camp
tocamp

INNOVATIVE SOLUTIONS
BY OPEN SOURCE EXPERTS

# Ein Ansatz zur Optimierung von raumbezogenen Datenbankabfragen mit PostGIS

Dominik Frey

# Über uns

Dominik Frey

- Full stack GIS development
- https://github.com/domfre

# Über Camptocamp

## Ihr Partner für Erfolg

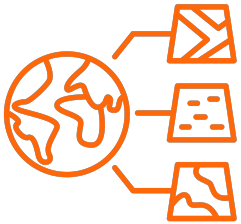
- Gegründet 2001
- 200+ Mitarbeitende
- 3 Länder:
  - Schweiz, Frankreich, Deutschland
- Ein wichtiger europäischer Akteur in Open Source Technologien
- Unabhängige Tochter der Swisscom Gruppe

# Mein Ziel für Heute

- Einen Erfahrungsbericht aus einer Performance-Analyse raumbezogener Datenbankabfragen mit PostGIS teilen
- Dazu möchte ich:
  - Den Kontext dieser Analyse erläutern (im Rahmen der Entwicklung einer Web-GIS-Anwendung)
  - Eine kurze Einführung in PostGIS geben
  - Meine Erfahrungen und Erkenntnisse präsentieren

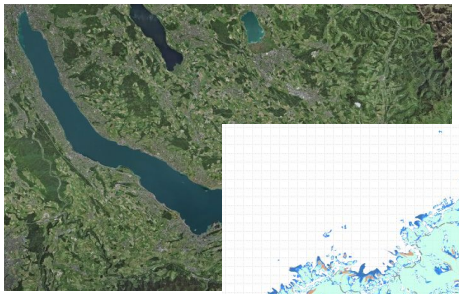
# Agenda

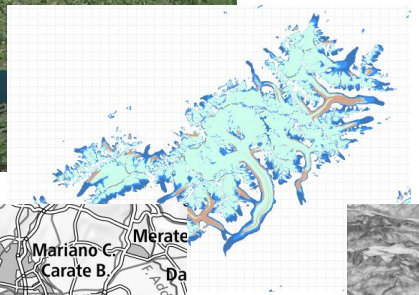
1. Was ist eine GIS-Applikation?
2. Was ist PostGIS und warum benutzen wir es?
3. Ein Ansatz zur Optimierung raumbezogener Datenbankabfragen
4. Erkenntnisse aus unserer Analyse



# Was ist eine GIS-Applikation?

# Was ist eine GIS-Applikation?



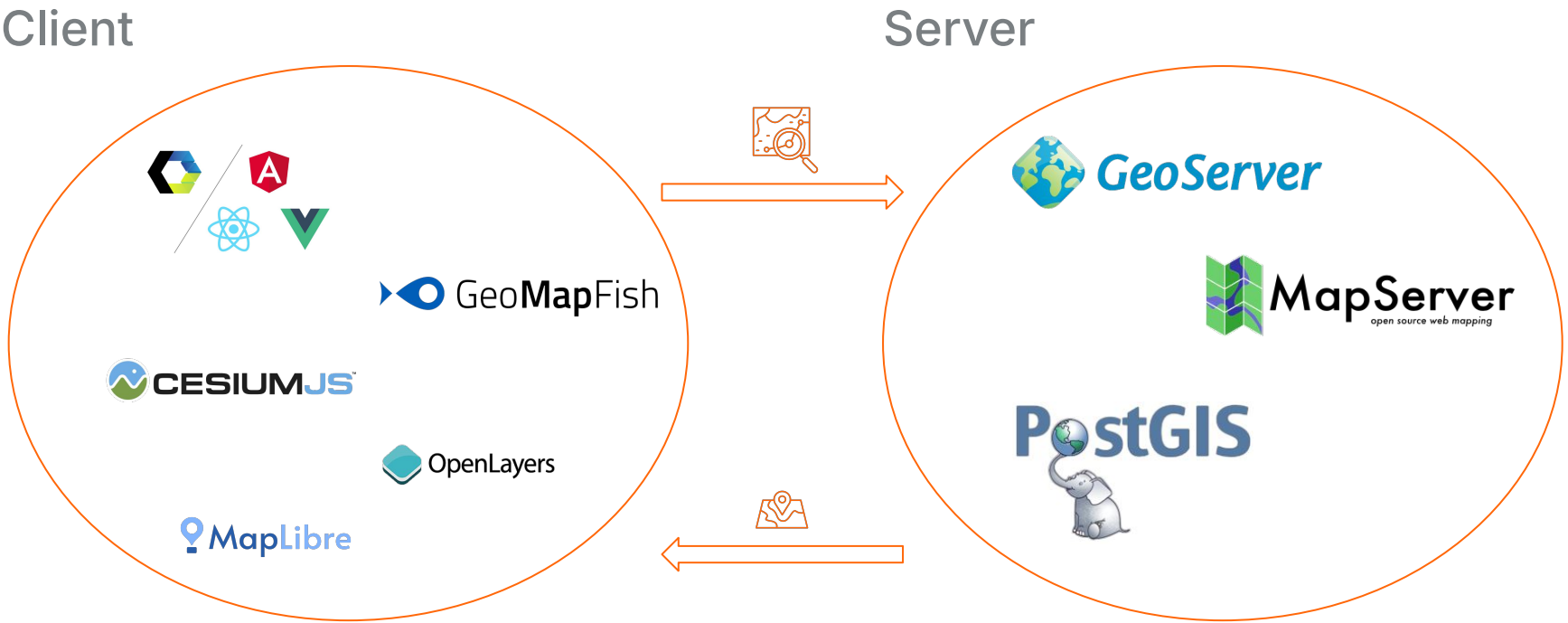



Eine Anwendung zur Darstellung von Geodaten:

➔ Jegliche Art von Daten mit räumlichen Information

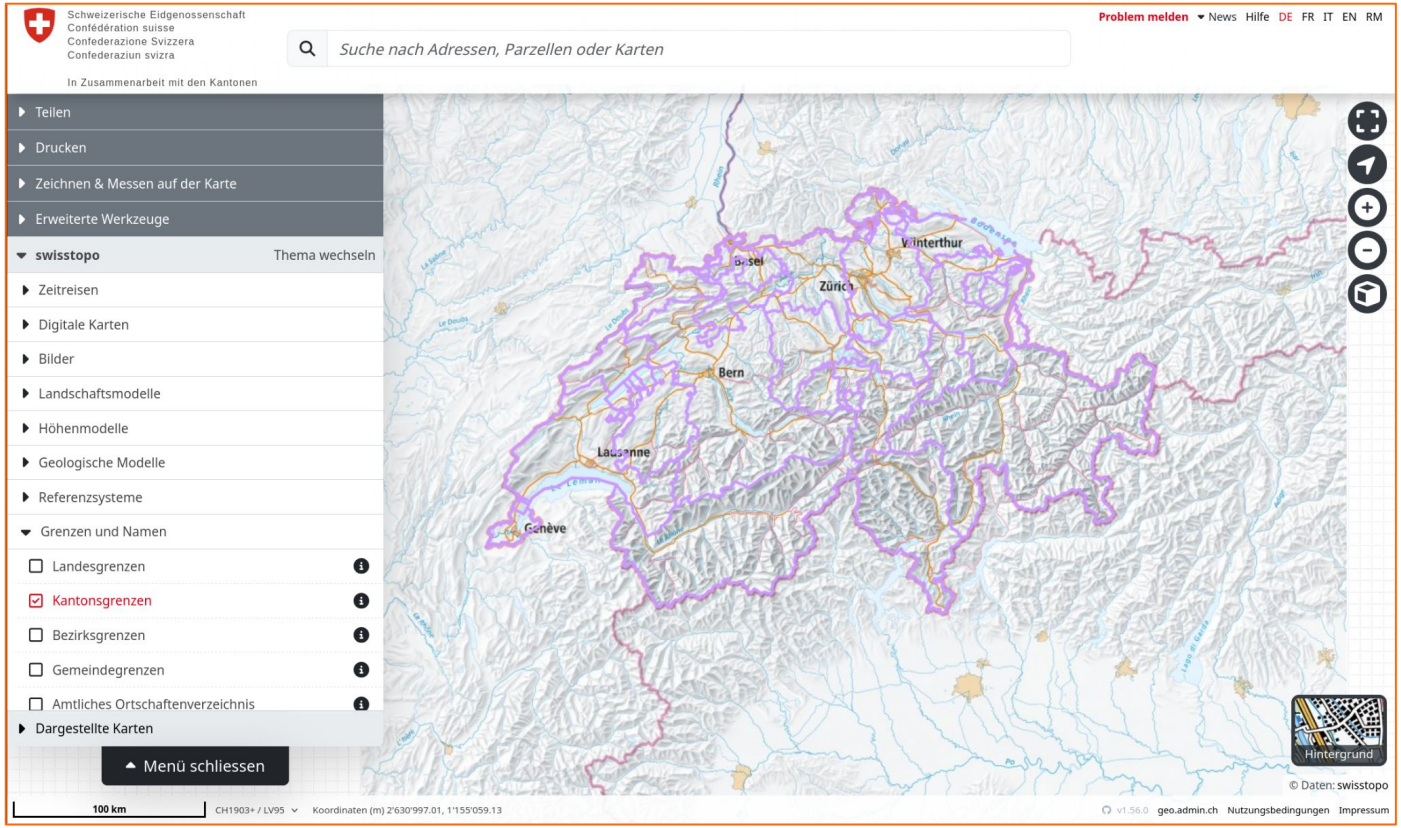
# Ein Beispiel einer GIS-Applikation

## Eine Web-GIS-Applikation



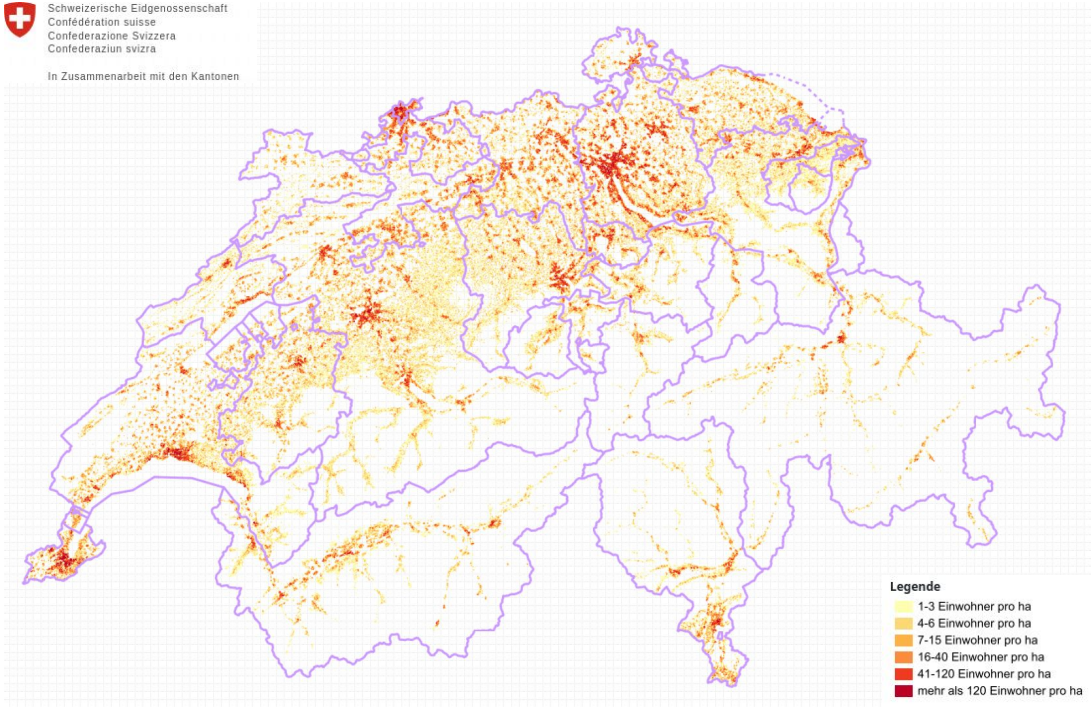
# Beispiel einer Web-GIS-Applikation

## Die Swisstopo Karten Applikation



# Unser Anwendungsfall

## Aggregieren von Daten mit einer räumlichen Beziehung




Kontext:

- ➔ Ein Projekt für das BABS zur Unterstützung der Katastrophenplanung

Ziel:

- ➔ Visualisierung der Nr. aktuell in einem Gebiet anwesenden Personen

Anforderung:

- ➔ Zeitnahe Bereitstellung der angeforderten Daten

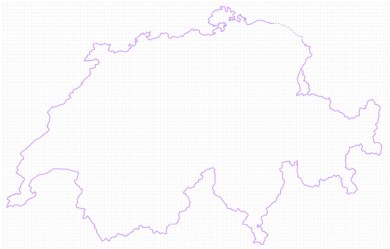
Herausforderungen:

- ➔ Umfangreiche Abfragen räumlicher Daten sind rechenintensiv

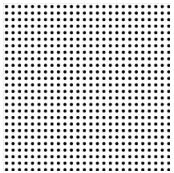
# Unser Anwendungsfall

## Aggregieren von Daten mit einer räumlichen Beziehung

Gewünscht:
- ➔ Web-GIS-Anwendung



&



Gegeben:
- ➔ Punkt-Gitter mit Nr. Personen/Punkt
- ➔ Punkt = Mittelpunkt eines 100m x 100m Quadrates

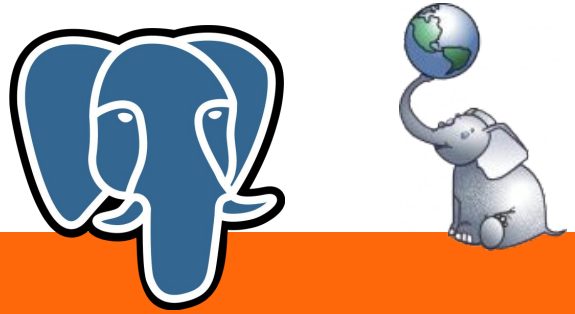
Geeignetste Technologie zur Datenspeicherung- und abfrage:
- ➔ PostGIS

Kontext:
- ➔ Ein Projekt für das BABS zur Unterstützung der Katastrophenplanung

Ziel:
- ➔ Visualisierung der Nr. aktuell in einem Gebiet anwesenden Personen

Anforderung:
- ➔ Zeitnahe Bereitstellung der angeforderten Daten

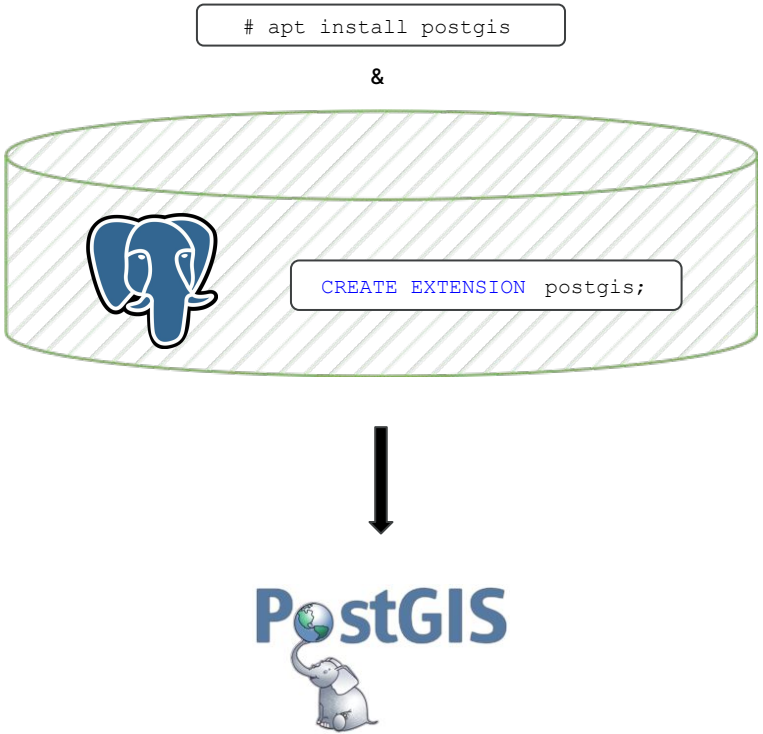
Herausforderungen:
- ➔ Umfangreiche Abfragen räumlicher Daten sind rechenintensiv



# Was ist PostGIS und warum benutzen wir es?

# Was ist PostGIS und warum benutzen wir es?

## Eine Datenbank für raumbezogene Daten



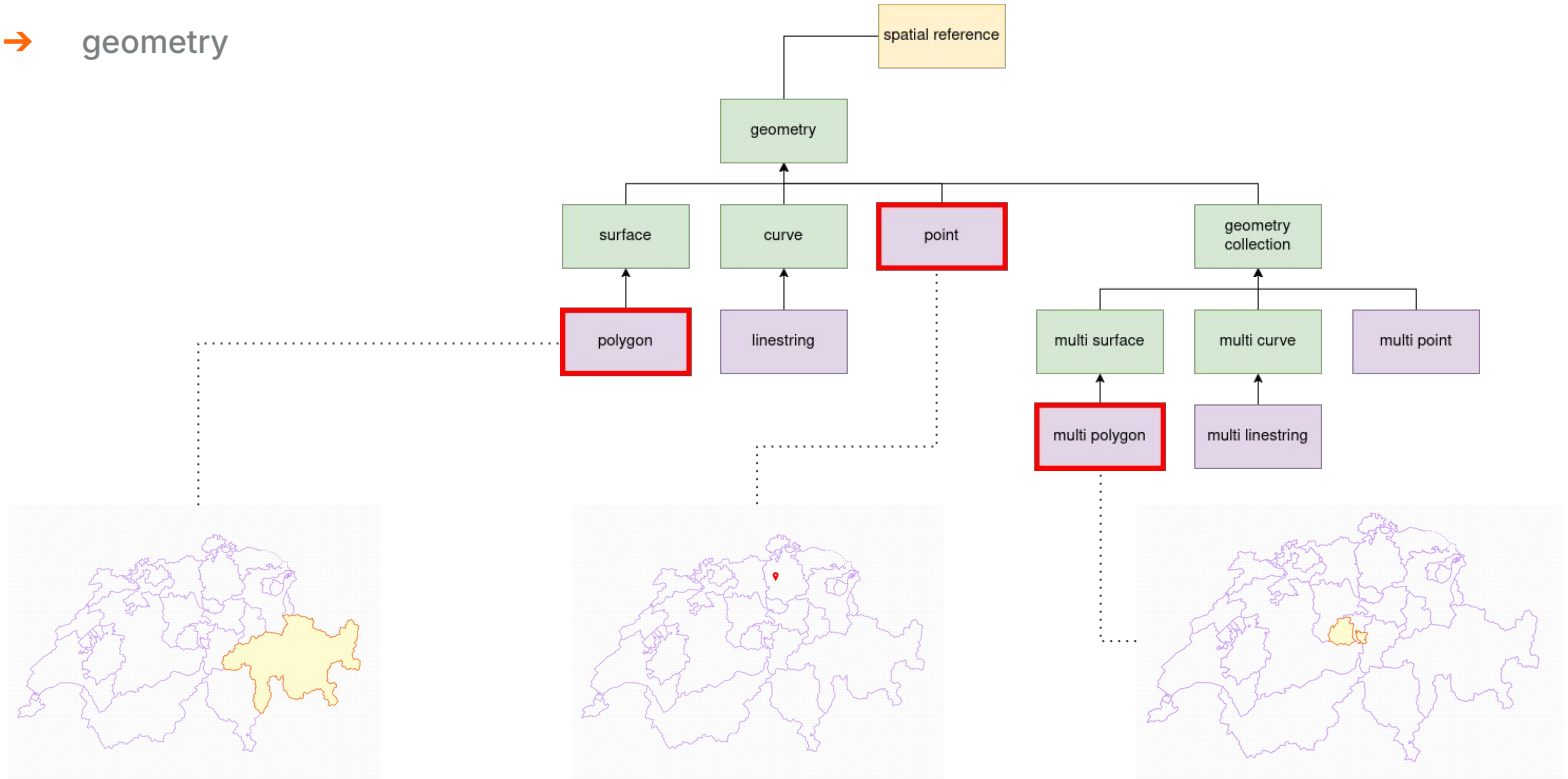

PostGIS ist eine Datenbank für raumbezogene Daten:

- ➔ Raumbezogene Datentypen:
  - Geometry (Point, Line, Polygon)
  - Raster Daten
- ➔ Raumbezogene Indexierung
  - Optimiert für raumbezogene Daten
- ➔ Raumbezogene Funktionen:
  - ST_Within
  - ST_Union
  - ...

# Raumbezogene Datentypen

## Vektordaten

- geometry

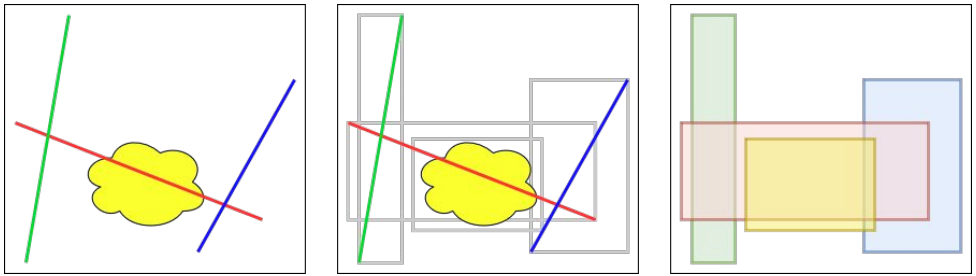


# Raumbezogene Datentypen

## Rasterdaten

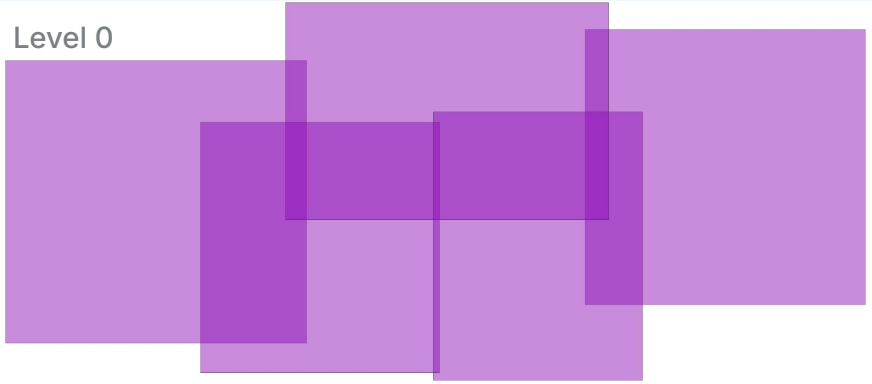
# Raumbezogene Indexierung

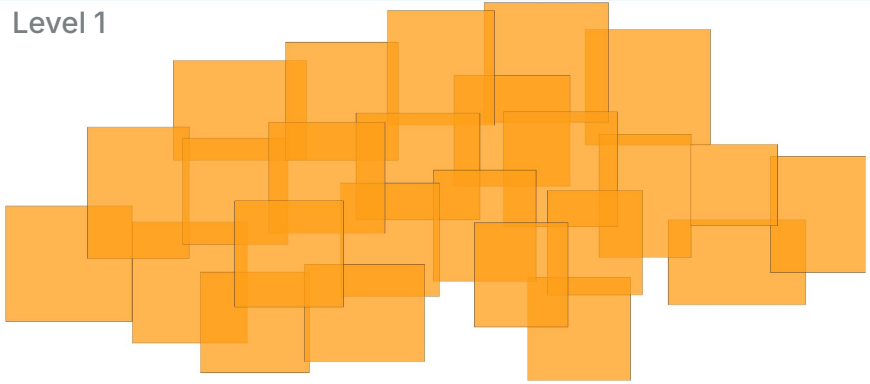
## Indexierung optimiert für räumliche Daten

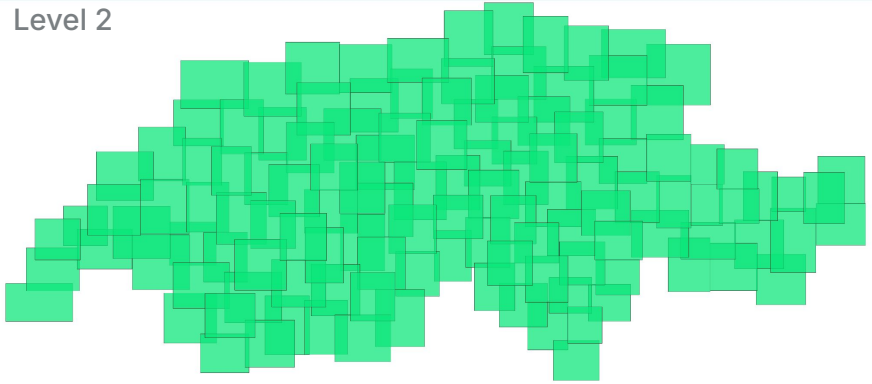


# Raumbezogene Indexierung

## Indexierung optimiert für räumliche Daten

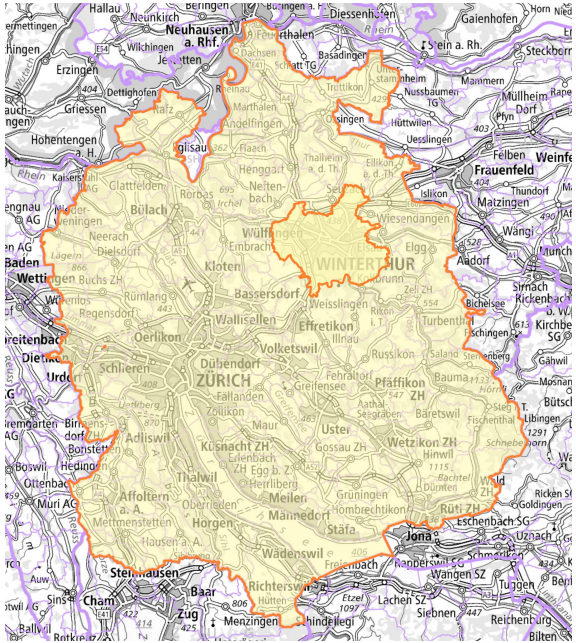



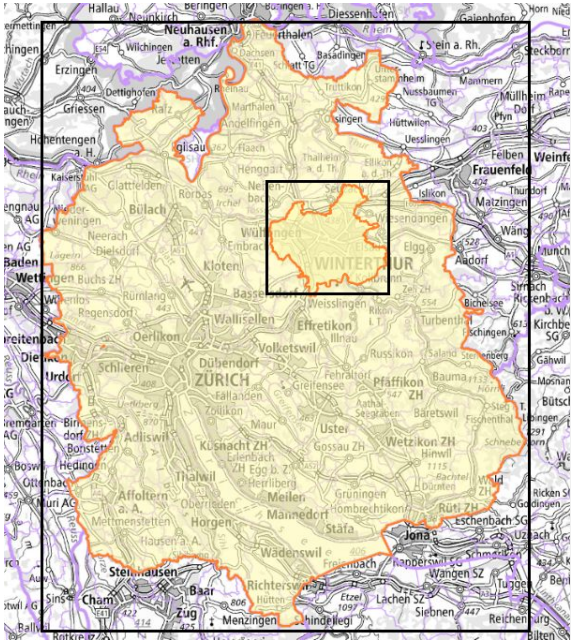
# Raumbezogene Funktionen

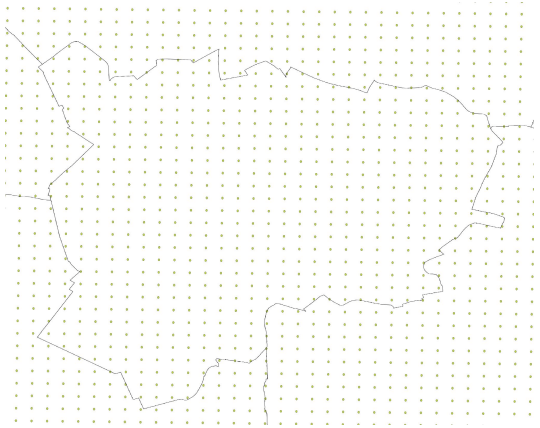
## ST_Within

ST_Within(geometry A,geometry B)

$$ST\_Within(A, B) \Leftrightarrow (A \cap B = A) \wedge (Int(A) \cap Int(B) \neq \varnothing)$$



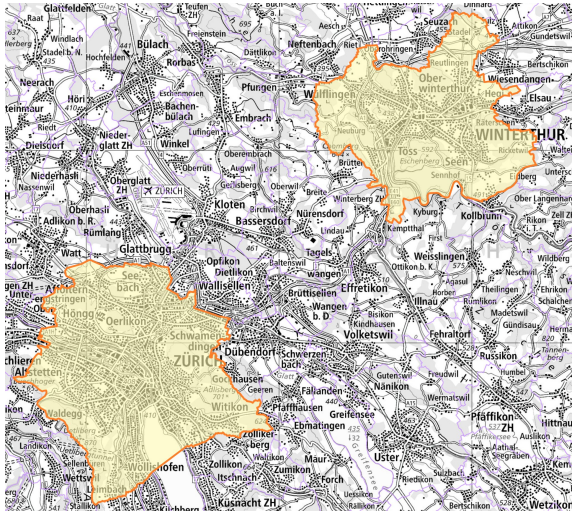
# Raumbezogene Funktionen

## ST_Union

```
ST_Union(geometry A,geometry B)
```

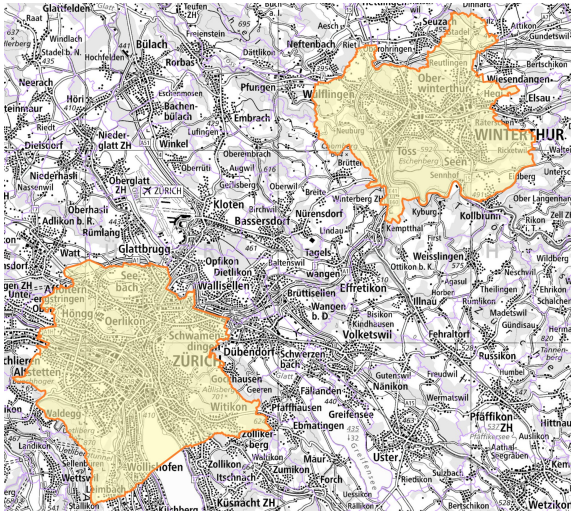
```
ST_Union(geometry[] geometry_array)
```

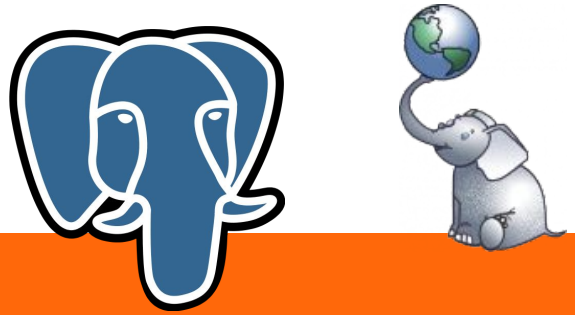
2 separate Geometrien vom Typ `Polygon`



ST_Union

1 einzelne Geometrie vom Typ `Multi Polygon`

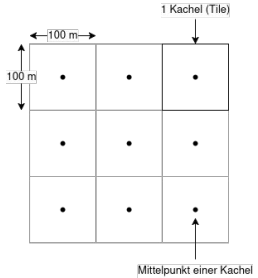




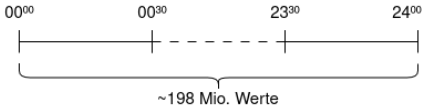
# Ein Ansatz zur Optimierung raumbezogener Datenbankabfragen

# Der Datensatz für die Performance-Tests

- Das Punkt-Gitter:
  - Ganze Fläche der Schweiz abgedeckt
  - Kachel-Gitter: 1 Kachel = 100m x 100m Quadrat
  - Jede Kachel durch ihren Mittelpunkt repräsentiert
  - ⇒Punkt-Gitter mit ~ 4.13 Mio. Punkt-Geometrien

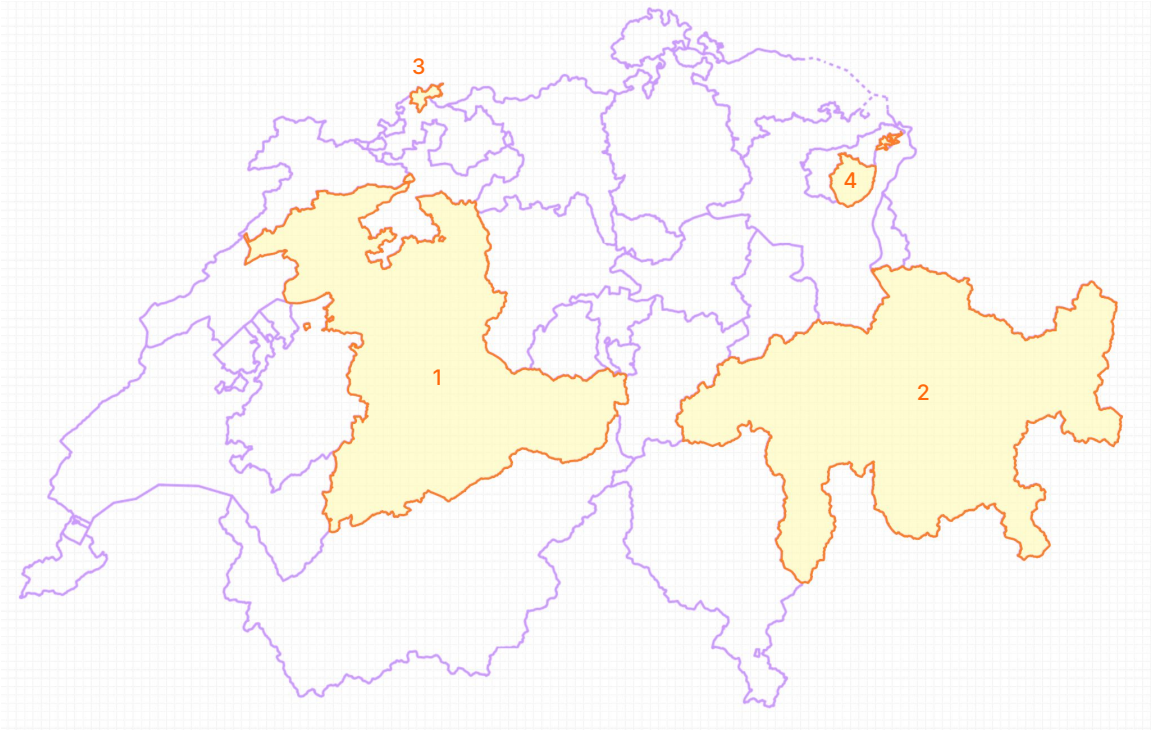



- Die numerischen Werte (Nr. Personen):
  - Synthetische Daten verwendet
  - Daten für 24h, 30min-Intervalle ⇒ 48 Datensätze
  - 1 Datensatz: 1 Wert pro Punkt-Geometrie
    ⇒ ~ 4.13 Mio Werte pro Datensatz
  - ⇒ 48 Datensätze x 4.13 Mio. Werte ≃ 198 Mio. Werte



# Verwendete Polygone

1. Bern
2. Graubünden
3. Basel-Stadt
4. Appenzell-Innerrhoden



# Getestete Datenbank-Strukturen

## ➔ Einzelne Tabelle:

| Einzelne Tabelle | | |
|---|---|---|
| PK | point_id | BIGINT NOT NULL |
| PK | timestamp | TIMESTAMP WITH TIME ZONE NOT NULL |
| | geom | GEOMETRY(POINT, 4326) NOT NULL |
| | count | INTEGER NOT NULL |

- Indizes
  - `timestamp`
  - GIST Index auf `geom`

## ➔ Zwei Tabellen:

| Punkt-Geometrien Tabelle | | |
|---|---|---|
| PK | point_id | BIGINT NOT NULL |
| | geom | GEOMETRY(POINT, 4326) NOT NULL |

| Werte Tabelle | | |
|---|---|---|
| PK, FK | point_id | BIGINT NOT NULL |
| PK | timestamp | TIMESTAMP WITH TIME ZONE NOT NULL |
| | count | INTEGER NOT NULL |

- Zwei Varianten:
  - Werte Tabelle nicht partitioniert
  - Werte Tabelle partitioniert (1 Partition pro Zeitstempel)
- Indizes
  - `timestamp` (bei nicht partitionierter Version)
  - GIST Index auf `geom`

! Indexierung auf Geometrien ist sehr rechenintensiv
⇒ importieren neuer Daten in eine einzelne Tabelle keine Option

# Getestete benutzerdefinierte Funktionen

1. `ST_Within` auf **vereinigte** Input-Geometrien (→ `ST_Union`)

```
CREATE OR REPLACE FUNCTION  query_within_unionized(timestamps  TIMESTAMP WITH TIME ZONE[] , target_geom
GEOMETRY[] )
        RETURNS TABLE  (point_id  BIGINT, point_geom  GEOMETRY) AS $$
BEGIN
        RETURN QUERY
        SELECT t_point_grid.id, geom
        FROM t_count JOIN t_point_grid ON point_id=t_point_grid.id
        WHERE timestamp = ANY (timestamps)  AND ST_Within(geom, ST_Union(target_geom));
END;
$$ LANGUAGE plpgsql;
```

2. `ST_Within` auf **einzelne** Input-Geometrien (→ `unnest`)

```
CREATE OR REPLACE FUNCTION  query_within_join_each(timestamps  TIMESTAMP WITH TIME ZONE[] , target_geom
GEOMETRY[] )
        RETURNS TABLE  (point_id  BIGINT, point_geom  GEOMETRY) AS $$
BEGIN
        RETURN QUERY
        WITH each_within (tile_id, tile_geom)  AS (
                SELECT t_point_grid.*
                FROM unnest(target_geom)  AS g(geom)
                        JOIN t_point_grid
                                ON ST_Within(t_point_grid.geom, g.geom)
        )
        SELECT DISTINCT tile_id, tile_geom
        FROM t_count JOIN each_within ON point_id=each_within.tile_id
        WHERE timestamp = ANY (timestamps);
END;
$$ LANGUAGE plpgsql;
```

# Performance Beobachtungen

## Kleine Polygone

| Partitionierung | Query | Input-Polygone | Execution Time |
|---|---|---|---|
| ❌ | `query_within_unionized` | Basel-Stadt<br>Appenzell-Innerrhoden | 1 s 481 ms |
| ✅ | `query_within_unionized` | Basel-Stadt<br>Appenzell-Innerrhoden | 996 ms |
| ❌ | `query_within_join_each` | Basel-Stadt<br>Appenzell-Innerrhoden | 747 ms |
| ✅ | `query_within_join_each` | Basel-Stadt<br>Appenzell-Innerrhoden | 651 ms |

# Performance Beobachtungen

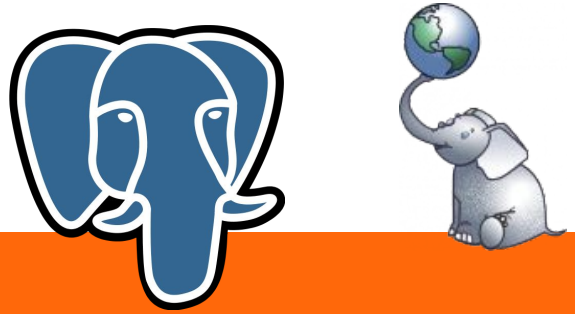
## Grosse Polygone

| Partitionierung | Query | Input-Polygone | Execution Time |
|---|---|---|---|
| ❌ | `query_within_unionized` | Bern<br>Graubünden | 21 s 626 ms |
| ✅ | `query_within_unionized` | Bern<br>Graubünden | 22 s 156 ms |
| ❌ | `query_within_join_each` | Bern<br>Graubünden | 23 s 368 ms |
| ✅ | `query_within_join_each` | Bern<br>Graubünden | 20 s 121 ms |

# Performance Beobachtungen

## Größere Anzahl Input-Polygone

| Partitionierung | Query | Input-Polygone | Execution Time |
|---|---|---|---|
| ✅ | `query_within_unionized` | Basel-Stadt<br>Appenzell-Innerrhoden<br>Bern<br>Graubünden | 39 s 562 ms |
| ✅ | `query_within_join_each` | Basel-Stadt<br>Appenzell-Innerrhoden<br>Bern<br>Graubünden | 21 s 644 ms |



# Erkenntnisse aus unserer Analyse

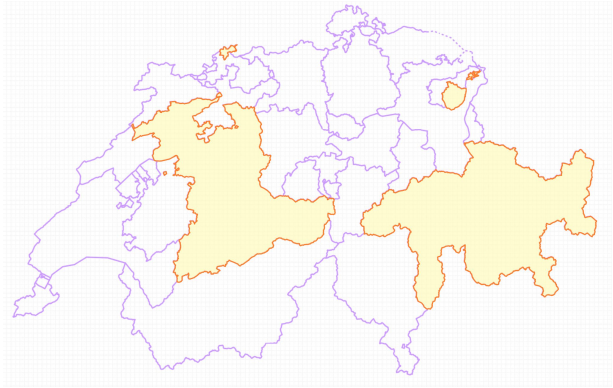
# Performance Analyse

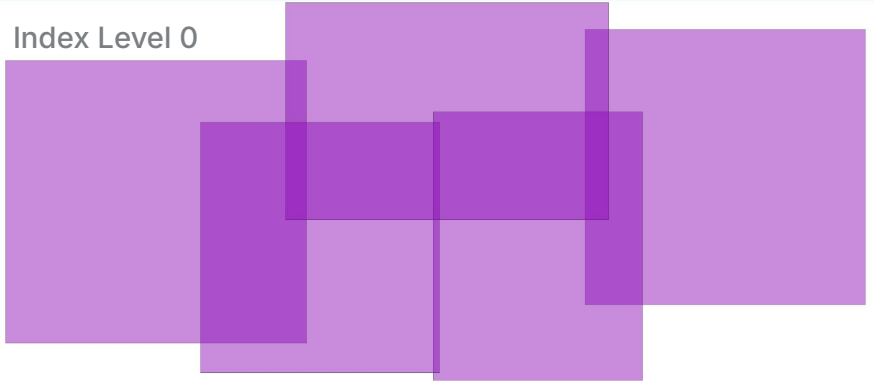
## Erkenntnisse

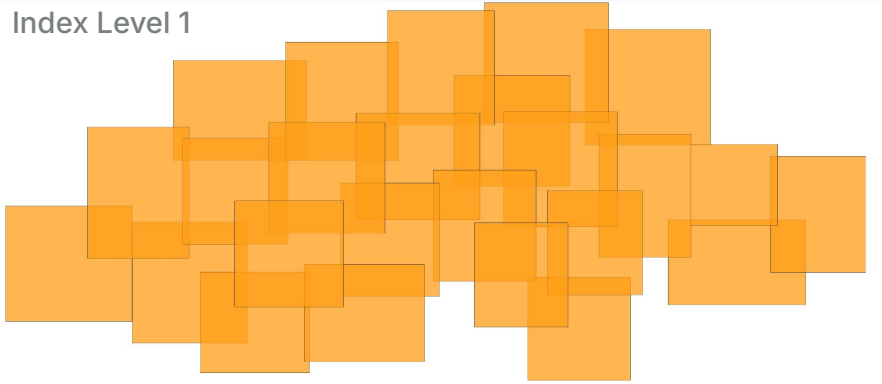
- ➔ Rechnerischer Aufwand für `ST_Union` steigt überproportional mit Anzahl Input-Polygonen
  - `ST_Union` generiert eine einzige (potentiell große und komplexe) Geometrie
  - Dies erschwert/verunmöglicht den effizienten Gebrauch von räumlichen Indizes
  - Des Weiteren braucht die Operation `ST_Union` selbst rechnerische Ressourcen
- ➔ `query_within_join_each` wertet die Input-Polygone einzeln aus, was den effizienten Gebrauch von räumlichen Indizes ermöglicht

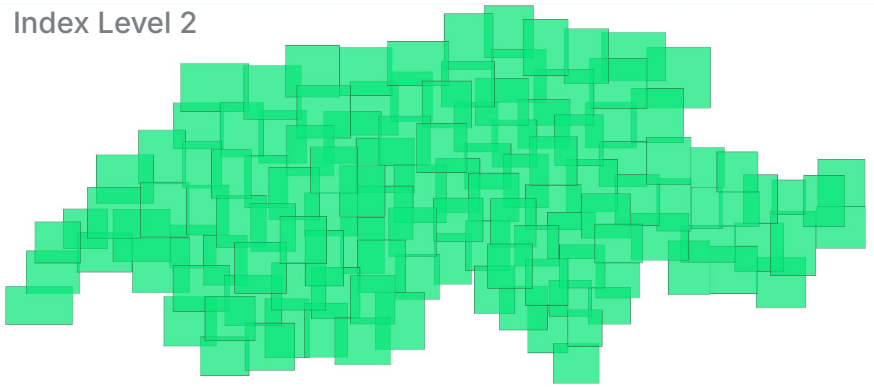
# Performance Analyse

## Erkenntnisse





camptocamp
INNOVATIVE SOLUTIONS
BY OPEN SOURCE EXPERTS

# Vielen Dank für Ihre Aufmerksamkeit.

https://github.com/camptocamp

https://www.camptocamp.com

**Dominik Frey**

dominik.frey@camptocamp.com

camptocamp

INNOVATIVE SOLUTIONS
BY OPEN SOURCE EXPERTS