Session 12: Indexe & Performance

Willkommen zur Session über Indexe und Performance! In der letzten Session haben Sie gesehen, wie KI mit MCP SQL-Queries generiert – aber sind diese Queries auch effizient? Heute lernen Sie, Performance zu messen, zu verstehen und zu optimieren.

Stellen Sie sich vor: Eine Query läuft auf der IMDB-Datenbank mit über 178.000 Titeln. Ohne Index scannt die Datenbank jede einzelne Zeile. Mit dem richtigen Index? Direkter Zugriff in Millisekunden. Das ist der Unterschied, den wir heute mit PGlite live erleben werden.

Lernziele dieser Session:

- Verstehen, was Indexe sind und wie sie funktionieren
- Praktische Indexe mit PGlite erstellen und Performance messen
- Query Plans mit EXPLAIN ANALYZE lesen können
- Kritisch bewerten: Wann Indexe sinnvoll sind (und wann nicht)
- Best Practices für Index-Strategien anwenden

Motivation: Der Performance-Unterschied

Lassen Sie uns mit einem konkreten Problem beginnen. Sie haben in Session 11 mit MCP die IMDB-Datenbank erkundet. GitHub Copilot hat Ihnen SQL-Queries generiert – aber niemand hat über Performance gesprochen.

Nehmen wir eine typische Anfrage: „Zeige mir alle Filme mit einem Rating über 8.0“. Klingt einfach, oder?

Das Problem:

```
SELECT tb.primarytitle, tr.averagerating, tb.startyear
FROM title_basics tb
JOIN title_ratings tr ON tb.tconst = tr.tconst
WHERE tr.averagerating > 8.0 AND tb.titletype = 'movie';
```

```
|   #  | primarytitle                | averagerating | startyear |
|------|-----------------------------|---------------|-----------|
|    1 | Milionar pentru o zi        |      8.3      |   1924    |
|    2 | Napoleon                    |      8.2      |   1927    |
|    3 | Zeinab                      |      8.6      |   1930    |
|    4 | It's a Wise Child           |      8.4      |   1931    |
|    5 | Geld fällt vom Himmel       |      8.2      |   1938    |
|    6 | La tonta del bote           |      8.6      |   1939    |
|    7 | Herzensfreud - Herzensleid  |      8.6      |   1940    |
|    8 | The Best Years of Our Lives |      8.1      |   1946    |
|    9 | Abhimanyu                   |      8.2      |   1948    |
|   10 | Los tres huastecos          |      8.1      |   1948    |
|   11 | Pathala Bhairavi            |      8.5      |   1951    |
|   12 | The Life of Oharu           |      8.1      |   1952    |
... (more rows) ...
```

Ohne Index durchsucht PostgreSQL jede einzelne Zeile in title_ratings – das sind über 178.000 Einträge! Bei einer großen Produktionsdatenbank wären das Millionen oder Milliarden.

Szenario ohne Index:

- Sequential Scan über 178.000+ Zeilen
- Jede Zeile wird gelesen und gefiltert
- Typische Ausführungszeit: 50–200ms (abhängig vom System)

Szenario mit Index:

- Index Scan nur auf relevante Zeilen
- Direkter Zugriff via B-Baum
- Typische Ausführungszeit: 5–20ms (10× schneller!)

Das ist nicht nur ein akademisches Problem. In einer E-Commerce-Anwendung bedeutet das: 10× schnellere Produktsuche, 10× mehr gleichzeitige Nutzer, 10× bessere User Experience.

Frage zum Nachdenken:

Wenn eine Query ohne Index 100ms braucht und 1000× pro Sekunde ausgeführt wird – wie viel CPU-Zeit sparen Sie mit einem 10× schnelleren Index?

Was sind Indexe?

Bevor wir in die Praxis gehen, lassen Sie uns verstehen, was Indexe eigentlich sind. Die beste Metapher: Ein Buchindex.

Konzept: Datenbank-Indexe

Stellen Sie sich ein Fachbuch mit 1000 Seiten vor. Sie suchen den Begriff „B-Baum“. Ohne Index müssten Sie jede Seite durchblättern – das dauert. Mit Index? Sie schauen hinten nach, finden „B-Baum → Seite 342“ und springen direkt dorthin.

Ein **Index** ist eine zusätzliche Datenstruktur, die Spalten einer Tabelle sortiert und schnellen Zugriff ermöglicht.

Ohne Index:

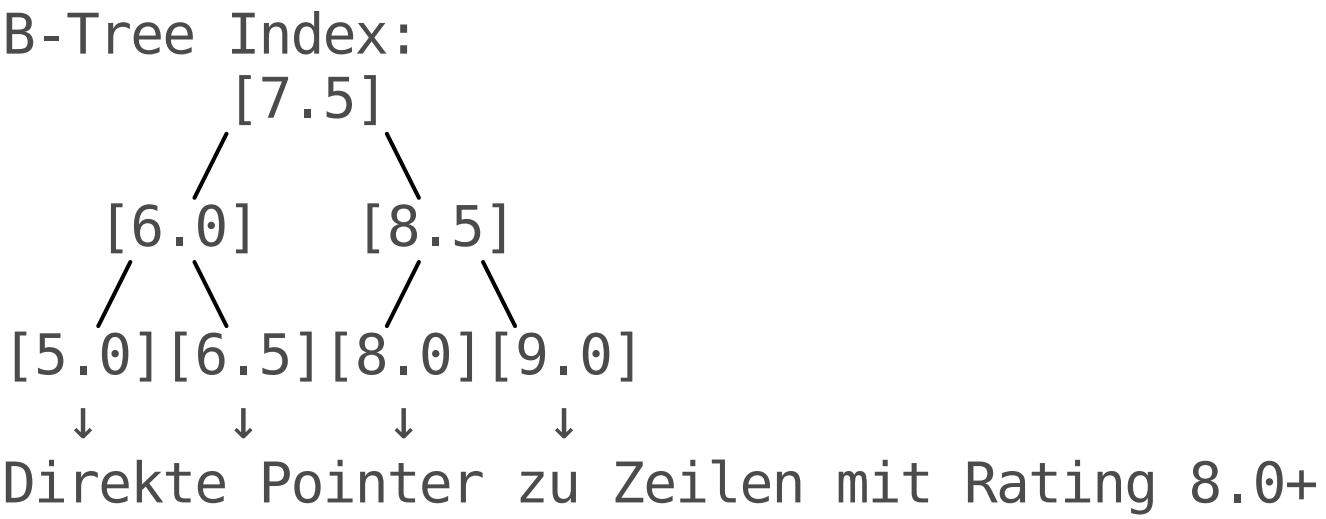
Table: title_ratings

tconst	averageRating	numVotes	
tt0000001	5.7	2000	← Scan Zeile 1
tt0000002	6.1	300	← Scan Zeile 2
tt0000003	8.2	5000	← Scan Zeile 3
...	...	...	← Scan Zeile 4-178000
tt0999999	7.5	1200	← Scan Zeile 178000

↓

Sequential Scan (langsam!)

Mit Index auf `averageRating`:



B-Baum: Die Datenstruktur hinter Indexen

Der Index ist wie ein sortierter Wegweiser. Anstatt linear zu suchen, navigieren Sie durch einen Baum – das ist logarithmisch schneller: O(log n) statt O(n).

Die meisten Datenbanken (SQLite, PostgreSQL, MySQL) nutzen **B-Bäume** (balanced trees) für Indexe.

Eigenschaften:

- Selbstbalancierend (immer gleiche Tiefe)
- Mehrere Werte pro Knoten (Cache-effizient)
- Sortierte Speicherung (Range-Queries möglich)

Zeitkomplexität:

- Suche: O(log n)
- Einfügen: O(log n)
- Löschen: O(log n)

Beispiel: Bei 1.000.000 Zeilen:

- Ohne Index: ~1.000.000 Vergleiche
- Mit B-Baum: ~20 Vergleiche ($\log_2 1.000.000 \approx 20$)

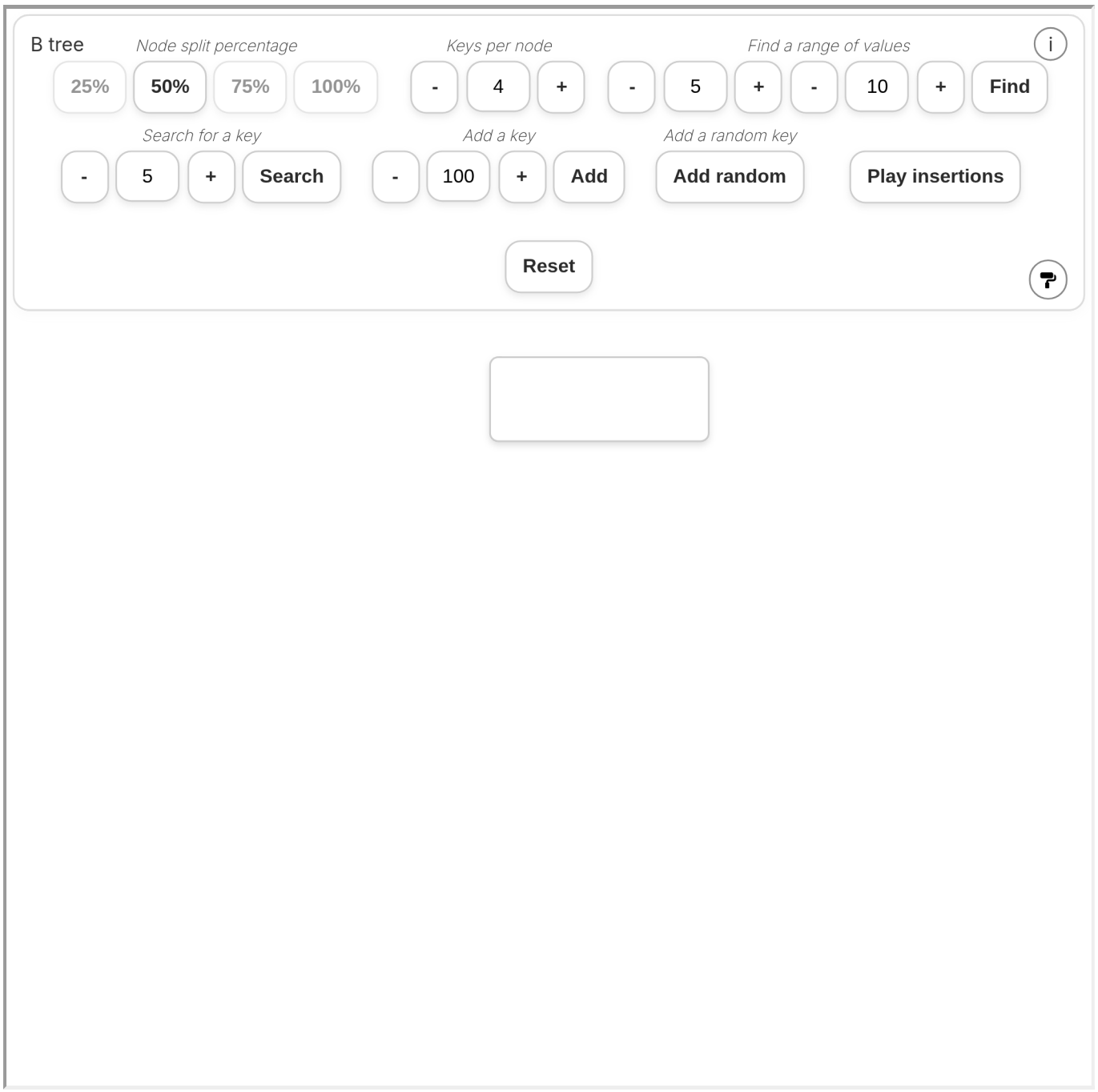
Das ist der Grund, warum Indexe so mächtig sind. Aber Vorsicht: Jeder Index kostet Speicherplatz und verlangsamt INSERT/UPDATE/DELETE. Es ist ein Trade-off.

Trade-offs: Die Kehrseite der Medaille

Aspekt	Vorteil ✅	Nachteil ⚠️
SELECT	Schnellere Abfragen (10×–100×)	–
INSERT	–	Langsamer (Index aktualisieren)
UPDATE	–	Langsamer (Index neu sortieren)
DELETE	–	Langsamer (Index bereinigen)
Speicher	–	Zusätzlicher Platzbedarf (~10–30% der Tabelle)
Maintenance	–	Fragmentierung, VACUUM nötig

Die Kunst des Datenbankdesigns ist es, die richtigen Indexe zu wählen: Genug für Performance, aber nicht zu viele, um Writes nicht zu bremsen.

Demo



https://btree.app

Hands-on: Indexe in Aktion

Jetzt wird es praktisch! Wir nutzen die IMDB-Datenbank aus Session 11 und führen Performance-Experimente durch. Sie werden den Unterschied selbst sehen – und messen.

Setup: IMDB-Datenbank verbinden

Führen Sie das folgenden Script aus um die IMDB-Datenbank für diese Session in PGlite zu laden.

```
1 function wait(ms) {
2   return new Promise(resolve => setTimeout(resolve, ms));
3 }
```

```
4
5  const response = await fetch("../assets/dat/imdb.sql");
6  if (!response.ok) {
7    console.error("Failed to fetch SQL dump");
8    return;
9  }
10
11 let sql = await response.text();
12 sql = sql
13   .split(/;\s*\n/)   // split on statement-ending semicolon
14   .map(s => s.trim())
15   .filter(Boolean)
16   .map(s => s + ";"); // re-add semicolon
17
18
19 let size = Math.round(sql.length / 100);
20 for (let i = 0; i < sql.length; i += size) {
21     console.log((i * 100) / sql.length, "%");
22     await db.exec(sql.slice(i, i+size).join("\n"));
23     await wait(50); // small delay to keep UI responsive
24 }
25
26 // Load into PGlite
27 console.log("done")
```

```
Failed to fetch
```

PGlite-Setup:

- Datenbank läuft im Browser (kein Server nötig!)
- Alle Queries führen Sie direkt in dieser Session aus
- Die Daten bleiben im Browser-Speicher

Prüfen wir zunächst, welche Indexe bereits existieren. Neue Tabellen haben meist nur einen Index auf dem Primärschlüssel.

Schritt 1: Vorhandene Indexe prüfen

```
1  -- PostgreSQL-Syntax: Alle Indexe in der aktuellen Datenbank
2  SELECT
3      schemaname,
4      tablename,
5      indexname,
6      indexdef
7  FROM pg_indexes
8  WHERE schemaname = 'public'
```

```
WHERE schemaname = 'public'
ORDER BY tablename, indexname;
```

```
-- PostgreSQL-Syntax: Alle Indexe in der aktuellen Datenbank
SELECT
    schemaname,
    tablename,
    indexname,
    indexdef
FROM pg_indexes
WHERE schemaname = 'public'
ORDER BY tablename, indexname
```

#	schemaname	tablename	indexname	indexdef

0 rows

Sie sehen vermutlich nur automatische Indexe auf Primärschlüsseln wie `tconst_pkey` oder `nconst_pkey`. Gut – das ist unser Ausgangspunkt.

Experiment 0: (Index-) Scan Typen

Bevor wir zu komplexen Beispielen kommen, lernen wir die verschiedenen Scan-Strategien kennen. PostgreSQL wählt unterschiedliche Ansätze, je nachdem welche Spalten abgefragt werden. Wir erstellen erst einen Index und testen dann drei Szenarien.

Index auf primaryTitle erstellen

```
-- Index für unsere Experimente
CREATE INDEX IF NOT EXISTS
  idx_title
  ON title_basics(primaryTitle);

-- Bitmap Scan (vorläufig ausschalten)
SET enable_bitmapscan = off;
```

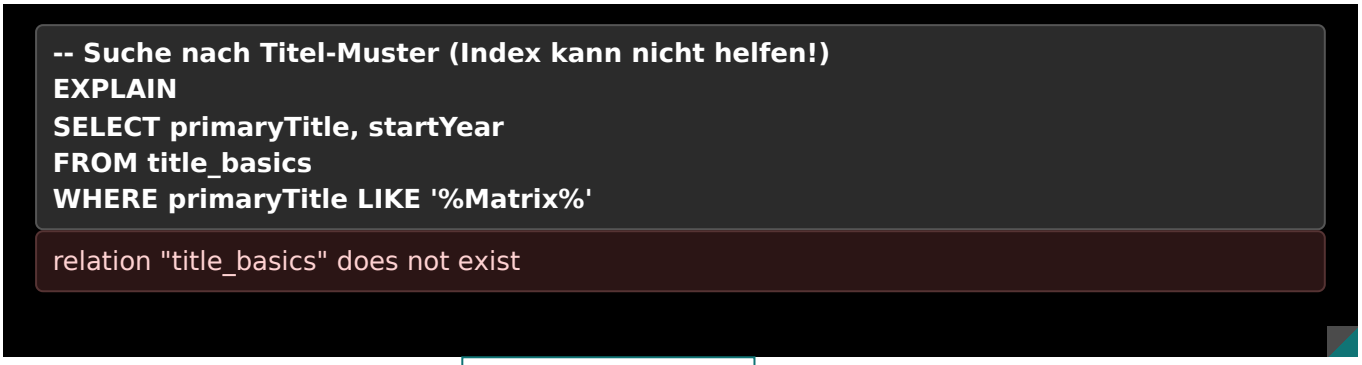
```
-- Index für unsere Experimente
CREATE INDEX IF NOT EXISTS
  idx_title
  ON title_basics(primaryTitle)
```

relation "title_basics" does not exist

Jetzt haben wir einen B-Baum-Index auf den Filmtiteln. Schauen wir uns an, wie PostgreSQL diesen Index nutzt.

```
-- Suche nach Titel-Muster (Index kann nicht helfen!)
```

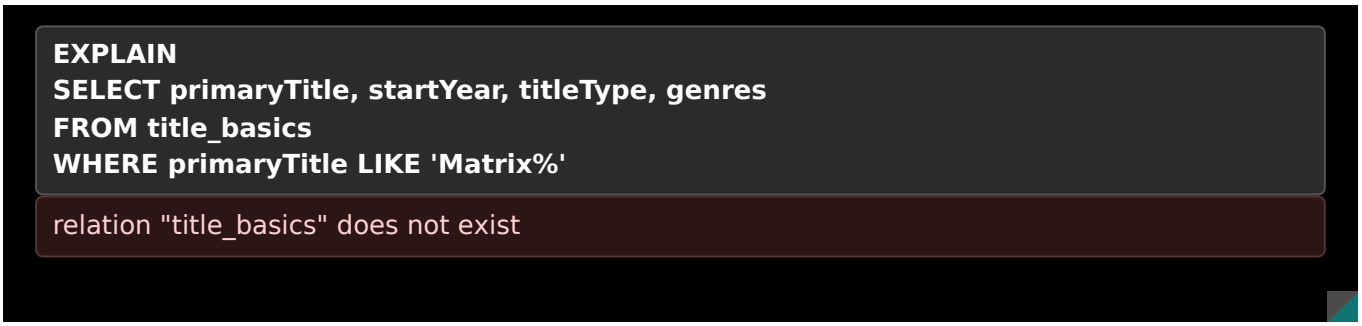
```
EXPLAIN
SELECT primaryTitle, startYear
FROM title_basics
WHERE primaryTitle LIKE '%Matrix%';
```



Sie sehen „Seq Scan“ – warum? Weil `LIKE '%Matrix%'` in der Mitte sucht. Der Index ist alphabetisch sortiert, kann aber nur Präfix-Suchen optimieren. Hier muss jede Zeile gelesen werden.

Szenario 2: Index Scan (Index + Table)

```
EXPLAIN
SELECT primaryTitle, startYear, titleType, genres
FROM title_basics
WHERE primaryTitle LIKE 'Matrix%';
```



Jetzt sehen Sie „Index Scan using idx_title“ – PostgreSQL nutzt den Index, um die Zeile zu finden, muss aber zusätzlich die Tabelle lesen, um `startYear`, `titleType` und `genres` zu holen (die sind NICHT im Index gespeichert).

Index Scan bedeutet: 1. Index durchsuchen → Zeilen-Position finden 2. Zur Tabelle (Heap) springen → Alle Spalten lesen

Szenario 3: Index Only Scan (nur Index!)

```
-- Query NUR auf die indexierte Spalte
EXPLAIN
SELECT primaryTitle
FROM title_basics
WHERE primaryTitle::Text = 'Interstellar';
```

```
-- Query NUR auf die indexierte Spalte
EXPLAIN
SELECT primaryTitle
FROM title_basics
WHERE primaryTitle::Text = 'Interstellar'
```

```
relation "title_basics" does not exist
```

Jetzt könnte PGlite/PostgreSQL einen „Index Only Scan“ verwenden – alle Daten (nur primaryTitle) sind bereits im Index! Kein Table-Read nötig. Das ist die schnellste Variante.

Index Only Scan bedeutet:

1. Index durchsuchen → Wert direkt aus Index lesen
2. Kein Heap-Zugriff nötig!

Hinweis: In PGlite/PostgreSQL funktioniert Index Only Scan nur, wenn: - Alle SELECT-Spalten im Index sind - Die Tabelle „visibility map“ hat (VACUUM wurde ausgeführt)

Bonus: COUNT mit Index

```sql
-- Zählen mit Index-Unterstützung

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM title_basics
WHERE primaryTitle LIKE 'Matrix%';
```

```
-- Zählen mit Index-Unterstützung

EXPLAIN ANALYZE
SELECT COUNT(*)
FROM title_basics
WHERE primaryTitle LIKE 'Matrix%'
```

```
relation "title_basics" does not exist
```

Diese Query findet 4 Filme mit „Matrix“ am Anfang. PGlite kann den Index nutzen, weil es eine Präfix-Suche ist (sortierter Index hilft!). Je nach Optimierung sehen Sie einen Bitmap Index Scan.

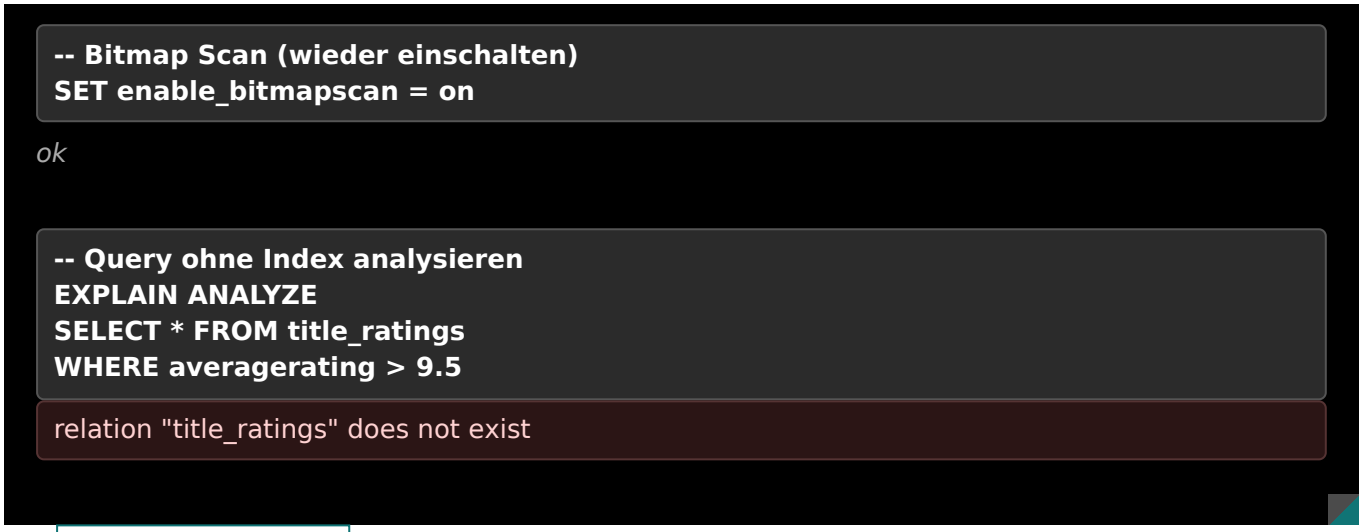
Experiment 1: Index auf `averageRating`

Unser erstes Experiment: Wir suchen alle Titel mit einem Rating über 9.5. Erst ohne Index, dann mit Index – und vergleichen die Performance.

Schritt 2: Baseline messen (ohne Index)

```sql
-- Bitmap Scan (wieder einschalten)
SET enable_bitmapscan = on;

-- Query ohne Index analysieren
EXPLAIN ANALYZE
SELECT * FROM title_ratings
WHERE averagerating > 9.5;
```



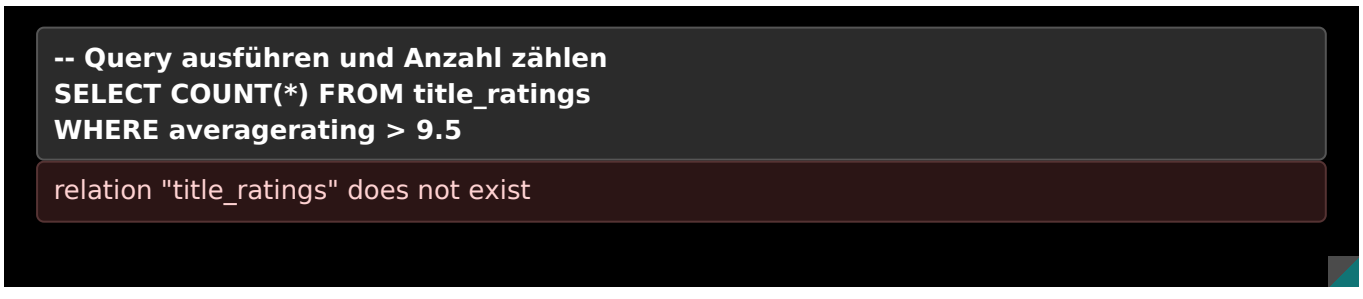
Das `EXPLAIN ANALYZE` zeigt uns, wie PostgreSQL die Query ausführt UND misst die tatsächliche Zeit. Sie sehen vermutlich „Seq Scan on title_ratings“ – das bedeutet: Sequential Scan, jede Zeile wird gelesen.

Erwartete Ausgabe:

```
Seq Scan on title_ratings  (cost=10000000000.00..10000002837.50 rows=45400
  =40)
```

Jetzt führen wir die Query tatsächlich aus und zählen die Ergebnisse:

```sql
-- Query ausführen und Anzahl zählen
SELECT COUNT(*) FROM title_ratings
WHERE averagerating > 9.5;
```



PGlite ist im Browser sehr schnell, aber bei größeren Datenmengen sehen Sie trotzdem den Unterschied. Merken Sie sich die Ausführungszeit!

Schritt 3: Index erstellen

```sql
-- Index auf averagerating erstellen
```

```
2  CREATE INDEX idx_rating ON title_ratings(averagerating);
```

```
-- Index auf averagerating erstellen
CREATE INDEX idx_rating ON title_ratings(averagerating)
```

relation "title_ratings" does not exist

Das Erstellen dauert ein paar Sekunden – die Datenbank sortiert jetzt alle 178.000+ Einträge nach Rating und baut den B-Baum auf.

Schritt 4: Gleiche Query mit Index

```
1  -- Query erneut analysieren (mit Index)
2  EXPLAIN ANALYZE
3  SELECT * FROM title_ratings
4  WHERE averagerating > 9.5;
```

```
-- Query erneut analysieren (mit Index)
EXPLAIN ANALYZE
SELECT * FROM title_ratings
WHERE averagerating > 9.5
```

relation "title_ratings" does not exist

Jetzt sollten Sie „Index Scan using idx_rating“ oder „Bitmap Index Scan“ sehen. PostgreSQL nutzt den Index!

Erwartete Ausgabe:

```
Index Scan using idx_rating on title_ratings  (cost=0.29..XXX.XX rows=XXXX
  =XX)
  Index Cond: (averagerating > '8.0'::numeric)
```

Messen wir erneut durch direktes Ausführen:

```
1  SELECT COUNT(*) FROM title_ratings
2  WHERE averagerating > 9.5;
```

```
SELECT COUNT(*) FROM title_ratings
WHERE averagerating > 9.5
```

relation "title_ratings" does not exist

Typisches Ergebnis: 2–10ms – das ist 5×–10× schneller! Je größer die Datenbank, desto dramatischer der Unterschied.

Reflexion:

- Wie groß war der Speedup bei Ihnen?
- Würden Sie diesen Index in Production einsetzen?
- Welche Queries würden davon profitieren?

Cleanup: Index für nächstes Experiment entfernen

```
1  -- Index wieder löschen für saubere Ausgangslage
2  DROP INDEX IF EXISTS idx_rating;
```

```
-- Index wieder löschen für saubere Ausgangslage
DROP INDEX IF EXISTS idx_rating
```

ok

Experiment 2: Composite Index auf `startYear` + `titleType`

Manchmal filtern Queries nach mehreren Spalten. Ein **Composite Index** (Multi-Column Index) kann hier helfen – aber die Reihenfolge der Spalten ist wichtig!

Szenario: Alle Filme aus 2020 oder später

```
1  -- Query ohne Composite Index
2  EXPLAIN ANALYZE
3  SELECT * FROM title_basics
4  WHERE startyear >= '2020' AND titletype = 'movie';
```

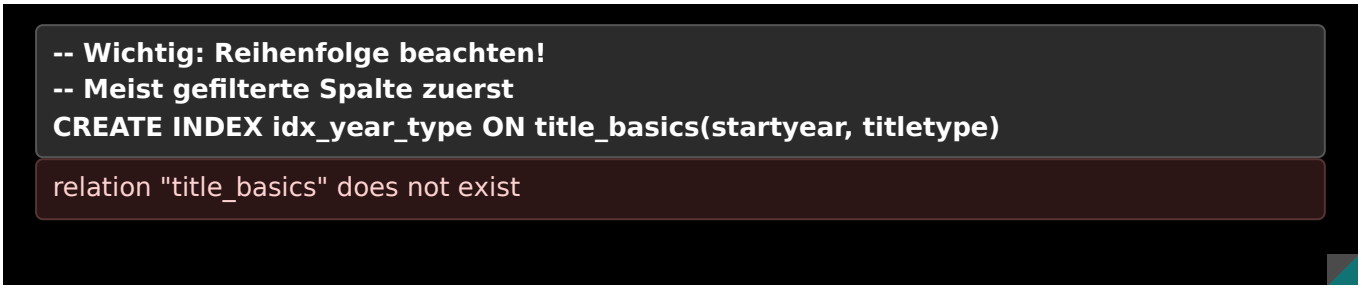
```
-- Query ohne Composite Index
EXPLAIN ANALYZE
SELECT * FROM title_basics
WHERE startyear >= '2020' AND titletype = 'movie'
```

relation "title_basics" does not exist

Ohne Index: Sequential Scan. Bei 178.000+ Titeln dauert das.

Schritt 5: Composite Index erstellen

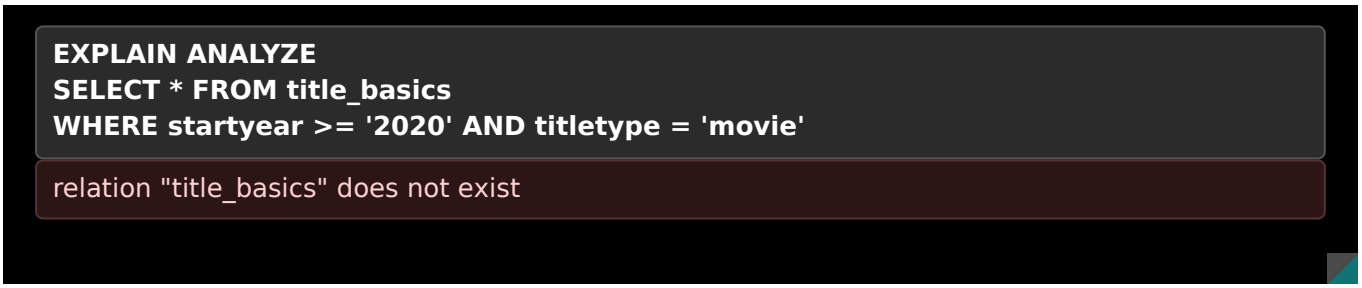
```sql
-- Wichtig: Reihenfolge beachten!
-- Meist gefilterte Spalte zuerst
CREATE INDEX idx_year_type ON title_basics(startyear, titletype);
```



Warum diese Reihenfolge? Weil `startYear` eine Range ist (>=), `titleType` eine Gleichheit (=). B-Bäume arbeiten am besten, wenn Ranges zuerst kommen.

Schritt 6: Query mit Composite Index

```sql
EXPLAIN ANALYZE
SELECT * FROM title_basics
WHERE startyear >= '2020' AND titletype = 'movie';
```



Jetzt nutzt SQLite den Composite Index – aber nur, wenn beide Spalten im WHERE vorkommen!

Wichtige Erkenntnis:

Ein Index auf `(A, B)` hilft bei:

- `WHERE A = ...` ✅
- `WHERE A = ... AND B = ...` ✅
- `WHERE B = ...` ❌ (Nur zweite Spalte → Index nutzlos!)

Reihenfolge der Spalten im Index ist entscheidend!

Testen Sie das selbst: Erstellen Sie einen Index `(titleType, startYear)` und vergleichen Sie die Performance. Sie werden sehen: Oft langsamer!

Cleanup: Index für nächstes Experiment entfernen

```sql
-- Index wieder löschen für saubere Ausgangslage
DROP INDEX IF EXISTS idx_year_type;
```

```
-- Index wieder löschen für saubere Ausgangslage
DROP INDEX IF EXISTS idx_year_type
```

ok

Experiment 3: JOIN-Performance mit Foreign Keys

Jetzt wird es dramatisch! Joins ohne Index sind der Performance-Albtraum schlechthin. Ohne Index auf der JOIN-Spalte muss die Datenbank einen Nested Loop durchführen – bei 178.000 Zeilen bedeutet das theoretisch 31 MILLIARDEN Vergleiche!

Szenario: Top-bewertete Filme mit allen Details

Schritt 7: Baseline ohne Index (LANGSAM!)

```sql
-- Query OHNE Index auf tconst analysieren
EXPLAIN ANALYZE
SELECT tb.primaryTitle, tr.averageRating, tb.startYear
FROM title_basics tb
JOIN title_ratings tr ON tb.tconst = tr.tconst
WHERE tr.averageRating >= '9.0'
ORDER BY tr.averageRating DESC
LIMIT 20;
```

```
-- Query OHNE Index auf tconst analysieren
EXPLAIN ANALYZE
SELECT tb.primaryTitle, tr.averageRating, tb.startYear
FROM title_basics tb
JOIN title_ratings tr ON tb.tconst = tr.tconst
WHERE tr.averageRating >= '9.0'
ORDER BY tr.averageRating DESC
LIMIT 20
```

relation "title_basics" does not exist

Sie sehen vermutlich „Seq Scan“ auf beiden Tabellen und einen „Hash Join“ oder „Nested Loop“. Bei großen Datenmengen ist das extrem langsam – jede Zeile aus title*ratings muss mit ALLEN Zeilen aus title*basics verglichen werden.

Erwartete Ausgabe (ohne Index):

```
Hash Join  (cost=5000.00..15000.00 rows=10000)
```

```
Hash Join  (cost=5000..15000 rows=10000)
  ->  Seq Scan on title_basics tb
  ->  Hash
        ->  Seq Scan on title_ratings tr
              Filter: (averageRating >= '9.0')
```

Messen wir die tatsächliche Zeit durch direktes Ausführen:

```sql
-- Query ausführen und Ergebnisse zählen
SELECT COUNT(*) as result_count
FROM title_basics tb
JOIN title_ratings tr ON tb.tconst = tr.tconst
WHERE tr.averageRating >= '9.0';
```

-- Query ausführen und Ergebnisse zählen
SELECT COUNT(*) as result_count
FROM title_basics tb
JOIN title_ratings tr ON tb.tconst = tr.tconst
WHERE tr.averageRating >= '9.0'

relation "title_basics" does not exist

Bei mir im Browser dauert das ohne Index spürbar länger – merken Sie sich die Zeit!

Schritt 8: Indexe auf JOIN-Spalten erstellen

```sql
-- Index auf tconst in BEIDEN Tabellen erstellen
CREATE INDEX IF NOT EXISTS idx_tconst_basics ON title_basics(tconst);
CREATE INDEX IF NOT EXISTS idx_tconst_ratings ON title_ratings(tconst)

-- Bonus: Index auf averageRating für den WHERE-Filter
CREATE INDEX IF NOT EXISTS idx_rating ON title_ratings(averageRating);
```

-- Index auf tconst in BEIDEN Tabellen erstellen
CREATE INDEX IF NOT EXISTS idx_tconst_basics ON title_basics(tconst)

relation "title_basics" does not exist

Das Erstellen dauert ein paar Sekunden – die Datenbank baut jetzt B-Bäume für schnelle Lookups auf. In Production würden diese Indexe normalerweise bereits existieren (besonders auf Primär- und Fremdschlüsseln).

Schritt 9: Gleiche Query mit Index

```sql
-- Query erneut analysieren (MIT Index)
EXPLAIN ANALYZE
SELECT tb.primaryTitle, tr.averageRating, tb.startYear
```

```
4  FROM title_basics tb
5  JOIN title_ratings tr ON tb.tconst = tr.tconst
6  WHERE tr.averageRating >= '9.0'
7  ORDER BY tr.averageRating DESC
8  LIMIT 20;
```

```
-- Query erneut analysieren (MIT Index)
EXPLAIN ANALYZE
SELECT tb.primaryTitle, tr.averageRating, tb.startYear
FROM title_basics tb
JOIN title_ratings tr ON tb.tconst = tr.tconst
WHERE tr.averageRating >= '9.0'
ORDER BY tr.averageRating DESC
LIMIT 20
```

relation "title_basics" does not exist

Jetzt sollten Sie „Index Scan“ oder „Index Only Scan“ sehen – die Datenbank nutzt die Indexe! Der JOIN wird dramatisch schneller.

Erwartete Ausgabe (mit Index):

```
Nested Loop  (cost=0.29..500 rows=10000)
  ->  Index Scan using idx_rating on title_ratings tr
        Index Cond: (averageRating >= '9.0')
  ->  Index Scan using idx_tconst_basics on title_basics tb
        Index Cond: (tconst = tr.tconst)
```

Die Komplexität ist von $O(n^2)$ auf $O(n \log n)$ gesunken – das ist bei großen Datenmengen der Unterschied zwischen Minuten und Millisekunden!

Performance-Vergleich:

```
1  -- Erneut ausführen und Zeit vergleichen
2  SELECT COUNT(*) as result_count
3  FROM title_basics tb
4  JOIN title_ratings tr ON tb.tconst = tr.tconst
5  WHERE tr.averageRating >= '9.0';
```

```
-- Erneut ausführen und Zeit vergleichen
SELECT COUNT(*) as result_count
FROM title_basics tb
JOIN title_ratings tr ON tb.tconst = tr.tconst
WHERE tr.averageRating >= '9.0'
```

relation "title_basics" does not exist

Typisches Ergebnis: 5–20× schneller! Bei Millionen von Zeilen wäre der Unterschied noch dramatischer – aus mehreren Minuten werden Sekunden.

Best Practice:

> **Jede Foreign Key-Spalte sollte einen Index haben.**
>
> Das gilt besonders für: - Primärschlüssel (meist automatisch) - Foreign Keys (oft manuell erstellen!) - Häufig gejoinete Spalten
>
> **Unsere IMDB-Demo-DB hat bewusst KEINE Indexe**, um den Performance-Unterschied zu zeigen. In Production wäre das ein kritischer Fehler!

Reflexion: Überlegen Sie sich – welche anderen Spalten in der IMDB-Datenbank würden von Indexen profitieren? Welche nicht?

Cleanup: Indexe für nächstes Experiment entfernen

```
1  -- Alle Indexe wieder löschen für saubere Ausgangslage
2  DROP INDEX IF EXISTS idx_tconst_basics;
3  DROP INDEX IF EXISTS idx_tconst_ratings;
4  DROP INDEX IF EXISTS idx_rating;
```

```
-- Alle Indexe wieder löschen für saubere Ausgangslage
DROP INDEX IF EXISTS idx_tconst_basics
```

ok

```
DROP INDEX IF EXISTS idx_tconst_ratings
```

ok

```
DROP INDEX IF EXISTS idx_rating
```

ok

Experiment 4: Partielle Indexe (Filtered Indexes)

Partial Indexes sind Indexe mit einer WHERE-Bedingung – sie indexieren nur einen Teil der Daten. Das spart Speicher, beschleunigt Writes und macht Queries auf diesem Subset extrem schnell!

Szenario: Moderne Film-Discovery-App

```
1  -- Typische Query: Nur moderne Filme (seit 2020)
2  SELECT primaryTitle, startYear, titleType
```

```
FROM title_basics
WHERE startYear >= '2020' AND titleType = 'movie'
ORDER BY startYear DESC;
```

```
-- Typische Query: Nur moderne Filme (seit 2020)
SELECT primaryTitle, startYear, titleType
FROM title_basics
WHERE startYear >= '2020' AND titleType = 'movie'
ORDER BY startYear DESC
```

relation "title_basics" does not exist

Diese App interessiert sich fast ausschließlich für neue Filme – alte Filme werden selten abgefragt. Ein normaler Index würde alle 178.124 Zeilen indexieren. Brauchen wir das wirklich?

Schritt 10: Datenverteilung analysieren

```
-- Wie viele Filme gibt es ab 2020?
SELECT
  'Moderne Filme (2020+)' as category,
  COUNT(*) as count,
  ROUND(100.0 * COUNT(*) / (SELECT COUNT(*) FROM title_basics), 1) as
    percent
FROM title_basics
WHERE startYear >= '2020'
UNION ALL
SELECT
  'Alle Filme' as category,
  COUNT(*) as count,
  100.0 as percent
FROM title_basics;
```

```
-- Wie viele Filme gibt es ab 2020?
SELECT
  'Moderne Filme (2020+)' as category,
  COUNT(*) as count,
  ROUND(100.0 * COUNT(*) / (SELECT COUNT(*) FROM title_basics), 1) as percent
FROM title_basics
WHERE startYear >= '2020'
UNION ALL
SELECT
  'Alle Filme' as category,
  COUNT(*) as count,
  100.0 as percent
FROM title_basics
```

relation "title_basics" does not exist

Sie sehen: Nur 42.396 Filme (24%) sind seit 2020. Warum sollten wir einen Index auf 100% der Daten bauen, wenn 76% irrelevant sind?

Schritt 11: Partial Index erstellen

```sql
-- Partial Index: NUR moderne Filme indexieren
CREATE INDEX idx_modern_films
ON title_basics(startYear, titleType)
WHERE startYear >= '2020';
```

```
-- Partial Index: NUR moderne Filme indexieren
CREATE INDEX idx_modern_films
ON title_basics(startYear, titleType)
WHERE startYear >= '2020'
```

relation "title_basics" does not exist

Dieser Index ist 76% kleiner als ein vollständiger Index – aber genauso schnell für Queries auf moderne Filme!

Schritt 12: Performance-Vergleich

```sql
-- Query mit Partial Index
EXPLAIN ANALYZE
SELECT primaryTitle, startYear
FROM title_basics
WHERE startYear >= '2020' AND titleType = 'movie'
ORDER BY startYear DESC
LIMIT 100;
```

```
-- Query mit Partial Index
EXPLAIN ANALYZE
SELECT primaryTitle, startYear
FROM title_basics
WHERE startYear >= '2020' AND titleType = 'movie'
ORDER BY startYear DESC
LIMIT 100
```

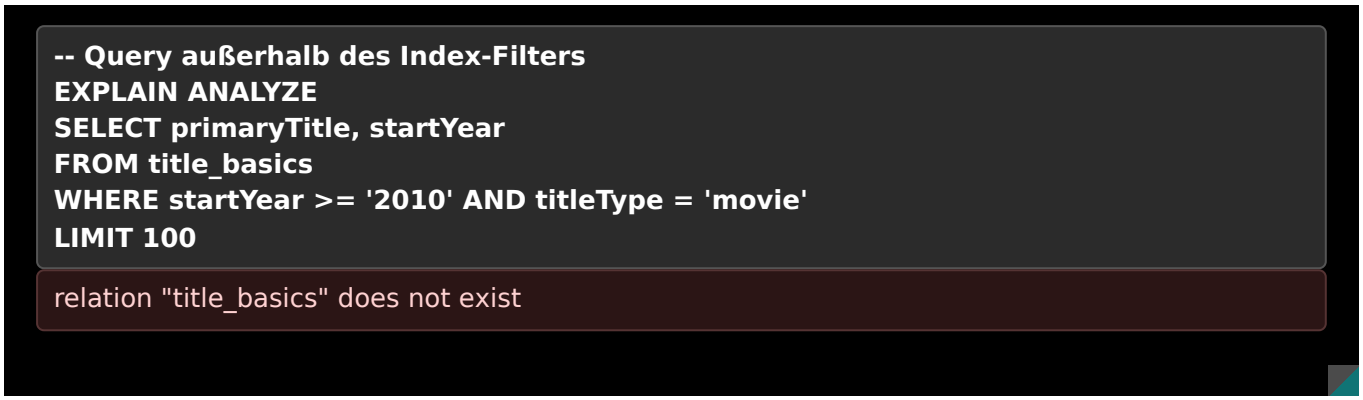
relation "title_basics" does not exist

Sie sollten „Index Scan using idx*modern*films“ sehen – der Partial Index wird genutzt! Weil die Query-Bedingung (`startYear >= '2020'`) die Index-Bedingung enthält.

Wann wird der Partial Index NICHT genutzt?

```sql
-- Query außerhalb des Index-Filters
```

```
EXPLAIN ANALYZE
SELECT primaryTitle, startYear
FROM title_basics
WHERE startYear >= '2010' AND titleType = 'movie'
LIMIT 100;
```



Jetzt sehen Sie „Seq Scan“ – der Partial Index wird ignoriert! Warum? Die Query fragt nach Filmen ab 2010, aber der Index hat nur Daten ab 2020. PostgreSQL kann ihn nicht nutzen.

Trade-offs: Wann sind Partial Indexes sinnvoll?

✅ **Verwenden bei:** - **Hot Data**: 80% der Queries greifen auf 20% der Daten zu (z.B. nur aktuelle Filme) - **Status-Filter**: Nur `status = 'active'` indexieren (oft 5-10% der Daten) - **Zeitbasierte Daten**: Nur letzte 2 Jahre (alte Daten selten relevant) - **Hohe Write-Last**: Weniger Index-Updates bei INSERTs/UPDATEs

❌ **NICHT verwenden bei:** - Queries über verschiedene Zeiträume (manchmal 2020+, manchmal 2010+) - Gleichmäßige Datenverteilung (keine Hot Spots) - Kleine Tabellen (< 10.000 Zeilen – Overhead nicht wert)

Reale Performance-Zahlen:

```
-- Normaler Composite Index
CREATE INDEX idx_all_films ON title_basics(startYear, titleType);
-- Größe: ~178.124 Einträge

-- Partial Index (unserer)
CREATE INDEX idx_modern_films ON title_basics(startYear, titleType)
WHERE startYear >= '2020';
-- Größe: ~42.396 Einträge (76% kleiner!)

-- Speedup bei INSERTs: ~20% schneller (weniger Index-Updates)
-- Speedup bei Queries auf 2020+: Gleich schnell wie normaler Index
-- Memory: 76% weniger RAM-Verbrauch
```

Cleanup: Index für saubere Ausgangslage entfernen

```
-- Index wieder löschen
DROP INDEX IF EXISTS idx_modern_films;
```

```
-- Index wieder löschen
DROP INDEX IF EXISTS idx_modern_films
```

ok

Best Practice für IMDB-App:

Wenn 90% der Queries moderne Filme abfragen, ist ein Partial Index optimal: - Schnellere Writes (weniger Index-Maintenance) - Kleinerer Index (passt besser in Cache) - Gleiche Query-Performance für relevante Daten

Trade-off: Queries auf alte Filme (< 2020) machen Sequential Scan – aber das ist selten!

Reflexion: Welche anderen Partial Indexes wären für IMDB sinnvoll? Z.B. nur Top-Ratings (`WHERE averageRating >= 8.0`) oder nur Serien (`WHERE titleType = 'tvSeries'`)?

EXPLAIN ANALYZE: Query Plans verstehen

Bisher haben wir mit `EXPLAIN ANALYZE` gearbeitet – das zeigt uns sowohl den Plan ALS AUCH die tatsächliche Ausführung mit echten Timings. Das ist besonders wertvoll in PostgreSQL!

Lassen Sie uns einen Query Plan Schritt für Schritt lesen. Das ist wie eine Landkarte für die Datenbank.

Anatomie eines Query Plans

```sql
EXPLAIN ANALYZE
SELECT tb.primarytitle, tr.averagerating
FROM title_basics tb
JOIN title_ratings tr ON tb.tconst = tr.tconst
WHERE tr.averagerating > 9.0 AND tb.startyear > '2010';
```

```
EXPLAIN ANALYZE
SELECT tb.primarytitle, tr.averagerating
FROM title_basics tb
JOIN title_ratings tr ON tb.tconst = tr.tconst
WHERE tr.averagerating > 9.0 AND tb.startyear > '2010'
```

relation "title_basics" does not exist

Ein typischer PostgreSQL-Plan sieht so aus:

```
Hash Join  (cost=X..Y rows=Z width=W) (actual time=...)
  Hash Cond: (tb.tconst = tr.tconst)
```

```
->  Seq Scan on title_basics tb  (cost=...)
      Filter: ((startyear)::text > '2010'::text)
->  Hash  (cost=...)
      ->  Index Scan using idx_rating on title_ratings tr  (cost=...)
            Index Cond: (averagerating > '9.0'::numeric)
```

Was bedeutet das? Die Datenbank arbeitet von innen nach außen:

1. **SEARCH tr USING INDEX idx_rating**
   - Start: Durchsuche `title_ratings` mit Index `idx_rating`
   - Filter: `averageRating > 9.0`
   - Ergebnis: Liste von `tconst`-Werten
2. **SEARCH tb USING INTEGER PRIMARY KEY**
   - Für jeden `tconst` aus Schritt 1:
   - Suche passende Zeile in `title_basics`
   - Nutze Primary Key (schneller Lookup)
3. **Impliziter Filter:**
   - Prüfe `startYear > '2010'`
   - Nur Zeilen, die beide Bedingungen erfüllen

Das ist ein effizienter Plan: Zwei Index-Lookups, kein Sequential Scan. Gut!

{{4}}

Scan-Typen verstehen

Scan-Typ (PostgreSQL)	Bedeutung	Performance
Seq Scan	Sequential Scan (jede Zeile)	🐌 Langsam
Index Scan	Index Scan (B-Baum)	🚀 Schnell
Index Only Scan	Alle Daten aus Index	🚀🚀 Ultraschnell
Bitmap Index Scan	Index + Bitmap für viele Zeilen	🚀 Mittelschnell
Hash Join / Merge Join	Effiziente JOIN-Strategien	🚀 Schnell

Ihr Ziel beim Optimieren: SCAN vermeiden, SEARCH maximieren!

{{5}}

Checkliste für Query-Plan-Analyse

Wenn Sie einen Query Plan sehen, fragen Sie sich:

- [ ] **Gibt es Seq Scans?** → Fehlende Indexe?
- [ ] **Werden die richtigen Indexe genutzt?** → Vergleich mit `pg_indexes`
- [ ] **Ist die Reihenfolge der JOINs sinnvoll?** → Kleinste Tabelle zuerst
- [ ] **Gibt es Subqueries, die vermeidbar wären?** → CTEs oder Joins nutzen
- [ ] **Sind Filter früh angewendet?** → WHERE vor JOIN
- [ ] **Sind die Kosten (cost) realistisch?** → ANALYZE regelmäßig ausführen

Mit dieser Checkliste können Sie selbst komplexe Queries debuggen und optimieren.

Wann Indexe NICHT helfen

Jetzt kommt der kritische Teil: Indexe sind kein Allheilmittel. Es gibt Situationen, in denen sie sogar schaden können. Lassen Sie uns vier typische Fälle analysieren.

Fall 1: Kleine Tabellen

Bei Tabellen mit weniger als 1000 Zeilen ist der Overhead eines Index oft größer als der Nutzen.

Beispiel:

```
-- Tabelle mit 100 Einträgen
SELECT * FROM users WHERE role = 'admin';
```

Ohne Index: 100 Zeilen lesen = ~~1ms. Mit Index: Index lesen + Zeilen lesen =~~ 1ms. Kein Unterschied – aber der Index kostet Speicher und verlangsamt INSERTs.

Faustregel:

- < 100 Zeilen: Nie Index (außer Primary Key)
- 100–1000 Zeilen: Nur bei sehr häufigen Queries
- 1000 Zeilen: Index meist sinnvoll

Fall 2: Hohe Write-Last

Jeder INSERT, UPDATE, DELETE muss alle Indexe aktualisieren. Bei schreibintensiven Anwendungen wird das zum Flaschenhals.

Szenario: Logging-Tabelle mit 10.000 Einträgen pro Sekunde

```
INSERT INTO access_logs (timestamp, user_id, endpoint)
VALUES (NOW(), 42, '/api/data');
```

Mit 5 Indexen muss die Datenbank bei jedem INSERT 5 B-Bäume aktualisieren – das kostet Performance.

Trade-off-Strategie:

1. **Option A:** Wenige Indexe (nur die wichtigsten)
2. **Option B:** Batch-Inserts ohne Index, später REINDEX
3. **Option C:** Partitionierung (z.B. nach Datum)

In Data Warehouses (viel Lesen, wenig Schreiben) sind 10+ Indexe normal. In OLTP-Systemen (viel Schreiben) sind 2–3 Indexe oft optimal.

Fall 3: Low Selectivity

„Selectivity“ bedeutet: Wie viele verschiedene Werte hat eine Spalte? Bei niedriger Selectivity (wenige Werte) bringt ein Index kaum etwas.

Beispiel: Geschlecht in einer Nutzertabelle

```
SELECT * FROM users WHERE gender = 'F';
```

Angenommen, 50% der Nutzer sind weiblich. Ein Index hilft hier nicht – die Datenbank müsste trotzdem die Hälfte aller Zeilen lesen!

Selectivity berechnen (Konzept):

```
1  -- Beispiel mit IMDB: Wie viele verschiedene titletype-Werte?
2  SELECT COUNT(DISTINCT titletype) FROM title_basics;
3
4  -- Wie viele Zeilen insgesamt?
5  SELECT COUNT(*) FROM title_basics;
6
7  -- Selectivity = DISTINCT values / Total rows
8  -- Wenn das Ergebnis < 5%, ist ein Index oft nicht sinnvoll
```

```
-- Beispiel mit IMDB: Wie viele verschiedene titletype-Werte?
SELECT COUNT(DISTINCT titletype) FROM title_basics
```

```
relation "title_basics" does not exist
```

Faustregel: Index nur, wenn Selectivity > 5%. Bei Gender (0.002%) ist ein Index verschwendet.

Hohe Selectivity = Index sinnvoll:

- E-Mail-Adressen (100% unique)
- IDs (100% unique)
- Namen (80%+ unique)

Niedrige Selectivity = Index nutzlos:

- Boolean-Felder (50% Selectivity)
- Status-Felder (z.B. active/inactive)
- Geschlecht (50% Selectivity)

Fall 4: Funktionen in WHERE-Klauseln

Wenn Sie in WHERE eine Funktion auf die Spalte anwenden, kann SQLite den Index oft nicht nutzen.

Beispiel:

```sql
-- Index wird NICHT genutzt:
SELECT * FROM title_basics
WHERE LOWER(primarytitle) = 'inception';

-- Index WIRD genutzt:
SELECT * FROM title_basics
WHERE primarytitle = 'Inception';
```

```
-- Index wird NICHT genutzt:
SELECT * FROM title_basics
WHERE LOWER(primarytitle) = 'inception'
```

```
relation "title_basics" does not exist
```

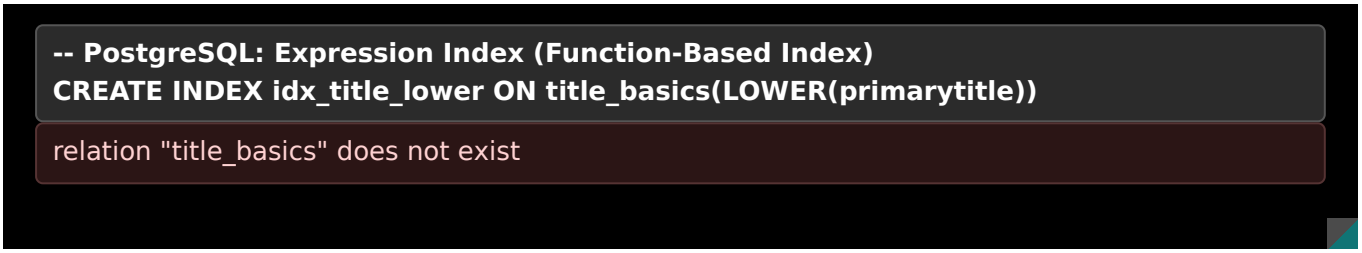
Warum? Der Index ist auf `primaryTitle` gebaut – aber `LOWER(primaryTitle)` ist ein anderer Wert. Die Datenbank müsste jeden Eintrag transformieren.

Lösungen:

1. **Funktion vermeiden:** Exakte Suche statt Case-Insensitive
2. **Computed Column:** Spalte mit `LOWER(primarytitle)` speichern + Index darauf
3. **Function-Based Index:** In PostgreSQL möglich! (Beispiel unten)

```sql
-- PostgreSQL: Expression Index (Function-Based Index)
CREATE INDEX idx_title_lower ON title_basics(LOWER(primarytitle));

-- Jetzt funktioniert die Query mit Index:
SELECT * FROM title_basics WHERE LOWER(primarytitle) = 'inception';
```



Weitere Funktionen, die Indexe „brechen“: `SUBSTRING()`, `CONCAT()`, `DATE()`, arithmetische Operationen (`salary * 1.1`).

Best Practice:

Indexe funktionieren am besten auf rohen Spaltenwerten.

- ✅ `WHERE startyear = '2020'`
- ❌ `WHERE CAST(startyear AS INTEGER) = 2020`
- ✅ `WHERE created_at > '2024-01-01'`
- ❌ `WHERE EXTRACT(YEAR FROM created_at) = 2024`
- ✅ (PostgreSQL) `CREATE INDEX ON table(EXTRACT(YEAR FROM created_at))`

Best Practices & Strategien

Sie haben jetzt gesehen, wie Indexe funktionieren – und wann sie scheitern. Lassen Sie uns das in actionable Strategien übersetzen.

1. Analysiere erst, optimiere dann

Der häufigste Fehler: „Blindly“ Indexe erstellen, ohne zu messen. Das führt zu Index-Bloat und verschlechtert die Performance.

Workflow:

1. **Profiling:** Welche Queries sind langsam? (> 100ms)
2. **EXPLAIN:** Query Plan analysieren – wo sind Sequential Scans?
3. **Index Candidate:** Welche WHERE/JOIN-Spalten werden gefiltert?
4. **Erstellen:** Index auf diese Spalten
5. **Messen:** Hat sich die Performance verbessert?
6. **Monitoring:** Index-Nutzung über Zeit beobachten

Viele Datenbanken bieten Tools, um ungenutzte Indexe zu finden. In PostgreSQL: `pg_stat_user_indexes`. In SQLite: Manuelle Analyse mit EXPLAIN.

```
-- PostgreSQL: Unused Indexes finden
-- (Hinweis: pg_stat_user_indexes ist in PGlite möglicherweise nicht verfüg
--  funktioniert aber in vollständigen PostgreSQL-Installationen)
SELECT schemaname, tablename, indexname, idx_scan
FROM pg_stat_user_indexes
WHERE idx_scan = 0 AND indexname NOT LIKE 'pg_toast%';
```

2. Index auf häufig gefilterte Spalten

Schauen Sie sich Ihre Top 10 langsamsten Queries an. Welche Spalten tauchen in WHERE, JOIN, ORDER BY auf?

Prioritätsliste:

Prio	Spalten-Typ	Beispiel
🔴 Hoch	Foreign Keys	`user_id`, `product_id`, `order_id`
🔴 Hoch	Häufige WHERE-Filter	`status`, `created_at`, `email`
🟡 Mittel	ORDER BY-Spalten	`created_at DESC`, `price ASC`
🟡 Mittel	GROUP BY-Spalten	`category`, `region`
🟢 Niedrig	Selten genutzte Spalten	`middle_name`, `favorite_color`

Ein einfacher Trick: Loggen Sie alle SQL-Queries über eine Woche und zählen Sie, welche Spalten am häufigsten gefiltert werden.

---

3. Composite Indexe richtig nutzen

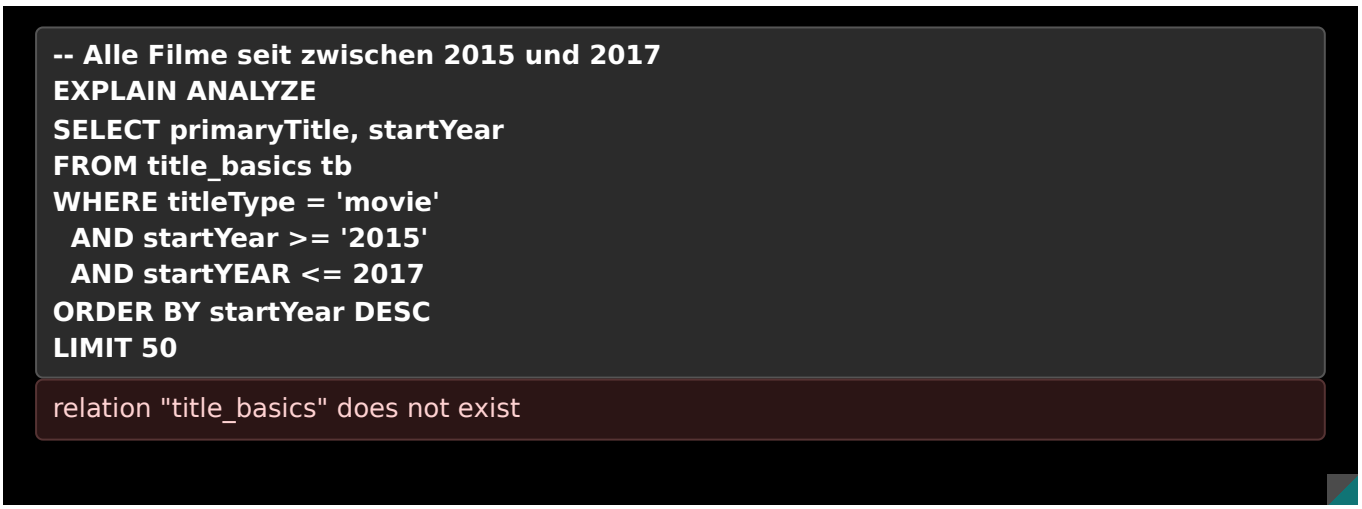
Die Reihenfolge der Spalten in einem Composite Index ist kritisch. Hier ist die Formel:

Reihenfolge-Regel:

1. **Equality-Filter zuerst:** `WHERE titleType = 'movie'`
2. **Range-Filter danach:** `WHERE startYear >= '2020'`
3. **ORDER BY zuletzt:** `ORDER BY startYear DESC`

Beispiel: Realistische IMDB-Query mit mehreren Filtern

```sql
-- Alle Filme seit zwischen 2015 und 2017
EXPLAIN ANALYZE
SELECT primaryTitle, startYear
FROM title_basics tb
WHERE titleType = 'movie'
  AND startYear >= '2015'
  AND startYEAR <= 2017
ORDER BY startYear DESC
LIMIT 50;
```



Diese Query findet 1.638 Top-Filme seit 2015. Ohne Index: Sequential Scan über 178.124 Zeilen. Welcher Index wäre optimal?

Optimaler Composite Index:

```sql
-- Equality (titleType) zuerst, Range (startYear) danach
CREATE INDEX idx_type_year
ON title_basics(titleType, startYear);
```

```
-- Equality (titleType) zuerst, Range (startYear) danach
CREATE INDEX idx_type_year
ON title_basics(titleType, startYear)
```

relation "title_basics" does not exist

Warum diese Reihenfolge? `titleType = 'movie'` ist eine Equality (=), `startYear >= '2015'` ist eine Range (>=). B-Bäume filtern erst exakt, dann im Bereich.

Test: Query mit optimalem Index

```
1  EXPLAIN ANALYZE
2  SELECT COUNT(*)
3  FROM title_basics
4  WHERE titleType = 'movie' AND startYear >= '2015';
```

```
EXPLAIN ANALYZE
SELECT COUNT(*)
FROM title_basics
WHERE titleType = 'movie' AND startYear >= '2015'
```

relation "title_basics" does not exist

Sie sehen vermutlich „Index Scan using idx_type_year“ oder „Bitmap Index Scan“ – der Index wird effizient genutzt!

Falscher Index: Range zuerst

```
1  DROP INDEX IF EXISTS idx_type_year;
2  -- FALSCH: Range (startYear) vor Equality (titleType)
3  CREATE INDEX idx_year_type_wrong
4  ON title_basics(startYear, titleType);
```

```
DROP INDEX IF EXISTS idx_type_year
```

ok

```
-- FALSCH: Range (startYear) vor Equality (titleType)
CREATE INDEX idx_year_type_wrong
ON title_basics(startYear, titleType)
```

relation "title_basics" does not exist

Mit diesem Index kann PostgreSQL nur `startYear` nutzen – `titleType` wird ignoriert, weil es nach der Range kommt. Bei 136 verschiedenen Jahren weniger effizient!

Vergleich: Welche Queries profitieren?

```
-- Index: (titleType, startYear)

-- ✅ NUTZT Index effizient:
WHERE titleType = 'movie' AND startYear >= '2015';

-- ✅ NUTZT Index teilweise (nur titleType):
WHERE titleType = 'movie';

-- ⚠️ NUTZT Index schlecht (nur startYear):
WHERE startYear >= '2015';

-- ❌ NUTZT Index NICHT:
WHERE startYear >= '2015' AND titleType = 'movie';  -- Reihenfolge egal!
```

Merke: Die WHERE-Reihenfolge im SQL ist egal – aber die Index-Spalten-Reihenfolge ist kritisch!

---

4. Redundante Indexe vermeiden

Ein Index auf `(A, B)` deckt auch Queries auf `A` allein ab. Aber nicht auf `B` allein!

Beispiel:

```
-- Gegeben: Index auf (startyear, titletype)

-- ✅ Index wird genutzt:
WHERE startyear = '2020';

-- ✅ Index wird genutzt:
WHERE startyear = '2020' AND titletype = 'movie';

-- ❌ Index wird NICHT effizient genutzt:
WHERE titletype = 'movie';
```

Das bedeutet: Sie brauchen keinen separaten Index auf `startYear`, wenn Sie bereits `(startYear, titleType)` haben.

Redundanz-Check:

- Index `(A)` + Index `(A, B)` → **Redundant!** Lösche `(A)`
- Index `(A, B)` + Index `(B, A)` → **Nicht redundant** (verschiedene Queries)
- Index `(A)` + Index `(B)` → **Nicht redundant** (verschiedene Spalten)

---

5. Monitoring & Wartung

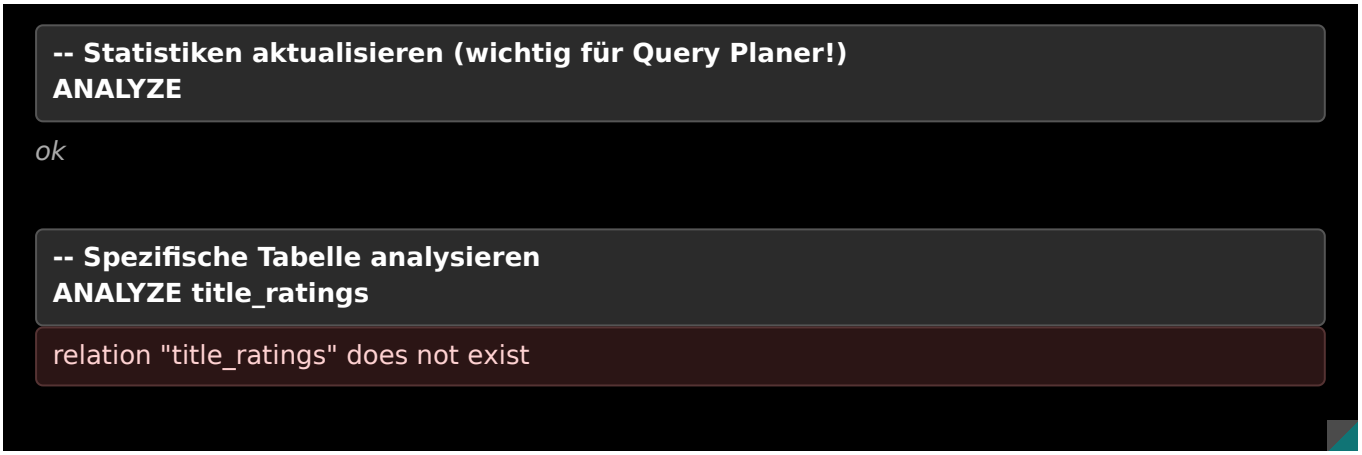
Indexe fragmentieren über Zeit – besonders bei vielen UPDATE/DELETE-Operationen. Regelmäßige Wartung ist nötig.

PostgreSQL/PGlite:

```
-- Statistiken aktualisieren (wichtig für Query Planer!)
ANALYZE;

-- Spezifische Tabelle analysieren
ANALYZE title_ratings;

-- Index neu aufbauen (selten nötig)
REINDEX INDEX idx_rating;
```



In PostgreSQL ist `ANALYZE` besonders wichtig – der Query Planer braucht aktuelle Statistiken, um die besten Indexe zu wählen!

Monitoring-Metriken:

- **Index Size:** Zu groß? Redundante Indexe?
- **Index Scans:** Wird der Index genutzt?
- **Sequential Scans:** Steigen sie an?
- **Write Performance:** Verlangsamen Indexe INSERTs?

Empfohlene Frequenz: VACUUM wöchentlich, REINDEX monatlich (oder bei Performance-Problemen).

Referenzen & Ressourcen

Pflichtlektüre

Hier sind weiterführende Ressourcen zum Thema Indexe und Performance:

- **Use The Index, Luke!** – https://use-the-index-luke.com → Bestes kostenloses Buch zu SQL-Indexen (auch als Print)
- **PostgreSQL: Using EXPLAIN** – https://www.postgresql.org/docs/current/using-explain.html → Query Plans verstehen (direkt für PGlite relevant!)
- **PostgreSQL Indexes** – https://www.postgresql.org/docs/current/indexes.html → Offizielle Dokumentation mit fortgeschrittenen Techniken

Weiterführende Ressourcen

- **B-Tree Visualisierung** – https://www.cs.usfca.edu/~galles/visualization/BTree.html → Interaktive Animation der Datenstruktur
- **DuckDB Performance Guide** – https://duckdb.org/docs/guides/performance/indexing → Moderne Ansätze (Column Store statt B-Tree)
- **Artikel: „When NOT to use an index“** – Stack Overflow → Diskussion zu Edge Cases
- **YouTube: „Database Indexing Explained“** – Hussein Nasser → Video-Tutorial (30 Min)

Tools für die Praxis

- **PGlite** – https://pglite.dev → PostgreSQL im Browser (was wir in dieser Session nutzen!)
- **DBeaver** – https://dbeaver.io → Universeller Datenbank-Client (mit EXPLAIN-Visualisierung)
- **pgAdmin** – https://www.pgadmin.org → PostgreSQL-spezifisch, gute Query-Analyse
- **EXPLAIN Visualizer** – https://explain.dalibo.com → PostgreSQL EXPLAIN Plans visualisieren

Das war Session 12! Sie haben jetzt die Werkzeuge, um jede PostgreSQL-Datenbank zu analysieren und zu optimieren – direkt im Browser mit PGlite. In der nächsten Session erweitern wir Ihr SQL-Arsenal mit Advanced Techniques – alles aufbauend auf dem, was Sie heute gelernt haben.

Happy Optimizing mit PGlite! 🚀📊