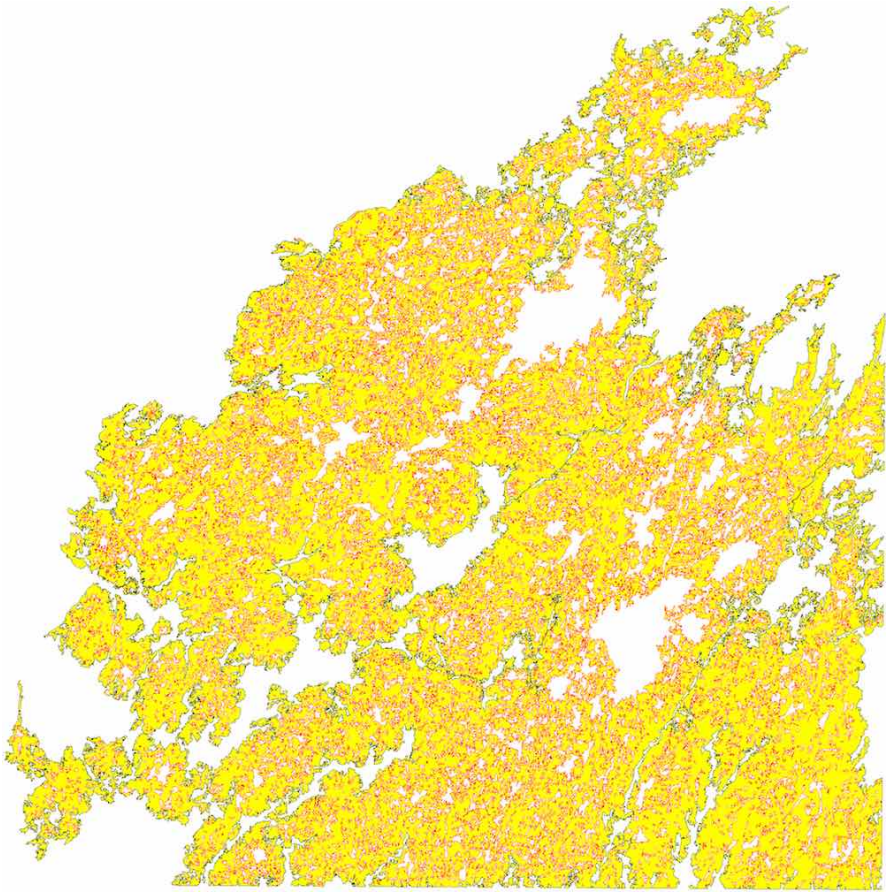
(a)



(b)



(c)

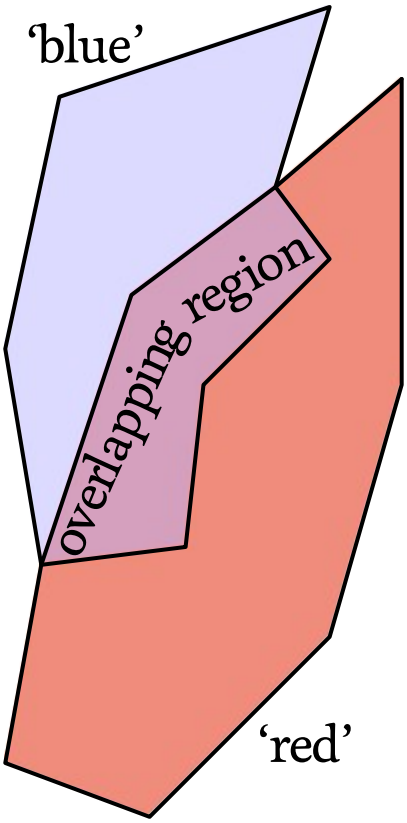


(d)

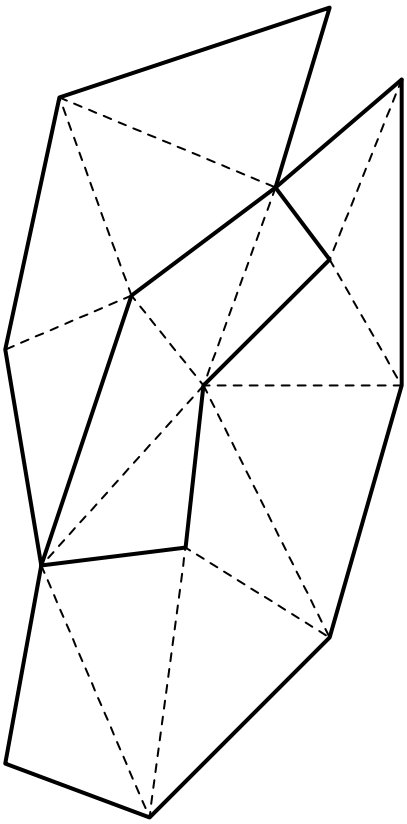
Figure 10.6: Processing the largest polygon in the CORINE land cover dataset: (a) outer and inner boundaries, (b) triangulation, (c) labelling, (d) reconstruction.

The method to **repair a planar partition** then uses polygons which are known to be valid based on the previous method. It consists of four main steps, shown in Figure 10.7, and is as follows:

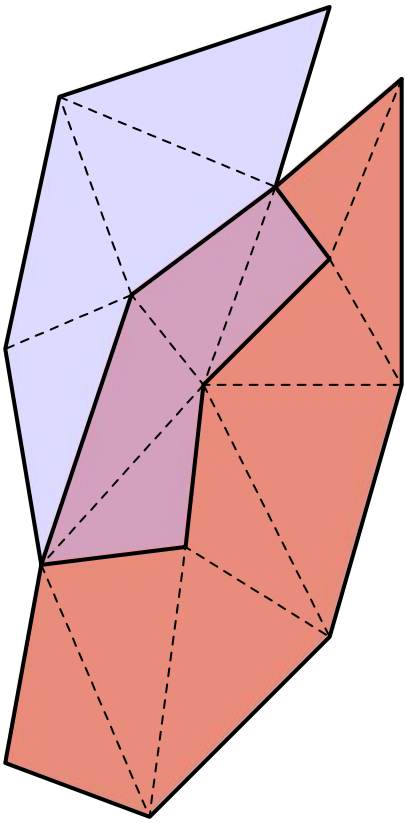
1. the constrained triangulation of the input segments forming the (now valid) polygons is constructed;
2. each triangle is labelled with the labels of the polygons inside which it is located, such that problems are detected by identifying triangles having no or multiple labels;
3. problems are fixed by re-labelling triangles according to customisable criteria, such that each triangle has exactly one label;
4. the polygons are reconstructed from the triangulation.



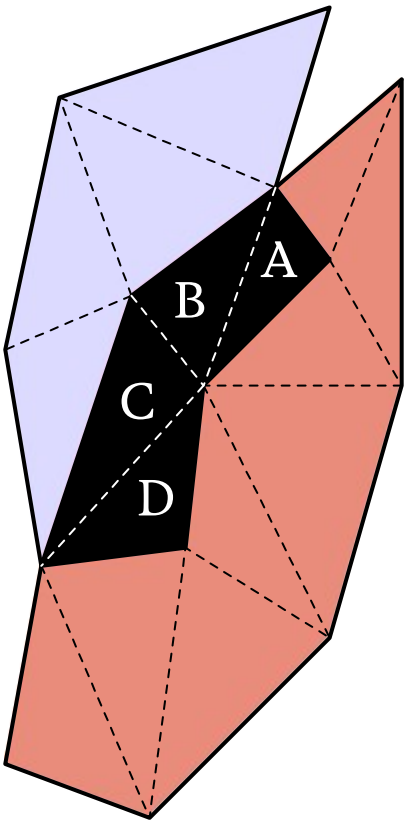

(a)



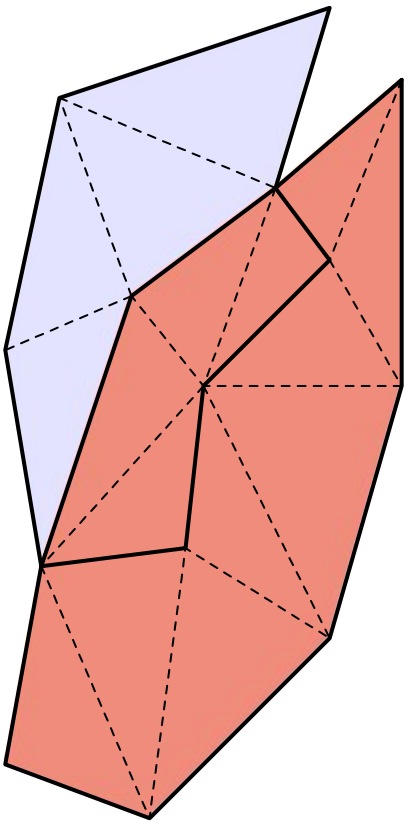
(b)



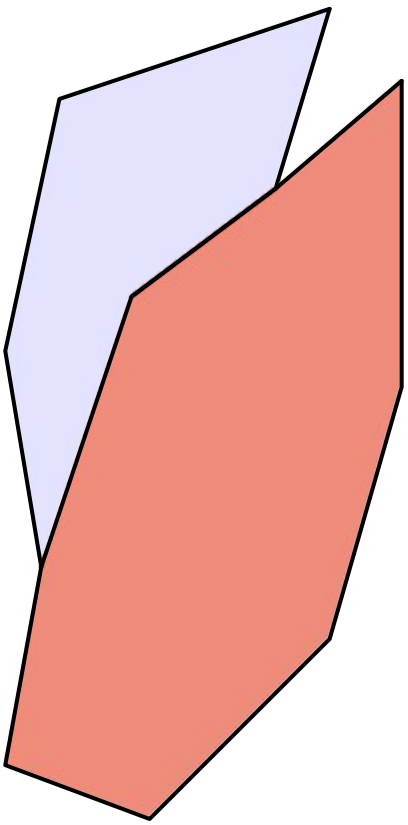
(c)




(d)



(e)



(f)

Figure 10.7: Steps to repair a planar partition using a constrained triangulation: (a) input data, (b) triangulation, (c) labelling, (d–e) relabelling problematic triangles, (f) reconstruction. ↩

Various local repair methods can thus be defined, all of which are based on choosing how to relabel a triangle which no label or with multiple labels. Figure 10.8 shows a few examples of such methods. Within this thesis, planar partitions are obtained by repairing invalid regions using the longest boundary with a neighbour (Figure 10.8c) if possible, as this tends to produce a cartographically more pleasing result, and a random neighbour (Figure 10.8d) otherwise.

(a)

(b)

(c)

(d)

Figure 10.8: Various repair methods can be defined based on relabelling triangles or sets of connected triangles. For instance, based on (a) the input data, it is possible to relabel: (b) an invalid triangle based using its longest boundary with a neighbour, (c) an invalid region of connected triangles using its longest boundary with a neighbour, or (d) an invalid region of connected triangles using a random neighbour.

This method is able to process large datasets quickly, such as the one shown in Figure 10.9, which consists of 16 tiles of the CORINE land cover dataset in a 4×4 configuration. It has 63 868 polygons with a total of 6 622 133 vertices and was processed in 4 minutes 47 seconds. By comparison, both ArcGIS and GRASS are unable to repair this dataset by snapping geometries, while FME repairs it in 15 minutes 48 seconds[146], also using snapping. Note that apart from the fact that snapping is slower, it does not guarantee a valid result.

### 1.3.2 Delaunay Triangulation with PostGIS

Quelle: http://postgis.net/docs/ST_DelaunayTriangles.html

## 1.4 Difference

Alle Polygone vereinen und um dessen maximal einschließendes Rechteck MER einen 1m Puffer legen als Außenbox.
Die vereinten Polygone vom Außenbox abziehen und davon den Geometrieteil als Außenpolygon extrahieren, der das gleiche MER hat wie die Außenbox. Das Außenpolygon von der Außenbox abziehen. Das verbleibende Innenpolygon um 0.1m nach außen und wieder mit -0.1m nach innen puffern. Das entstehende neue Innenpolygon von Außenbox abziehen und so ein verbessertes Außenpolygon erzeugen.

## 1.5 Aufbereitung der Geometrien mit pg_topolgy

- Löschen doppelte Punkte innerhalb eines Abstandes mit outer_poly.
- Erzeugen einer Toplogy mit einer Tolleranz
- Extrahieren der Geometrien der Faces der Topology als neue Polygonen
- Zuordnen der Sachdaten zu den neuen Geometrien durch Verschneidung
- Zuordnen von Lücken zu Nachbarflächen durch GeomUnion

# 2 Umsetzung mit Verschneidung

## 2.1 Ausgangstabelle vorbereiten und Hilfsfunktionen

- Oids einstellen
- Spalte geom_neu anlegen
- Funktion zum Löschen von kleinen inneren Ringen (filter_rings) erstellen
- Daten aufbereiten mit
  - ST_Transform to metric system
  - ST_SimplifyPreserveTopology 0,001
  - ST_MakeValid
  - ST_CollectionExtract 3
  - filter_rings 0.01
  - ST_Mulit

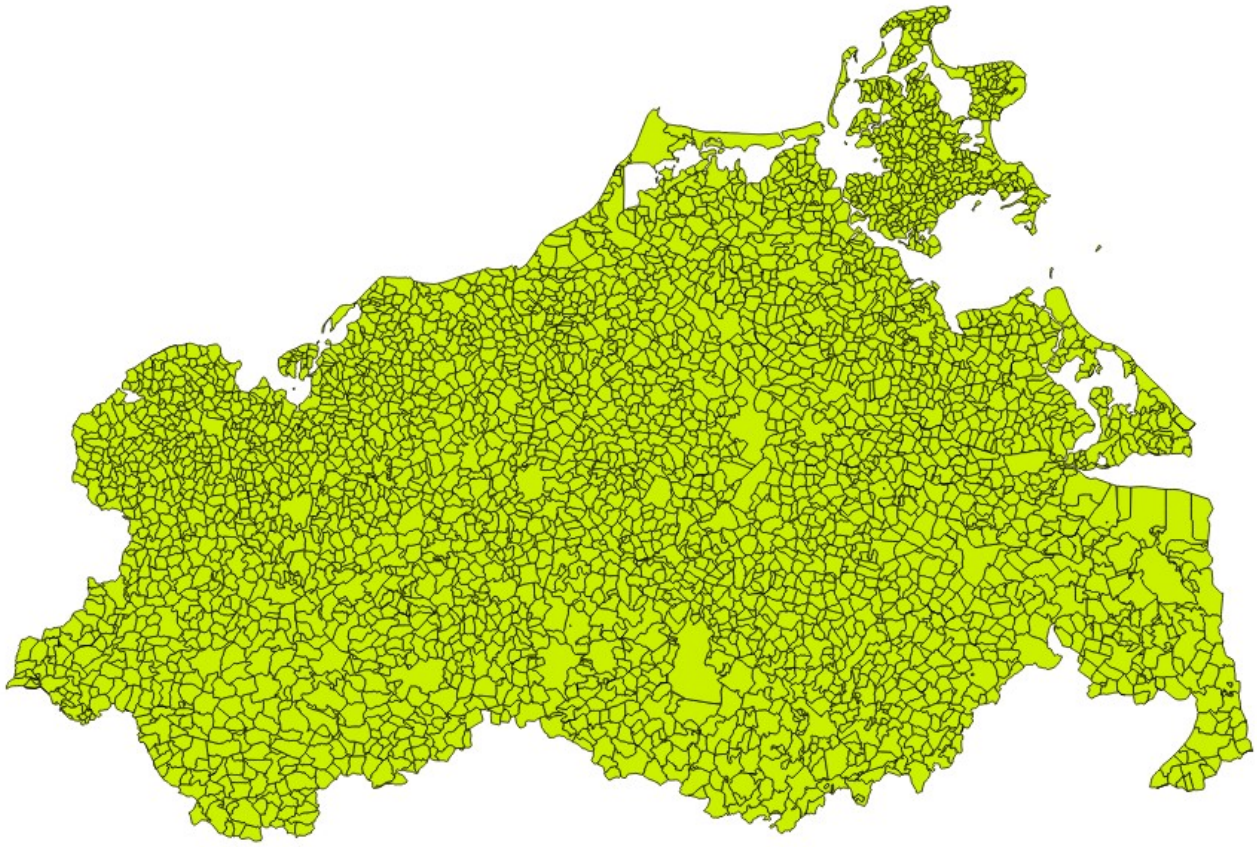


**Abbildung 1: Ausgangsdatensätze**

## 2.2 outerbox

- Spalte outerbox anlegen
- update outerbox = box2d(geom_neu)

## 2.3 Tabelle für Aggregationen anlegen

- Spalte id als serial
- SELECT addgeometrycolumn('public', 'gemeinden_mv_agg', 'overlap', 25833, 'MultiPolygon', 2);
- SELECT addgeometrycolumn('public', 'gemeinden_mv_agg', 'mer_diff', 25833, 'MultiPolygon', 2);
- SELECT addgeometrycolumn('public', 'gemeinden_mv_agg', 'outer_poly', 25833, 'Polygon', 2);
- SELECT addgeometrycolumn('public', 'gemeinden_mv_agg', 'inner_poly', 25833, 'MultiPolygon', 2);
- SELECT addgeometrycolumn('public', 'gemeinden_mv_agg', 'outer_conc', 25833, 'MultiPolygon', 2);
- SELECT addgeometrycolumn('public', 'gemeinden_mv_agg', 'mer_agg', 25833, 'MultiPolygon', 2);

## 2.4 Außenpolygon berechnen

- Berechne Rechteck der gesamten Ausdehnung aller Polygone (mer)

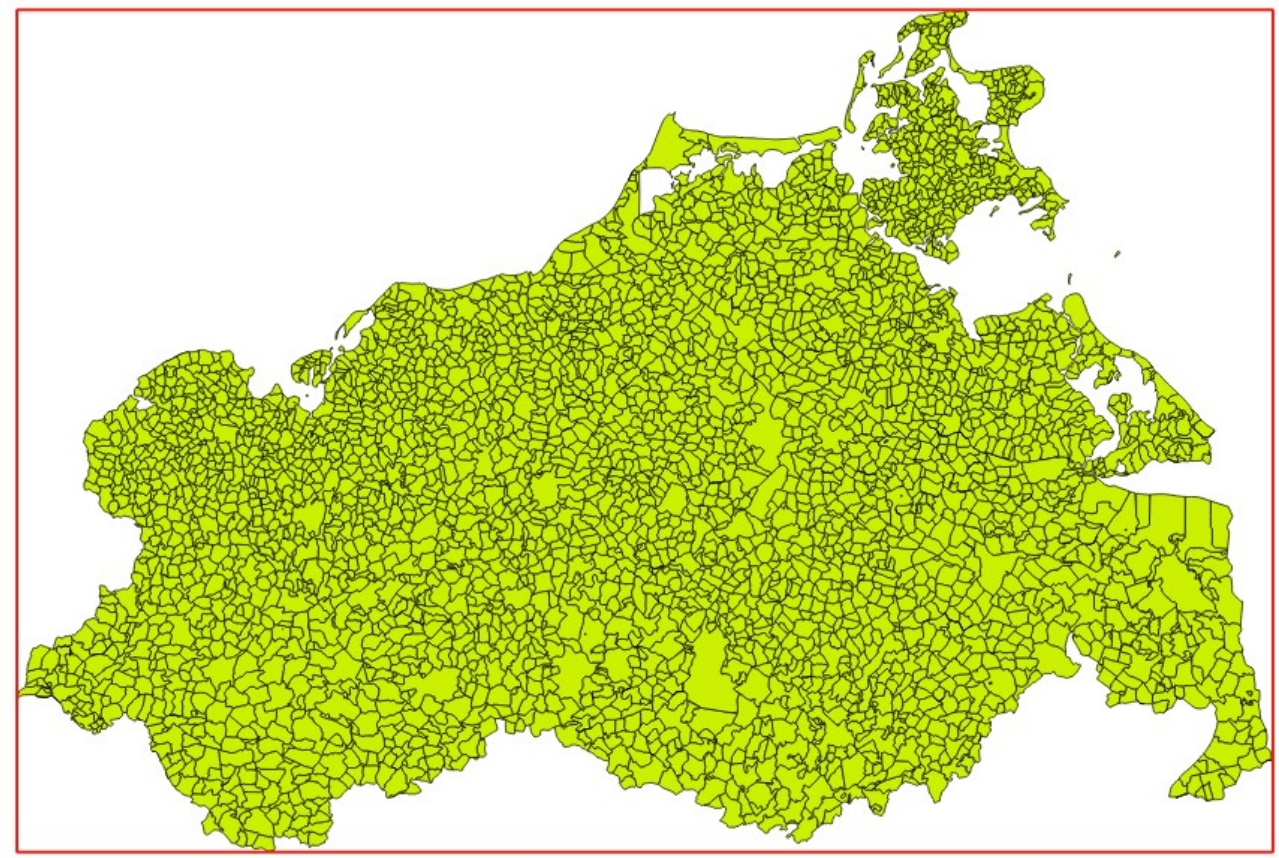


**Abbildung 2: Begrenzende Box mit Puffer (MER)**

- Ziehe vom mer alle boxen der Polygone ab und schreibe in outer_mer_hull

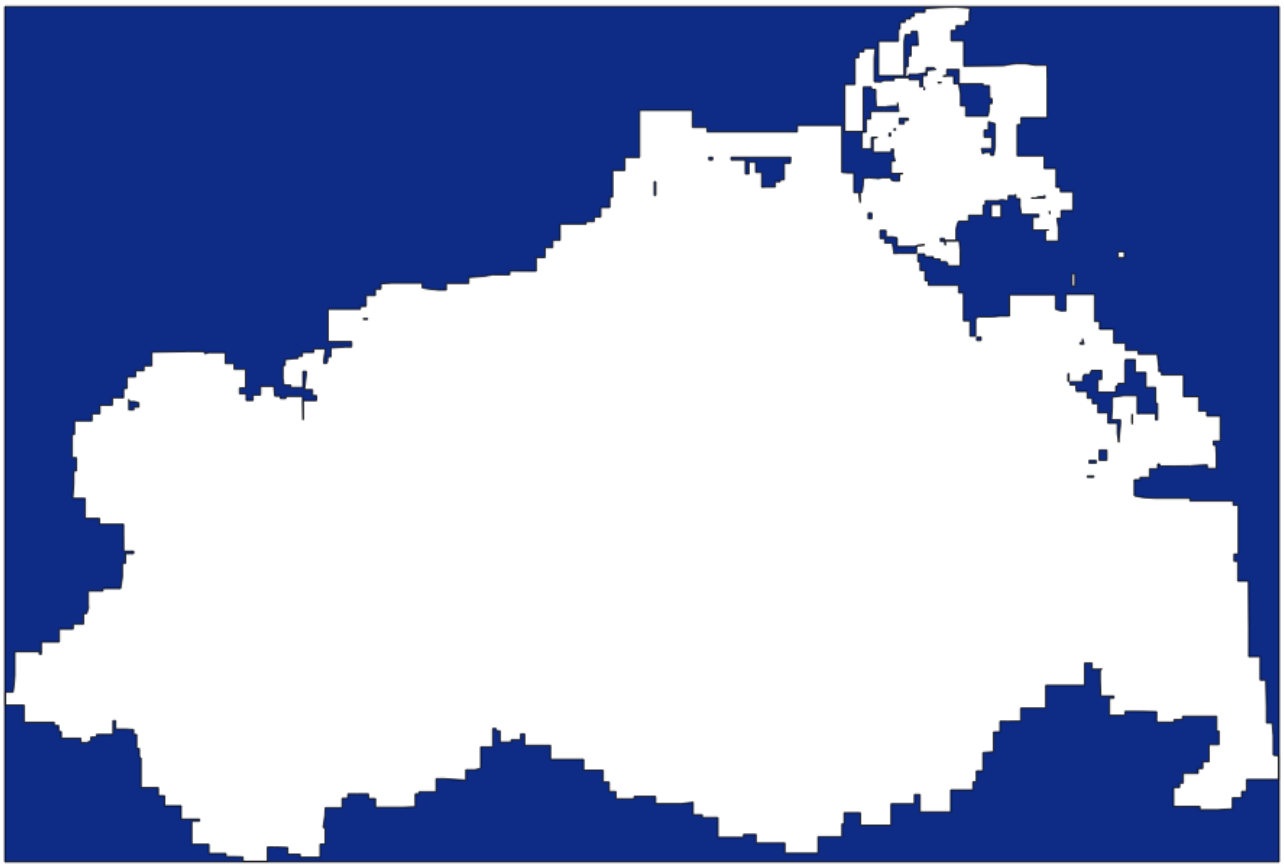


**Abbildung 3: MER abzüglich aller Boundingboxen der Datensätze**

- Markiere alle Polygone, deren Boxen am äußeren Rand des outer_mer_hull liegen in is_border_geom
- Markiere zusätzlich alle Polygone, die sich mit den Boxen der vorher markierten schneiden

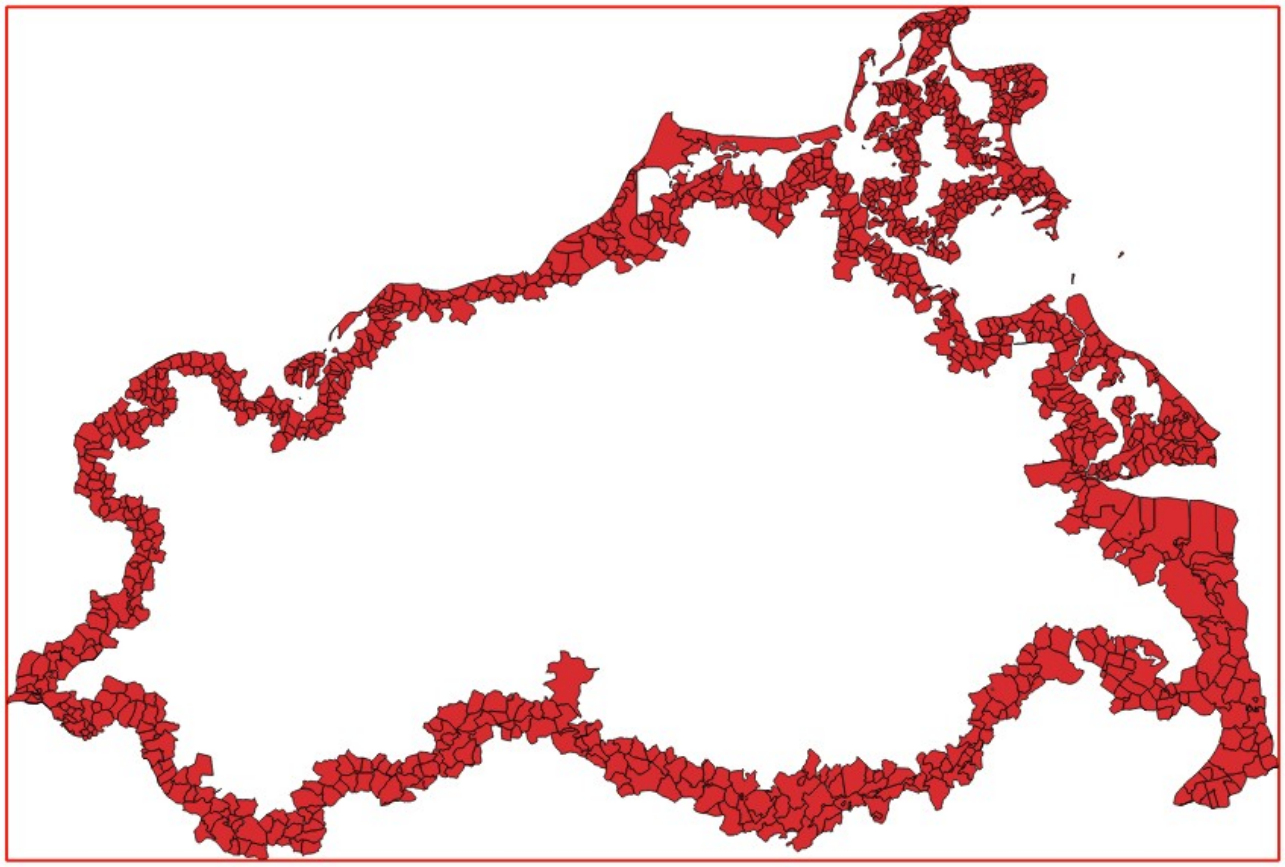


**Abbildung 4: Alle Polygone, dessen MER an das zuvor berechnete Außenpolygon grenzen + deren Nachbarflächen**

- Aggregiere alle die bisher is_border_geom sind und speicher in border_geom

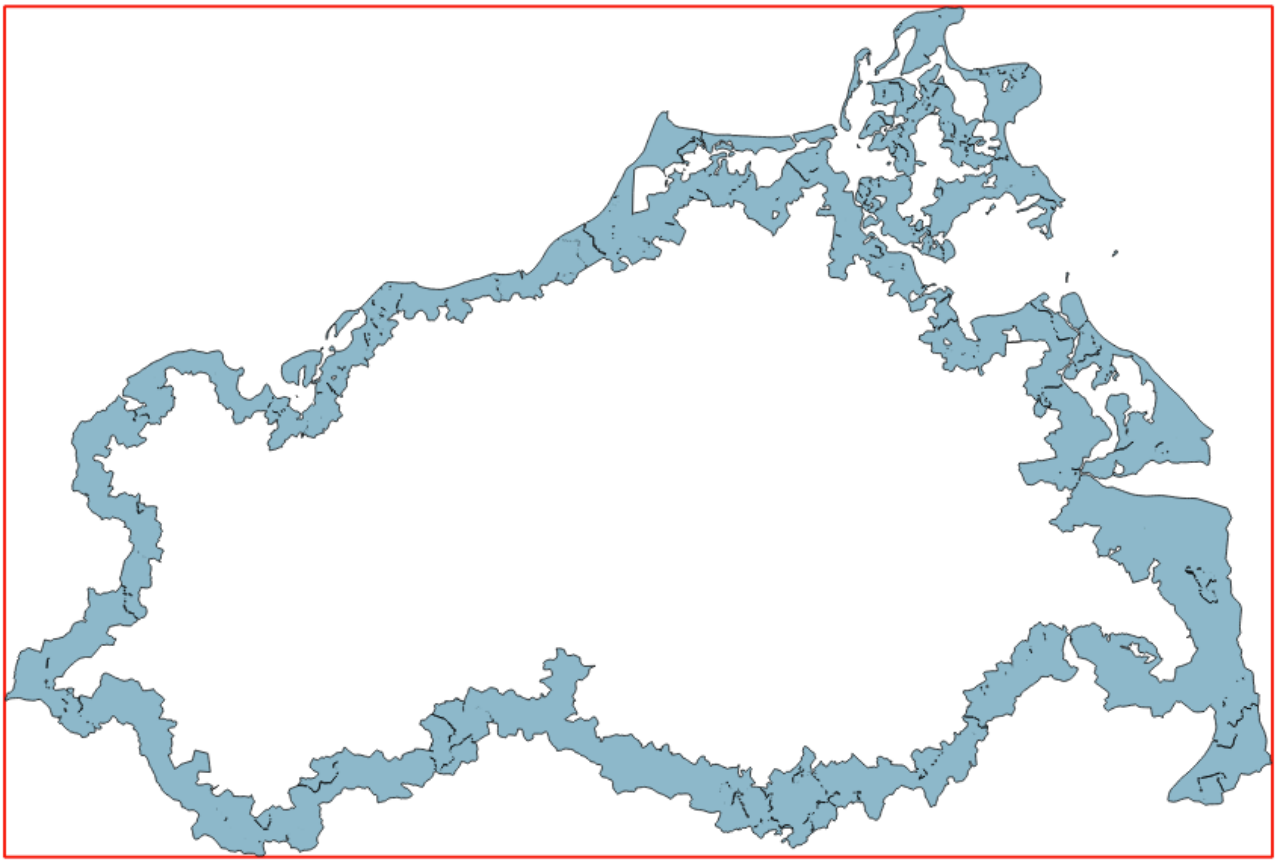


**Abbildung 5: Aggregation dieser am Rand liegenden Polygone**

- Lösche Einschnitte durch Pufferung und Repufferung

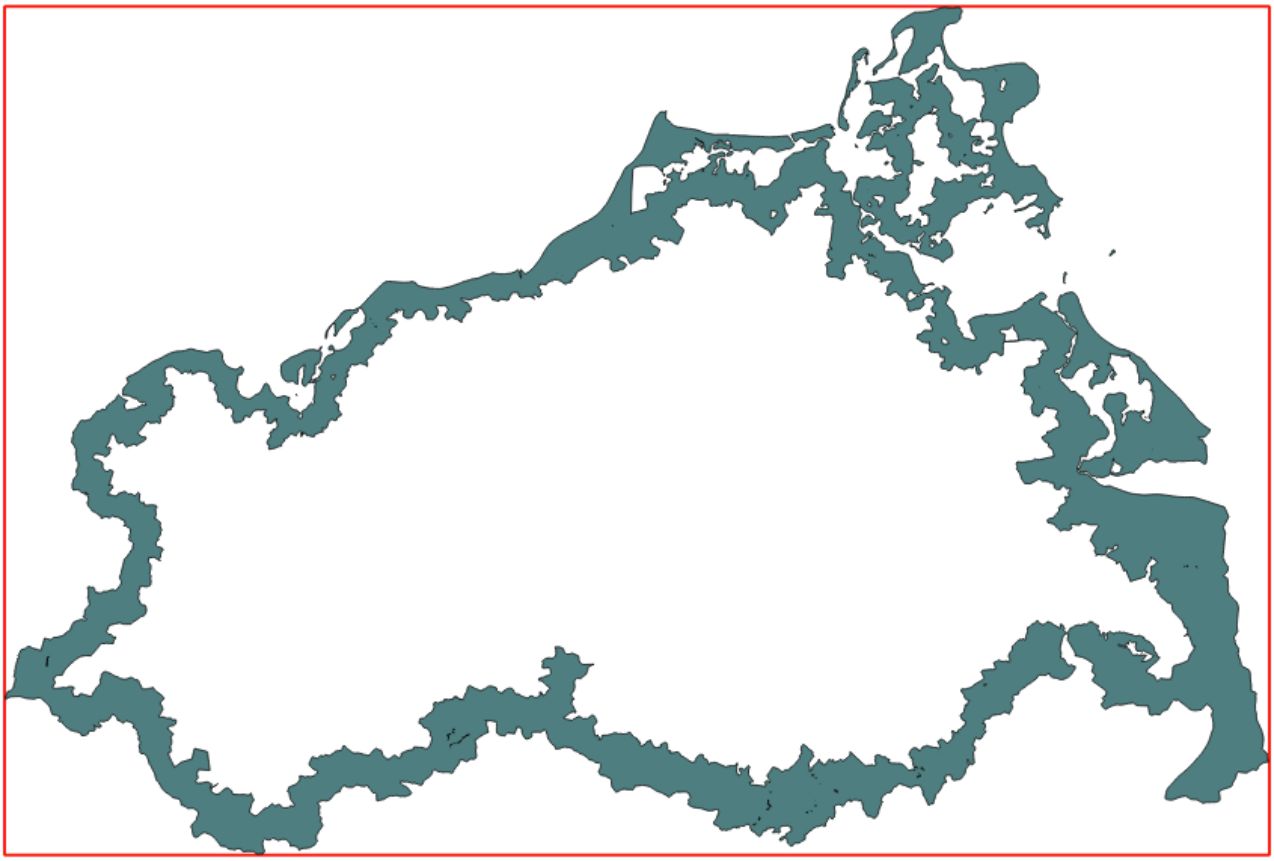


**Abbildung 6: Reduktion der Löcher und Einschnitte durch Puffer und Repuffer**

- Ziehe border_geom vom mer ab

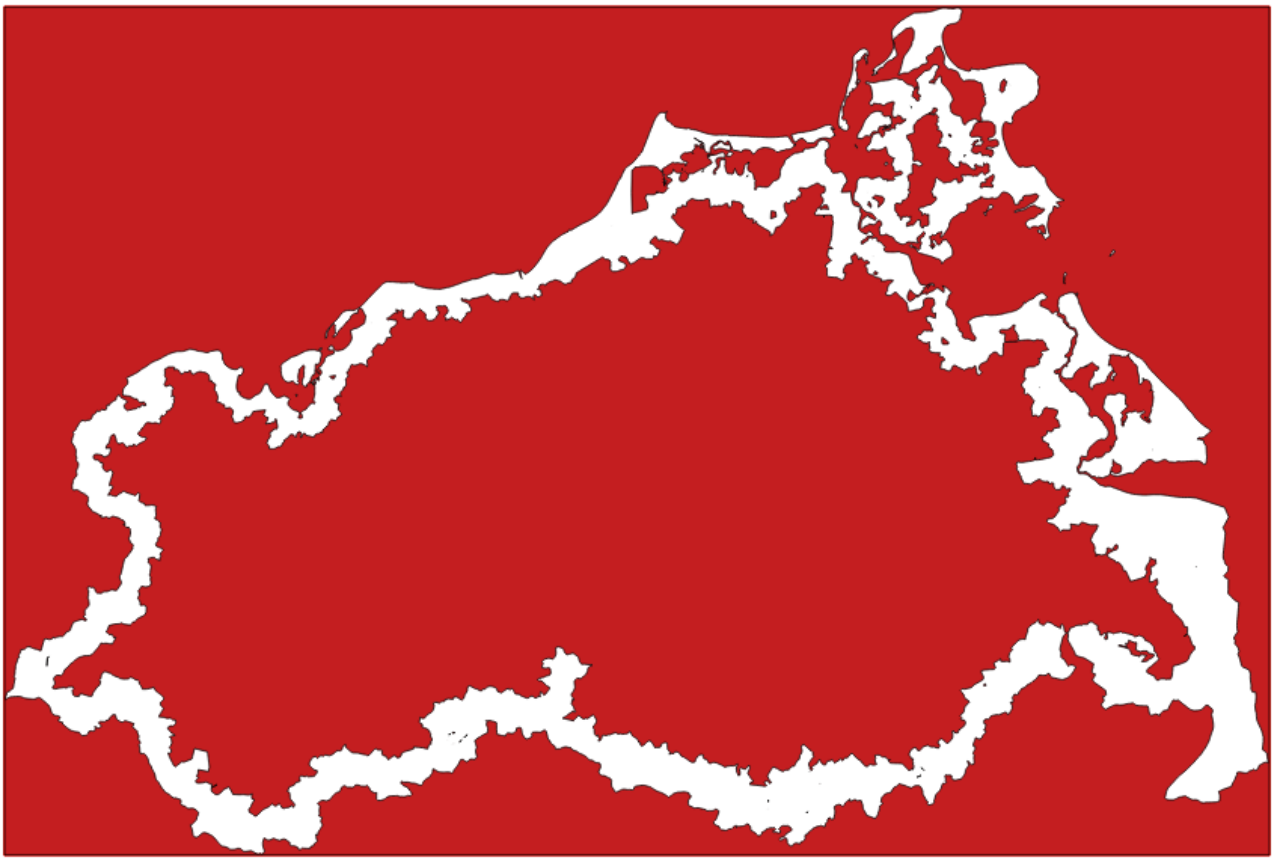


**Abbildung 7: MER abzüglich der aggregierten Außenpolygone**

**Abbildung 8: Ausschnitt, der Zeigt, dass das Außenpolygon geschlossen ist**

- Extrahiere die äußeren Ringe als outer_poly

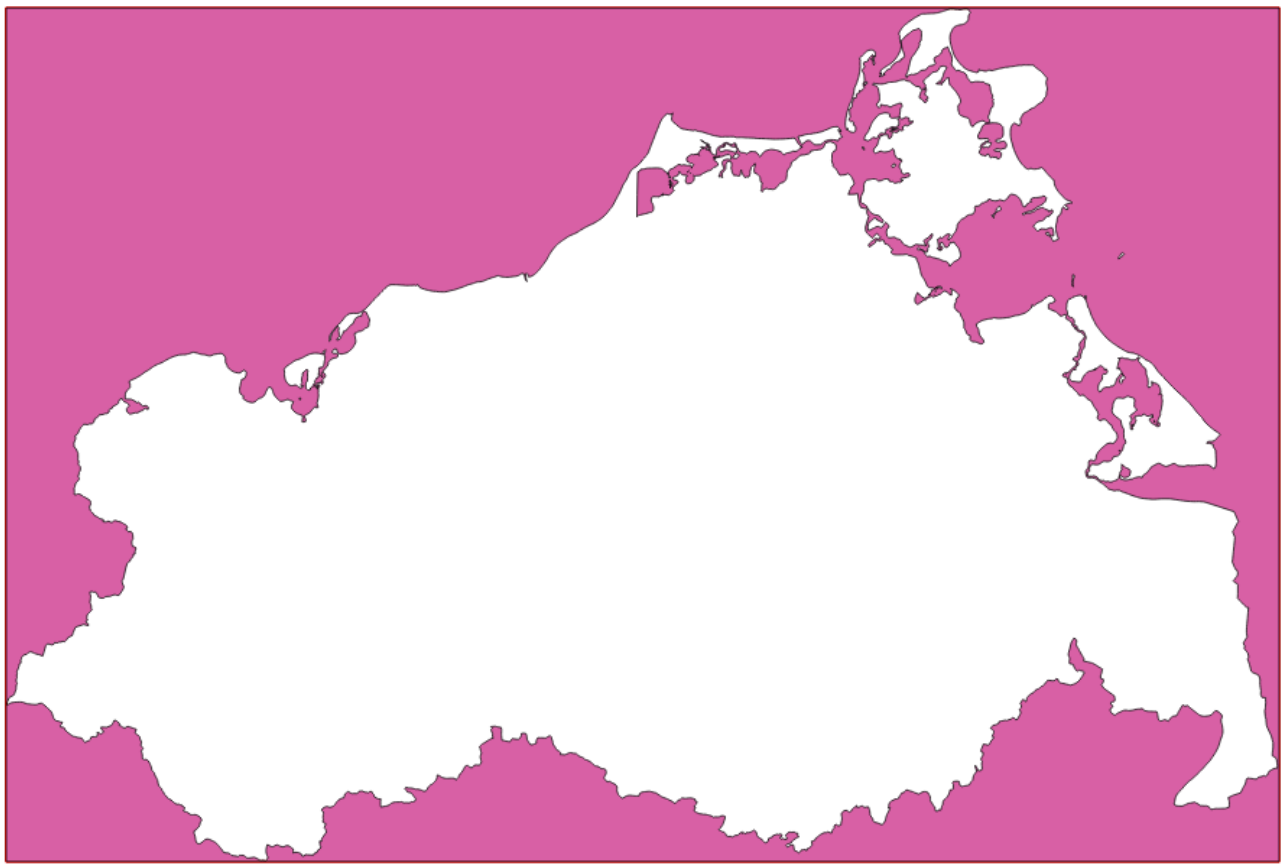


**Abbildung 9: Außenpolygon**

- Zerteile das Außenpolygon in kleinere Flächen, damit nicht immer das ganze Außenpolygon verwendet werde muss.

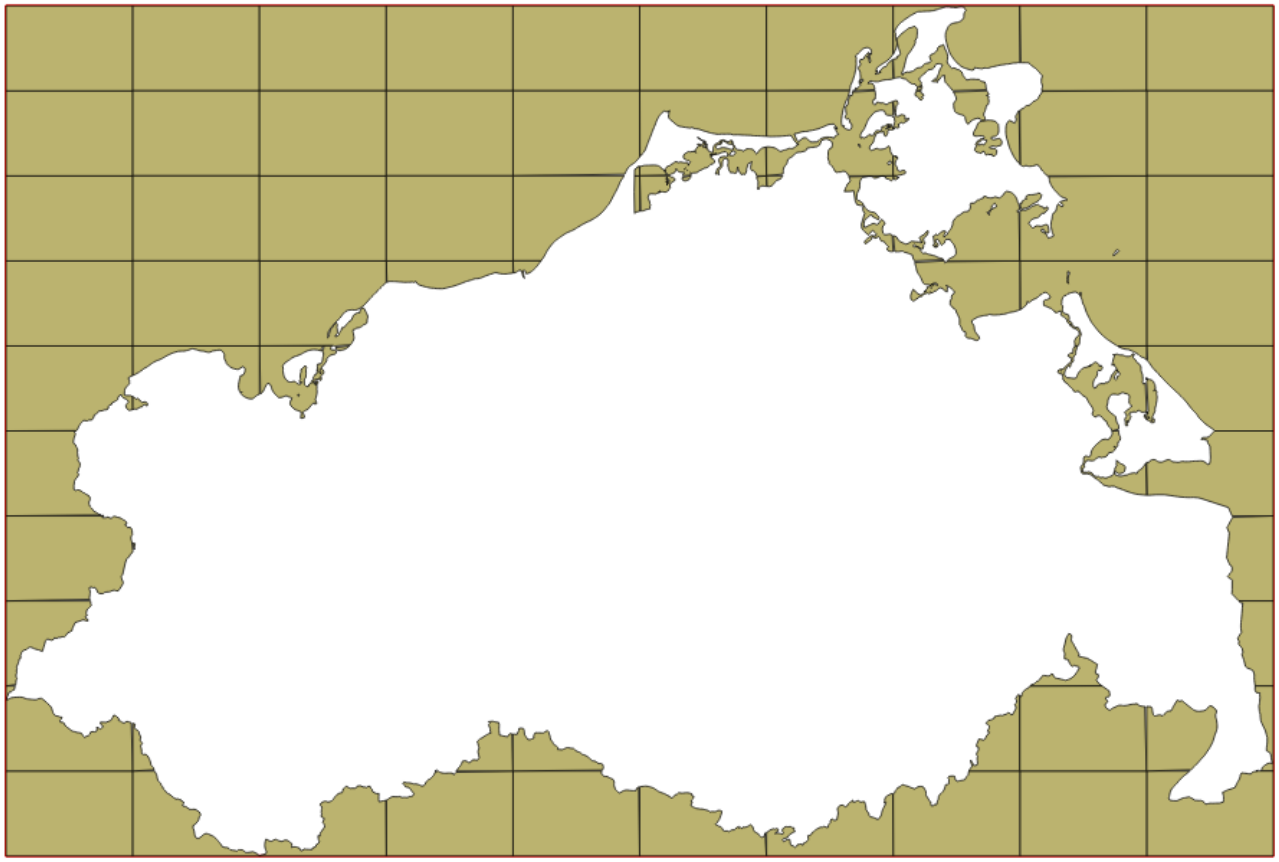


**Abbildung 10: Zerteiltes Außenpolygon**

- Invertiere um die Gesamtfläche als inner_poly zu bekommen (wird aber eigentlich nicht gebraucht. Zeigt nur wie der innere Bereich aussieht und kann ggf. hinterher zum Flächenvergleich genutzt werden.

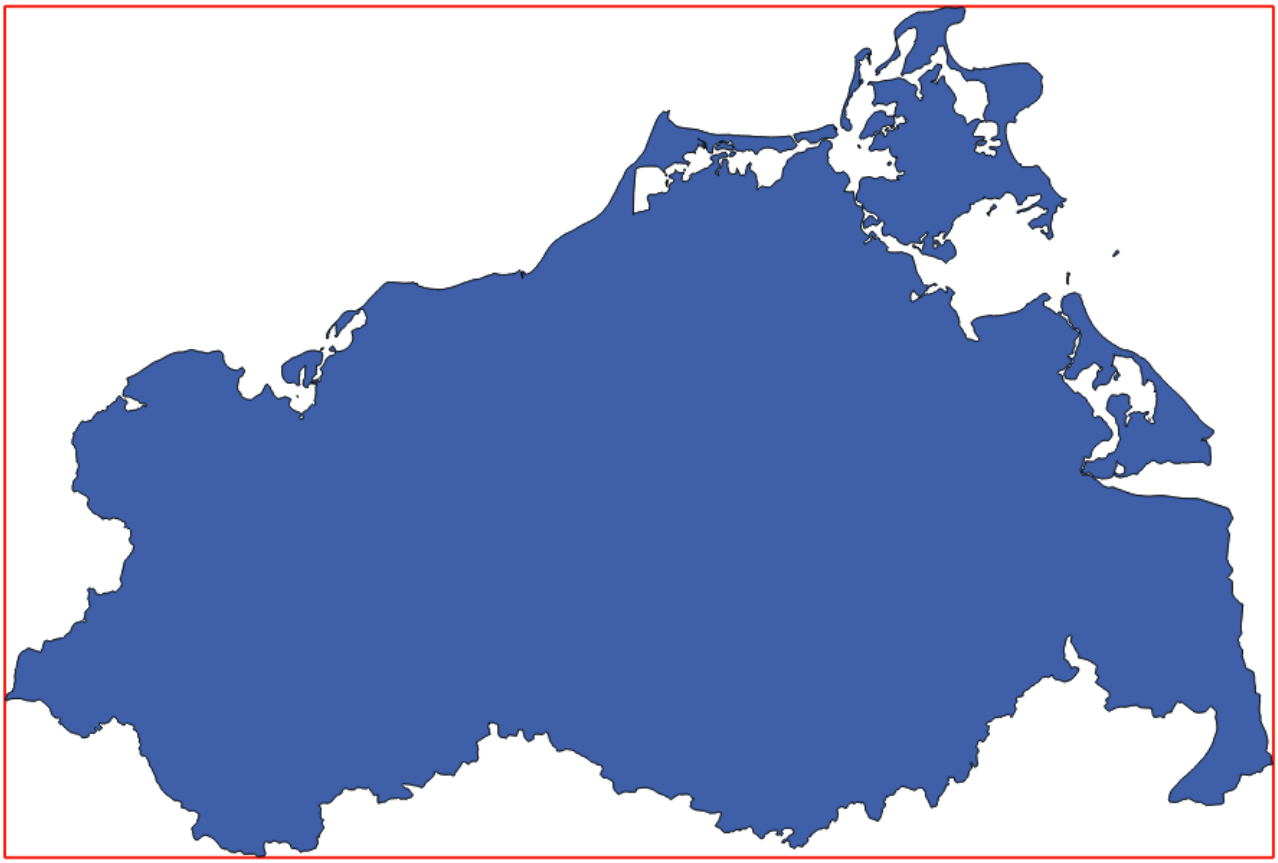


**Abbildung 11: Innenpolygon**

## 2.5 Funktion zum Updaten der Flächen

- Für alle Flächen durchlaufe:
- Berechne Polygon aller bis 100 m entfernten Nachbarflächen als umschließendes Polygon, Umschließendes Polygon schließen durch Verschnitt mit Box und Bildung eines outerpoly
- Extrahiere alle inneren Ringe, die sich mit der ursprünglichen Fläche des aktuell behandelten Polygons überlappen und bilde daraus ein Multipolygon.
- Verwende diese Geometrie als neue Geometrie in geom_neu und setze is_topo_poly_geom auf true

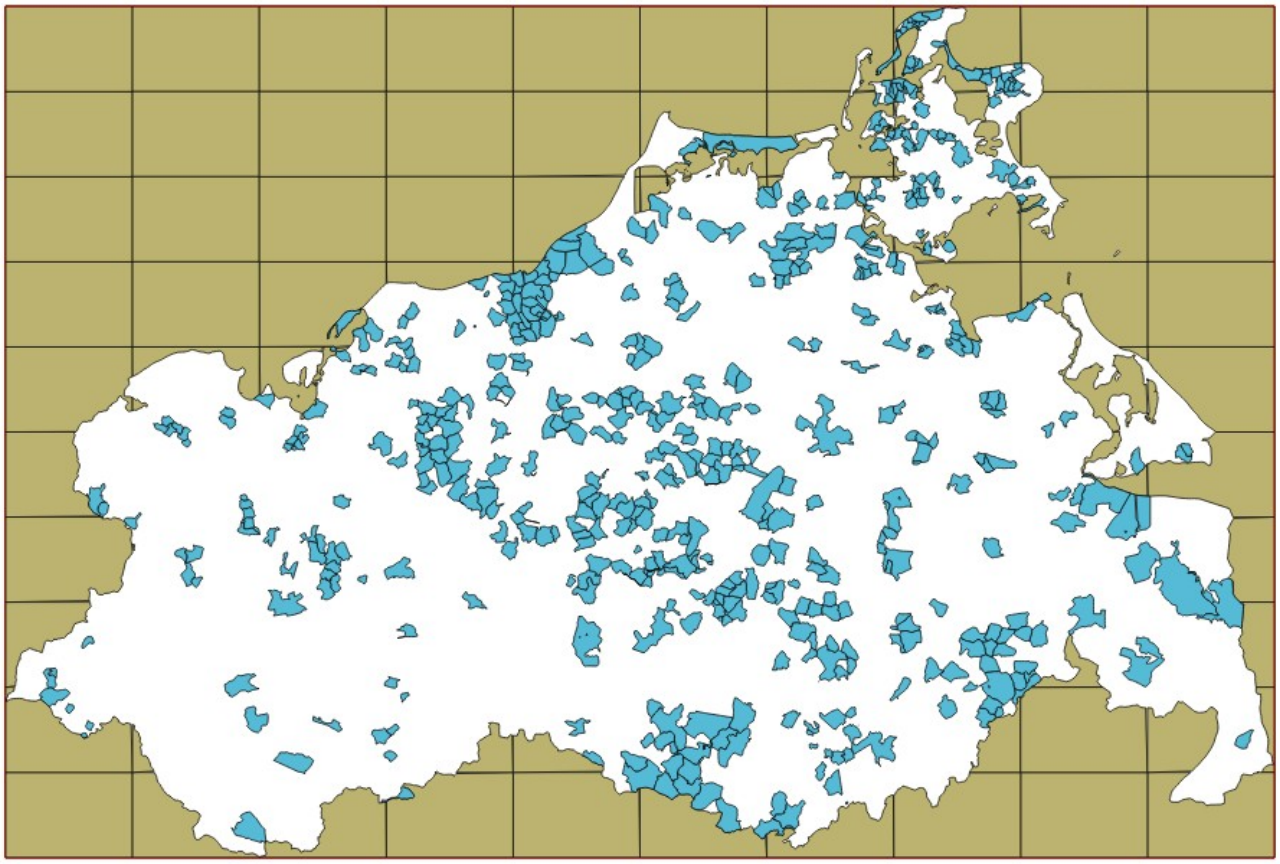


**Abbildung 12: Stand nach einem Durchlauf zur Berechnung der topologisch korrekten Flächen**

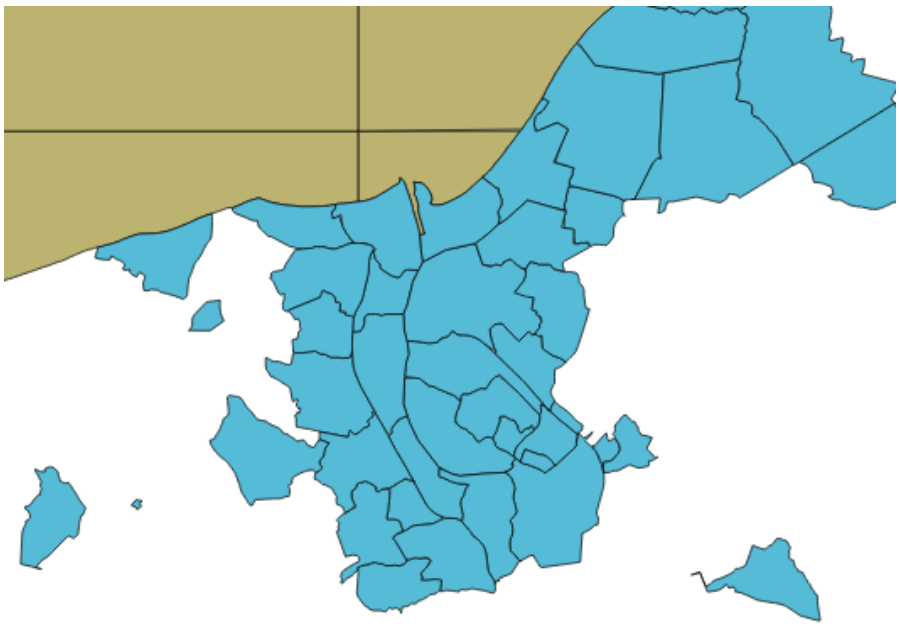


**Abbildung 13: Ausschnitt zum Stand um Rostock**

Die Funktion bricht ab mit Meldung, dass Union nicht durchgeführt werden kann wegen None-nuded intersection oder self-intersection. Schwer nachvollziehbar, weil es immer an bei irgendwelchen Aggregationen von benachbarten Polygonen passiert. Zur Analyse Attribut is_topo_geom eingeführt, damit

man sieht welche schon berechnet wurden und dann test mit einzelnen fortführen. Selbst dann ist Fehler beim Aggregiern mit union manchmal nicht plausibel.
Lösungen:

- Finde eine Lösung bei der immer garantiert beliebige Flächen Aggregiert werden können
- Aggregiere nicht vorher, sondern finde eine Lösung bei der die Nachbarflächen nach und nach von der zu behandelnden Fläche abgezogen werden. Bilde dazu ein Puffer der behandelten Fläche.

# 3 Umsetzung mit Topologie

## 3.1 Vorverarbeitung

Falls noch nicht vorhanden muss in der Datenbank die Topologieerweiterung eingeschaltet werden.

```
CREATE EXTENSION postgis_topology;
```

Dann wird eine leeren Topologie Tabelle angelegt.

```
SELECT topology.CreateTopology('topo', 25833, 2);
```

Diese ist anschließend in der Tabelle topology angelegt und es existiert ein Schema topo mit den Tabellen edge_data, face, node und relation.

| id<br>[PK] serial | name<br>character varying | srid<br>integer | precision<br>double precision | hasz<br>boolean |
|---|---|---|---|---|
| 14 | topo | 25833 | 10 | FALSE |

Bevor Geometrien in Topologien umgewandet werden können, müssen die Multipolygone aufbereitet und in Polygone aufgetrennt werden, damit es beim Erzeugen der Topologie möglichst keine Fehler gibt.

```
CREATE TABLE gemeinden_ausschnitt_poly AS
SELECT
 f.gid,
 ST_GeometryN(
  f.geom,
  generate_series(
   1,
   ST_NumGeometries(geom)
  )
 ) AS geom
 FROM
 (
  SELECT
   gid,
   ST_SimplifyPreserveTopology(
    ST_CollectionExtract(
     ST_MakeValid(
      ST_Transform(
       ST_GeometryN(
        geom,
        generate_series(
         1,
         ST_NumGeometries(geom)
        )
       ),
       25833
      )
```

```
        ),
        3
      ),
      0.1
    ) AS geom
  FROM
    gemeinden_ausschnitt
) f
ORDER BY gid
```

Die Geometrie wird in folgenden Schritten aufbereitet und in eine neue Tabelle mit dem Postfix _poly gespeichert.

- Auftrennen der Multipolygone in einzelne Polygone mit generate_series und ST_GeometryN
- Transformieren nach UTM Zone 33 um ein metrisches System zu haben (Die Toleranz der Topology wird im folgenden in Metern angegeben.)
- Reparieren invalider Polygone mit ST_MakeValid
- Extrahieren der dabei entstehenden Polygone mit ST_CollectionExtract
- Vereinfachen der Geometrien mit ST_SimplifyPreserveTolerance und dem Betrag distance tolerance
- Auftrennen der beim Reparieren neu entstandenen Multipolygone ebenfalls wieder mit generate_series und ST_GeometryN

Das Ergebnis ist eine Tabelle mit allen Polygonen und die jeweils dazugehörige gid. In dieser Tabelle wird nun eine Topology-Spalte hinzugefügt. Darin wird die Verknüpfungen zu den Topologie-Elementen hinterlegt, wodurch die Geometrien in Beziehung mit deren Topologien gesetzt werden können.

```
SELECT AddTopoGeometryColumn('topo', 'public', 'gemeinden_ausschnitt', 'geom_topo', 'Polygon');
```

Dieser Befehl legt auch einen Layer in der Tabelle topoglogy.layer an.

| topology_id [PK] integer | layer_id [PK] integer | schema_name character varying | table_name character varying | feature_column character varying | feature_type integer | level integer | child_id integer |
|---|---|---|---|---|---|---|---|
| 14 | 1 | public | gemeinden_ausschnitt_poly | geom_topo | 3 | 0 | |

## 3.2 Tolerance

Warum ist die Wahl der Toleranz beim Anlegen der Topologie relevant.

1. Sie legt fest wie groß die Abstände zwischen nodes der zu bildenden Topologie minimal sein sollen.
2. Ist sie zu groß angegeben, werden Flächen mit Kantenlängen, die kleiner sind als die Toleranz, „verschluckt“. Mit verschluckt ist gemeint, dass aus einer Fläche eine Linie oder ein Punkt werden kann und das führt zu Fehlermeldungen.

Punkt 1 ist klar. Man möchte ja gerade die Topologie-Funktionen zum topologisch korrekten Vereinen von Flächen nutzen. Dabei sollen Lücken geschlossen werden und Überlappungen beseitigt.
Auf das Problem von Fehlern bei der Erzeugung der Topologien kann man wie folgt reagieren:

1. Man ignoriert die Fehler einfach, weil man davon ausgeht, das diese ohnehin von zu kleinen Flächen ausgelöst werden.
2. Man führt eine Vorverarbeitung der Daten durch, so dass die Fehler nicht mehr auftreten können.

Bei 1. ist zu klären ob wirklich alle Fehler durch Polygone mit zu kleinen Kantenlängen entstehen.
Bei 2. ist zu klären ob dabei nicht ein Informationsverlust im Vorfeld entsteht.

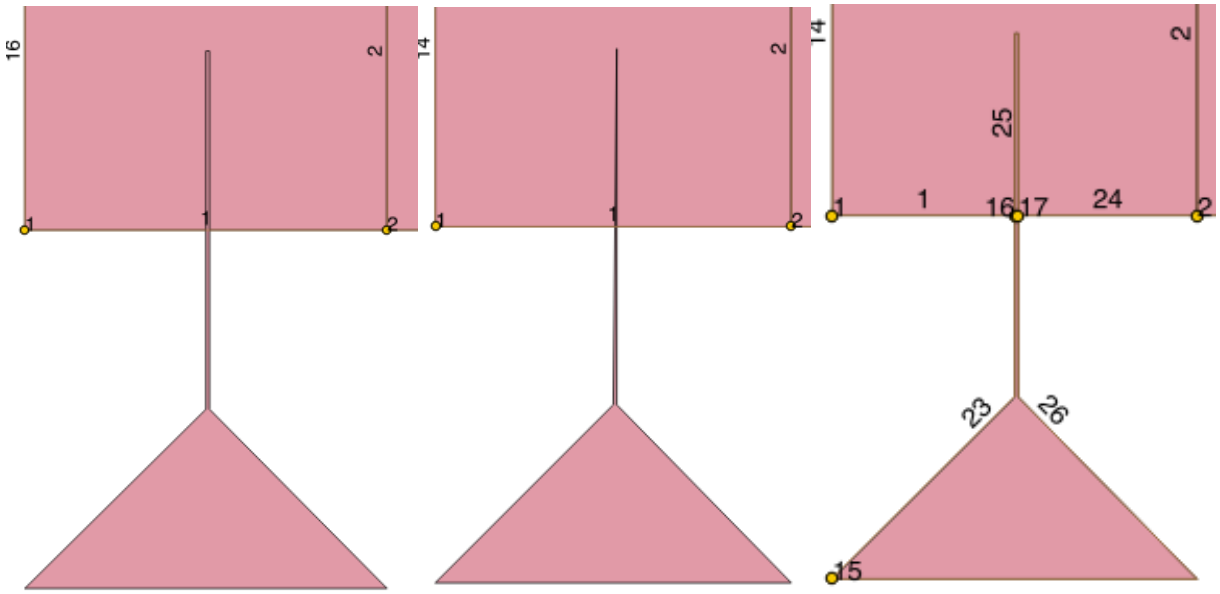


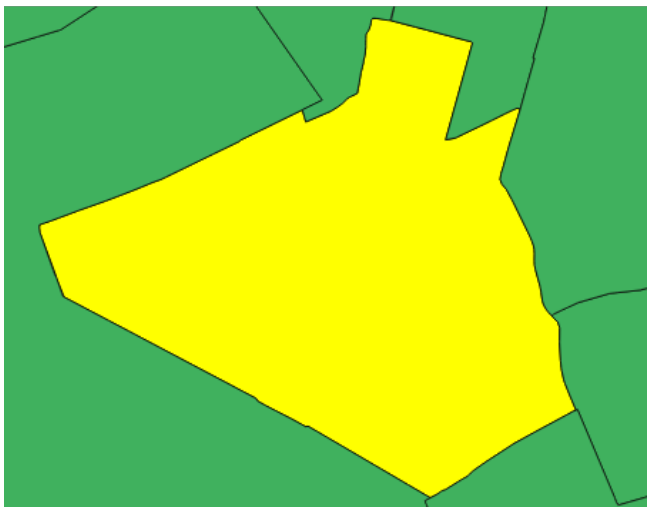

**Abbildung 14: Polygono mit Kantenlänge schmaler als Toleranz**

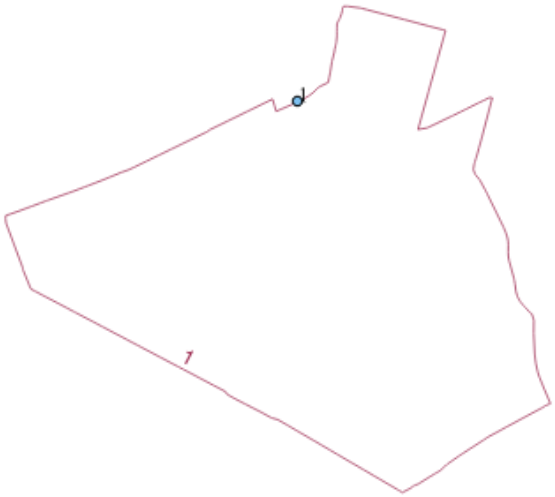
In Abbildung 14 ist gut zu sehen worum es geht. Das Polygon in Abbildung 14 (Links) hat oben eine Kantenlänge von 0,1m. Wird das Polygon mit Toleranz 1m zur Topologie hinzugefügt, würde eine sich selbst überlappende Linie entstehen. Die Fehlermeldung lautet: „SQL/MM Spatial exception - curve not simple“.
Mit ST_Simplify kann man zwar die Kante, die kleiner ist als die Topologietoleranz ist, eliminieren (in Abbildung 14 (Mitte) bleibt die Spitze übrig), die Fehlermeldung bei der Topologieerzeugung bleibt aber.
Verringert man statt dessen die Tolerance der Topologie auf eine Größe < 0,1m, klappt das Hinzufügen zur Topologie, siehe Abbildung 14 (Rechts). Was übrig bleibt ist eine Überlappung, die sich jedoch durch das Verfahren, welches in Abschnitt 3.4.2 beschrieben ist, beheben läßt.
Man sollte also mit ST_Simplify zunächst festlegen wie groß die kleinsten Kanten sein dürfen und anschließend eine Topologietoleranz wählen, die darunter liegt.

## 3.3 Topologie erzeugen

Jetzt können von den Polygonen die Topologien erzeugt werden. Zur Veranschaulichung was dabei passiert wird das schrittweise vorgenommen und illustriert.

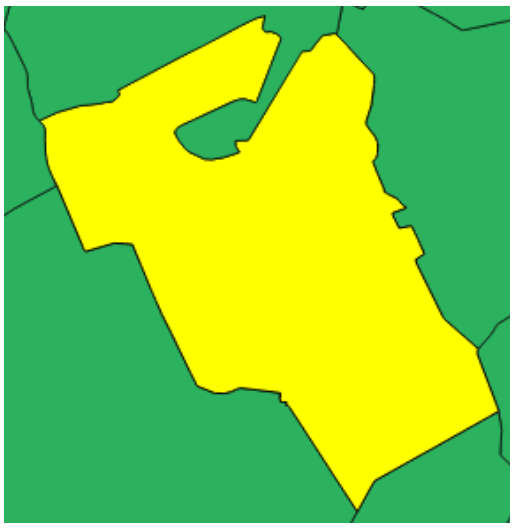
```
UPDATE gemeinden_ausschnitt_poly SET geom_topo = toTopoGeom(geom, 'topo', 1, 10) WHERE gid = 1;
```

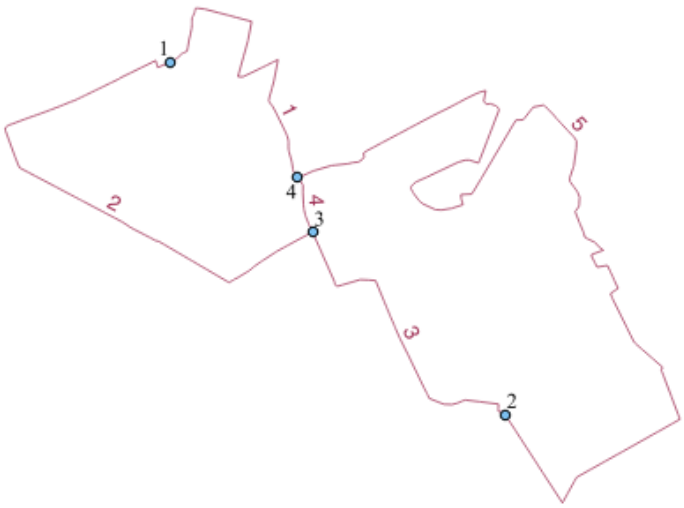





Von dem ausgewählten Polygon mit der gid=1 wird eine geschlossene Masche angelegt, mit einem Node und einer Edge. Wenn ein Nachbar hinzugefügt wird, entsteht eine weitere Masche mit den Nodes 4, 3 und der Edge 4, die sich beide teilen.

```
UPDATE gemeinden_ausschnitt_poly SET geom_topo = toTopoGeom(geom, 'topo', 1, 10) WHERE gid = 83;
```






Bei der Erzeugung der Edges werden Kanten, der Polygone, die kürzer als die Toleranz sind generalisiert, wie man in Abbildung 15.

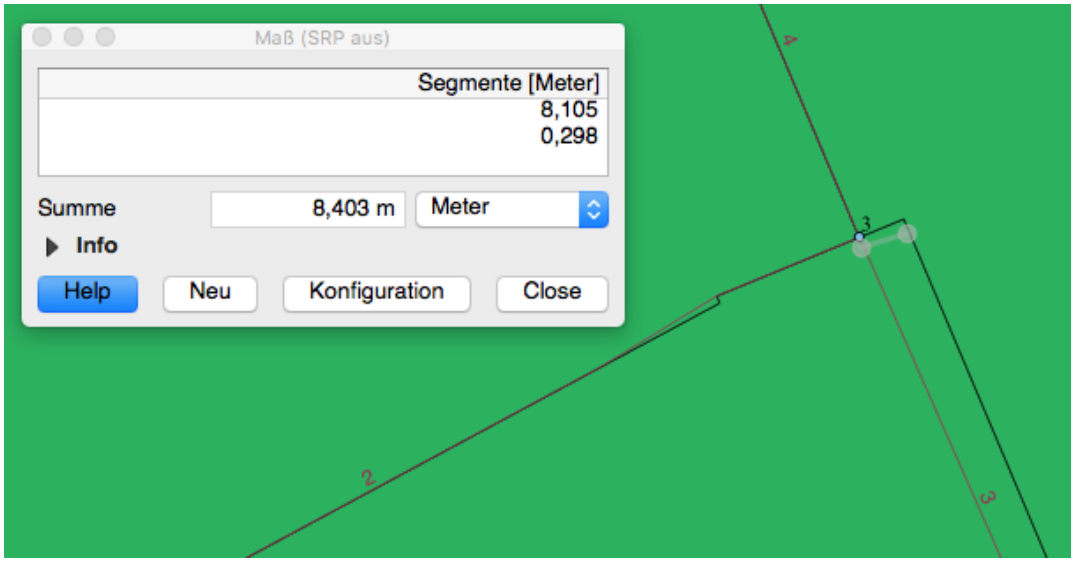



**Abbildung 15: Generalisierung mit der Toleranz der Topology**

```
UPDATE gemeinden_ausschnitt_poly SET geom_topo = toTopoGeom(geom, 'topo', 1, 10) WHERE gid = 36;
```

Fügt eine weitere Maschen hinzu, weil die Geometrie mit gid=36 zuvor ein Multipolygon war und nun aus zwei Polygonen besteht. Leider entstehen bei der Erzeugung der Topology auch Edges, die zwischen den gleichen Nodes verlaufen, wie im Beispiel zwischen Node 3 und 12, die Edges 15 und 16 und somit auch kleine faces, wie in Abbildung 16 face 2. Diese Edges können im nachhinein korrigiert werden. Dazu muss man sie zunächst programmatisch finden und entscheiden welche Edge gelöscht wird oder ob die Relation zum Face entfernt wird.

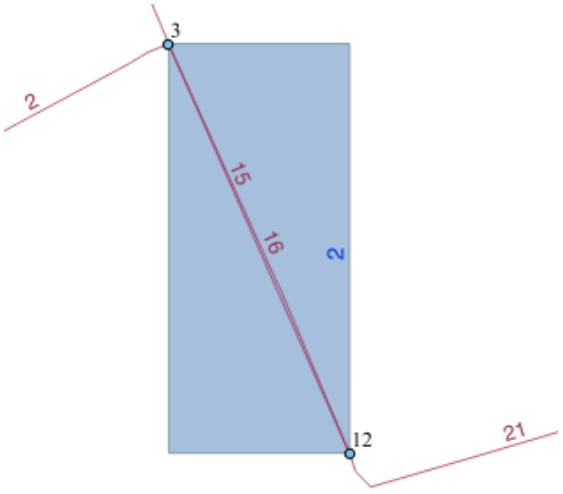



**Abbildung 16: Miniface**

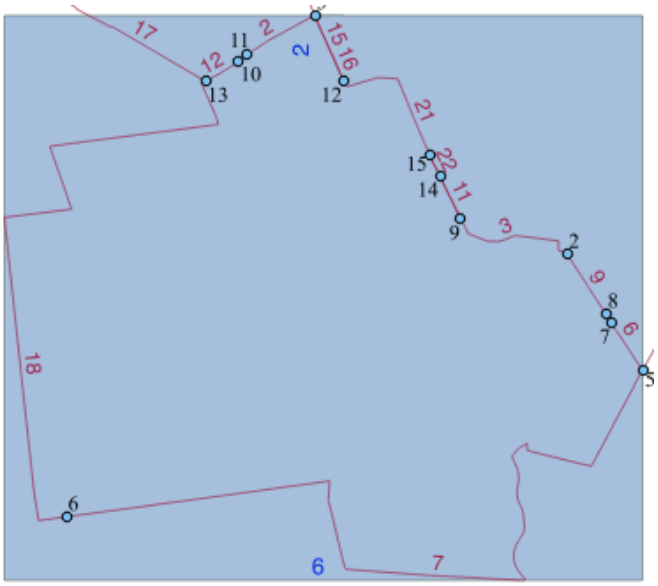



**Abbildung 17: Faces 2 und 6, die zu einer TopoGeometry gehören**

Doch zunächst noch einmal zur Logik wie die Geometrie mit der Topology verknüpft ist. Fragt man die TopoGeometry von gid 36 ab, kommen zwei Ergebnisse, weil die 36 ein Multipolygon hatte mit zwei Polygonen.

```
SELECT geom_topo FROM gemeinden_ausschnitt_poly WHERE gid = 36;
```

| geom_topo topogeometry | topogeo_id integer | layer_id integer | element_id integer | element_type integer |
|---|---|---|---|---|
| (14,1,4,3) | 3 | 1 | 2 | 3 |
| (14,1,3,3) | 3 | 1 | 6 | 3 |

Die 14 ist die ID der topology. Die 1 ist die ID des Layers. Das dritte Argument ist die ID der Relation, auch Topogeom ID genannt, in der die Elemente der Topology hinterlegt sind, die zur Geometrie gehören. Die 3 steht für den Topologytyp, hier Polygon. Fragt man nun die Relationen der Topology z.B. mit der ID 3 ab:

```
SELECT * FROM topo.relation where topogeo_id = 3;
```

erhält man die Elemente, die zur Topogeometry gehören. In diesem Fall sind das die Element mit der ID 2 und 6. Da es sich um den Typ 3=Polygon handelt, sind dies die ID‘s der Faces mit face_id = 2 und 6, siehe Abbildung 17. Die andere TopoGeometry hat einen Bezug zum Face 9. Man kann auch alle Elemente einer TopoGeometry abfragen mit dem Befehl

```
SELECT GetTopoGeomElements(geom_topo) FROM gemeinden_ausschnitt_poly WHERE gid = 36;
```

Das entspricht im Prinzip einem JOIN zwischen der Datentabelle gemeinden_ausschnitt und der Tabelle topo.relation über die topogeo_id.

| gettopogeomelements integer[] |
|---|
| {9,3} |
| {2,3} |
| {6,3} |

Mit der Funktion ST_GetFaceGeometry kann nun wieder die Geometrie abgefragt werden, die durch das Face und die darin enthaltenden Edges gebildet werden.

```
SELECT ST_GetFaceGeometry('topo', (GetTopoGeomElements(geom_topo))[1]) FROM
gemeinden_ausschnitt_poly WHERE gid = 36;
```

Liefert die Geometrien der topologisch korrekten Polygone, die zu dem Objekt mit der gid 36 gehören.

```
UPDATE
  gemeinden_ausschnitt alt
SET
  geom_neu = neu.geom
FROM
  (
    SELECT
      gid,
      ST_Multi(ST_Union(geom)) AS geom
    FROM
      (
        SELECT gid, ST_GetFaceGeometry('topo', (GetTopoGeomElements(geom_topo))[1]) AS geom
FROM gemeinden_ausschnitt_poly
      ) foo
    GROUP BY
      gid
```

```
  ) neu
WHERE
  alt.gid = neu.gid
```

Werden diese Geometrien per ST_Union wieder zu Mutlipolygonen zusammengefasst entsteht die korrigierte Geometrie der ursprünglichen Objekte.

Da Face 2 zu zwei Geometrien gehört,

```
SELECT * FROM topo.relation WHERE layer_id = 1 AND element_id = 2 AND element_type = 3
```

| topogeo_id integer | layer_id integer | element_id integer | element_type integer |
|---|---|---|---|
| 2 | 1 | 2 | 3 |
| 3 | 1 | 2 | 3 |

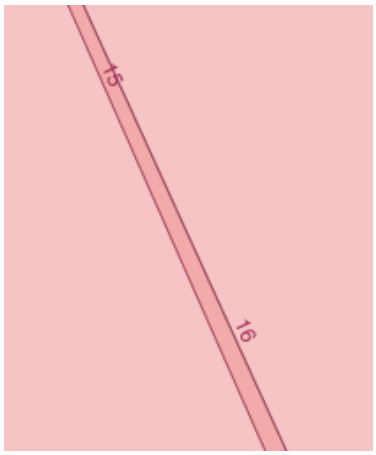
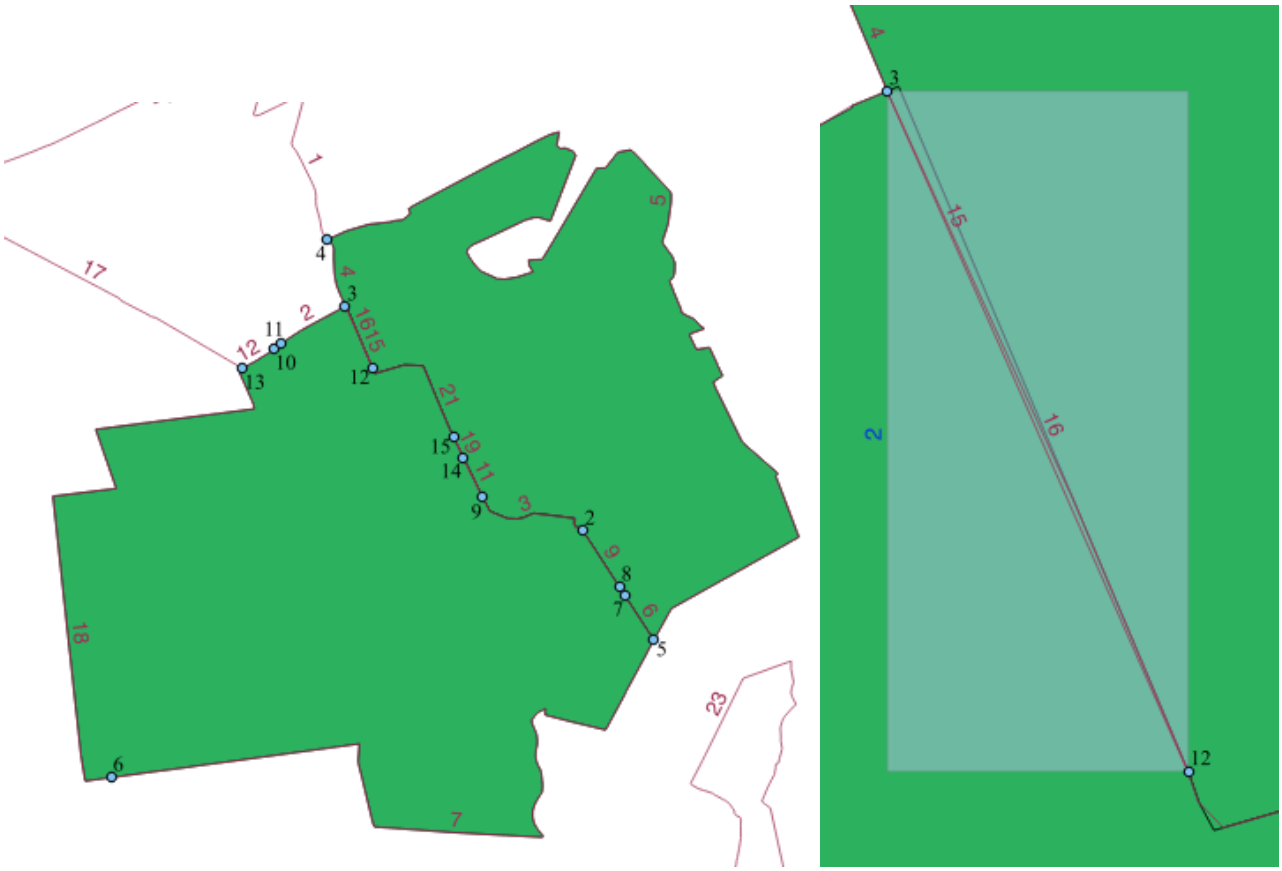



**Abbildung 18: Zwei Polygone überschneiden sich weil sie das gleiche Face nutzen**

muss man sich für eine entscheiden und vorher ein Face entfernen. Die Abfrage

```
SELECT
  gid
FROM
  (
    SELECT
      gid,
      (GetTopoGeomElements(geom_topo))[1] AS face_id
    FROM
      gemeinden_ausschnitt_poly
  ) faces
WHERE
  face_id = 2;
```

liefert die gid‘s 83 und 36. Die durch den UPDATE Befehl erzeugte neue Geometrie für diese beiden Polygone überschneiden sich also noch, siehe Abbildung 18. Jedes Face soll also nur zu genau einem Polygon gehören, damit es nach dem Union keine Überschneidungen mehr gibt.



Zur Frage von oben. Wir identifizieren also die Faces, die zu mehr als einem Polygon gehören und löschen alle bis auf eins und bilden erst danach die neuen Polygone mit ST_Union.

```
SELECT
  TopoGeom_remElement(geom_topo, Array[duplicates.element_id, 3]::topology.topoelement)
FROM
  gemeinden_ausschnitt_poly g JOIN
  (
    SELECT
      a.topogeo_id,
      a.element_id
    FROM
      topo.relation a JOIN
      topo.relation b ON (a.topogeo_id > b.topogeo_id AND a.element_id = b.element_id)
  ) duplicates ON (g.geom_topo).id = duplicates.topogeo_id
```

Die Unterabfrage duplicates enthält jeweils alle Topology-Elemente eines faces außer das mit der kleinsten ID. Geliefert wird die topogeo_id, die für den JOIN mit der Objekttabelle gemeinden_ausschnitt_poly verwendet wird. TopoGeom_remElement entfernt dann schließlich das Face in jeder TopoGeometry, die in der Unterabfrage herauskommt. Wird das UPDATE Statement von oben noch mal ausgeführt, sind dieses mal dann keine sich überlappenden Geometrien erzeugt worden, siehe Abbildung 19.

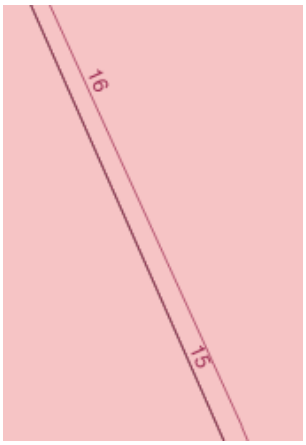



**Abbildung 19: Bereinigte Überlappung durch Entfernung der Face-Relation**

Die Variante des Löschens des Face-Elements im TopoGeometry-Objekt hat den Nachteil, dass die erzeugte Geometrie nich konsistent gegenüber der Topologie ist. Würde man eine solche gemergete Geometry in eine Topologie umwandeln, würde das kleine Face nicht erzeugt werden, da die Geometrie ja nicht mehr überlappt. Das kleine Face ist aber noch in der Topologie vorhanden. Es wäre also besser das Face zu löschen und nicht nur die Zuordnung zu der topoGeometrie.

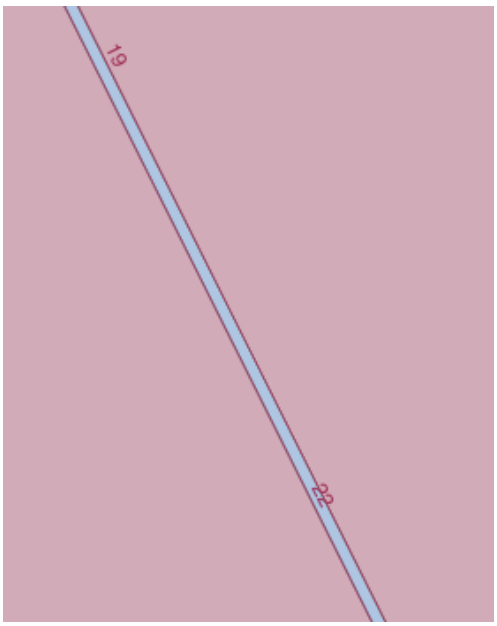



**Abbildung 20: Zwischenräume zwischen Polygonen**

Auf der anderen Seite kann es vorkommen, dass sich zwischen den Polygonen Zwischenräume ergeben. Diese sind dann nicht nur in der Geometrie, sondern auch in der Topologie. Im Beispiel Abbildung 20 gibt es ein Face id=8 welches von den edges 19 und 22 gebildet wird und nicht in der relation Tabelle auftaucht.
Hier besteht die Aufgabe nun darin die Fläche des Face der einen oder anderen Nachbarflächezuzuordnen. Welche Edges das sind, zwischen denen das nicht zugeordnete Face liegt läßt sich einfach abfragen mit:

```
SELECT
  edge_id,
  left_face,
  right_face
FROM
  topo.edge_data
```

```
WHERE
  left_face = 8 OR right_face = 8
```

| edge_id integer | left_face integer | right_face integer |
|---|---|---|
| 19 | 8 | 5 |
| 22 | 8 | 6 |

Eine Liste aller Faces, die keinen Zuordnung zu Geometrien haben und ihre jeweiligen Nachbarn bekommt man mit der Abfrage:

```
SELECT
  left_face,
  right_face
FROm
  topo.edge_data ed LEFT JOIN
  topo.relation rl ON ed.left_face = rl.element_id
WHERE
  rl.element_id IS NULL AND
  ed.left_face > 0
```

Somit haben wir alle Abfragen parat mit denen wir sowohl sich überlappende Flächen als auch Lücken finden. Jetzt behandeln wir das Thema wie wir diese kleinen Faces aus der Topologie entfernen können durch Löschen der Edges.

Bei der Erzeugung von Topologien können auch Fehler auftreten, insbesondere, wenn die Polygone Nasen oder Kerben haben, die in der Topologie zu toten Enden führen. Deshalb beginnt der nächste Abschnitt mit der Bereinigung von Kerben und Nasen.

## 3.4 Topologie bereinigen

### 3.4.1 Nasen und Kerben

Manche Ausgangspolygone haben sehr schmale Ausbuchtungen oder Einkerbungen, siehe Abbildung 21. Die Ausbuchtung von Polygon 1735 ist nur ca. 1,5 mm.

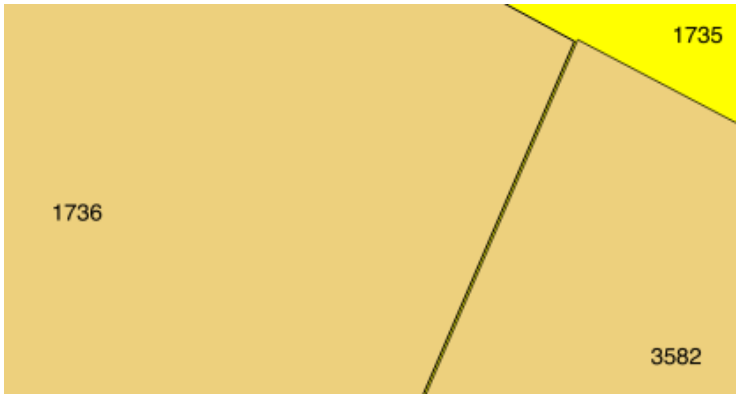
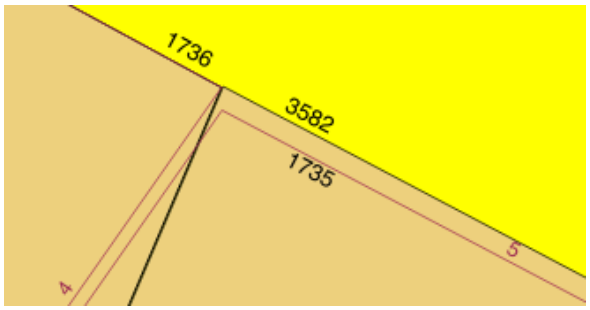
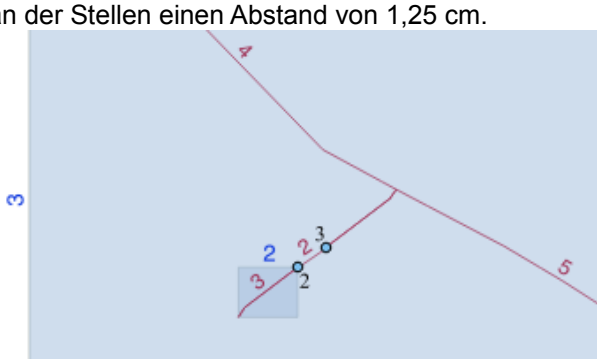



**Abbildung 21: Nase im Polygon 1735**

Das führt beim Erzeugen von Topologien zu Nasen und Kerben.




Die Kanten 4 und 5 entstehen merkwürdigerweise trotz Toleranz von 10m nicht aneinander sondern haben an der Stellen einen Abstand von 1,25 cm.




| st_getfaceedges getfaceedges_returntype |
|---|
| (1,-4) |
| (2,-5) |

Die Begrenzung des face 3 geht mit node 4 bis zu node 3 und verläuft weiter mit edge 5. Die edge 2 zwischen node 2 und 3 ist keinem face zugeordnet. Weil

```
SELECT ST_GetFaceEdges(,topo‘, 3)
```

nur edge 4 und 5 liefert und in der TopoGeom von Polygon mit der ID 1735 nur die Elemente 2 und 3 liefert. Edge 2 hängt sozusagen in der Luft und hat als left_face und right_face nur die 0

```
SELECT * FROM topo.edge_data WHERE edge_id = 2
```

| edge_id integer | start_node integer | end_node integer | next_left_edge integer | abs_next_left_edge integer | next_right_edge integer | abs_next_right_edge integer | left_face integer | right_face integer | g g |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 3 | 4 | 4 | 3 | 3 | 0 | 0 | 0 |

Die Edge 3 hat wenigstens ein eigenes Face id = 2, wäre aber ohne edge 2 isoliert von face 3. Edge 2 ist also falsch und muss gelöscht werden und face 2 ist zwar nicht falsch im topologischem Sinne, aber dennoch auch ein Teil der zu eliminierenden Nase.
Es ergeben sich also 2 Unterfälle zur Eliminierung von Nasen. Das Löschen von edges, die zu keinem face gehören und das Löschen von zu kleinen isolierten faces und der dazugehörigen edges und nodes.

#### 3.4.1.1 Löschen von edges, die zu keinen faces gehören

Mit der Abfrage

```
SELECT ST_RemEdgeModFace('topo', edge_id) FROM topo.edge_data WHERE right_face = 0 and left_face = 0
```

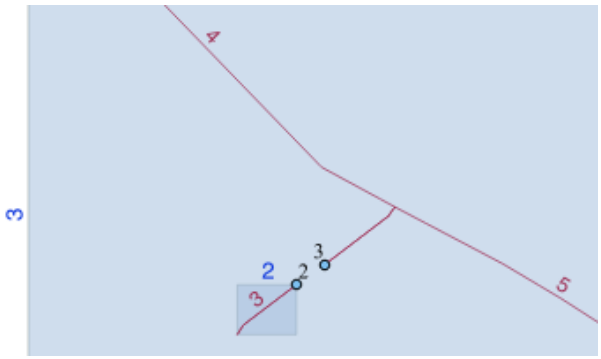



**Abbildung 22: Nach Löschen des edge ohne face**

ist die Edge 2 gelöscht, siehe Abbildung 22.

#### 3.4.1.2 Löschen von faces und edges, die isoliert und sehr klein sind

Der Algorithmus zum Finden und eliminieren dieser faces geht wie folgt:

- Finde alle faces, deren Fläche kleiner x ist.
- Entferne alle edge-Elemente aus der TopoGeometry
- Lösche die edges
- Lösche die nodes
- Lösche das verbliebene face.face

Mit Abfrage

```
WITH edges AS (
SELECT start_node, end_node
FROM topo.edge
WHERE left_face = 2
OR right_face = 2
)
SELECT start_node FROM edges UNION
SELECT end_node FROM edges;
```

werden alle nodes des face 2 abgefragt. Mit Abfrage

```
SELECT TopoGeom_remElement(geom_topo, Array[2,3]::topology.TopoElement) FrOM
gemeinden_mv_topo WHERE gid = 1735
```

wird das face von der TopoGeometrie entfernt.

```
SELECT ST_RemEdgeModFace('topo', abs((ST_GetFaceEdges('topo', 2)).edge))
```

sind alle edges des face 2 gelöscht, siehe Abbildung 23 und mit Abfrage

```
SELECT ST_RemoveIsoNode('topo', 2);
```

werden die verbleibenden nodes gelöscht, die oben mit der WITH Abfrage abgefragt wurden. Hier node 2.

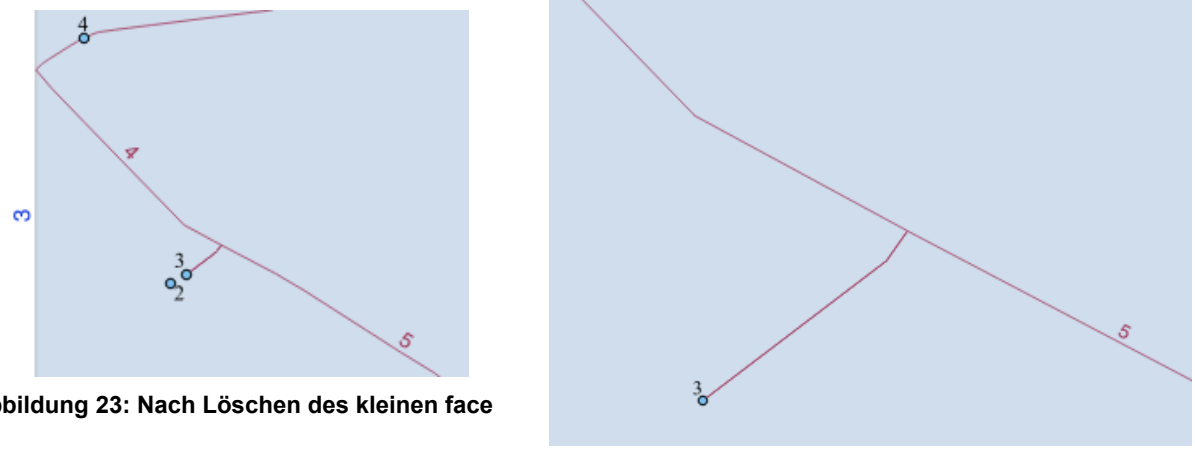



**Abbildung 23: Nach Löschen des kleinen face**

Was jetzt noch übrig geblieben ist, ist die Nase zu node 3.

Eine Schlüssel-Funktion zur Bereinigung in Topologien, die Splitter-Faces durch Überlappung und Lücken aufweisen, spielt ST_RemEdgeModFace().

## 3.4.2 Überlappungen

Überlappungen entstehen, wenn sich benachbarte Linienzüge überschneiden, siehe Abbildung 24.

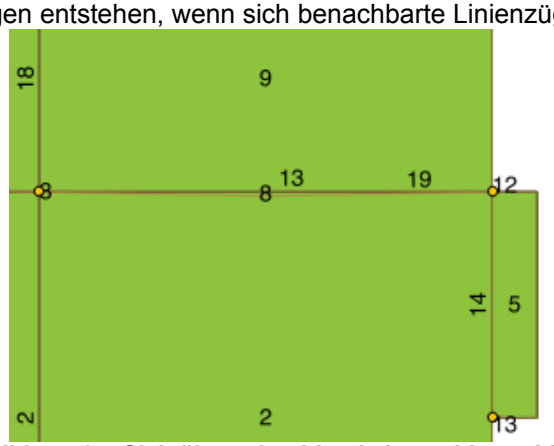



**Abbildung 24: Sich überschneidende benachbarte Linien**

In dem Fall wird im Überlappungsbereich ein face gebildet, welches von beiden benachbarten Ausgangsgeometrien genutzt wird.
Versucht man eine Edge eines solchen Face zu löschen, welches durch eine Überlappung entstanden ist, bekommt man eine Fehlermeldung:

```
SELECT ST_RemEdgeModFace('topo', 13);
```

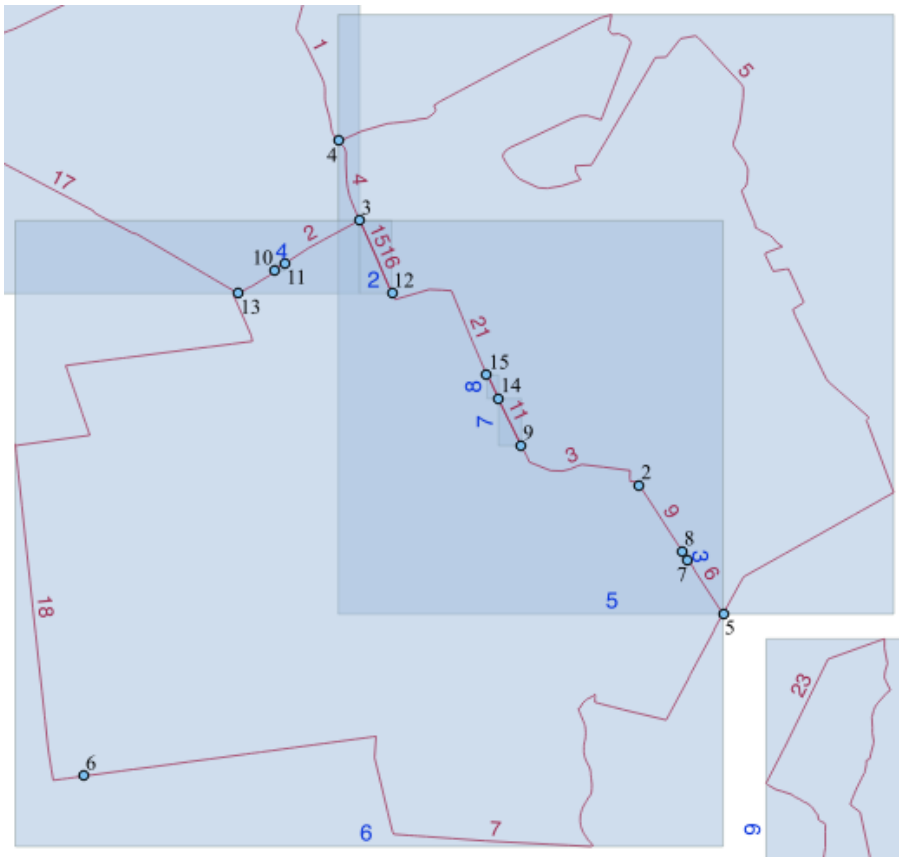
FEHLER: TopoGeom 3 in layer 1 (public.gemeinden_ausschnitt_poly.geom_topo) cannot be represented healing faces 8 and 9
TopGeom 3 hat die Relation zu face 8 und 2. Wird durch das Löschen von edge 13 das face 8 mit face 9 vereint, kann die TopoGeom 3 nicht mehr repräsentiert werden.
Das ist einleuchtend. Man muss also erst die Relation des zu löschenden face und der TopoGeom eines Nachbarn entfernen mit:

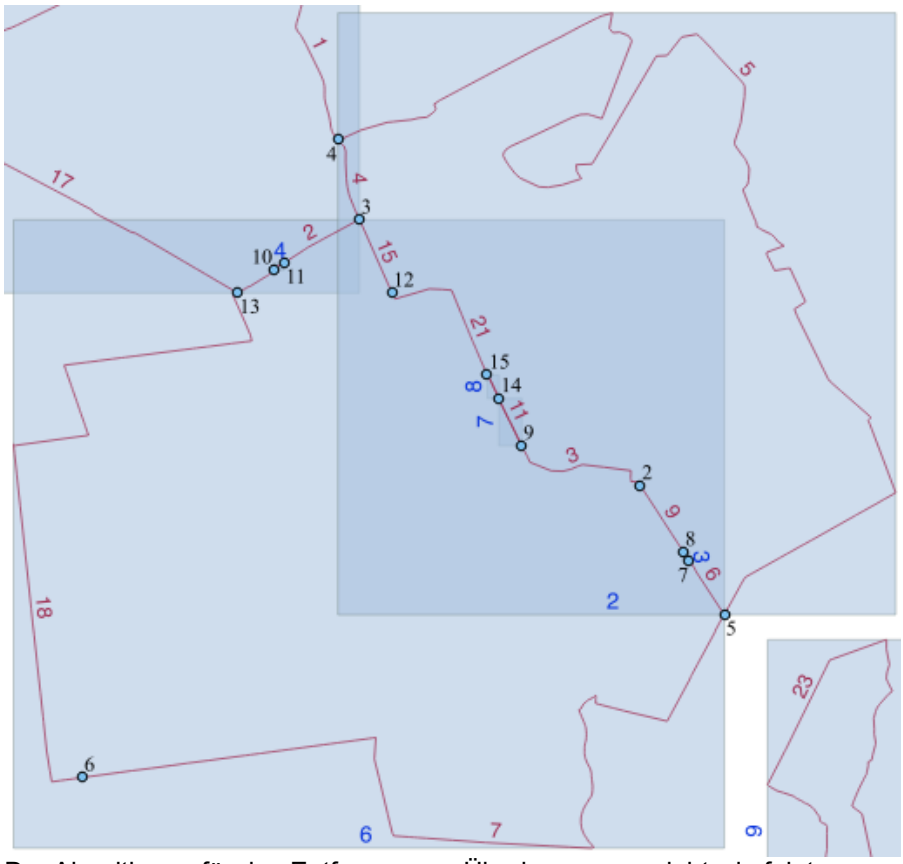
```
SELECT TopoGeom_remElement(geom_topo, '{2, 3}') FROM gemeinden_ausschnitt_poly WHERE gid = 36;
```

Danach kann auch die Edge gelöscht werden. Im Ergebnis ist da wo vorher die edge 13 und 19 das face 8

gebildet haben nur noch die edge 19 und das face 9 ersetzt face 8 und wird von edge 19 begrenzt. Face 2 ist unverändert und wird auch edge 19 begrenzt. Nach der Bereinigung gibt es also nur noch eine edge 19 face_left = 9 und face_right = 2. Die mit dem obigen Update erzeugten Polygone überlappen sich nicht mehr, siehe auch Beispiel mit edge 15 und 16 wo 16 gelöscht wird.

Wenn mit TopoGeom_remElement ein Element einer topoGeometrie gelöscht wird, ist die Topologie nicht mehr konsistent. Das lässt sich dadurch heilen, dass anschließend die Geometrie aus der dann neuen topoGeometry erzeugt wird.
Also noch mal zur Verdeutlichung. Zwei Flächen (A und B) überlappen sich in einem Bereich C. Es entstehen 3 faces (a, b und c). A liegt über a und c, B liegt über b und c. Jetzt wird erst face c von Fläche A mit TopoGeom_remElement entfernt, damit gehört A nur noch zu a und B zu b und c. Dann wird die Edge zwischen b und c gelöscht so dass nur noch b übrig bleibt mit ST_RemEdgeModFace. Jetzt gehört in der Topologie a zu A und b zu B. In Wirklichkeit sind die Geometrien aber immer noch überlappend. Um das zu Bereinigen muss die Geometrie von A und B mit ST_GetFaceGeometry aktualisiert werden. Somit ist von A nur noch der Teil des faces a übrig und B besteht aus den vereinigten faces b und c. Hätte man das face c von Fläche B entfernt und die edge zwischen face a und c, wäre die Überlappungsfläche A zugeschlagen worden statt B.





Der Algorithmus für das Entfernen von Überlappungen sieht wie folgt aus:

- Finde alle faces, die zu mehr als einer TopoGeom eine relation haben und die dazugehörigen topoGeoms
- Entferne die faces in den topoGeoms
- Suche die Edge, zwischen dem zu löschenden Face und dem Nachbarn, der es noch in relation hat

```
CREATE OR REPLACE FUNCTION kvw_RemoveTopoOverlaps(
  topo_name CHARACTER VARYING,
  table_name character varying,
  topo_geom_column CHARACTER VARYING)
  RETURNS BOOLEAN AS
$BODY$
  DECLARE
    sql text;
    overlap RECORD;
  BEGIN
    sql = '
      SELECT
        a.topogeo_id,
        a.element_id AS face_id
      FROM
        ' || topo_name || '.relation a JOIN
        ' || topo_name || '.relation b ON (a.topogeo_id > b.topogeo_id AND a.element_id = b.element_id)
    ';
    --RAISE NOTICE 'Loop over sql: %', sql;

    FOR overlap IN EXECUTE sql LOOP
      RAISE NOTICE 'Remove overlap at face % in topogeo id %', overlap.face_id, overlap.topogeo_id;
```

```
    sql = '
      SELECT
        TopoGeom_remElement(' || topo_geom_column || ', ARRAY[' || overlap.face_id || ',
3]::topology.TopoElement)
      FROM
        ' || table_name || '
      WHERE
        (' || topo_geom_column || ').id = ' || overlap.topogeo_id || '
    ';
    RAISE NOTICE 'Execute sql to remove the face: %', sql;
    EXECUTE sql;

    sql = '
      SELECT
        ST_RemEdgeModFace(''' || topo_name || ''', e.edge_id)
      FROM
        topo.relation r JOIN
        topo.edge_data e ON (
          (e.left_face = ' || overlap.face_id || ' AND e.right_face != r.element_id) OR
          (left_face != r.element_id AND e.right_face = ' || overlap.face_id || ')
        )
      WHERE
        r.topogeo_id = ' || overlap.topogeo_id || '
    ';

    RAISE NOTICE 'Execute sql to remove edge: %', sql;
    EXECUTE sql;

  END LOOP;
  RETURN TRUE;
 END;
$BODY$
 LANGUAGE plpgsql VOLATILE COST 100;
```

ToDo: Korrigiere, dass dieser Fehler nicht mehr auftritt:
entsteht wenn gid 3582 nach 1735 und 1736 hinzugefügt wurden und mit kvw_RemoveTopoOverlaps bearbeitet werden. kvw_CleanPolygonTopo geht auch nicht in diesem Zustand. Hier will er auch was Löschen was er nicht darf.

FEHLER: TopoGeom 1 in layer 1 (public.gemeinden_mv_topo.geom_topo) cannot be represented healing faces 2 and 9 CONTEXT: SQL-Anweisung » SELECT ST_RemEdgeModFace('topo', e.edge_id) FROM topo.relation r JOIN topo.edge_data e ON ( (e.left_face = 2 AND e.right_face != r.element_id) OR (left_face != r.element_id AND e.right_face = 2) ) WHERE r.topogeo_id = 2
Das Problem mit den Überlappungen ist größer, dann es können auch mehr als 2 Flächen überlappen.

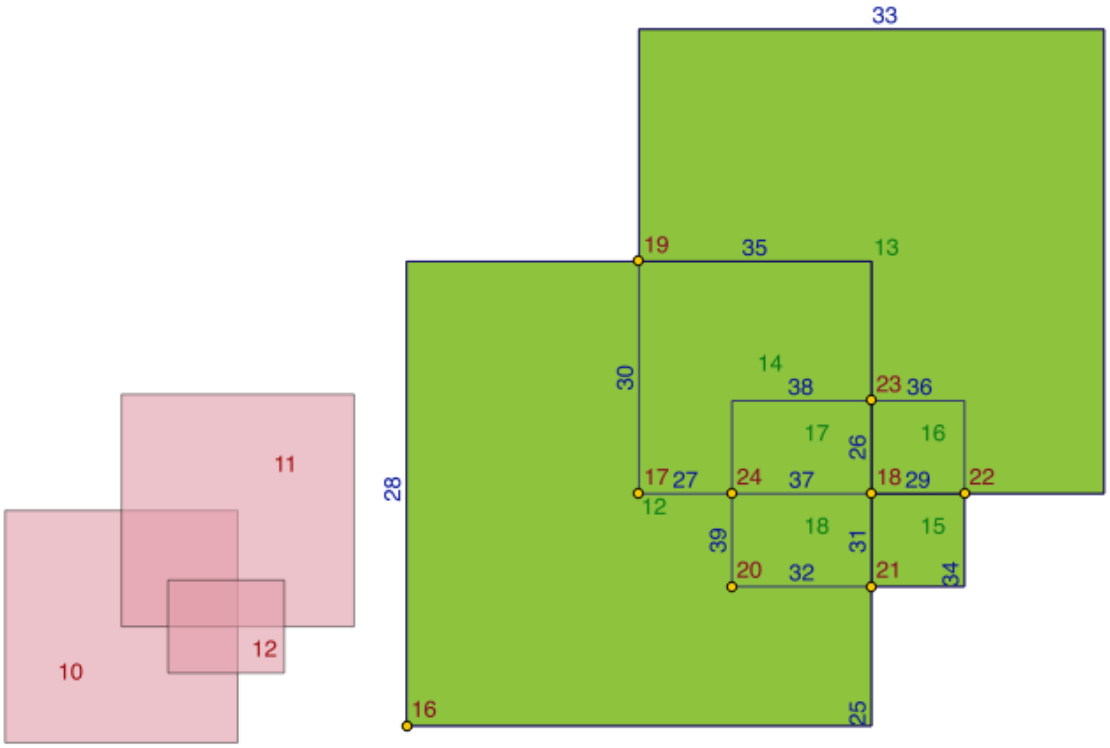
### 3.4.3 Mehrflächenüberlappung

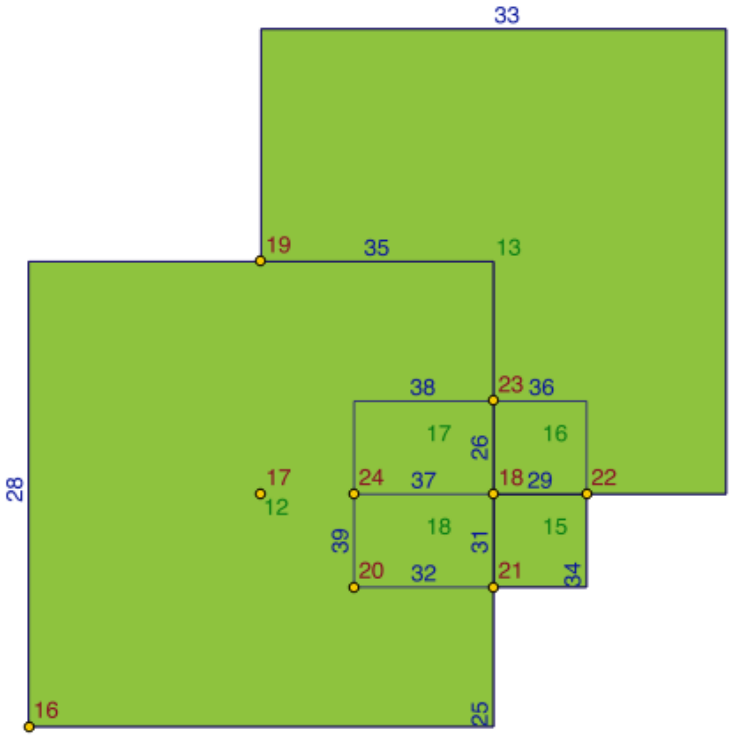
Im Beispiel sehen wir 3 Polygone (10, 11, 12) die sich überlappen. Um diese aufzulösen kann folgender Algorithmus angewendet werden:

1. Finde alle faces F die nur zu einem Polygon gehören und durchlaufe eine Schleife von i = 1-n
2. Für $F_i$ finde alle Nachbar faces N, die genau zwei Zuordnungen zu Polygonen haben und durchlaufe mit ihnen eine Schleife von j = 1-m
3. Für $N_{ij}$ lösche $N_{ij}$ bei Polygon des Nachbarn und Edges zwischen $F_i$ und $N_{ij}$

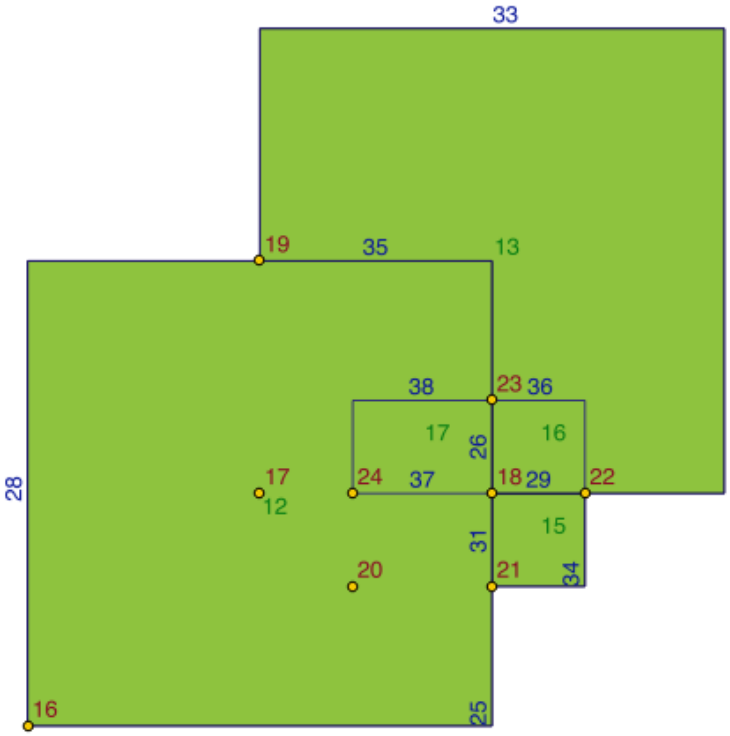
Der Ablauf für das Beispiel sähe so aus:

- Finde faces 12, 13 und 15
- Für face 12 finde als Nachbarn mit zwei Zuordnungen faces 14 und 18
- lösche face 14 in Polygon 11 und edges 27 und 30

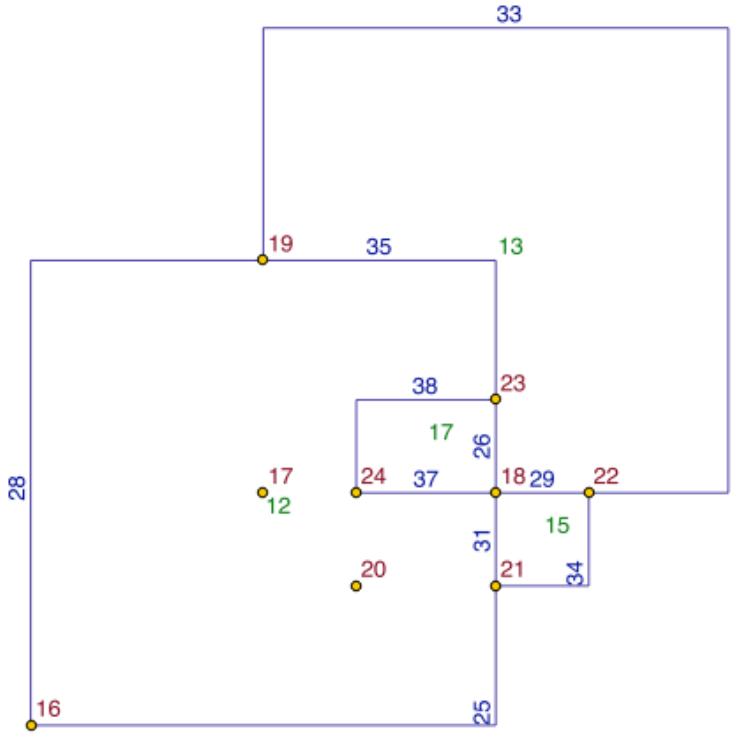



- Polygon 10 besteht jetzt nur noch aus face 12, 17 und 18 und Polygon 11 aus face 13, 17 und 16
- lösche face 18 in Polygon 12 und edges 32 und 39




- Polygon 10 besteht jetzt nur noch aus face 12 und 17
- Für face 13 finde als Nachbarn mit zwei Zuordnungen noch face 16
- lösche face 16 in Polygon 12 und edge 36




- Für face 15 werden keine Nachbarn mit zwei Zuordnungen mehr gefunden.

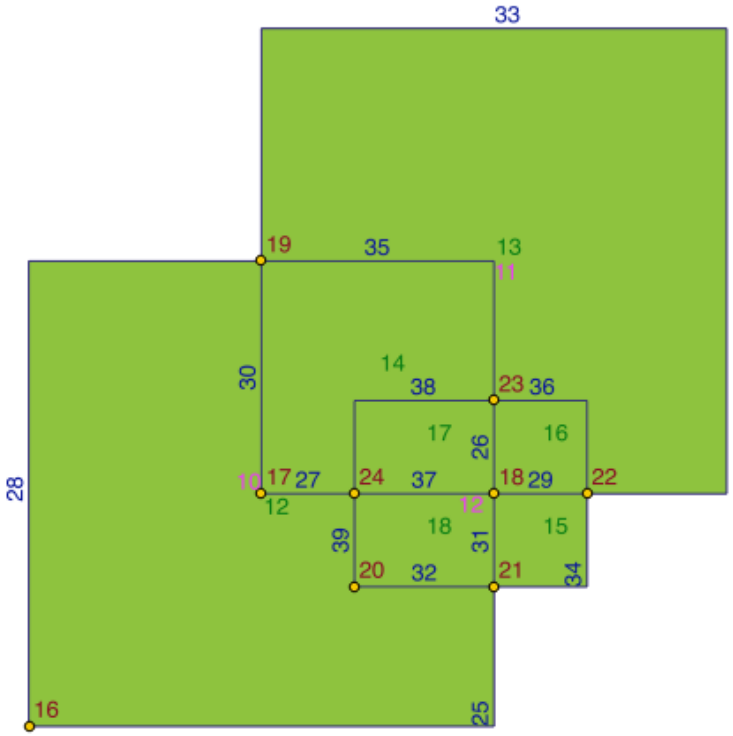
Gelöscht werden muss jetzt noch face 17. Daher wird der Algorithmus auf die faces mit mehr als zwei Zuordnungen zu Nachbarn erweitert.

1. Finde alle faces F, die nur eine Zuordnung haben und sortiere nach deren Flächen aufsteigend (nehme den mit der kleinsten Fläche zu erst, weil ihm sonst wohl am meisten weggeschnitten

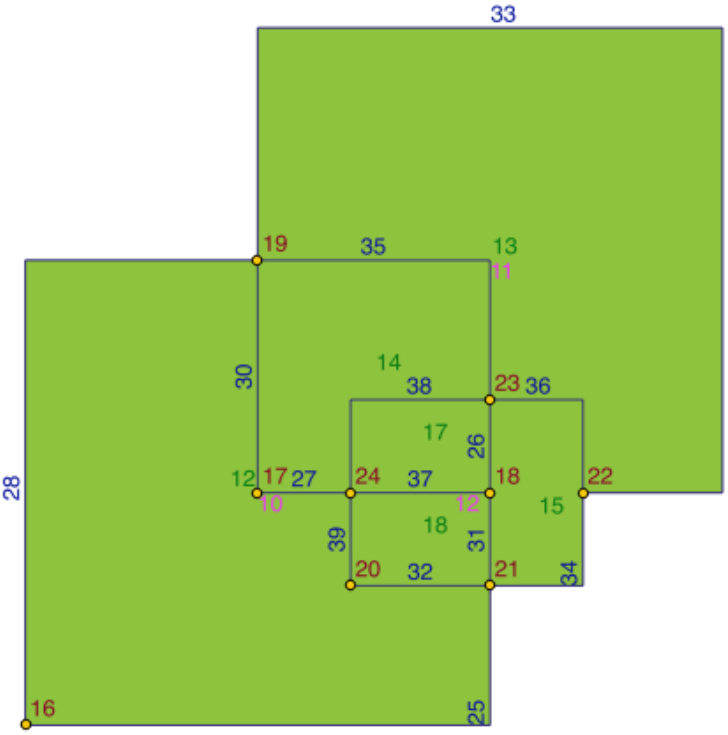
werden würde) und durchlaufe eine Schleife von i = 1-n

2. Für $F_i$ finde alle Nachbar faces N, die mehr als eine Zuordnung zu Polygonen haben und durchlaufe mit ihnen eine Schleife von j = 1-m
3. Für $N_{ij}$ lösche $N_{ij}$ bei dessen Polygon, Edges zwischen $F_i$ und $N_{ij}$ sowie isolierte nodes des $F_i$

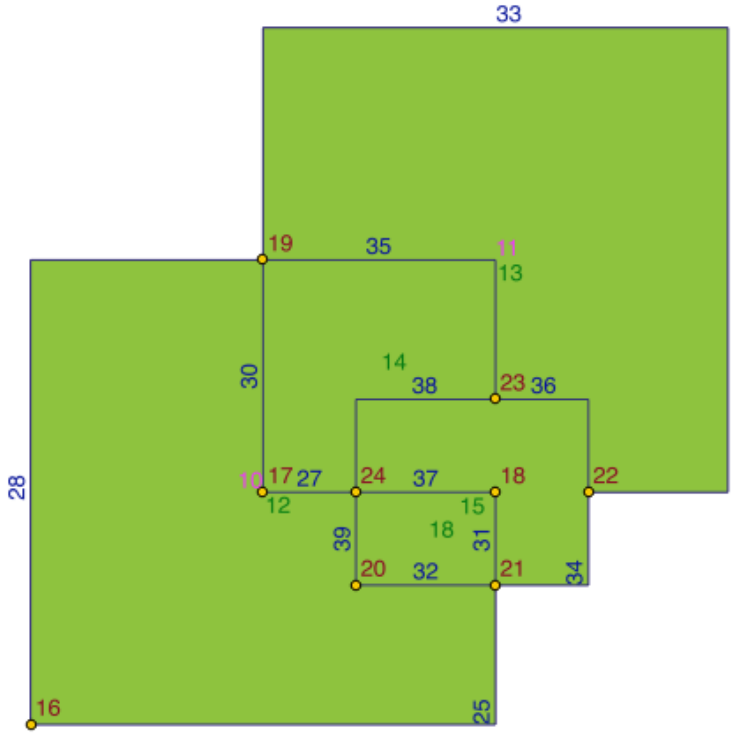
Für diesen Algorithmus sehen die Schritte wie folgt aus:




- Finde faces 15, 12 und 13 (gründe Ziffern)
- Für $F_0$ = face 15 finde als Nachbarn mit mehr Zuordnungen (faces 16, 17 und 18)
- Lösche $N_{00}$ = face 16 in Polygon 11 (pinke Ziffer)
- Lösche edge 29 (blaue Ziffer) zwischen $F_0$ = face 15 und $N_{00}$ = face 16
- Lösche isolierte nodes containing $F_0$ = face 15

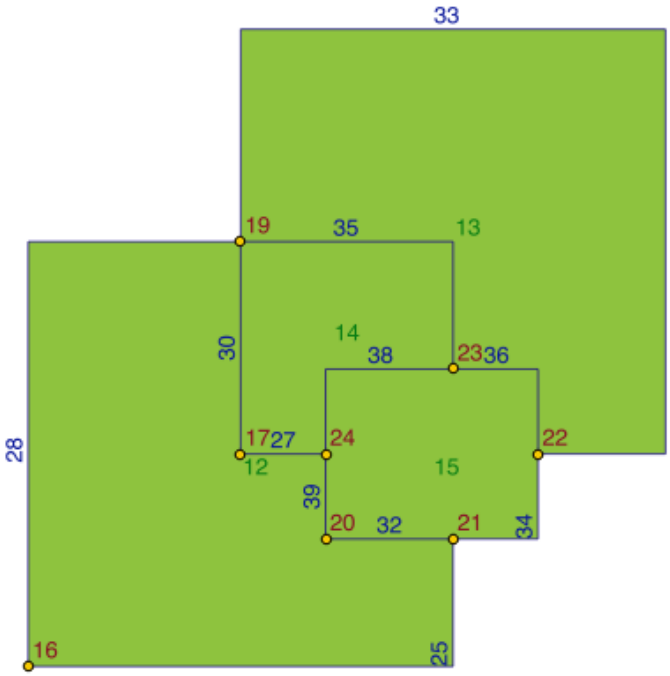



- Polygon 12 hat jetzt noch face 15, 17 und 18 und Polygon 11 hat face 13, 14 und 17
- Lösche $N_{01}$ = face 17 in Polygon 11 und Polygon 10
- Lösche edge 26 zwischen $N_0$ = face 15 und $N_{01}$ = face 17
- Lösche isolierte nodes containing $F_0$ = face 15

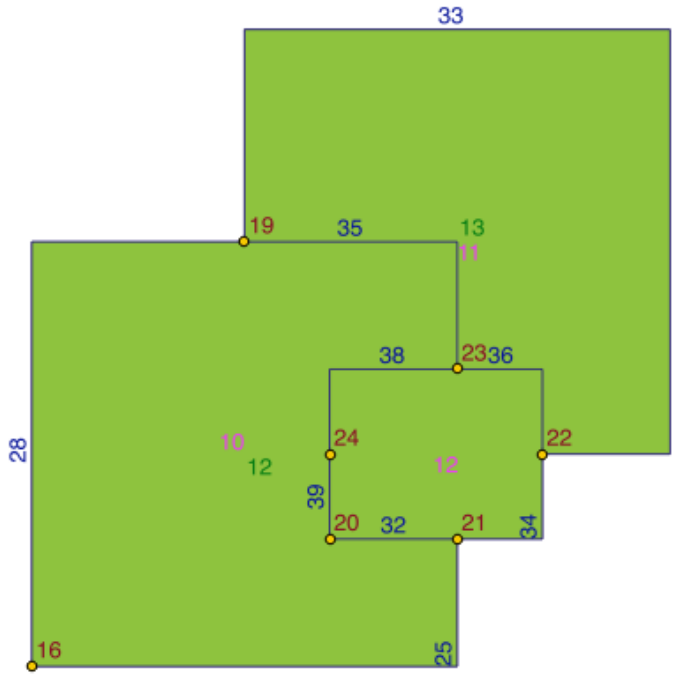



- Polygon 12 hat jetzt noch face 15 und 18, Polygon 11 hat face 13 und 14 und Polygon 10 hat face 12, 14 und 18
- Lösche $N_{02}$ = face 18 in Polygon 10
- Lösche edge 31 und 37 zwischen $N_0$ = face 15 und $N_{02}$ = face 18

- Lösche isolierte nodes containing $F_0$ = face 15




- Polygon 10 hat dann noch face 12 und 14, Polygon 12 face 18 und Polygon 11 face 13 und 14
- Für $F_1$ = face 12 finde als Nachbarn mit mehr Zuordnungen face 14
- Lösche $N_{10}$ = face 14 in Polygon 11
- Lösche edge 30 und 27 zwischen $F_1$ = face 12 und $N_{10}$ = face 14
- Lösche isolierte nodes containing $F_1$ = face 12

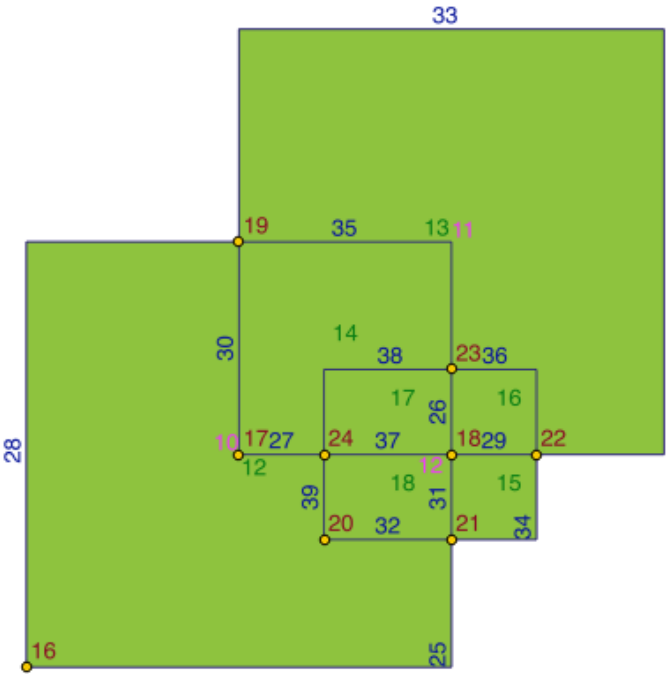



Auf das löschen der isolierten Nodes kann man auch verzichten, wenn man vor jedem Löschen von edges die Funktion ST_ModEdgeHeal für diese ausführt um die nodes dazwischen zu löschen. Das ist aber sicher ein größerer Aufwand, weil man da erst die dazwischen liegenden nodes finden müsste.
Sinnvoller ist wohl ganz zum Schluss noch mal die Funktion gdi_ModEdgeHealException anzuwenden um generell alle unnötigen nodes zu löschen.
Dieser Ansatz geht also von den faces aus, die keine Überlappungen haben. Anders herum kann man auch

von den faces ausgehen, die Überlappungen haben. Der Algorithmus ginge dann so:

1. Durchlaufe alle Polygone P der Größe nach, mit dem kleinsten beginnend von i = 1-n
2. Für $P_i$ lösche alle Zuordnungen von faces zu anderen Polygonen außer zu $P_i$
3. Für Pi lösche alle edges, die links und rechts faces haben, die zu $P_i$ gehören. Damit werden alle innen liegenden edges gelöscht und die faces zu einer vereint.

Die einzelnen Schritte laufen dann wie folgt ab:




- Finde Polygone 12, 10 und 11 in der Reihenfolge
- Für $P_0$ = P12 Lösche face 16 in P 11, face 17 in P 11, face 17 in P 10 und face 18 in P 10
- Für $P_0$ = P12 Lösche edge 26, 37, 29 und 31
- Lösche isolierten node des face, welches dann noch zu $P_0$ = P12 gehört node 18
- Für $P_1$ = P10 Lösche face 14 in P11
- Für $P_1$ = P10 Lösche edge 30 und 27
- Lösche isolierten node des face, welches dann noch zu $P_1$ = P10 gehört node 17

Zum Löschen der faces kann folgendes Statement verwendet werden:

```
SELECT
--  'Remove element ' || r1.element_id || ' in polygon ' || t.id,
  TopoGeom_remElement(topo_geom, ARRAY[r1.element_id, 3]::topology.TopoElement)
FROM
  test_topo.relation r1 JOIN
  test_topo.relation r2 ON r1.topogeo_id < r2.topogeo_id AND r1.element_id = r2.element_id JOIN
  test t ON r1.topogeo_id = (t.topo_geom).id
WHERE
  r2.topogeo_id = P0
```

Zum Löschen der edges kann das folgende Statement verwendet werden.

```
DELTE FROM
  test_topo.edge_data e JOIN
  test_topo.relation r1 ON e.left_face = r1.element_id JOIN
  test_topo.relation r2 ON e.right_face = r2.element_id AND r1.topogeo_id = r2.topogeo_id JOIN
  test t ON r1.topogeo_id = (t.topo_geom).id
WHERE
```

t.id = $P_i$.id

(Statement funktioniert für Select und muss noch auf DELETE getestet werden.

### 3.4.4 Lücken

Lücken entstehen, wenn benachbarte Linienzüge sich nicht überschneiden, siehe Abbildung 25.

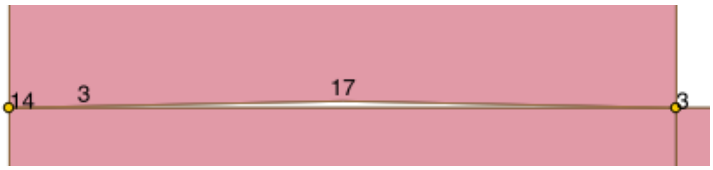
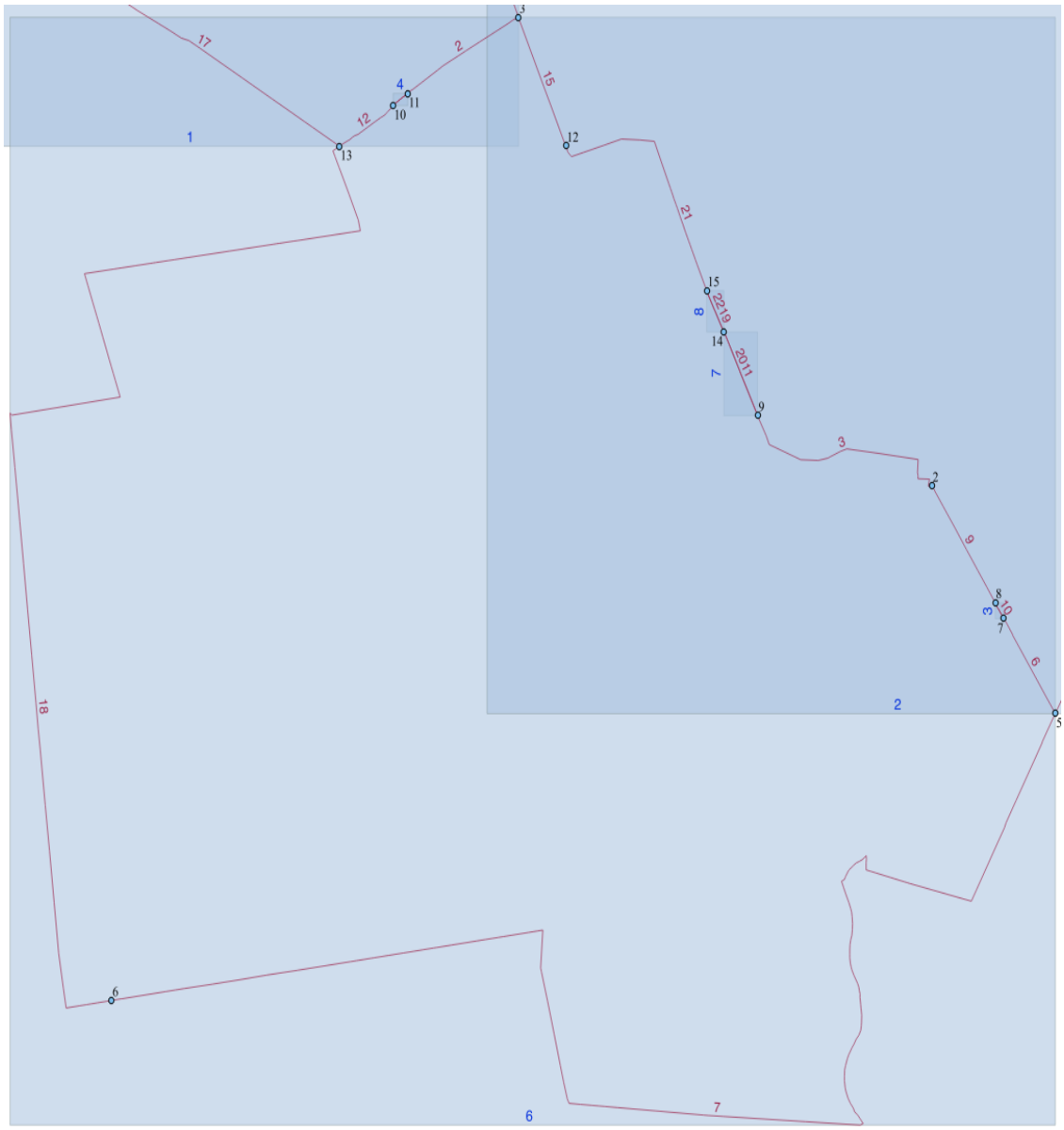
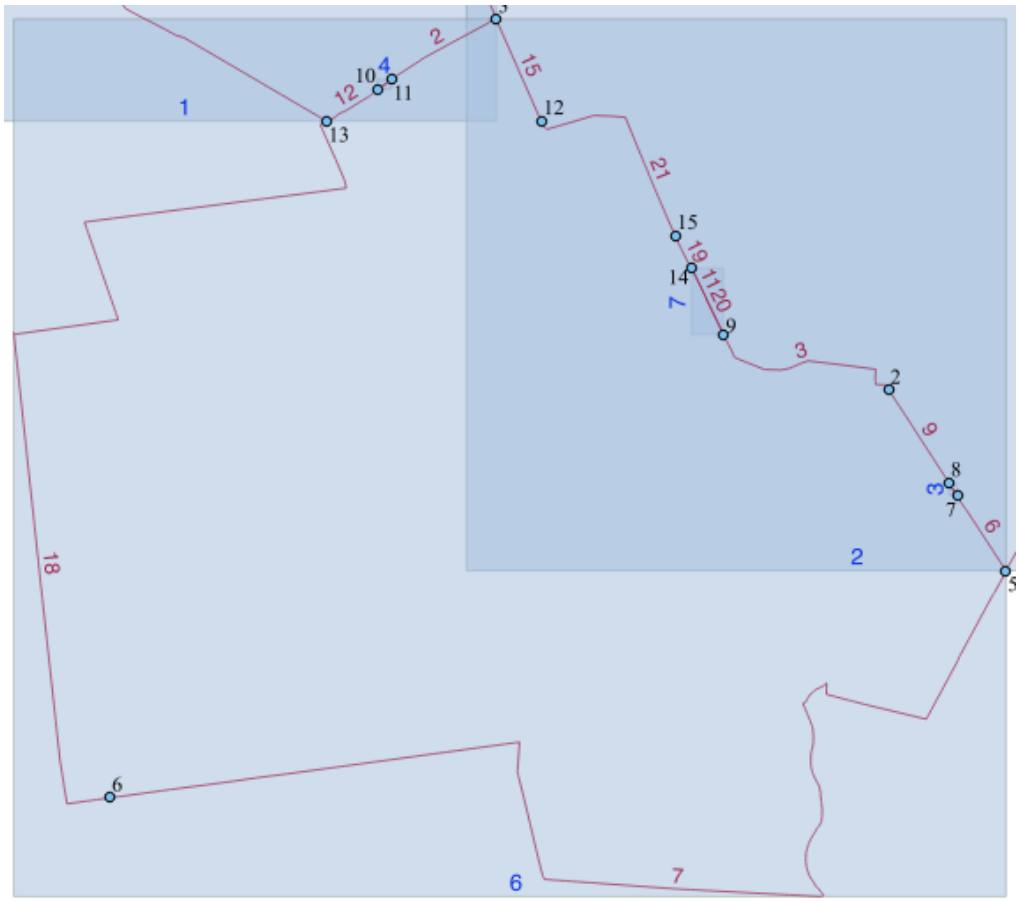



**Abbildung 25: Lücke zwischen Flächen**

Bei Lücken, wie sie zwischen 19 und 22 auftreten, kann auch die Funktion ST_RemEdgeModFace zum Einsatz kommen. In diesem Fall muss aber vorher das Face, welches die Lücke abdeckt vorher der Nachbarfläche zugeschlagen werden. Im Beispiel wird das Face 8 der TopoGeom 3 zugeschlagen, die die Geometrie links der Lücke repräsentiert und schon zu face 6 gehörte. Anschließend wird die edge 22 gelöscht, die zwischen dem Face 8 und 6 lag. Damit wird face 8 gelöscht und übrig bleibt face 6. Die andere Seite mit face 2 bleibt unberührt.


17
2
4
11
10
12
1
13
3
15
12
21
15
2219
8
14
2011
7
9
3
2
9
8
10
3
7
6
18
2
5
6
6
7




Der Algorithmus zum Schließen der Lücken lautet:

- Finde alle faces, die keine relation haben außer face_id = 0.
- Vom jeweils 1 gefundenen Face der Gruppen gleicher left_faces suche die TopoGeom, zu der auch schon das right_face gehört, füge das left_face hinzu und lösche die edge aus dieser Zeile.

| edge_id integer | left_face integer | right_face integer |
|---|---|---|
| 8 | 3 | 2 |
| 10 | 3 | 6 |
| 13 | 4 | 1 |
| 14 | 4 | 6 |
| 11 | 7 | 2 |
| 20 | 7 | 6 |

Im Beispiel z.b. Hänge face 3 an die TopoGeom, die schon face 2 hat an und lösche edge 8. Hänge face 4 an die TopoGeom, die schon face 1 hat an und lösche edge 13 usw.

```
CREATE OR REPLACE FUNCTION kvw_CloseTopoGaps(
  topo_name CHARACTER VARYING,
  table_name character varying,
  topo_geom_column CHARACTER VARYING)
  RETURNS BOOLEAN AS
$BODY$
  DECLARE
    sql text;
    gap RECORD;
```

```
 BEGIN
  sql = '
   SELECT
    gap.edge_id,
    g.' || topo_geom_column || ' AS geom_topo,
    gap.left_face,
    gap.right_face,
    r.topogeo_id
   FROM
    (
     SELECT DISTINCT ON (left_face)
      edge_id,
      left_face,
      right_face
     FROM
      ' || topo_name || '.edge_data ed LEFT JOIN
      ' || topo_name || '.relation rl ON ed.left_face = rl.element_id
     WHERE
      rl.element_id IS NULL AND
      ed.left_face > 0
    ) gap JOIN
    ' || topo_name || '.relation r ON gap.right_face = r.element_id JOIN
    ' || table_name || ' g ON r.topogeo_id = (g.' || topo_geom_column || ').id
  ';
  --RAISE NOTICE 'Loop over sql: %', sql;

  FOR gap IN EXECUTE sql LOOP
   RAISE NOTICE 'Close gap at face % by adding it to topogeom id %: % from face % and remove edge
%', gap.left_face, gap.topogeo_id, gap.geom_topo, gap.right_face, gap.edge_id;

   sql = '
    SELECT
     TopoGeom_addElement(' || topo_geom_column || ', ARRAY[' || gap.left_face || ', 3]::TopoElement)
    FROM
     ' || table_name || '
    WHERE
     (' || topo_geom_column || ').id = ' || gap.topogeo_id || '
   ';
   RAISE NOTICE 'Execute sql to add the face: %', sql;
   EXECUTE sql;

   sql = 'SELECT ST_RemEdgeModFace(''' || topo_name || ''', ' || gap.edge_id || ')';
   RAISE NOTICE 'Execute sql to remove edge: %', sql;
   EXECUTE sql;

  END LOOP;
  RETURN TRUE;
 END;
$BODY$
 LANGUAGE plpgsql VOLATILE COST 100;
```

Aufgerufen wird das mit:

```
SELECT kvw_CloseTopoGaps('topo', 'gemeinden_ausschnitt_poly', 'geom_topo');
```

Hier noch ein Beispiel:

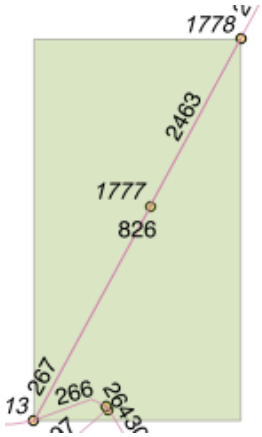



**Abbildung 26: Lücke mit 3 begrenzenden Kanten**

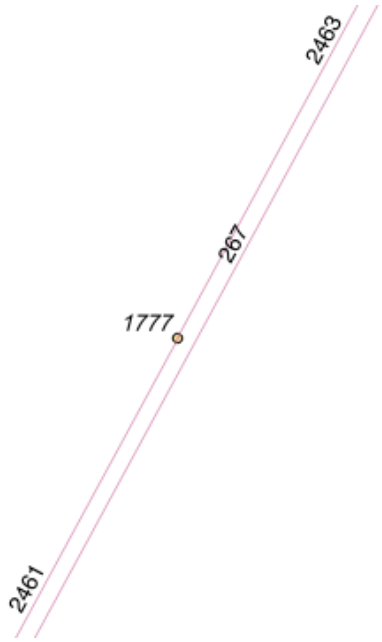



**Abbildung 27: Lücke mit 3 begrenzenden Kanten**

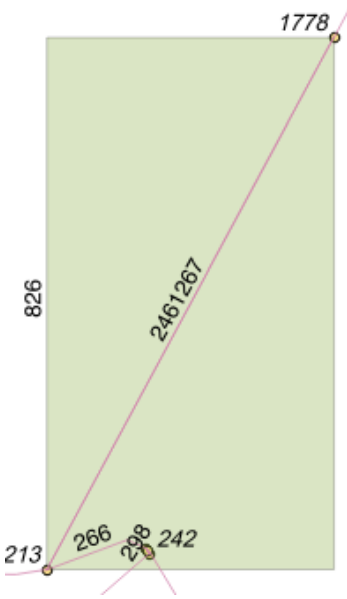



**Abbildung 28: Lücke nach gelöschtem Node. Nur noch 2 Kanten bleiben übrig.**

Das Face 826 wird begrenzt von edge 2461, 2463 und 267. Edge 2461 und 2463 haben auf der anderen Seite face 827 und edge 267 face 1633. Fragt man die Relation dieser faces ab, findet man nur Einträge für face 827 und 1633. Face 826 hat also keine Zuordnung zu einem Polygon und ist eine Lücke.

| edge_id integer | left_face integer | right_face integer |
|---|---|---|
| 267 | 826 | 1633 |
| 2461 | 826 | 827 |
| 2463 | 826 | 827 |

| topogeo_id integer | layer_id integer | element_id integer | element_type integer |
|---|---|---|---|
| 529 | 1 | 827 | 3 |
| 109 | 1 | 1633 | 3 |

Nun muss man entscheiden welche Kante gelöscht wird um die Lücke dem Nachbarn zuzuordnen. Im obigen Fall in Abbildung 15 gibt es nur zwei Kanten, die die Lücke begrenzen. In diesem Fall in Abbildung 26 und Abbildung 27 hat die Lücke mehr als nur zwei Kanten als Begrenzung. Es muss also entschieden werden welche Kante gelöscht wird. In diesem konkreten Fall könnte man einfach den node löschen, siehe Abschnitt 3.4.5. Damit hätte man dann wieder nur zwei begrenzende Kanten. Theoretisch könnte von diesem node aber auch eine edge abgehen. Dann bleibt die Lücke von 3 Kanten begrenzt und ich muss mich entscheiden welche. In Abbildung 29 kann man sehen, dass es noch weitere Beispiele von Lücken gibt, wo es nicht immer klar ist welche Kante gelöscht wird.

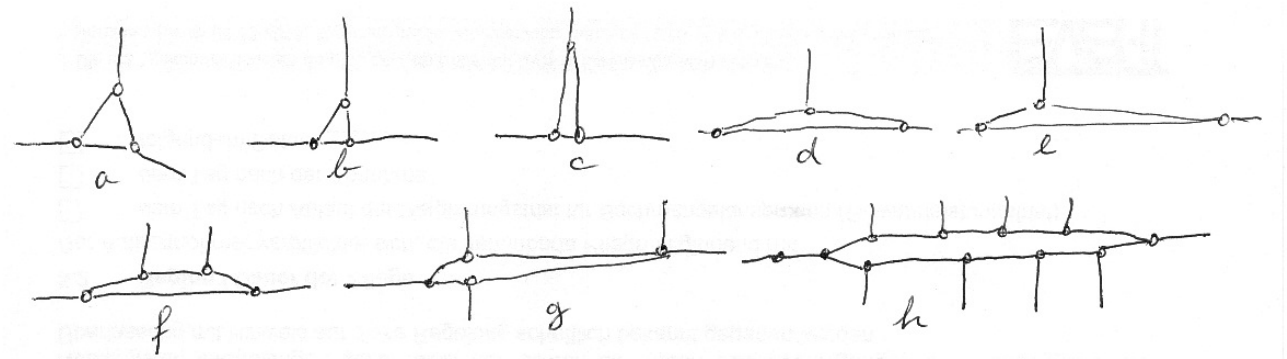



**Abbildung 29: Varianten von Lücken mit mehr als nur 2 umschließenden Kanten**

Im Fall a sind alle 3 Kanten gleich lang. Welcher Fläche soll die Lücke zugeordnet werden. Eigentlich müsste man jedem ein drittel zuordnen. In b sieht man, dass die Lücke vielleicht am ehesten zur linken oberen Fläche gehört. In c wiederum müsste man die Lücke an die Teilflächen oben links und rechts aufteilen. Auf

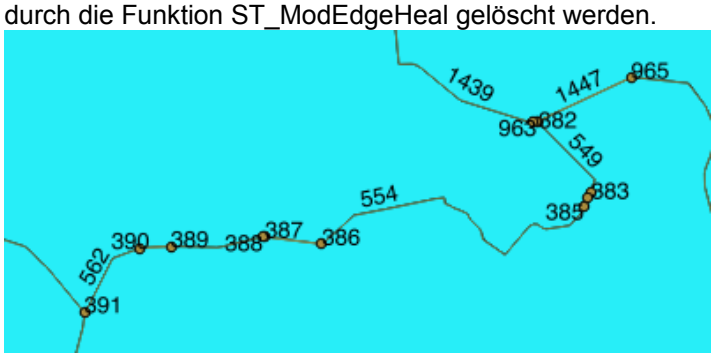
jeden Fall aber der unteren Fläche nichts zuordnen, weil dann wieder so ein schmaler Streifen entsteht. In d und e ist wohl auch klar, dass die Lücke der unteren Fläche zugeschlagen wird. In f ist es schon wieder schwieriger, weil es oben drei Flächen sind. Aber das Prinzip ist das gleiche. Wenn man die Lücke schließen will durch Löschung nur einer Kante, sieht es hinterher am besten aus, wenn man die längste Kante löscht. Beispiel g und h zeigen aber, dass auch dann spitze Ausbuchtungen entstehen können oder ungerechte Zuordnungen von Flächen. In Beispiel h wäre es in jedem Fall falsch die Lücke irgendeiner anderen Fläche allein zuzuordnen. In der Praxis kommen aber kaum Fälle ab f vor. In der Regel gibt es Berührungen mit maximal 3 Flächen. Die Beispiel b bis e zeigen, dass das Löschen der längsten Kante wohl die beste Lösung ist. Der Fall a dürfte kaum vorkommen und wenn dabei irgendeine Kante gelöscht wird ist das Resultat geometrisch gesehen auch nicht so schlecht (zumindest kein spitzer Winkel, sondern mindestens 60°).
Um die größte Kante zu finden muss der SQL-Teil wie folgt heißen:

```
SELECT DISTINCT ON (left_face)
  edge_id,
  left_face,
  right_face,
  St_Length(geom) AS length,
  count(edge_id) OVER (PARTITON BY left_face) AS num_edges
FROM
  ' || topo_name || '.edge_data ed LEFT JOIN
  ' || topo_name || '.relation rl ON ed.left_face = rl.element_id
WHERE
  rl.element_id IS NULL AND
  ed.left_face > 0
ORDER BY left_face, length DESC
```

**Fazit: In dem Algorithmus zum Füllen von Lücken müssen vorher Knoten von denen nur zwei Kanten abgehen, die selbst kein eigenes Face bilden gelöscht werden und anschließend die Lücke der Fläche zugewiesen werden, die die längste Kante an der Lücke hat.**

## 3.4.5 Topology vereinfachen

Nach allen Bereinigungen kommt es dennoch vor, dass Kanten unnötig durch einzelne nodes zersetzt sind, von denen nicht mehr als zwei edges abgehen. Diese nodes sind im Sinne der Topology unnütz und können durch die Funktion ST_ModEdgeHeal gelöscht werden.




Dazu muss man wissen welche das betrifft. Die folgende Abfrage findet alle edges Paare, die einen node zwischen sich haben von denen keine weiteren edges abgehen.

```
SELECT
  gdi_ModEdgeHealException('radrouten_topo', min(edge_id), max(edge_id))
FROM
(
  SELECT node_id, abs((GetNodeEdges('radrouten_topo', node_id)).edge) edge_id FROM
radrouten_topo.node order by node_id
) edges
```

```
GROUP BY node_id
HAVING count(node_id) = 2
```

Hier muss aber wohl auch noch abgefangen werden dass nodes nicht gelöscht werden wenn doch noch andere Objekte davon abhängen.

Man kann aber auch einfach alle Löschen und bei denen, die sich nicht löschen lassen, weil mehr als zwei edges dranhängen fängt man über eine Exception ab. So ist es in der Funktion gdi_ModEdgeHealException umgesetzt.

```
  CREATE OR REPLACE FUNCTION gdi_ModEdgeHealException(atopology CHARACTER VARYING,
                                anedge INTEGER, anotheredge INTEGER)
  RETURNS integer AS $$
DECLARE
  node_id INTEGER;
BEGIN
  SELECT ST_ModEdgeHeal($1, $2 , $3) INTO node_id;
  RETURN node_id;
EXCEPTION WHEN others THEN
  RETURN 0;
END;
$$ LANGUAGE plpgsql;
```

Jetzt kann man einfach alle edges links und rechts von einem löschen.

```
SELECT gdi_ModEdgeHealException('radrouten_topo', edge_id, abs_next_left_edge) from
radrouten_topo.edge_data;
SELECT gdi_ModEdgeHealException('radrouten_topo', edge_id, abs_next_right_edge) from
radrouten_topo.edge_data;
```

Beide Abfragen muss man aber mehrmals ausführen, weil sich nach dem Löschen von nodes ja wieder edges zusammengefunden haben können, die nur einen node haben, wo keine weiteren edges abgehen. Das ganze muss man so oft wiederholen, bis die Funktion nur noch Nullen liefert. Die Abfrage wird immer kürzer, weil es immer weniger edges gibt und diese immer länger (größer) werden, siehe auch Abschnitt 3.5.1.6.

## 3.5 Version 1

Schließlich wurde eine finale Version der Bereinigung der Flächen mit Topology erstellt. Dabei kommen folgende Funktionen zum Einsatz. Eine Beschreibung der Funktionen findet sich auch jeweils im Comment der Funktion.

#### 3.5.1.1 gdi_CreateTopo

```
CREATE OR REPLACE FUNCTION gdi_CreateTopo(
  schema_name CHARACTER VARYING,
  table_name character varying,
  id_column CHARACTER VARYING,
  geom_column CHARACTER VARYING,
  epsg INTEGER,
  simplify_tolerance DOUBLE PRECISION,
  simplify_angle_tolerance DOUBLE PRECISION,
  topo_tolerance DOUBLE PRECISION,
  area_tolerance DOUBLE PRECISION,
  prepare_topo BOOLEAN,
  expression CHARACTER VARYING
)
```

```
RETURNS BOOLEAN AS
$BODY$
  DECLARE
    sql text;
    polygon RECORD;
    topo_name CHARACTER VARYING = table_name || '_topo';
    debug BOOLEAN = false;
  BEGIN

    IF prepare_topo THEN
      -- Prepare the polygon topology
      IF debug THEN RAISE NOTICE 'Prepare Topology'; END IF;
      EXECUTE 'SELECT gdi_PreparePolygonTopo(''' || topo_name || ''', ''' || schema_name || ''', ''' ||
table_name || ''', ''' || id_column || ''', ''' || geom_column || ''', ' || epsg || ', ' || simplify_tolerance || ', ' ||
simplify_angle_tolerance || ', ' || topo_tolerance || ')';
    END IF;

    -- query polygons
    sql = '
      SELECT polygon_id AS id
      FROM ' || topo_name || '.topo_geom
      WHERE ' || expression || '
      ORDER BY polygon_id
    ';
    IF debug THEN RAISE NOTICE 'Query objects for loop: %', sql; END IF;
    FOR polygon IN EXECUTE sql LOOP
      RAISE NOTICE 'Create TopGeom for object with polygon_id = %', polygon.id;

      BEGIN
        EXECUTE '
          UPDATE ' || topo_name || '.topo_geom
          SET ' || geom_column || '_topo = toTopoGeom(' || geom_column || ', ''' || topo_name || ''', 1, ' ||
topo_tolerance || ')
          WHERE polygon_id = ' || polygon.id || '
        ';
        EXECUTE 'SELECT gdi_CleanPolygonTopo(''' || topo_name || ''', ''' || topo_name || ''', "topo_geom", '''
|| geom_column || '_topo", ' || area_tolerance || ', ' || polygon.id || ')';
      EXCEPTION WHEN OTHERS THEN
        RAISE WARNING 'Loading of record polygon_id: % failed: %', polygon.id, SQLERRM;
        EXECUTE '
          UPDATE ' || topo_name || '.topo_geom
          SET err_msg = ''' || SQLERRM || '''
          WHERE polygon_id = ' || polygon.id || '
        ';
      END;

    END LOOP;

    BEGIN
      EXECUTE 'SELECT gdi_RemoveTopoOverlaps(
        ''' || topo_name || ''',
        ''' || topo_name || ''',
        "topo_geom",
        ''' || id_column || ''',
        ''' || geom_column || ''',
        ''' || geom_column || '_topo"
```

```
    )';
  EXCEPTION WHEN OTHERS THEN
    RAISE WARNING 'Removing of Overlaps failed: %', SQLERRM;
  END;

  EXECUTE '
    INSERT INTO ' || topo_name || '.statistic (key, value, description) VALUES
    (
      "Fläche der gelöschten Topologieüberlappungen",
      (SELECT Round(Sum(ST_Area(face_geom))) FROM ' || topo_name || '.removed_overlaps),
      "m2"
    )
  ';

  EXECUTE 'SELECT gdi_RemoveNodesBetweenEdges("' || topo_name || '")';

  BEGIN
    EXECUTE 'SELECT gdi_CloseTopoGaps("' || topo_name || '", "' || topo_name || '", "topo_geom", "' ||
geom_column || '_topo")';
  EXCEPTION WHEN OTHERS THEN
    RAISE WARNING 'Closing of Gaps failed: %', SQLERRM;
  END;

  EXECUTE '
    INSERT INTO ' || topo_name || '.statistic (key, value, description) VALUES
    (
      "Fläche der gefüllten Topologielücken",
      (SELECT Round(Sum(ST_Area(face_geom))) FROM ' || topo_name || '.filled_gaps),
      "m2"
    )
  ';

  EXECUTE 'SELECT gdi_RemoveNodesBetweenEdges("' || topo_name || '")';

  -- Zurückschreiben der korrigierten Geometrien in die Spalte geom_column_topo_corrected in der
Originaltabelle
  RAISE NOTICE 'Schreibe korrigierte Geometrie zurück in Tabelle der Ausgangsdaten: %.% Spalte:
%_topo_corrected', schema_name, table_name , geom_column;
  EXECUTE '
    UPDATE
      ' || schema_name || '.' || table_name || ' AS alt
    SET
      ' || geom_column || '_topo_corrected = neu.geom
    FROM
      (
        SELECT
          id,
          ST_Multi(ST_Union(geom)) AS geom
        FROM
          (
            SELECT ' || id_column || ' AS id, ST_GetFaceGeometry("' || topo_name || '",
(GetTopoGeomElements(' || geom_column || '_topo))[1]) AS geom FROM ' || topo_name || '.topo_geom
          ) foo
        GROUP BY
          id
      ) neu
```

```
    WHERE
      alt.' || id_column || ' = neu.id
  ';

  EXECUTE 'CREATE INDEX ' || table_name || '_' || geom_column || '_topo_corrected_gist ON ' ||
schema_name || '.' || table_name || ' USING gist(' || geom_column || '_topo_corrected)';

  EXECUTE '
    INSERT INTO ' || topo_name || '.intersections (step, polygon_a_id, polygon_b_id, the_geom)
    SELECT
      "nach TopoCorrection",
      a.gid,
      b.gid,
      ST_Multi(
        ST_CollectionExtract(
          ST_MakeValid(st_intersection(a.the_geom_topo_corrected, b.the_geom_topo_corrected))
          ,3
        )
      )
    FROM
      ' || schema_name || '.' || table_name || ' a JOIN
      ' || schema_name || '.' || table_name || ' b ON (
        a.gid > b.gid AND
        ST_Intersects(a.the_geom_topo_corrected, b.the_geom_topo_corrected) AND
        NOT ST_Touches(a.the_geom_topo_corrected, b.the_geom_topo_corrected)
      )
    ORDER BY
      a.gid, b.gid
  ';

  EXECUTE '
    INSERT INTO ' || topo_name || '.statistic (key, value, description) VALUES
    (
      "Fläche der Überlappungen korrigierter Geometrien",
      COALESCE(
        (
          SELECT Round(Sum(ST_Area(the_geom)))
          FROM ' || topo_name || '.intersections
          WHERE step = "nach TopoCorrection"
        ),
        0
      ),
      "m2"
    ), (
      "Gesamtfläche korrigierter Geometrien", (
        SELECT Round(Sum(ST_Area(' || geom_column || '_topo_corrected)))/10000
        FROM ' || schema_name || '.' || table_name || '
      ),
      "ha"
    ), (
      "Anzahl Flächen nach Korrektur", (SELECT count(*) FROM ' || schema_name || '.' || table_name || '),
"Stück"
    ), (
      "Anzahl Stützpunkte hinterher", (
        SELECT Sum(ST_NPoints(' || geom_column || '_topo_corrected))
        FROM ' || schema_name || '.' || table_name || '
```

```
      ),
      "Stück"
    )
  ';

  EXECUTE '
    INSERT INTO ' || topo_name || '.statistic (key, value, description) VALUES
    (
      "Flächendifferenz nach - vor Topo-Korrektur",
      (SELECT round(value * 10000) FROM ' || topo_name || '.statistic WHERE key = "Gesamtfläche
korrigierter Geometrien") -
      (SELECT round(value * 10000) FROM ' || topo_name || '.statistic WHERE key = "Gesamtfläche nach
NoseRemove"),
      "m2"
    )
  ';


  EXECUTE '
    INSERT INTO ' || topo_name || '.statistic (key, value, description) VALUES
    (
      "Flächendifferenz gesamt",
      (SELECT round(value * 10000) FROM ' || topo_name || '.statistic WHERE key = "Gesamtfläche
korrigierter Geometrien") -
      (SELECT round(value * 10000) FROM ' || topo_name || '.statistic WHERE key = "Gesamtfläche
vorher"),
      "m2"
    )
  ';

  EXECUTE '
    INSERT INTO ' || topo_name || '.statistic (key, value, description) VALUES
    (
      "Flächendifferenz Lücken - Überlappungen",
      (SELECT value FROM ' || topo_name || '.statistic WHERE key = "Fläche der gefüllten
Topologielücken") -
      (SELECT value FROM ' || topo_name || '.statistic WHERE key = "Fläche der gelöschten
Topologieüberlappungen"),
      "m2"
    )
  ';

  EXECUTE '
    INSERT INTO ' || topo_name || '.statistic (key, value, description) VALUES
    (
      "Absoluter Betrag der Differenz der Flächendifferenz gesamt / Lücken-Überlappungen",
      ABS ((SELECT value FROM ' || topo_name || '.statistic WHERE key = "Flächendifferenz gesamt") -
      (SELECT value FROM ' || topo_name || '.statistic WHERE key = "Flächendifferenz Lücken -
Überlappungen")),
      "m2"
    )
  ';

  RETURN TRUE;
 END;
$BODY$
```

```
 LANGUAGE plpgsql VOLATILE COST 100;
COMMENT ON FUNCTION gdi_CreateTopo(character varying, character varying, character varying,
character varying, INTEGER, DOUBLE PRECISION, DOUBLE PRECISION, DOUBLE PRECISION,
DOUBLE PRECISION, BOOLEAN, character varying) IS 'Erzeugt die Topologie <table_name>_topo der
Tabelle <table_name> in der temporären Tabelle <table_name>_topo mit einer Toleranz von <tolerance>
für alle Geometrie aus Spalte <geom_column>, die der Bedingung <expression> genügen. Ist
<prepare_topo> false, wird PreparePolygonTopo nicht ausgeführt.';
```

Die Funktion gdi_CreateTopo ist die Hülle, die die Polygone topologisch korrekt erzeugt. Mit der Funktion gdi_CreateTopo kann man 3 Varianten fahren:

Nur Anlegen der Zwischentabelle, Ergebnisspalte und der Topologie

```
SELECT gdi_CreateTopo('public', 'gemeinden_mv', 'gid', 'the_geom', 25833, 0.1, 0.01, 0.001, 1, true,
'false');
```

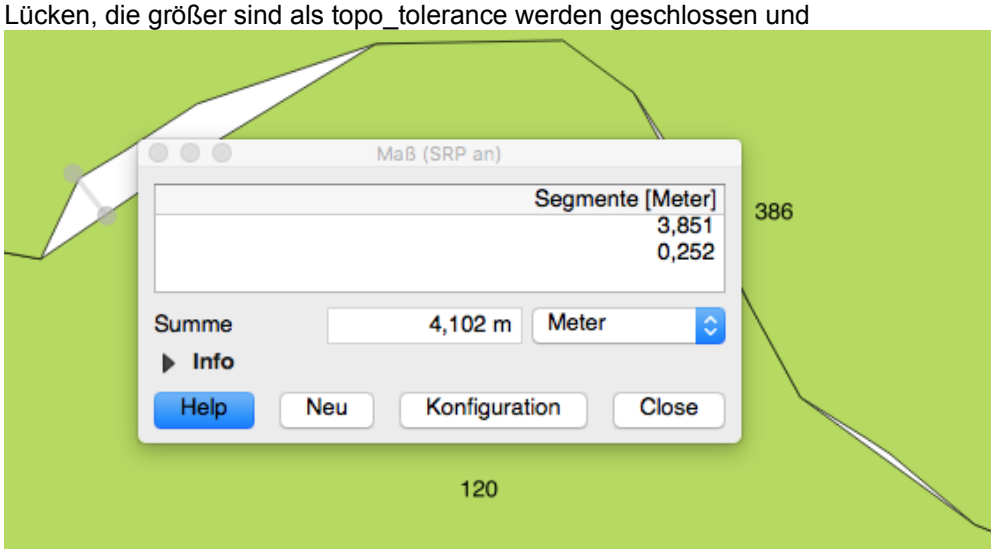
Bereinigen von einigen gegebenen Geometrien

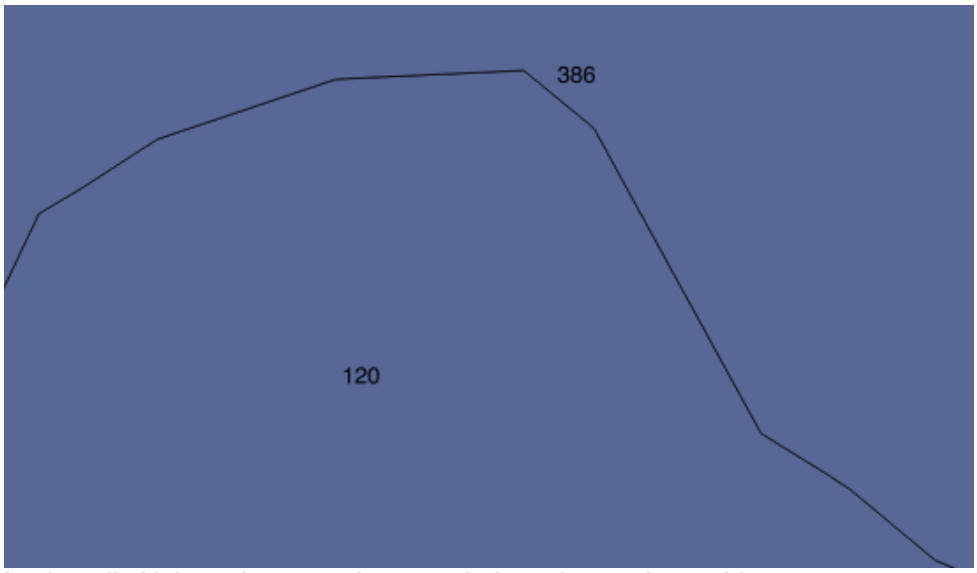
```
SELECT gdi_CreateTopo('public', 'gemeinden_mv', 'gid', 'the_geom', 25833, 0.1, 0.01, 0.001, 1, true, 'gid
IN (457, 120, 150, 304, 312, 333, 386, 425, 124, 662, 679, 685)');
```

Vorbereitung und Bereinigung aller Geometrien einer Tabelle

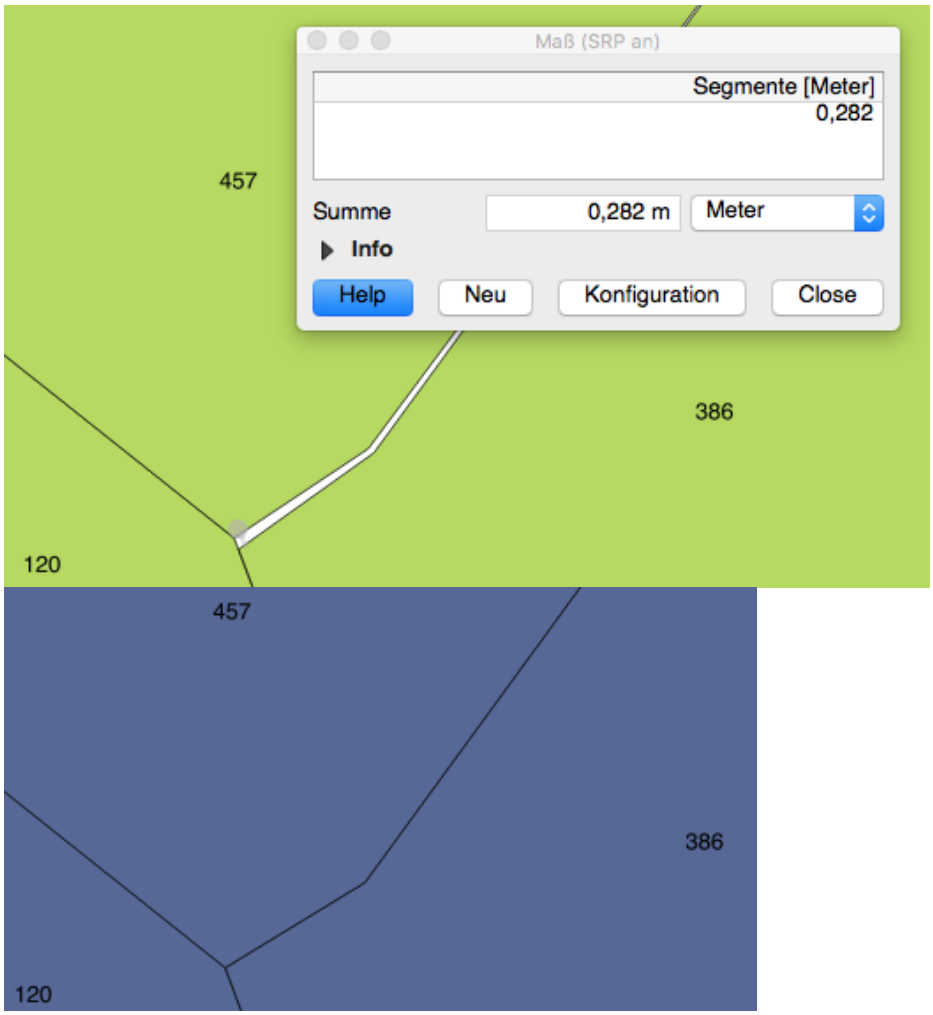
```
SELECT gdi_CreateTopo('public', 'gemeinden_mv', 'gid', 'the_geom', 25833, 0.1, 0.01, 0.001, 1, true,
'true');
```

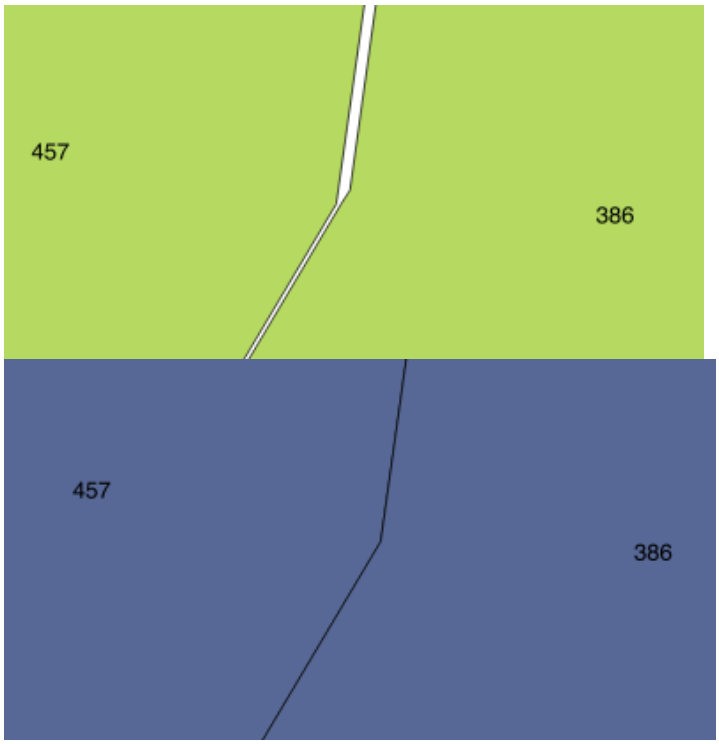
Der Parameter prepare_topo ist tatsächlich BOOLEAN und der letzte Parameter expression als CHARACTER VARYING zu definieren.
Lücken, die größer sind als topo_tolerance werden geschlossen und



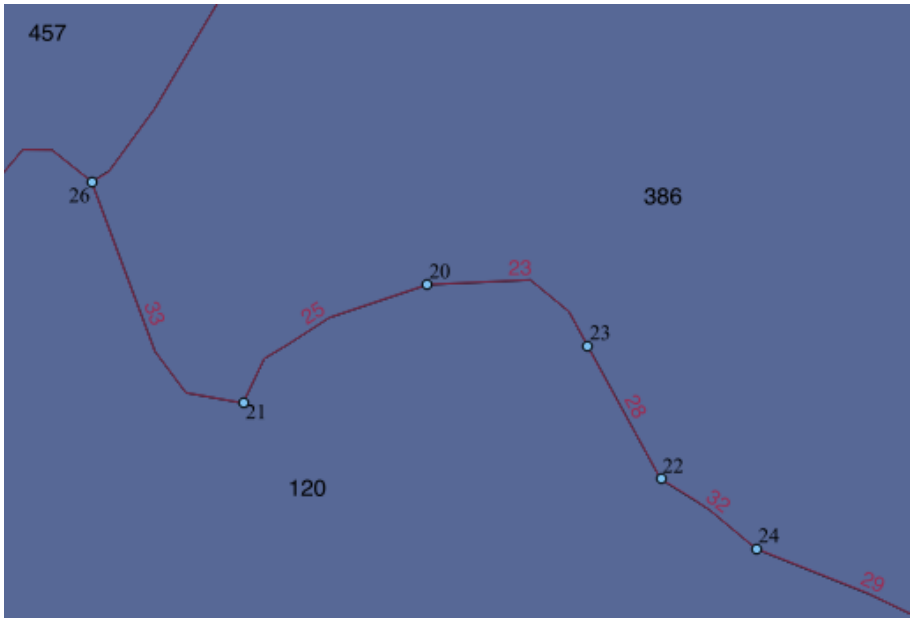



Lücken die kleiner als topo_tolerance sind werden auch geschlossen.






Eckpunkte von 3 Polygonen werden als ein Node zusammengefasst.




Die Werte topo_tolerance, distance_tolerance, angle_tolerance und area_tolerance können variiert werden. Es ergeben sich vor allem Fehlermeldungen wenn die topo_tolerance größer als die distance_tolerance ist. Dabei treten Fehler wie:
"Spatial exception - geometry intersects edge 1451",
"SQL/MM Spatial exception - curve not simple" oder
"Invalid edge (no two distinct vertices exist)" auf.
Im Beispiel traten bei distance_tolerance = 0,001 m und topo_tolerance = 10m beim Hinzufügen von 939 Polygonen zur Topologie 22 Fehler auf.

Bei distance_tolerance = 0.001 m und topo_tolerance = 1m traten keine Fehler auf.
Sinnvolle Werte für die Toleranzen sind:

- distance_tolerance = 0.1 (1dm)
- angle_tolerance = 0.01 (0.573 Grad)
- topo_tolerance = 0.001 (1 mm),
- area_tolerance = 1 (1 m²)

Zur Funktionsweise von gdi_CreateTopo: Im ersten Schritt wird die Funktion gdi_PreparePolygonTopo aufgerufen, siehe Abschnitt 3.5.1.2.

### 3.5.1.2 gdi_PreparePolygonTopo

In der Funktion gdi_PreparePolygonTopo wird zunächst geprüft ob die geforderten Parameter angegeben sind. Dann wird eine Topologie mit dem Namen <table_name>_topo angelegt. Das führt dazu, dass ein gleichnamiges Schema entsteht.

```
CREATE OR REPLACE FUNCTION gdi_PreparePolygonTopo(
  topo_name CHARACTER VARYING,
  schema_name CHARACTER VARYING,
  table_name character varying,
  id_column CHARACTER VARYING,
  geom_column CHARACTER VARYING,
  epsg_code INTEGER,
  distance_tolerance DOUBLE PRECISION,
  angle_tolerance DOUBLE PRECISION,
  topo_tolerance DOUBLE PRECISION
)
RETURNS BOOLEAN AS
$BODY$
  DECLARE
    sql text;
    result RECORD;
    debug BOOLEAN = FALSE;
  BEGIN

    -- Prüfe ob ein topo_name angegeben wurde
    IF topo_name = '' OR topo_name IS NULL THEN
      RAISE EXCEPTION 'Es muss ein Name für die Topographie angegeben werden!';
    END IF;

    -- Ersetze schema_name mit public when leer oder NULL und prüfe ob es das Schema gibt
    IF schema_name = '' OR schema_name IS NULL THEN
      schema_name = 'public';
    END IF;
    EXECUTE 'SELECT schema_name FROM information_schema.schemata WHERE schema_name = '''
|| schema_name || ''' INTO result;
    IF result IS NULL THEN
      RAISE EXCEPTION 'Schema % nicht gefunden!', schema_name;
    END IF;

    -- Prüfe ob ein Tabellenname angegeben wurde und ob es die Tabelle in dem Schema gibt
    IF table_name = '' OR table_name IS NULL THEN
      RAISE EXCEPTION 'Es muss ein Tabellenname angegeben werden!';
    END IF;
    EXECUTE 'SELECT table_name from information_schema.tables WHERE table_schema =  ''' ||
schema_name || ''' AND table_name = ''' || table_name || ''' INTO result;
    IF result IS NULL THEN
```

```
    RAISE EXCEPTION 'Tabelle % im Schema % nicht gefunden!', table_name, schema_name;
  END IF;

  -- Prüfe ob die id Spalte angegeben worden ist und in der Tabelle vorhanden ist
  IF id_column = '' OR id_column IS NULL THEN
    RAISE EXCEPTION 'Es muss ein Name für die ID-Spalte in der Tabelle angegeben sein!';
  END IF;
  EXECUTE 'SELECT column_name from information_schema.columns WHERE table_schema =  ''' ||
schema_name || ''' AND table_name = ''' || table_name || ''' AND column_name = ''' || id_column || '''' INTO
result;
  IF result IS NULL THEN
    RAISE EXCEPTION 'Spalte % in Tabelle % nicht gefunden!', id_column, table_name;
  END IF;

  -- Prüfe ob die geom Spalte angegeben worden, in der Tabelle vorhanden und vom Geometrietyp
Polygon oder MultiPolygon ist
  IF geom_column = '' OR geom_column IS NULL THEN
    RAISE EXCEPTION 'Es muss ein Name für die Geometriespalte in der Tabelle angegeben sein!';
  END IF;
  EXECUTE 'SELECT type FROM geometry_columns WHERE f_table_schema =  ''' || schema_name || '''
AND f_table_name = ''' || table_name || ''' AND f_geometry_column = ''' || geom_column || '''' INTO result;
  IF result IS NULL THEN
    RAISE EXCEPTION 'Geometriespalte % in Tabelle % nicht gefunden oder nicht vom Typ geometry!',
geom_column, table_name;
  END IF;
  IF Position('POLYGON' IN result.type) = 0 THEN
    RAISE EXCEPTION 'Geometriespalte % in Tabelle % ist nicht vom Typ Polygon oder MultiPolygon!',
geom_column, table_name;
  END IF;

  -- Prüfe ob die epsg_code Spalte angegeben worden ist und ob es diesen gibt
  IF epsg_code IS NULL THEN
    RAISE EXCEPTION 'Es muss ein EPSG-Code für die zu korrigierende Geometriespalte in der Tabelle
angegeben sein!';
  END IF;
  EXECUTE 'SELECT srid FROM spatial_ref_sys WHERE srid = ' || epsg_code INTO result;
  IF result IS NULL THEN
    RAISE EXCEPTION 'EPSG-Code % existiert nicht!', epsg_code;
  END IF;

  -- Prüfe ob die distance-Toleranz angegeben worden ist
  IF distance_tolerance = 0 OR distance_tolerance IS NULL THEN
    RAISE EXCEPTION 'Es muss eine Toleranz angegeben werden für die Löschung von zu eng
beieinander liegenden Punkten!';
  END IF;

  -- Prüfe ob die Topo-Toleranz angegeben worden ist
  IF topo_tolerance = 0 OR topo_tolerance IS NULL THEN
    RAISE EXCEPTION 'Es muss eine Toleranz angegeben werden für die Bildung der Topologie!';
  END IF;

  -- drop topology
  IF debug THEN RAISE NOTICE 'Drop Topology: %', topo_name; END IF;
  EXECUTE '
    SELECT topology.DropTopology(''' || topo_name || ''')
    WHERE EXISTS (
```

```
      SELECT * FROM topology.topology WHERE name = ''' || topo_name || '''
    )
  ';
  -- create topology
  if debug THEN RAISE NOTICE 'Create Topology: %', topo_name; END IF;
  EXECUTE 'SELECT topology.CreateTopology(''' || topo_name || ''', '|| epsg_code || ', ' || topo_tolerance ||
')';

  -- create tables for logging results
  if debug THEN RAISE NOTICE 'Create tables for logging results'; END IF;

  EXECUTE 'DROP TABLE IF EXISTS ' || topo_name || '.intersections';
  EXECUTE '
    CREATE TABLE ' || topo_name || '.intersections (
      step character varying,
      polygon_a_id integer,
      polygon_b_id integer,
      the_geom geometry(MULTIPOLYGON, ' || epsg_code || '),
      CONSTRAINT intersections_pkey PRIMARY KEY (step, polygon_a_id, polygon_b_id)
    )
  ';

  EXECUTE 'DROP TABLE IF EXISTS ' || topo_name || '.removed_spikes';
  EXECUTE '
    CREATE TABLE ' || topo_name || '.removed_spikes (
      id serial,
      polygon_id integer,
      geom geometry(POINT, ' || epsg_code || '),
      CONSTRAINT removed_spikes_pkey PRIMARY KEY (id)
    )
  ';

  EXECUTE 'DROP TABLE IF EXISTS ' || topo_name || '.removed_overlaps';
  EXECUTE '
    CREATE TABLE ' || topo_name || '.removed_overlaps (
      polygon_id integer,
      face_id integer,
      face_geom geometry(POLYGON, ' || epsg_code || '),
      CONSTRAINT removed_overlaps_pkey PRIMARY KEY (polygon_id, face_id)
    )
  ';

  EXECUTE 'DROP TABLE IF EXISTS ' || topo_name || '.filled_gaps';
  EXECUTE '
    CREATE TABLE ' || topo_name || '.filled_gaps (
      polygon_id integer,
      face_id integer,
      num_edges integer,
      face_geom geometry(POLYGON, ' || epsg_code || '),
      CONSTRAINT filled_gaps_pkey PRIMARY KEY (polygon_id, face_id)
    )
  ';

  EXECUTE 'DROP TABLE IF EXISTS ' || topo_name || '.statistic';
  EXECUTE '
    CREATE TABLE ' || topo_name || '.statistic (
```

```
      nr serial,
      key character varying,
      value double precision,
      description text,
      CONSTRAINT statistic_pkey PRIMARY KEY (nr)
    )
  ';

  EXECUTE 'DROP TABLE IF EXISTS ' || topo_name || '.removed_nodes';
  EXECUTE '
    CREATE TABLE ' || topo_name || '.removed_nodes (
      node_id integer,
      geom geometry(POINT, ' || epsg_code || '),
      CONSTRAINT removed_nodes_pkey PRIMARY KEY (node_id)
    )
  ';

  if debug THEN RAISE NOTICE 'Write first 7 statistics data'; END IF;
  EXECUTE '
    INSERT INTO ' || topo_name || '.statistic (key, value, description) VALUES
    (''Name der Topologie'', NULL, ''' || topo_name || '''),
    (''Ursprüngliche Tabelle'', NULL, '''|| schema_name || '.' || table_name || '''),
    (''Geometriespalte'', NULL, ''' || geom_column || '''),
    (''Distanz Toleranz'', ' || distance_tolerance || ', ''m''),
    (''Angle Toleranz'', ' || angle_tolerance || ', ''m''),
    (''Topology Toleranz'', ' || topo_tolerance || ', ''m''),
    (''Gesamtfläche vorher'', (SELECT Round(Sum(ST_Area(ST_Transform(' || geom_column || ', ' ||
epsg_code || '))))/10000 FROM ' || schema_name || '.' || table_name || '), ''ha''),
    (''Anzahl Flächen vorher'', (SELECT count(*) FROM ' || schema_name || '.' || table_name || '), ''Stück''),
    (''Anzahl Stützpunkte vorher'', (SELECT Sum(ST_NPoints(' || geom_column || ')) FROM ' ||
schema_name || '.' || table_name || '), ''Stück'')
  ';

  RAISE NOTICE 'Calculate Intersections of original geometry in table intersections.';
  EXECUTE '
    INSERT INTO ' || topo_name || '.intersections (step, polygon_a_id, polygon_b_id, the_geom)
    SELECT
      ''vor Polygonaufbereitung'' AS step,
      a.gid AS polygon_a_id,
      b.gid AS polygon_b_id,
      ST_Transform(ST_Multi(
        ST_CollectionExtract(
          ST_MakeValid(st_intersection(a.the_geom, b.the_geom))
          ,3
        )
      ), ' || epsg_code || ') AS the_geom
    FROM
      ' || schema_name || '.' || table_name || ' a JOIN
      ' || schema_name || '.' || table_name || ' b ON ST_Intersects(a.the_geom, b.the_geom) AND a.gid >
b.gid AND NOT ST_Touches(a.the_geom, b.the_geom)
    ORDER BY
      a.gid, b.gid
  ';

  RAISE NOTICE 'Create table % in topology schema % and prepare polygons.', table_name,
topo_name;
```

```
  -- create working table for topological corrected polygons
  EXECUTE 'DROP TABLE IF EXISTS ' || topo_name || '.topo_geom';
  EXECUTE '
    CREATE TABLE ' || topo_name || '.topo_geom AS
    SELECT
      f.' || id_column || ',
      ST_GeometryN(
        f.geom,
        generate_series(
          1,
          ST_NumGeometries(f.geom)
        )
      ) AS ' || geom_column || ',
      ""::CHARACTER VARYING AS err_msg
    FROM
      (
        SELECT
          ' || id_column || ',
          ST_SimplifyPreserveTopology(
            ST_CollectionExtract(
              ST_MakeValid(
                ST_Transform(
                  ST_GeometryN(
                    ' || geom_column || ',
                    generate_series(
                      1,
                      ST_NumGeometries(' || geom_column || ')
                    )
                  ),
                  ' || epsg_code || '
                )
              ),
              3
            ),
            ' || distance_tolerance || '
          ) AS geom
        FROM
          ' || schema_name || '.' || table_name || '
      ) f
    ORDER BY ' || id_column || '
  ';

  IF debug THEN RAISE NOTICE 'Add columns polygon_id, %_topo, %_corrected_geom and indexes',
table_name, table_name; END IF;
  BEGIN
    EXECUTE 'CREATE INDEX ' || table_name || '_' || geom_column ||'_gist ON ' || schema_name || '.' ||
table_name || ' USING gist(' || geom_column || ')';
  EXCEPTION
    WHEN duplicate_table
    THEN RAISE NOTICE 'Index: %_%_gist on table: % already exists, skipping!', table_name,
geom_column, table_name;
  END;
  EXECUTE 'ALTER TABLE ' || topo_name || '.topo_geom ADD COLUMN polygon_id serial NOT NULL';
  EXECUTE 'ALTER TABLE ' || topo_name || '.topo_geom ADD CONSTRAINT ' || table_name ||
'_topo_pkey PRIMARY KEY (polygon_id)';
  EXECUTE 'CREATE INDEX topo_geom_' || id_column || '_idx ON ' || topo_name || '.topo_geom USING
```

```
btree (' || id_column || ')';
    EXECUTE 'CREATE INDEX topo_geom_' || geom_column || '_gist ON ' || topo_name || '.topo_geom
USING gist(' || geom_column || ')';
    EXECUTE 'SELECT AddTopoGeometryColumn('' || topo_name || '', '' || topo_name || '', "topo_geom", ''
|| geom_column || '_topo", "Polygon")';
    IF debug THEN RAISE NOTICE 'Drop column %_topo_corrected if exists!', geom_column; END IF;
    EXECUTE 'ALTER TABLE ' || schema_name || '.' || table_name || ' DROP COLUMN IF EXISTS ' ||
geom_column || '_topo_corrected';
    EXECUTE 'SELECT AddGeometryColumn('' || schema_name || '', '' || table_name || '', '' ||
geom_column || '_topo_corrected", ' || epsg_code || ', "MultiPolygon", 2)';

    RAISE NOTICE 'Calculate intersections after polygon preparation in table intersections';
    EXECUTE '
      INSERT INTO ' || topo_name || '.intersections (step, polygon_a_id, polygon_b_id, the_geom)
      SELECT
        "nach Polygonaufbereitung",
        a.polygon_id,
        b.polygon_id,
        ST_Multi(
          ST_CollectionExtract(
            ST_MakeValid(st_intersection(a.the_geom, b.the_geom))
            ,3
          )
      )
      FROM
        ' || topo_name || '.topo_geom a JOIN
        ' || topo_name || '.topo_geom b ON ST_Intersects(a.the_geom, b.the_geom) AND a.polygon_id >
b.polygon_id AND NOT ST_Touches(a.the_geom, b.the_geom)
      ORDER BY
        a.polygon_id, b.polygon_id
    ';

    EXECUTE '
      INSERT INTO ' || topo_name || '.statistic (key, value, description) VALUES
      (
        "Fläche der Überlappungen nach Polygonaufbereitung",
        (
          SELECT Round(Sum(ST_Area(the_geom)))
          FROM ' || topo_name || '.intersections
          WHERE step = "nach Polygonaufbereitung"
        ),
        "m2"
      ), (
        "Gesamtfläche nach Polygonaufbereitung",
        (
          SELECT
            Round(Sum(ST_Area(' || geom_column || ')))/10000
          FROM
            ' || topo_name || '.topo_geom
        ),
        "ha"
      ), (
        "Anzahl der Polygone", (SELECT count(*) FROM ' || topo_name || '.topo_geom), "Stück")
    ';

    EXECUTE '
```

```
    INSERT INTO ' || topo_name || '.statistic (key, value, description) VALUES
    (
      "Flächendifferenz nach - vor Polygonaufbereitung",
      (SELECT Round(value * 10000) FROM ' || topo_name || '.statistic WHERE key = "Gesamtfläche nach
Polygonaufbereitung") -
      (SELECT Round(value * 10000) FROM ' || topo_name || '.statistic WHERE key = "Gesamtfläche
vorher"),
      "m2"
    )
  ';

  RAISE NOTICE 'Do NoseRemove and update topo_geom.';
  EXECUTE '
    UPDATE ' || topo_name || '.topo_geom
    SET ' || geom_column || ' = gdi_NoseRemove("' || topo_name || '", polygon_id, ' || geom_column || ', ' ||
angle_tolerance || ', ' || distance_tolerance || ')
  ';

  RAISE NOTICE 'Calculate Intersection after NoseRemove in table intersections.';
  EXECUTE '
    INSERT INTO ' || topo_name || '.intersections (step, polygon_a_id, polygon_b_id, the_geom)
    SELECT
      "nach NoseRemove",
      a.polygon_id,
      b.polygon_id,
      ST_Multi(
        ST_CollectionExtract(
          ST_MakeValid(st_intersection(a.the_geom, b.the_geom))
          ,3
        )
    )
    FROM
      ' || topo_name || '.topo_geom a JOIN
      ' || topo_name || '.topo_geom b ON ST_Intersects(a.the_geom, b.the_geom) AND a.polygon_id >
b.polygon_id AND NOT ST_Touches(a.the_geom, b.the_geom)
    ORDER BY
      a.polygon_id, b.polygon_id
  ';

  EXECUTE '
    INSERT INTO ' || topo_name || '.statistic (key, value, description) VALUES
    (
      "Fläche der Überlappungen nach NoseRemove",
      (
        SELECT Round(Sum(ST_Area(the_geom)))
        FROM ' || topo_name || '.intersections
        WHERE step = "nach NoseRemove"
      ),
      "m2"
    ), (
      "Gesamtfläche nach NoseRemove",
      (
        SELECT
          Round(Sum(ST_Area(' || geom_column || ')))/10000
        FROM
          ' || topo_name || '.topo_geom
```

```
      ),
      "ha"
    )
  ';

  EXECUTE '
    INSERT INTO ' || topo_name || '.statistic (key, value, description) VALUES
    (
      "Flächendifferenz nach - vor NoseRemove",
      (SELECT Round(value * 10000) FROM ' || topo_name || '.statistic WHERE key = "Gesamtfläche nach
NoseRemove") -
      (SELECT Round(value * 10000) FROM ' || topo_name || '.statistic WHERE key = "Gesamtfläche nach
Polygonaufbereitung"),
      "m2"
    );
  ';

  RETURN TRUE;
 END;
$BODY$
 LANGUAGE plpgsql VOLATILE COST 100;
COMMENT ON FUNCTION gdi_preparepolygontopo(character varying, character varying, character
varying, character varying, character varying, integer, DOUBLE PRECISION, DOUBLE PRECISION,
DOUBLE PRECISION) IS 'Bereitet die Erzeugung einer Topologie vor in dem die Geometrien der
betroffenen Tabelle zunächst in einzelne Polygone zerlegt, transformiert, valide und mit distance_tolerance
vereinfacht werden. Die Polygone werden in eine temporäre Tabelle kopiert und dort eine TopGeom Spalte
angelegt. Eine vorhandene Topologie und temporäre Tabelle mit gleichem Namen wird vorher gelöscht.';
```

Im Topologieschema werden alle für die Bereinigung der Geometrien verwendeten Daten abgelegt, außer die finale korrigierte Geometrie. Die wird in eine neu angelegte Spalte <*geom_column*>_topo_corrected in der Originaltabelle gespeichert in dem System welches mit *epsg_code* angegeben ist.
Die MultiPolygone werden in der Tabelle topo_geom in Polygone aufgeteilt, valide gemacht nach *epsg_code* transformiert sowie mit gdi_NoseRemove (Abschnitt 3.5.1.4 und 3.5.1.3) Nasen und Kerben entfernt.
Im zweiten Schritt werden die Polygone selektiert, für die die Bereinigung stattfinden soll. In einer Schleife werden diese der Topology mit *topo_tolerance* hinzugefügt und gdi_CleanPolygonTopo (Abschnitt 3.5.1.9) aufgerufen. Damit werden zu kleine geschlossene Polygone und Kanten ohne Relationen zu Faces entfernt.
Kann ein Polygon nicht zur Topology hinzugefügt werden, weil ein Fehler auftritt, wird statt dessen eine Fehlermeldung in die Spalte err_msg der Tabelle topo_geom geschrieben.

Nach Ablauf der Schleife werden die Funktionen gdi_RemoveTopoOverlaps (Abschnitt 3.5.1.5) und gdi_CloseTopoGaps (Abschnitt 3.5.1.7) aufgerufen, die Überlappungen und Lücken zwischen den topologischen Flächen entfernen.

#### 3.5.1.3 gdi_NoseRemoveCore

```
CREATE OR REPLACE FUNCTION public.gdi_NoseRemoveCore(
  topo_name CHARACTER VARYING,
  polygon_id INTEGER,
  geometry,
  angle_tolerance double precision,
  distance_tolerance double precision)
RETURNS geometry AS
$BODY$
DECLARE
  ingeom    alias for $3;
  lineusp geometry;
```

```
  linenew geometry;
  newgeom geometry;
  testgeom varchar;
  remove_point boolean;
  removed_point_geom geometry;
  newb boolean;
  changed boolean;
  point_id integer;
  numpoints integer;
  angle_in_point float;
  distance_to_next_point FLOAT;
DECLARE
  debug BOOLEAN = FALSE;
BEGIN
  -- input geometry or rather set as default for the output
  newgeom := ingeom;

  IF debug THEN RAISE NOTICE 'Start function gdi_NoseRemoveCore'; END IF;
  -- check polygon
  if (select ST_GeometryType(ingeom)) = 'ST_Polygon' then
    IF (SELECT debug) THEN RAISE NOTICE 'ingeom is of type ST_Polygon'; END IF;
    IF (SELECT ST_NumInteriorRings(ingeom)) = 0 then
      IF (SELECT debug) THEN RAISE NOTICE 'num interior ring is 0'; END IF;
      --save the polygon boundary as a line
      lineusp := ST_Boundary(ingeom) as line;
      -- number of tags
      numpoints := ST_NumPoints(lineusp);
      IF (numpoints > 3) THEN
        IF (SELECT debug) THEN RAISE NOTICE 'num points of the line: %', numpoints; END IF;
        -- default value of the loop indicates if the geometry has been changed
        newb := true;
        -- globale changevariable
        changed := false;

        -- loop (to remove several points)
        WHILE newb = true loop
          -- default values
          remove_point := false;
          newb := false;
          point_id := 1;
          numpoints := ST_NumPoints(lineusp) - 1;
          IF (numpoints > 3) THEN
            -- the geometry passes pointwisely until spike has been found and point removed
            WHILE (point_id <= numpoints) AND (remove_point = false) LOOP
              -- the check of the angle at the current point of a spike including the special case, that it is the first
point.
                angle_in_point = (
                select
                  abs(
                    pi() -
                    abs(
                      ST_Azimuth(
                        ST_PointN(lineusp, case when point_id = 1 then ST_NumPoints(lineusp) - 1 else point_id -
1 end),
                        ST_PointN(lineusp, point_id)
                      ) -
```

```
                ST_Azimuth(
                  ST_PointN(lineusp, point_id),
                  ST_PointN(lineusp, point_id + 1)
                )
              )
            )
          );
          distance_to_next_point = (
            SELECT ST_Distance(
              ST_PointN(lineusp, point_id),
              ST_PointN(lineusp, point_id + 1)
            )
          );
          IF debug THEN RAISE NOTICE 'P: %, d: %, ß: %, a in P % (%): %, a in P % (%): %',
            point_id,
            distance_to_next_point,
            angle_in_point,
            case when point_id = 1 then ST_NumPoints(lineusp) - 1 else point_id - 1 end,
            ST_AsText(ST_PointN(lineusp, case when point_id = 1 then ST_NumPoints(lineusp) - 1 else
point_id - 1 end)),
            ST_Azimuth(
              ST_PointN(lineusp, case when point_id = 1 then ST_NumPoints(lineusp) - 1 else point_id - 1
end),
              ST_PointN(lineusp, point_id)
            ),
            point_id,
            ST_AsText(ST_PointN(lineusp, point_id)),
            ST_Azimuth(
              ST_PointN(lineusp, point_id),
              ST_PointN(lineusp, point_id + 1)
            );
          END IF;

          IF angle_in_point < angle_tolerance OR distance_to_next_point < distance_tolerance then
            -- remove point
            removed_point_geom = ST_PointN(lineusp, point_id + 1);
            linenew := ST_RemovePoint(lineusp, point_id - 1);

            IF linenew is not null THEN
              RAISE NOTICE '---> point % removed (%)', point_id, ST_AsText(removed_point_geom);
              EXECUTE '
                INSERT INTO ' || topo_name || '.removed_spikes (polygon_id, geom) VALUES
                  (' || polygon_id || ', ''' || removed_point_geom::text || ''')
              ';
              lineusp := linenew;
              remove_point := true;

              -- if the first point is concerned, the last point must also be changed to close the line again.
              IF point_id = 1 THEN
                linenew := ST_SetPoint(lineusp, numpoints - 2, ST_PointN(lineusp, 1));
                lineusp := linenew;
              END IF;
            END IF;
          END IF;
          point_id = point_id + 1;
        END LOOP; -- end of pointwisely loop to remove a spike
```

```
        END IF;

        -- remove point
        IF remove_point = true then
          numpoints := ST_NumPoints(lineusp);
          newb := true;
          point_id := 0;
          changed := true;
        END IF; -- point has been removed
      END LOOP; -- end of loop to remove several points

      --with the change it is tried to change back the new line geometry in a polygon. if this is not possible,
the existing geometry is used
      IF changed = true then
        newgeom :=  ST_BuildArea(lineusp) as geom;
        -- errorhandling
        IF newgeom is not null THEN
          raise notice 'new geometry created!';
        ELSE
          newgeom := ingeom;
          raise notice '-------------- area could not be created !!! --------------';
          testgeom := ST_AsText(lineusp);
          raise notice 'geometry %', testgeom;
        END IF; -- newgeom is not null
      END IF; -- geom has been changed
    ELSE
      IF (SELECT debug) THEN RAISE NOTICE 'Break loop due to num points of the line is only %',
numpoints; END IF;
    END IF; -- ingeom has more than 3 points
  end if; -- ingeom has 0 interior rings
 end if; -- ingeom is of type ST_Polygon
 -- return value
 RETURN newgeom;
END;
$BODY$
LANGUAGE plpgsql VOLATILE COST 100;
COMMENT ON FUNCTION gdi_NoseRemoveCore(CHARACTER VARYING, INTEGER, geometry, double
precision, double precision) IS 'Entfernt schmale Nasen und Kerben in der Umrandung von Polygonen
durch abwechslendes Löschen von Punkten mit Abständen < <distance_tolerance> und von
Scheitelpunkten mit spitzen Winkeln < <angle_tolerance> in arc';
```

#### 3.5.1.4 gdi_NoseRemove

Vor dem Ausführen der Funktion können Polygone aussehen wie in Abbildung 30.


**Abbildung 30: Nase am Polygon**

Nach der Ausführung ist diese Nase entfernt, siehe Abbildung 30 rechts.

```
CREATE OR REPLACE FUNCTION public.gdi_NoseRemove(
  topo_name CHARACTER VARYING,
  polygon_id INTEGER,
  geometry,
  angle double precision,
  tolerance double precision)
RETURNS geometry AS
$BODY$
  SELECT ST_MakePolygon(
    (
      --outer ring of polygon
      SELECT ST_ExteriorRing(gdi_NoseRemoveCore($1, $2, geom, $4, $5)) as outer_ring
      FROM ST_DumpRings($3)
      where path[1] = 0
    ),
    array(
      --all inner rings
      SELECT ST_ExteriorRing(gdi_NoseRemoveCore($1, $2, geom, $4, $5)) as inner_rings
      FROM ST_DumpRings($3)
      WHERE path[1] > 0
    )
) as geom
$BODY$
LANGUAGE sql IMMUTABLE COST 100;
COMMENT ON FUNCTION gdi_NoseRemove(CHARACTER VARYING, INTEGER, geometry, double
precision, double precision) IS 'Entfernt schmale Nasen und Kerben in Polygongeometrie durch Aufruf von
der Funktion gdi_NoseRemoveCore für jeden inneren und äußeren Ring und anschließendes wieder
zusammenfügen zu Polygon.';
```

#### 3.5.1.5 gdi_RemoveTopoOverlaps

```
CREATE OR REPLACE FUNCTION public.gdi_RemoveTopoOverlaps(
    topo_name character varying,
    schema_name character varying,
    table_name character varying,
    id_column character varying,
    geom_column character varying,
    topo_geom_column character varying)
  RETURNS boolean AS
$BODY$
  DECLARE
    sql text;
    polygon RECORD;
    debug BOOLEAN = false;
  BEGIN
    -- Finde alle Polygone, die mehr als eine
    sql = '
      SELECT
        t.' || id_column || ' AS id,
        r.topogeo_id,
        ST_Area(t.' || geom_column || '),
        count(topogeo_id)
      FROM
        ' || schema_name || '.' || table_name || ' t JOIN
```

```
      ' || topo_name || '.relation r ON (t.' || topo_geom_column || ').id = r.topogeo_id
    GROUP BY t.' || id_column || ', r.topogeo_id, ST_Area(t.' || geom_column || ')
    HAVING count(r.topogeo_id) > 1
    ORDER BY ST_Area(t.' || geom_column || ')
  ';
  IF debug THEN RAISE NOTICE 'Finde sich überlappende faces in schema: % tabelle: % mit sql: %',
schema_name, table_name, sql; END IF;

  FOR polygon IN EXECUTE sql LOOP
    RAISE NOTICE 'Remove and log overlaping faces for polygon_id %', polygon.id;
    sql = '
      INSERT INTO ' || topo_name || '.removed_overlaps (polygon_id, face_id, face_geom)
      SELECT
        ' || polygon.id || ',
        face_id,
        face_geom
      FROM
        (
          SELECT
            r1.element_id face_id,
            ST_GetFaceGeometry('' || topo_name || '', r1.element_id) AS face_geom,
            TopoGeom_remElement(t1.' || topo_geom_column || ', ARRAY[r1.element_id,
3]::topology.TopoElement)
          FROM
            ' || topo_name  || '.relation r1 JOIN
            ' || topo_name  || '.relation r2 ON r1.element_id = r2.element_id JOIN
            ' || schema_name  || '.' || table_name || ' t1 ON r1.topogeo_id = (t1.' || topo_geom_column || ').id
JOIN
            ' || schema_name  || '.' || table_name || ' t2 ON r2.topogeo_id = (t2.' || topo_geom_column || ').id
          WHERE
            r1.topogeo_id != r2.topogeo_id AND
            r2.topogeo_id = ' || polygon.topogeo_id || '
        ) AS overlaps_table
    ';
    IF debug THEN RAISE NOTICE 'Execute sql: % to remove the faces of polygon: %', sql, polygon.id;
END IF;
    EXECUTE sql;

    sql = '
      SELECT
        ST_RemEdgeModFace('' || topo_name || '', e.edge_id)
      FROM
        ' || topo_name || '.edge_data e JOIN
        ' || topo_name || '.relation r1 ON e.left_face = r1.element_id JOIN
        ' || topo_name || '.relation r2 ON e.right_face = r2.element_id AND r1.topogeo_id = r2.topogeo_id
      WHERE
        r1.topogeo_id = ' || polygon.topogeo_id || '
    ';
    if debug THEN RAISE NOTICE 'Execute sql: % to remove edges in polygon: %', sql, polygon.id; END
IF;
    EXECUTE sql;

    sql = '
      SELECT
        ST_RemoveIsoNode('' || topo_name || '', node_id)
      FROM
```

```
        ' || topo_name || '.relation r JOIN
        ' || topo_name || '.node n ON r.element_id = n.containing_face
      WHERE
        r.topogeo_id = ' || polygon.topogeo_id || '
    ';
    if debug THEN RAISE NOTICE 'Execute sql: % to remove isolated nodes in polygon: %', sql,
polygon.id; END IF;
    EXECUTE sql;

  END LOOP;
  RETURN TRUE;
 END;
$BODY$
LANGUAGE plpgsql VOLATILE COST 100;
COMMENT ON FUNCTION public.gdi_RemoveTopoOverlaps(character varying, character varying,
character varying, character varying, character varying, character varying) IS 'Entfernt Faces, die zu mehr
als einer Topo-Geometrie zugeordnet sind durch Löschen der Face-Zuordnungen und schließlich dem
Zuschlagen des Faces durch Löschen der Edges zwischen der Überlappung und der Fläche, der es
zugeschlagen wird.';
```

#### 3.5.1.6 gdi_RemoveNodesBetweenEdges

Nach dem Entfernen von Überlappungen und dem Füllen von Lücken bleiben nodes übrig, die links und rechts nur von jeweils einer edge begrenzt sind, siehe Abbildung 31. Diese sind aus topologische Sicht überflüssig und können auch noch gelöscht werden. Das erfolgt grundsätzlich mit Hilfe der Funktion ST_ModEdgeHieal

```
int ST_ModEdgeHeal(varchar atopology, integer anedge, integer anotheredge);
```

Die Funktion nimmt den Namen der Topologie entgegen sowie die ID‘s der beiden Edges, die verbunden werden sollen.

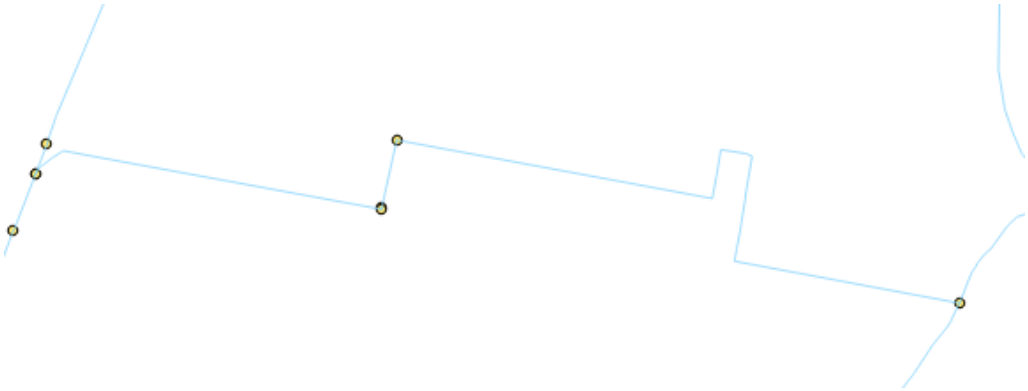


**Abbildung 31: Überflüssige nodes zwischen jeweils nur zwei edges.**

In Abschnitt 3.4.5 wird beschrieben wie die nodes, die gelöscht werden sollen gefunden werden. In der Finalen Version ist das Löschen in der Funktionen gdi_RemoveNodesBetweenEdges umgesetzt worden.

```
CREATE OR REPLACE FUNCTION gdi_RemoveNodesBetweenEdges(
  topo_name CHARACTER VARYING
)
RETURNS BOOLEAN AS
$BODY$
  DECLARE
    sql text;
```

```
  num_nodes INTEGER = 1;
  debug BOOLEAN = false;
 BEGIN
  WHILE num_nodes > 0 LOOP
   EXECUTE '
    WITH node_id_rows AS (
     INSERT INTO gemeinden_mv_topo.removed_nodes (node_id, geom)
     SELECT
      removed_nodes.node_id,
      geom
     FROM
      (
       SELECT
        gdi_ModEdgeHealException("gemeinden_mv_topo", edge_left_id, edge_right_id) node_id
       FROM
        (
         SELECT abs_next_left_edge edge_left_id, edge_id edge_right_id FROM
gemeinden_mv_topo.edge_data
         UNION
         SELECT edge_id edge_left_id, abs_next_right_edge edge_right_id FROM
gemeinden_mv_topo.edge_data
        ) edges
      ) removed_nodes JOIN
      gemeinden_mv_topo.node ON removed_nodes.node_id = node.node_id
     WHERE
      removed_nodes.node_id > 0
     RETURNING removed_nodes.node_id
    )
    SELECT count(*) FROM node_id_rows
   ' INTO num_nodes;
   RAISE NOTICE 'Anzahl gelöschter Nodes: % ', num_nodes;
  END LOOP;
  RETURN TRUE;
 END;
$BODY$
 LANGUAGE plpgsql VOLATILE COST 100;
COMMENT ON FUNCTION gdi_RemoveNodesBetweenEdges(CHARACTER VARYING) IS 'Die Funktion
entfernt alle überflüssigen Knoten, die nur von zwei Kanten begrenzt werden, die selber kein eigenes Face
bilden.'
```

Dabei werden zunächst alle Edge Paare abgefragt und dann mit der Funktion gdi_ModEdgeHealException versucht die Nodes dazwischen zu löschen. Das gelingt nur für die, die wirklich nur an zwei edges angeschlossen sind. Die Funktion liefert die ID der gelöschten nodes zurück, die dann zusammen mit der Punktgeometrie in die Tabelle r*emoved_nodes* geschrieben wird.

Diese Funktion muss vor gdi_CloseTopoGaps aufgerufen werden, weil diese Funktion nur richtig funktioniert, wenn gaps von nur zwei Kanten begrenzt sind.

#### 3.5.1.7 gdi_CloseTopoGaps

```
CREATE OR REPLACE FUNCTION gdi_CloseTopoGaps(
 topo_name CHARACTER VARYING,
 schema_name CHARACTER VARYING,
 table_name CHARACTER VARYING,
```

```
  topo_geom_column CHARACTER VARYING)
  RETURNS BOOLEAN AS
$BODY$
  DECLARE
    sql text;
    gap RECORD;
    num_edges INTEGER = 2;
    debug BOOLEAN = FALSE;
  BEGIN
    RAISE NOTICE 'Closing TopoGaps';

    sql = '
      SELECT
        gap.edge_id,
        g.' || topo_geom_column || ' AS geom_topo,
        gap.left_face,
        gap.right_face,
        num_edges,
        r.topogeo_id
      FROM
        (
          SELECT DISTINCT ON (left_face)
            edge_id,
            left_face,
            right_face,
            St_Length(geom) AS length,
            count(edge_id) OVER (PARTITION BY left_face) AS num_edges
          FROM
            ' || topo_name || '.edge_data ed LEFT JOIN
            ' || topo_name || '.relation rl ON ed.left_face = rl.element_id
          WHERE
            rl.element_id IS NULL AND
            ed.left_face > 0
          ORDER BY left_face, length DESC
        ) gap JOIN
        ' || topo_name || '.relation r ON gap.right_face = r.element_id JOIN
        ' || schema_name || '.' || table_name || ' g ON r.topogeo_id = (g.' || topo_geom_column || ').id
    ';
    IF debug THEN RAISE NOTICE 'Find gaps in topology with sql: %', sql; END IF;

    FOR gap IN EXECUTE sql LOOP
      RAISE NOTICE 'Close and log gap covert by % edges at face % by adding it to topogeom id %: %
from face % and remove edge %', gap.num_edges, gap.left_face, gap.topogeo_id, gap.geom_topo,
gap.right_face, gap.edge_id;
      sql = '
        INSERT INTO ' || topo_name || '.filled_gaps (polygon_id, face_id, num_edges, face_geom)
        SELECT
          polygon_id,
          face_id,
          num_edges,
          face_geom
        FROM
          (
            SELECT
              polygon_id,
```

```
            ' || gap.left_face  || ' AS face_id,
            ' || gap.num_edges || ' AS num_edges,
            ST_GetFaceGeometry('' || topo_name || '', ' || gap.left_face || ') AS face_geom,
            TopoGeom_addElement(' || topo_geom_column || ', ARRAY[' || gap.left_face || ', 3]::TopoElement)
          FROM
            ' || schema_name || '.' || table_name || '
          WHERE
            (' || topo_geom_column || ').id = ' || gap.topogeo_id || '
        ) AS gaps_table
    ';
    IF debug THEN RAISE NOTICE 'Execute sql to add the face: %', sql; END IF;
    EXECUTE sql;

    sql = 'SELECT ST_RemEdgeModFace('' || topo_name || '', ' || gap.edge_id || ')';
    IF debug THEN RAISE NOTICE 'Execute sql to remove edge: %', sql; END IF;
    EXECUTE sql;

   END LOOP;
   RETURN TRUE;
 END;
$BODY$
 LANGUAGE plpgsql VOLATILE COST 100;
COMMENT ON FUNCTION gdi_closetopogaps(CHARACTER VARYING, CHARACTER VARYING,
CHARACTER VARYING, CHARACTER VARYING) IS 'Entfernt faces, die keine Relation zu Polygonen
haben, also Lücken zwischen anderen darstellen und ordnet die Fläche dem benachbarten Face und
damit Polygon zu, welches die längste Kante an der Lücke hat.';
```

#### 3.5.1.8 gdi_ModEdgeHealException

```
CREATE OR REPLACE FUNCTION public.gdi_ModEdgeHealException(
    atopology character varying,
    anedge integer,
    anotheredge integer)
 RETURNS integer AS
$BODY$
DECLARE
 node_id INTEGER;
BEGIN
 SELECT ST_ModEdgeHeal($1, $2 , $3) INTO node_id;
 RETURN node_id;
EXCEPTION WHEN others THEN
 RETURN 0;
END;
$BODY$
LANGUAGE plpgsql VOLATILE COST 100;
COMMENT ON FUNCTION public.gdi_ModEdgeHealException(character varying, integer, integer) IS
'Führt zwei benachbarte Kanten zu einer zusammen, wenn von dem Knoten dazwischen keine weiter';
```

#### 3.5.1.9 gdi_CleanPolygonTopo

```
CREATE OR REPLACE FUNCTION gdi_CleanPolygonTopo(
 topo_name CHARACTER VARYING,
```

```
  schema_name CHARACTER VARYING,
  table_name character varying,
  geom_column CHARACTER VARYING,
  area_tolerance DOUBLE PRECISION,
  polygon_id INTEGER
)
RETURNS BOOLEAN AS
$BODY$
  DECLARE
    sql text;
    small_face Record;
    node RECORD;
    i INTEGER;
    node_id INTEGER;
    nodes INTEGER[];
    debug BOOLEAN = false;
  BEGIN
    -- Remove all edges without relations to faces
    sql = 'SELECT ST_RemEdgeModFace(''' || topo_name || ''', edge_id) FROM ' || topo_name ||
'.edge_data WHERE right_face = 0 and left_face = 0';
    EXECUTE sql;

    -- query small closed faces of polygon with polygon_id (same start_ and end_node)
    sql = '
      SELECT
        faces.face_id
      FROM
        (
          SELECT
            (GetTopoGeomElements(' || geom_column || '))[1] AS face_id,
            ST_GetFaceGeometry(''' || topo_name || ''', (GetTopoGeomElements(' || geom_column || '))[1]) AS
geom
          FROM
            ' || schema_name || '.' || table_name || '
          WHERE
            polygon_id = ' || polygon_id || '
        ) faces JOIN
        ' || topo_name || '.edge_data e ON faces.face_id = e.left_face OR faces.face_id = e.right_face
      WHERE
        e.start_node = e.end_node AND
        ST_Area(faces.geom) < ' || area_tolerance || '
    ';
    IF debug THEN RAISE NOTICE 'Query small faces with sql: % ', sql; END IF;
    FOR small_face IN EXECUTE sql LOOP
      -- Frage vorher schon mal alle nodes des faces ab, weil die nach dem löschen des face schlechter
zufinden sind.
      sql = '
        WITH edges AS (
        SELECT
          start_node, end_node
        FROM
          ' || topo_name || '.edge
        WHERE
          left_face = ' || small_face.face_id || ' OR right_face = ' || small_face.face_id || '
        )
```

```
      SELECT start_node AS node_id FROM edges UNION
      SELECT end_node AS node_id FROM edges;
    ';
    i = 1;
    FOR node IN EXECUTE sql LOOP
      nodes[i] = node.node_id;
      i = i + 1;
    END LOOP;

    -- Entferne face von TopoGeom des features
    RAISE NOTICE 'Entferne zu kleines face % from Polygon with polygon_id %', small_face_id,
polygon_id;
    EXECUTE 'SELECT TopoGeom_remElement(' || geom_column || ', Array[' || small_face.face_id || ',
3]::topology.TopoElement) FROM ' || table_name || ' WHERE polygon_id = ' || polygon_id;

    -- Entferne alle edges des face aus der Topology und damit auch das face
    EXECUTE 'SELECT ST_RemEdgeModFace(''' || topo_name || ''', abs((ST_GetFaceEdges(''' ||
topo_name || ''', ' || small_face.face_id || ')).edge))';

    FOREACH node_id IN ARRAY nodes LOOP
      EXECUTE 'SELECT ST_RemoveIsoNode(''' || topo_name || ''', ' || node_id || ')';
    END LOOP;
  END LOOP;

  RETURN TRUE;
 END;
$BODY$
 LANGUAGE plpgsql VOLATILE COST 100;
COMMENT ON FUNCTION gdi_cleanpolygontopo(character varying, character varying, character varying,
character varying, double precision, integer) IS 'Entfernt alle edges ohne Relation zu Polygonen (Ja das
kann es geben bei der Erzeugung von TopoGeom.) und löscht Faces des Polygon mit polygon_id, die
kleiner als die angegebene area_tolerance in Quadratmetern sind.';
```

Zum Schluss werden schließlich die topologisch korrekten Flächen wieder über die Feature id *id_column* aggregiert und in die Spalte <*geom_column*>_topo_corrected geschrieben.

Vor und nach der Polygonaufbereitung, nach dem entfernen der Nasen und Kerben sowie nach der Entfernung von Überlappungen und Lücken werden die Überlappungen berechnet und in die Tabelle intersections gespeichert, sieh auch Abschnitt 4. Die Schritte in der Spalte step heißen:

- vor Polygonaufbereitung
- nach Polygonaufbereitung
- nach NoseRemove
- nach TopoCorrection

Wenn Einträge mit „nach TopoCorrection“ fehlen, liegt das daran, dass keine Überlappungen mehr vorhanden sind, was ja auch das Ziel des ganzen Prozesses ist.

## 3.6 Version 2

Da in der Variante 1 immer noch Fehler wie „SQL/MM Spatial exception - geometry crosses edge“ oder „SQL/MM Spatial exception - curve not simple“ auftraten wurde der Zeitpunkt wann die overlaps und gaps bearbeitet werden geändert. In dieser Version werden die Funktionen gdi_RemoveTopoOverlaps und gdi_CloseTopoGaps mit anschließendem gdi_RemoveNodesBetweenEdges jeweils nach dem Hinzufügen einer einzelnen TopoGeom aufgerufen. Werden also durch eine neue Geometrie Überlappungen oder

Lücken in der Topologie erzeugt, werden diese gleich wieder entfernt bevor die nächste Geometrie hinzugefügt wird.
Die Funktion gdi_CreateTopo kann nun zusätzlich mit dem Parameter debug aufgerufen werden. mit true liefert die Funktion die Laufzeiten der jeweilig aufgerufenen Funktionen in ms. Dadurch kann man leichter herausfinden welche Geometrien komplex sind und länger dauern. Es ist aber noch kein Muster zu erkennen gewesen, welches darauf hindeutet, dass die Funktionen länger laufen je größer die Topologie wird.
Des Weiteren wurde FilterRings eingeführt um kleine innere Ringe in der Vorverarbeitung der Polygone zu löschen.

# 4 Logging

In dem Schema in dem die Topologie der Tabellengeometrie hinterlegt ist (<table_name>_topo) werden folgende Tabellen mit Informationen über die Korrekturfunktion gespeichert.

| Log-Tabelle | Inhalt |
|---|---|
| statistic | Einzelne statistische Angaben zur Berechnung, jeweils mit Bezeichnung, Wert und Einheit oder Beschreibung |
| intersections | Überlappungen von Flächen während verschiedener Berechnungsphasen, gekennzeichnet mit den Werten „vor Polygonaufbereitung“, „nach Polygonaufbereitung“, „nach NoseRemove“ und „nach TopoCorrection“ in der Spalte step. |
| removed_spikes | Stützpunkte, die durch gdi_RemoveNose gelöscht wurden |
| removed_overlaps | Entfernte Überlappungen in der Topologie (zwei oder mehr Polygone waren zu dem selben face zugeordnet) |
| filled_gaps | Aufgefüllte Lücken in der Topologie (faces, die keine Zuordnung zu Polygonen hatten, mit polygon_id von der Fläche, der die Lücke zugeschlagen wird, edge_id der Kante die gelöscht wurde, num_edges Anzahl der Kanten, die die Lücke umschlossen haben und der face_geom, die gelöscht wurde) |
| removed_nodes | Gelöschte nodes, die nur Anschluss zu 2 edges hatten, dies selbst kein eigenes face bilden. |

Die Geometriespalten in den Tabellen können zur Visualisierung der Elemente verwendet werden.

Werte in der Statistiktabelle werden folgende hinterlegt:

| Nr | Schlüssel | Erläuterung |
|---|---|---|
| 1 | Name der Topologie | |
| 2 | Ursprüngliche Tabelle | |
| 3 | Geometriespalte | |
| 4 | Distanz Toleranz | |
| 5 | Angle Toleranz | |
| 6 | Topology Toleranz | |
| 7 | Gesamtfläche vorher | |
| 8 | Anzahl Flächen vorher | |
| 9 | Anzahl Stützpunkte vorher | |
| 10 | Fläche der Überlappungen nach Polygonaufbereitung | |
| 11 | Gesamtfläche nach Polygonaufbereitung | |
| 12 | Anzahl der Polygone | |
| 13 | Flächendifferenz nach - vor Polygonaufbereitung | 11 - 7 |

| | | |
|---|---|---|
| 14 | Fläche der Überlappungen nach NoseRemove | |
| 15 | Gesamtfläche nach NoseRemove | |
| 16 | Flächendifferenz nach - vor NoseRemove | 14 - 11 |
| 17 | Fläche der gelöschten Topologieüberlappungen | |
| 18 | Fläche der gefüllten Topologielücken | |
| 19 | Fläche der Überlappungen korrigierter Geometrien | |
| 20 | Gesamtfläche korrigierter Geometrien | |
| 21 | Anzahl Flächen nach Korrektur | |
| 22 | Anzahl Stützpunkte hinterher | |
| 23 | Flächendifferenz nach - vor Topo-Korrektur | 20 - 15 |
| 24 | Flächendifferenz gesamt | 20 - 7 |
| 25 | Flächendifferenz Lücken - Überlappungen | 18 - 17 |
| 26 | Absoluter Betrag der Differenz der Flächendifferenz gesamt / Lücken-Überlappungen | abs(25 - 24) |

In der Tabelle topo_geom ist zusätzlich eine Spalte err_msg eingefügt. In diesem Attribut sind Fehlermeldungen hinterlegt, wenn es beim Anlegen der Topologie für das Polygon einen Fehler gab. Eine Auswertung der Fehler erfolgt also mit dem Statement:

```
SELECT gid, polygon_id, err_msg FROM <table_name>_topo.topo_geom WHERE err_msg != '';
```

Die Funktionen liefern an einigen Stellen mit dem Befehl RAISE NOTICE Ausgaben. Damit wird kann ein wenig der Fortschritt der Ausführung der Funktionen nachvollzogen werden.

```
HINWEIS:  Create TopGeom for object with polygon_id = 669
HINWEIS:  Create TopGeom for object with polygon_id = 670
HINWEIS:  Create TopGeom for object with polygon_id = 671
HINWEIS:  Create TopGeom for object with polygon_id = 672
HINWEIS:  Create TopGeom for object with polygon_id = 673
HINWEIS:  Create TopGeom for object with polygon_id = 674
HINWEIS:  Create TopGeom for object with polygon_id = 675
HINWEIS:  Create TopGeom for object with polygon_id = 676
HINWEIS:  Create TopGeom for object with polygon_id = 677
HINWEIS:  Create TopGeom for object with polygon_id = 678
HINWEIS:  Create TopGeom for object with polygon_id = 679
HINWEIS:  Create TopGeom for object with polygon_id = 680
HINWEIS:  Create TopGeom for object with polygon_id = 681
```

In den Funktionen gdi_PreparePolygonTopo, gdi_NoseRemoveCore, gdi_RemoveTopoOverlaps, gdi_CloseTopoGaps, gdi_CleanPolygonTopo gibt es eine Variable debug. Ist diese auf true gesetzt werden mehr Ausgaben erzeugt, die auch zum debugging genutzt werden können. Dabei werden zum Teil auch verwendete SQL-Statements ausgegeben.

# 5 Test gemeinden_mv

Als Testdatensatz diente der Datensatz Gemeinden und gemeindefreie Kommunen Mecklenburg-Vorpommern, herunterzuladen unter https://www.opendata-hro.de/dataset/gemeinden_mecklenburg-vorpommern.
Die Shape-Datei wurde in eine Postgres-Datenbank in das Schema public importiert und die Tabelle gemeinden_mv bezeichnet. Der Datensatz deckt ganz Mecklenburg-Vorpommern ab, siehe Abbildung 32. Es handelt sich um 750 Flächen mit 920779 Stützpunkten und einer Gesamtfläche von 23291.350696 km².

Er enthält Überlappungen bei Polygonen, siehe Abbildung 33, 34, 35 und 36 und Lücken, siehe Abbildung 37.
Die Laufzeit für die gesamte Berechnung beträgt ca. 250 Sekunden.

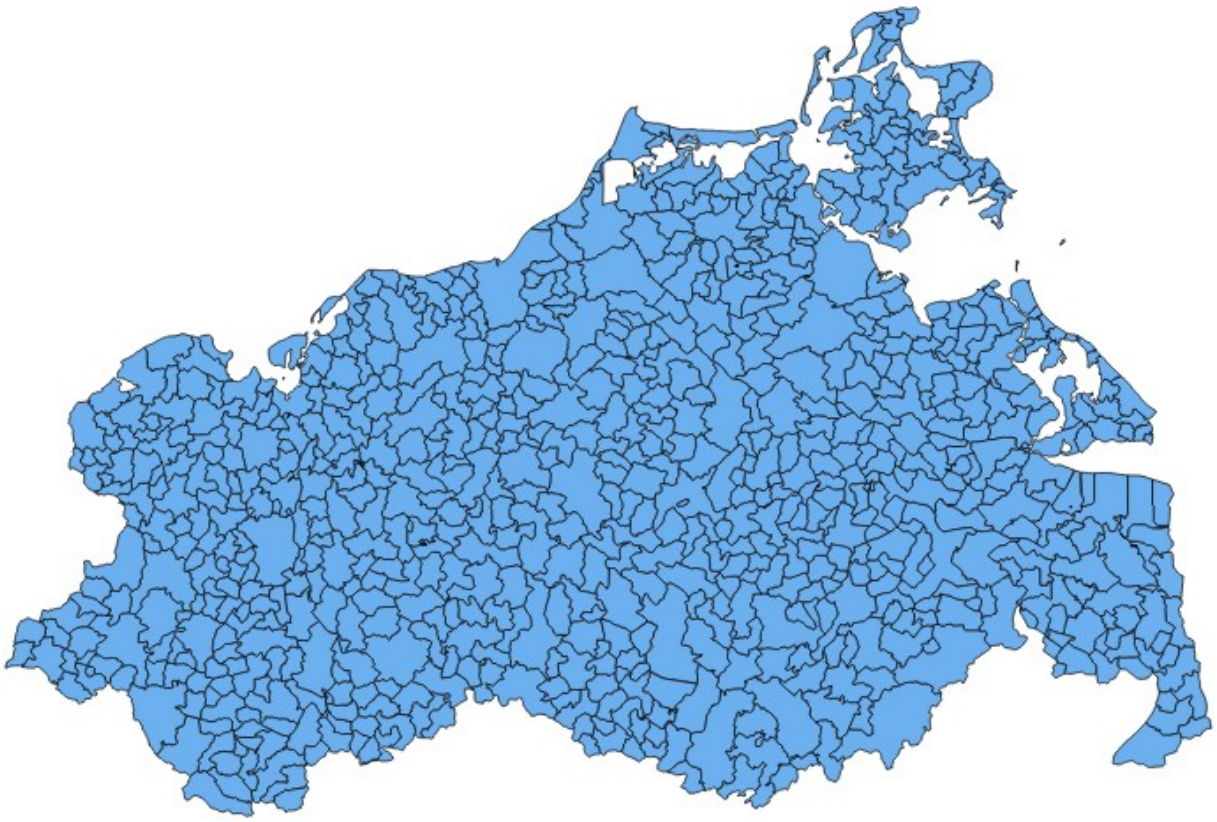


**Abbildung 32: Abdeckung des Datensatzes**

Auch wenn auf den folgenden Abbildungen nicht viel zu sehen ist, zeigen diese doch, dass es zahlreiche Überlappungen und Gaps gibt und zwar mit

```
SELECT step, count(*) FROM gemeinden_mv_topo.intersections GROUP BY step;
```

,

```
SELECT count(*) FROM gemeinden_mv_topo.removed_overlaps;
```

und

```
SELECT count(*) FROM gemeinden_mv_topo.filled_gaps;
```

ermittelt:

| step<br>character varying | count<br>bigint |
|---|---|
| vor Polygonaufbereitung | 42 |
| nach Polygonaufbereitung | 569 |
| nach NoseRemove | 569 |

| count<br>bigint |
|---|
| 386 |

| count<br>bigint |
|---|
| 343 |

**Abbildung 33: Überschneidungen vor der Polygonaufbereitung**

**Abbildung 34: Überlappungen nach Polygonaufbereitung**

**Abbildung 35: Überlappungen nach NoseRemove**

**Abbildung 36: Gelöschte Topologieüberlappungen**

**Abbildung 37: Gefüllte Lücken**

**Abbildung 38: Edges und Nodes der Topologie der Flächen in MV**

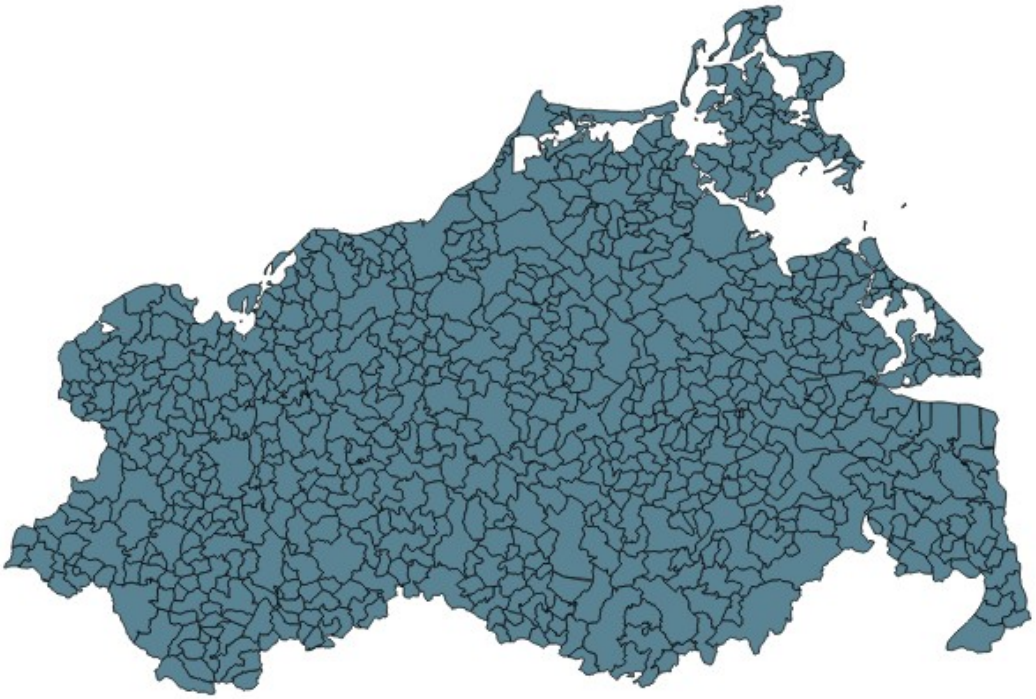


**Abbildung 39: Korrigierte Geometrie**

# 6 Zusammenfassung

Die Bereinigung von benachbarten Flächen zur Löschung von Überlappungen und dem Schließen von Lücken stellt eine große Herausforderung dar. Es wurden verschiedene Ansätze verfolgt, von denen sich die Berechnung einer Topologie mit Bereinigungsfunktion und anschließender Extraktion der Polygone als umsetzbar herausstellte. Die Variante mit der Berechnung der äußeren Hülle, Pufferung aller Flächen und sukzessive Differenzbildung von Nachbarflächen ist zwar logisch richtig, aber es traten immer wieder Fehler bei der GeometrieUnion auf, so dass diese Variante verworfen wurde. Die Performance ist auch viel schlechter als unter Verwendung der Topologie. Die Variante mit Triangulation hört sich auch vielversprechend an, wurde aber aus Zeitgründen verworfen. PostGIS bietet eine Funktion ST_DelaunayTriangles https://postgis.net/docs/ST_DelaunayTriangles.html zur Berechnung von flächendeckenden Dreiecken zwischen Punkten. Es wären jedoch im Anschluss zahlreiche Point in Polygontests und Unions nötig für die Rückberechnung auf die Feature-Polygone.
Bei der Variante der Topologie fällt nebenbei die Topologie ab, die für weitere Berechnungen oder Abfragen verwendet werden kann. Insbesondere wer sind die nächsten Nachbarn.
Die angegebenen Toleranzen für

- die Simplifizierung der Ausgangsgeometrie (Abstände benachbarter Punkte und minimaler Winkel),
- die Topologiebildung und
- die Genauigkeit beim Schließen der Lücken und Löschen von Überlappungen

entscheiden am Ende darüber ob die Berechnung fehlerfrei durchläuft, die Form der korrigierten Geometrie und über die Geschlossenheit des Polygonverbandes.
Die Methode bei der die Topologiebildung genutzt wird zeichnet sich am Ende am erfolgreichsten und schnellsten ab.

# Abbildungsverzeichnis