

PostGIS-Tuning

Performance-Tricks für Geodatenprofis

Warum ist Performance wichtig?

- Datenmenge wird größer → Abfragen werden langsamer?
- Abfragen werden komplexer → Abfragen werden langsamer?
- Ziel: Praxisnahe Tipps zur Optimierung
- Inhalt: Indizes, Datenorganisation, Abfrageoptimierung

Teil 1: Optimierung der Daten

Indizes und räumliche Sortierung

Indizes

- Selbstaktualisierende Datenstruktur
- Beschleunigung von Suchen durch einen Suchbaum (Search Tree)
- Beschleunigung von Table Joins
- Räumliche Indizes: Beschleunigen räuml. Funktionen

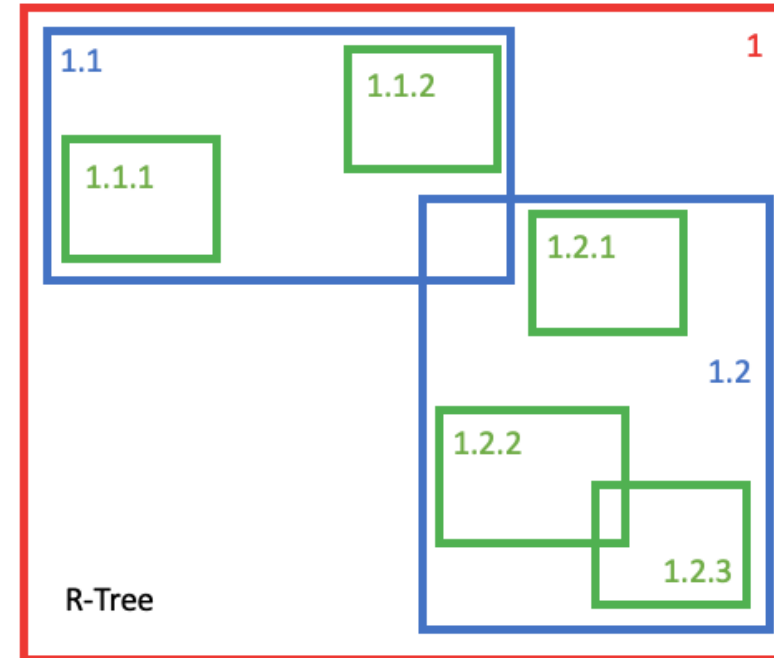
GiST-Index

(Generalized Search Tree)

- Allrounder: Ideal für die meisten Anwendungsfälle
- Indiziert die BBOX jeder Geometrie
- Benutzt R-Tree als hierarchische Struktur für schnelle Suchen

Beispiel:

```
CREATE INDEX idx_geom ON buildings USING gist (geom);
```



Grafik von: <https://www.crunchydata.com/blog/the-many-spatial-indexes-of-postgis>

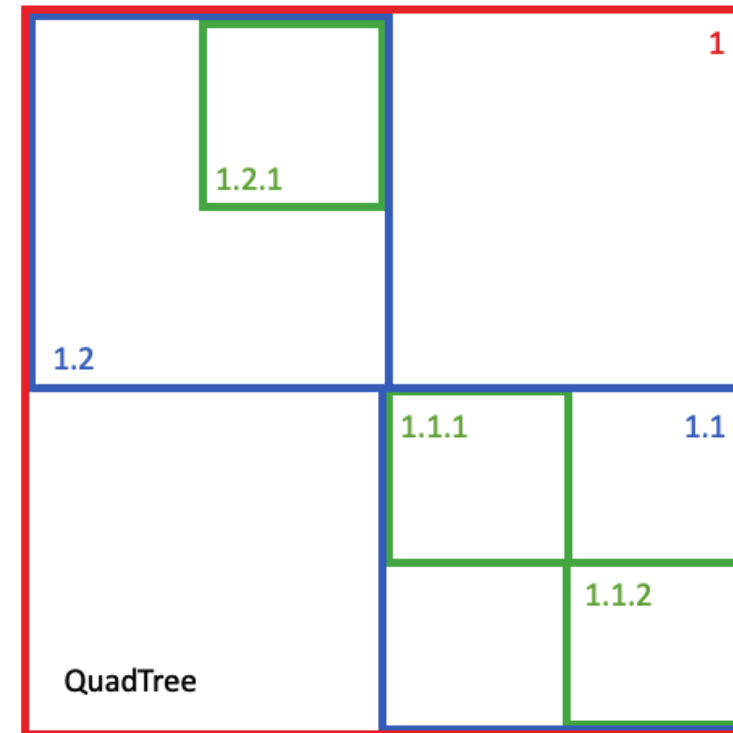
SP-GiST-Index

(Space-Partitioned GiST)

- Optimiert für nicht überlappende Geometrien, da ein Quad-Tree verwendet wird
- Gut für Punktgeometrien, da diese perfekt in Bereich unterteilt werden können
- In manchen Fällen schneller als GiST

Beispiel:

```
CREATE INDEX idx_geom ON buildings USING spgist (geom);
```



Grafik von: <https://www.crunchydata.com/blog/the-many-spatial-indexes-of-postgis>

BRIN-Index

(Block Range Index)

- Benötigt wenig Speicherplatz
- Sinnvoll bei sehr großen Tabellen, da Zusammenfassungen von Datenblöcken gespeichert werden
- Nur sinnvoll bei sortierten Daten!
 - Profitiert von Clustering oder GeoHashing

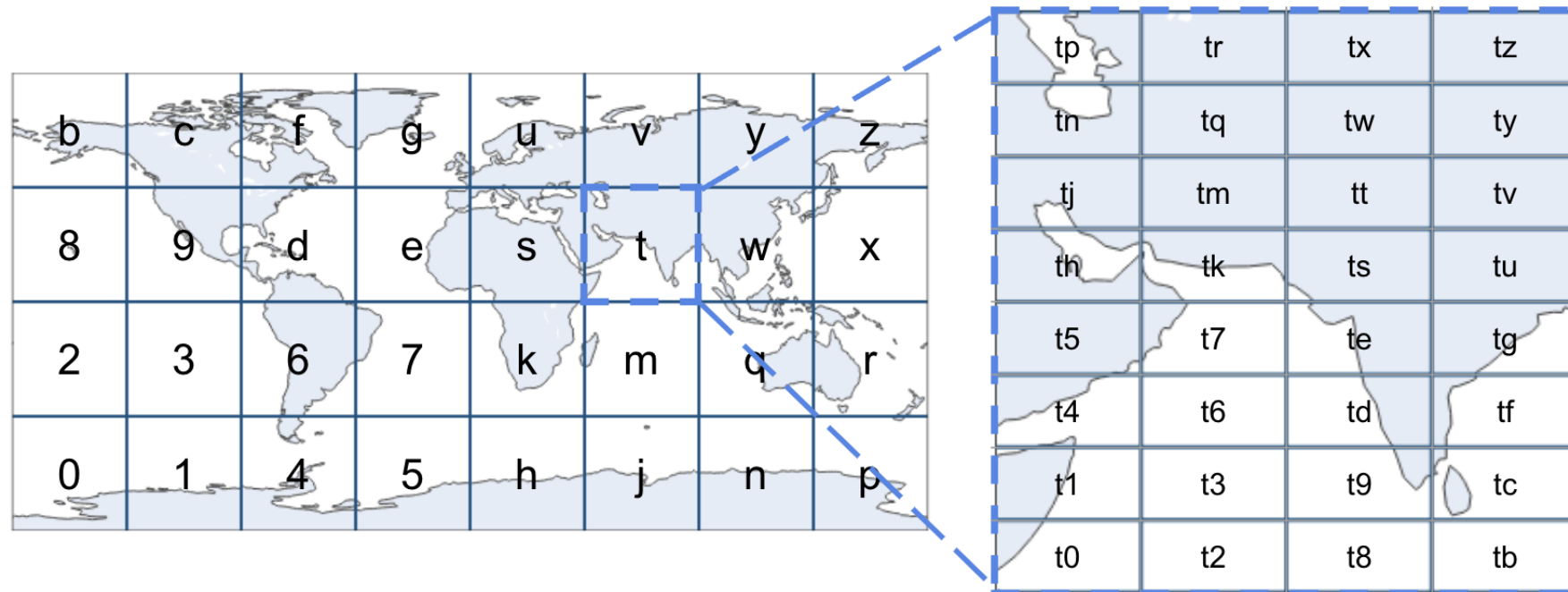
Beispiel:

```
CREATE INDEX idx_geom ON buildings USING brin (geom);
```

Datenorganisation

- **GeoHashing:** Kodierung von Koordinaten
- Repräsentiert Geometrie als Zeichenkette (1D)
- Ist sortierbar und durchsuchbar (Präfix)
- Je länger der GeoHash, desto genauer die Position

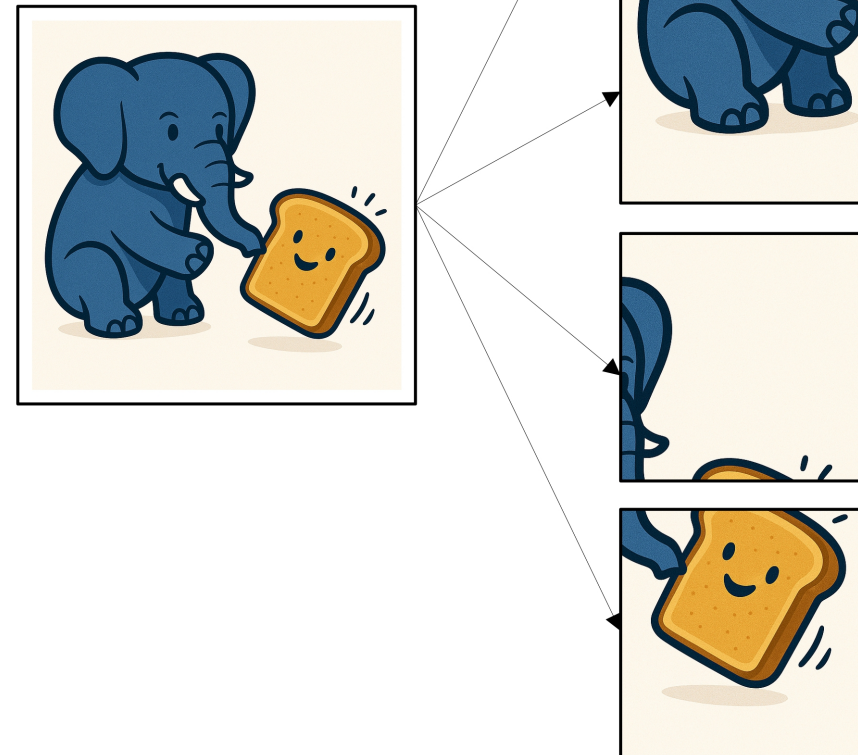
```
SELECT ST_GeoHash( ST_Point(-126,48), 5); -- Ergebnis: c0w3h
```



Grafik von: <https://benfeifke.com/posts/geospatial-indexing-explained/>

Optimierung großer Geometrien

- Problem: Große Geometrien und TOAST
(The **O**versized-**A**tttribute **S**torage **T**echnique)
 - Große Objekte werden in Postgres in Side Tables ausgelagert
 - Original-Objekt wird mit Verweisen zu Einzelteilen ersetzt
 - Herausholen solcher Objekte ist zeitaufwändig
→ alle Teile zusammensammeln und -setzen!

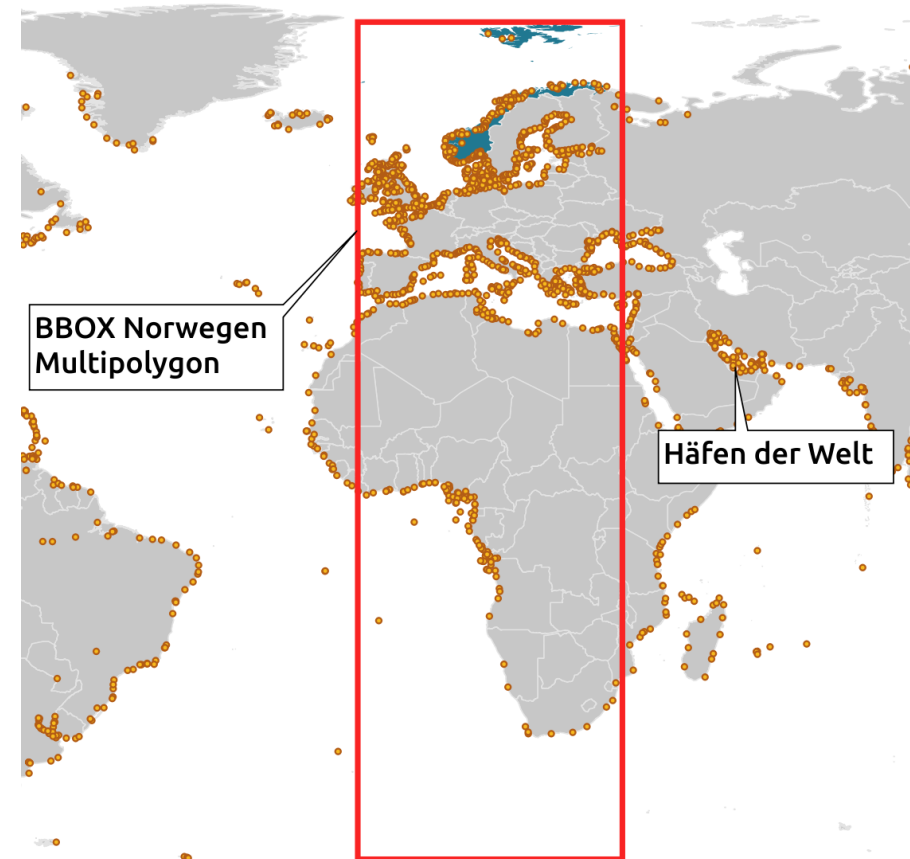


Optimierung großer Geometrien für Abfragen

- Grundsätzlich: Index macht räumliche Abfragen schneller!

AZ QUERY PLAN	AZ QUERY PLAN
Nested Loop (cost=0.00..92674.37 rows=7 width=9)	Nested Loop (cost=0.15..21.69 rows=7 width=9)
Join Filter: st_intersects(n.geom, p.geom)	-> Seq Scan on norway n (cost=0.00..1.01 rows=1)
Rows Removed by Join Filter: 7308	-> Index Scan using gist_world_ports_geom on w
-> Seq Scan on norway n (cost=0.00..1.01 rows=1)	Index Cond: (geom && n.geom)
-> Seq Scan on world_ports p (cost=0.00..812.43 r	Filter: st_intersects(n.geom, geom)
Planning Time: 0.138 ms	Rows Removed by Filter: 1748
Execution Time: 9.123 ms	Planning Time: 0.209 ms
	Execution Time: 4.564 ms

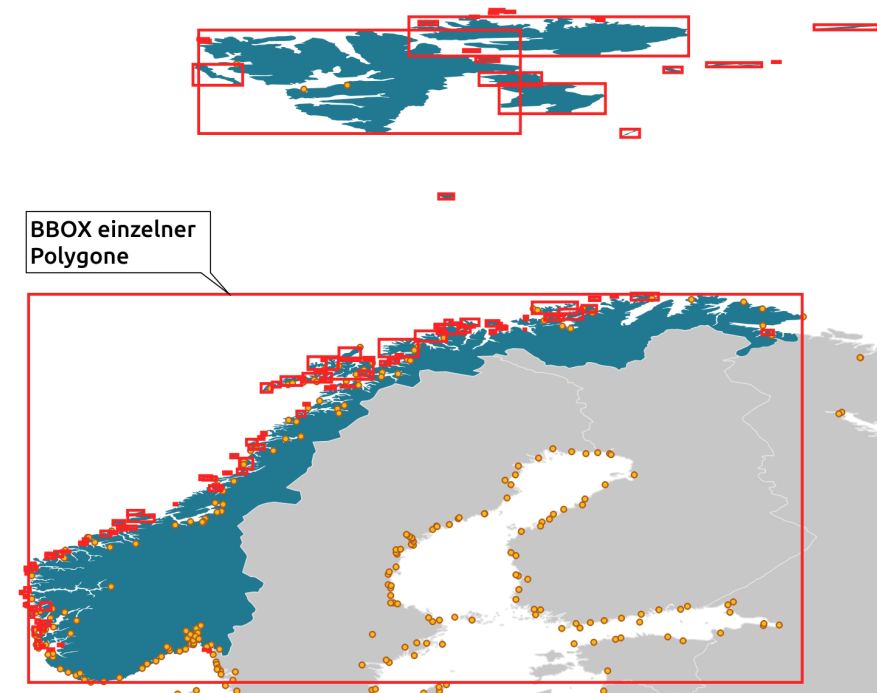
- Problem: Vereinfachung durch BBOX
 - Index wird durch BBOX zwar schnell aufgebaut, ist aber eine starke Abstraktion
 - Vorfilterung kann dadurch schlecht sein



Optimierung großer Geometrien für Abfragen

Lösung – ST_Dump: Multigeometrien in Einzelgeometrien zerteilen

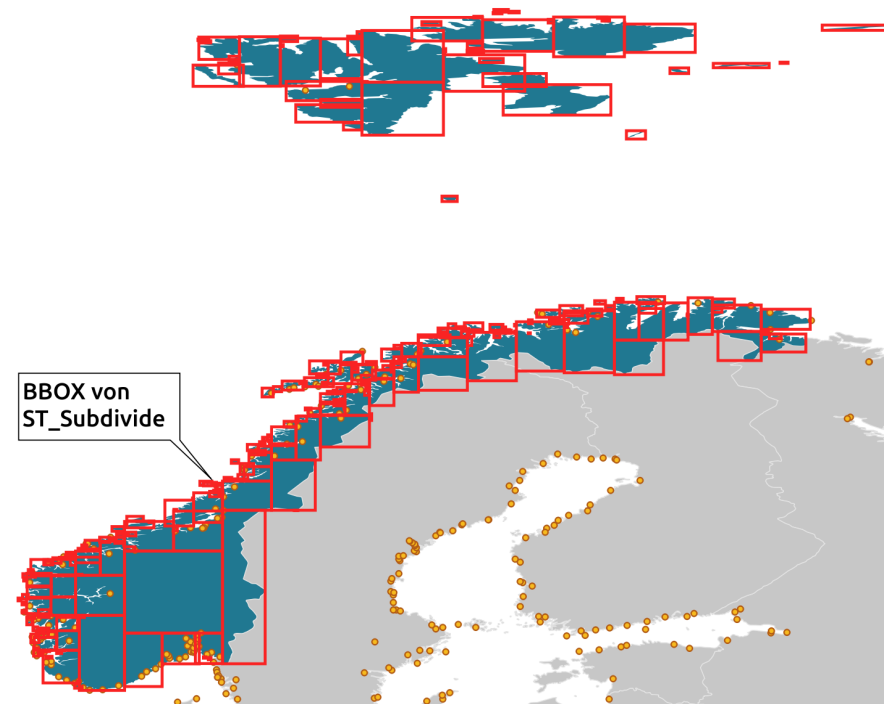
```
SELECT gid, name, (ST_Dump(geom)).geom AS einzel_geom  
FROM countries  
WHERE name = 'Norway';
```



Optimierung großer Geometrien für Abfragen

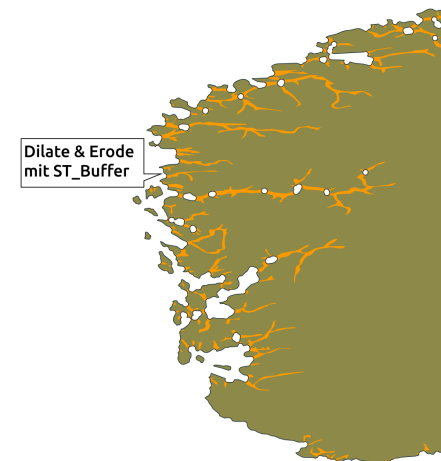
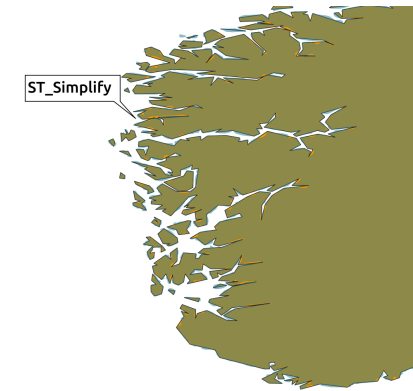
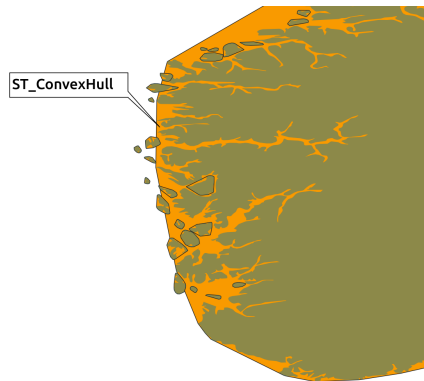
Lösung – ST_Subdivide: Zerlegen großer Geometrien in kleinere Teile

```
SELECT gid, name, ST_Subdivide(geom, 200) AS geteilte_geom  
FROM countries  
WHERE name = 'Norway';
```



Optimierung großer Geometrien zur Darstellung

- Lösung – Geometrien vereinfachen
 - **ST_Simplify:** Reduziert die Komplexität von Geometrien
 - **Dilate & Erode:** Puffern mit positivem & danach mit gleich großem negativem Puffer zum Glätten (evtl. anschließend noch ST_Simplify)
 - **ST_ConvexHull:** Füllt Einkerbungen; ursprüngliche Geometrie bleibt enthalten



Teil 2: Optimierung der Datenzugriffe

Effiziente Abfragen und Query-Optimierungen

Abfrageoptimierung mit EXPLAIN ANALYZE

- Ausführungsplan des Query Planners:
 - Welche Schritte werden ausgeführt?
 - Werden Indizes genutzt?
 - Was ist die geschätzte Ausführungszeit?
- Debugging-Tipps:
 - Wo werden Indizes genutzt und wo nicht?
 - Welche Teile der Query dauern besonders lange?

<pre>EXPLAIN ANALYZE SELECT admin_name FROM laender JOIN baum ON ST_Intersects(laender.geom, baum.geom)</pre>
QUERY PLAN
Nested Loop (cost=0.13..84.35 rows=1 width=102) (actual time=0.061..0.147 rows=5 loops=1)
-> Seq Scan on baum (cost=0.00..1.05 rows=5 width=32) (actual time=0.006..0.007 rows=5 loops=1)
-> Index Scan using laender_geom_idx on laender (cost=0.13..16.65 rows=1 width=134) (actual time=0.027..0.027 rows=1 loops=5)
Index Cond: (geom && baum.geom)
Filter: st_intersects(geom, baum.geom)
Rows Removed by Filter: 1
Planning Time: 0.096 ms
Execution Time: 0.166 ms

Allgemeine Tipps

- **Joins optimieren:** Direkte Joins verwenden statt alles in die WHERE-Bedingungen schreiben

Alles im Filter:

```
SELECT o.id, o.name, pp.plz_ort
FROM ortschaften o, plz_gebiete p
WHERE ST_Intersects(o.geom,p.geom)
AND p.plz_ort LIKE 'Berlin%';
```

```
SELECT o.id, o.name
FROM ortschaften o
WHERE o.plz IN (
    SELECT postleitzahl
    FROM plz_gebiete
    WHERE plz_ort LIKE 'Berlin%'
);
```

Expliziter JOIN

```
SELECT o.id, o.name, pp.plz_ort
FROM ortschaften o
JOIN plz_gebiete p
ON ST_Intersects(o.geom,p.geom)
WHERE p.plz_ort LIKE 'Berlin%';
```

```
SELECT o.id, o.name
FROM ortschaften o
JOIN plz_gebiete p
ON o.plz = p.postleitzahl
WHERE p.plz_ort LIKE 'Berlin%';
```

Allgemeine Tipps

- **Materialized Views:** Ergebnisse komplexer Abfragen speichern und indizieren

```
CREATE MATERIALIZED VIEW mv_schulen_500m_radius AS
  SELECT id, name, geom
  FROM schulen
  WHERE ST_DWithin(geom, ST_SetSRID(ST_Point(13.4, 52.5), 4326), 500);

-- Index anlegen
CREATE INDEX idx_mv_schulen_500m_radius_geom ON mv_schulen_500m_radius USING gist(geom);

-- Mat. View auffrischen
REFRESH MATERIALIZED VIEW mv_schulen_500m_radius;
```

Allgemeine Tipps

- **Unnötige CTEs vermeiden:** Können die Nutzung des Index verhindern!
- CTEs mit Zwischenergebnissen räumlicher Operationen besser in Temporary Tables oder Materialized Views speichern

```
-- Unnötige CTE, die Indexnutzung verhindert
WITH s AS (
  SELECT id, name, geom
  FROM schulen
)
SELECT id, name, geom
FROM s
JOIN plz_gebiete p
ON ST_Intersects(s.geom,p.geom)
WHERE p.plz_ort LIKE 'Berlin%';
```

Zusammenfassung

- Indizes sind der Schlüssel zur Performance
- Zugriff auf komplexe Geometrien minimieren
- Komplexe Geometrien zerteilen oder vereinfachen
- Abfrageoptimierung mit EXPLAIN ANALYZE!


PostGIS-Tuning – Performance-Tricks für Geodatenprofis

Interesse geweckt? – Fragen Sie gerne!



 Website

<https://wherogroup.com>

 E-Mail

annika.froede@wherogroup.com

 Telefon

+49 305 1302 78 82

 Adresse

WhereGroup GmbH

Bundesallee 23

10717 Berlin

 WhereGroup Shorts

<https://wherogroup.com/shorts/>