



**Hochschule  
Bonn-Rhein-Sieg**  
University of Applied Sciences

Projektarbeit

# **PostgresSQL - Rekursion auf Basis generischer Stored Procedures**

Fachbereich Informatik  
Referent: Prof. Dr. Harm Knolle

eingereicht von:  
Jennifer Wittling, Rolf Kimmelman, Jan Löffelsender

Sankt Augustin, den 09.12.2018

## Zusammenfassung

In den letzten Jahren haben Graphdatenbanken an Bedeutung gewonnen, da sich mit diesen bestimmte Fragestellungen besonders schnell lösen lassen. Graphdatenbanken haben den Vorteil, dass sich insbesondere Beziehungen zwischen Objekten gut abbilden und sehr performant abfragen lassen. Bei relationalen Datenbanken ist es zur Darstellung von Beziehungen zwischen Objekten erforderlich die verschiedenen Tabellen mittels des JOIN Operators zu verknüpfen. Diese Verknüpfungen können schnell zu einem großen Rechenaufwand und langen Laufzeiten führen. Es soll am Beispiel von Postgres untersucht werden, ob und wie sich Graphen in relationalen Datenbanken abbilden lassen. Weiterhin soll analysiert werden, ob und für welche Problemstellungen es sinnvoller ist Graphen in einer relationalen Datenbank statt einer Graphdatenbank abzubilden. Ist es zukünftig notwendig für die performante Verarbeitung steigender Datenmengen auf neue Technologien, wie Graphdatenbanken zu schwenken oder lassen sich die klassischen relationalen Datenbanken so erweitern, dass diese Problemstellungen ähnlich effizient lösen können.

Not only SQL (NoSQL) Datenbanken und insbesondere Graphdatenbanken sind im Gegensatz zu den relationalen Datenbanken flexibler und bei der Lösung bestimmter Probleme weniger rechen- und speicherintensiv. Insbesondere wenn es um die Auflösung von Beziehungen bzw. um die Traversierung über einen Graphen geht, bieten Graphdatenbanken Vorteile gegenüber den herkömmlichen relationalen Datenbanken. In der Praxis wurde jedoch auch die Beobachtung gemacht, dass durch die Verwendung von Stored Procedures die Traversierung über einen Graphen mittels einer relationalen Datenbank ähnlich schnell umgesetzt werden kann, wie mit einer Graphdatenbank.

Es soll das Modell als grundlegender technologische Aspekt von Graphdatenbanken kurz erläutert werden. Zielsetzung dieser Arbeit ist es einen Graphen in der relationalen Datenbank Postgres abzubilden und zu vergleichen, wie sich die Traversierung über diesen Graphen effizient umsetzen lässt. Zunächst soll die Umsetzung mittels klassischer SQL Operationen erfolgen. Anschließend sollen die Problemstellungen mittels Stored Procedures, sowie der Rekursion mittels PL/SQL gelöst werden. Die Ergebnisse der verschiedenen Vorgehensweisen sollen miteinander verglichen werden.

## Vorwort

Kapitel	schriftlich	Umsetzung	Vortrag erstellt	Vortrag gehalten
1	Wittling	-	Wittling	Wittling
2	Löffelsender, Wittling	-	-	-
3	Löffelsender, Wittling	Löffelsender	Löffelsender	Löffelsender
4	Kimmelman	Kimmelman	Kimmelman	Kimmelman
Systemadministration		Kimmelman	-	-

Tabelle 0.1: Aufgabenverteilung

Die packages

- amsthm,
- lstautogobble,
- multirow

wurden zusätzlich zur Standardvorlage verwendet.

# Inhaltsverzeichnis

<b>Zusammenfassung</b>	<b>II</b>
<b>Vorwort</b>	<b>III</b>
<b>1 Graph-Datenbanken - Grundlegende technologische Aspekte</b>	<b>1</b>
1.1 Modell . . . . .	1
1.1.1 Graph . . . . .	2
1.1.2 Bäume . . . . .	3
1.1.3 Property Graphen . . . . .	4
1.1.4 Hypergraphen . . . . .	5
1.1.5 Verschachtelte Graphen . . . . .	5
<b>2 Graph-Datenbanken und -Frameworks - Ausgewählte Systeme</b>	<b>7</b>
2.1 PostgreSQL . . . . .	7
2.1.1 Visitenkarte des Systems . . . . .	7
<b>3 Graph-Datenbanken im praktischen Einsatz: Online Transaction Processing (OLTP)</b>	<b>9</b>
3.1 PostgresSQL: OLTP . . . . .	9
3.1.1 Installation von Postgres . . . . .	9
3.1.2 CSV-Import . . . . .	10
3.1.3 Datenbankschema . . . . .	10
3.1.4 Erstellen von Fremdschlüsseln, Indexen und Partitionen . . . . .	10
3.1.5 Ausführungsplan . . . . .	11
3.1.6 Graphtraversierung mit Hilfe von Standard Structured Query Language (SQL) . . . . .	11
3.1.7 Graphtraversierung mit Hilfe von verschachteltem SELECT Statement . . . . .	13
3.1.8 Graphtraversierung mit Hilfe von rekursiven INNER JOIN . . . . .	14
3.1.9 Graphtraversierung mit Hilfe von selbstgeschriebenen Stored Procedure . . . . .	15
<b>4 Graph-Datenbanken im praktischen Einsatz: OLAP</b>	<b>16</b>
4.1 PostgreSQL: OLAP . . . . .	16
4.1.1 Indexe und Partionierte Tabellen . . . . .	16
4.1.2 Benchmark . . . . .	17
4.1.3 Interpretation der Ergebnisse . . . . .	36

<b>5 Fazit</b>	<b>37</b>
5.1 Fazit . . . . .	37
<b>A Anhang</b>	<b>38</b>
A.1 Graph-Datenbanken - Grundlegende technologische Aspekte . . . . .	38
A.2 Graph-Datenbanken und -Frameworks - Ausgewählte Systeme . . . . .	38
A.3 Graph-Datenbanken im praktischen Einsatz: OLTP . . . . .	38
A.4 Graph-Datenbanken im praktischen Einsatz: OLAP . . . . .	45
<b>Abbildungsverzeichnis</b>	<b>46</b>
<b>Tabellenverzeichnis</b>	<b>47</b>
<b>Listings</b>	<b>48</b>
<b>Abkürzungsverzeichnis</b>	<b>49</b>
<b>Literaturverzeichnis</b>	<b>51</b>
<b>Eidesstattliche Erklärung</b>	<b>54</b>

# 1 Graph-Datenbanken - Grundlegende technologische Aspekte

## 1.1 Modell

Allgemein betrachtet ist ein Modell eine vereinfachte bzw. abstrahierte Darstellung von realen Gegenständen, Sachverhalten oder Problemen. Durch die Modellierung soll die Realität auf die wichtigsten Einflussfaktoren reduziert werden [KS96]. In der Datenbankwelt beschreibt das Modell die Struktur der Daten, Operationen zum manipulieren der Daten und Integritätsbedingungen [Cod81]. Das derzeit am häufigsten verwendete Modell ist das relationale Datenbankmodell, bei dem die Daten in Tabellen gespeichert werden und in der Regel die standardisierte Abfragesprache SQL eingesetzt wird. Der Schwachpunkt des relationalen Datenbankmodells liegt jedoch bei der Verarbeitung von Daten mit einer hohen Anzahl an Beziehungen [VMZ<sup>+</sup>10].

Da eine effiziente Verarbeitung von großen vernetzten Datenmengen immer wichtiger wird, haben Graphenmodelle in den letzten Jahren im Datenbankbereich stark an Bedeutung gewonnen. Graph-Datenbanken nutzen Graphen als Datenbankmodell und greifen auf graphenspezifische Operationen zur effizienten Verarbeitung vernetzter Daten zurück [AG08]. Obwohl sich die verschiedenen Graphdatenbanken im allgemeinen nur minimal bei der Strukturierung der Daten unterscheiden, gibt es sehr verschiedene Ansätze bei den Abfragesprachen [AG18]. Dennoch sind die kleinen Unterschiede in der Datenmodellierung oft entscheidend und es können je nach Anwendungsfall verschiedene Modelle sinnvoll sein [Ang12]. Im folgenden sollen verschiedene Graphdatenbankmodelle, sowie Aspekte der Graphentheorie, die zur Modellierung von Graphdatenbanken von Bedeutung sind, kurz vorgestellt werden.

### 1.1.1 Graph

Die Grundlage aller Graphdatenbankmodelle bietet die Definition eines einfachen Graphen:

**Definition.** Ein Graph  $G = (V, E, \gamma)$  ist ein Tripel bestehend aus:

- $V$ , einer nicht leeren, ungeordneten Menge von Knoten (vertices)
- $E$ , einer Menge von Kanten (edges)
- $\gamma$ , einer Inzidenzabbildung (incidence relation), mit  
 $\gamma : E \longrightarrow \{X \mid X \subseteq V, 1 \leq |X| \leq 2\}$

[Bec18, Seite 21]

Ein Knoten repräsentiert ein Element in einem Graphen und die Kanten stellen die Beziehung zwischen den einzelnen Knoten her. Bei einfachen Graphen können die Kanten nur die Kardinalität  $1 \leq |X| \leq 2$  haben. Als Kardinalität wird die Anzahl Knoten bezeichnet, die durch eine Kante in Beziehung gesetzt werden kann. Zwei Knoten heißen adjazent, wenn diese über eine Kante direkt miteinander verbunden sind. Eine Kante die mit einem Knoten verbunden ist wird als inzident zu diesem Knoten bezeichnet [KK15].

Graphen können gerichtet oder ungerichtet sein. Gerichtete Graphen zeichnen sich dadurch aus, dass die Kanten eine zugewiesene Richtung besitzen. Grafisch werden gerichtete Kanten in der Regel durch Pfeile dargestellt. Für die Modellierung von realen Gegebenheiten ist das Konzept der gerichteten Graphen sehr entscheidend, da dieses die Darstellung einseitiger Beziehungen zwischen den Entitäten des Modells erlaubt.

Um die Beziehung zwischen zwei Knoten genauer zu definieren, lassen sich die Kanten gewichten. Dabei werden den Kanten in der Regel numerische Werte zugeordnet und man bezeichnet diese Graphen als Gewichtete Graphen. Durch die Wichtung von Kanten lassen sich beispielsweise Kosten oder Distanzen zwischen den Entitäten definieren.

Ein Knoten ist isoliert, wenn er keine inzidenten Kanten und somit keine direkten Nachbarn hat [KK15]. Ein ungerichteter Graph heißt zusammenhängend, falls es zwischen zwei beliebigen Knoten  $a$  und  $b$  aus  $V$  einen ungerichteten Weg mit  $a$  als Startknoten und  $b$  als Endknoten gibt [KN12, 36-38]. Hat eine Kante als Start- und Endknoten den selben Knoten, verbindet also den Knoten mit sich selber, spricht man von einer Schlinge. Liegen zwischen zwei Knoten eines Graphen mehr als eine Kante, nennt man diese Multikante. Enthält ein Graph Multikanten und Schlingen ist dies kein einfacher Graph mehr sondern ein Multigraph [SF05].

Der Grad eines Knoten bezeichnet die Anzahl der inzidenten Kanten des Knoten. Dabei werden Schleifen doppelt gezählt [Rah17b]. Ein Graph, bei dem alle Knoten

den selben Knotengrad haben, wird als regulärer Graph bezeichnet [Fel03]. Sind bei einem Graphen alle Knoten mit allen übrigen Knoten verbunden spricht man von einem vollständigen Graphen:

$$K_n = ([n], \binom{[n]}{2})$$

Da bei einem Graphen nur die Struktur definiert ist, also welcher Knoten über welche Kante mit den anderen Knoten verbunden ist, können Graphen auf unterschiedliche weise gezeichnet werden und trotzdem gleich sein. Sind zwei Graphen gleich, bezeichnet man diese als isomorph [Rah17a, Seite 22].

### 1.1.2 Bäume

Ist ein Graph kreisfrei, es gibt keinen Weg bei dem der Start- gleich dem Endknoten ist, spricht man von einem Wald. Sind die Knoten eines Waldes zusammenhängend entsteht ein Baum. Knoten mit dem Grad  $n = 1$  werden als Blätter bezeichnet. Bäume können gerichtet und ungerichtet sein. Im Falle von gerichteten Bäumen spricht man auch von gewurzelten Bäumen, da der Ursprungsknoten als Wurzel bezeichnet wird. Abbildung 1.1.2 zeigt einen gewurzelten Baum, die Blätter sind hier grün dargestellt und die Wurzel rot. In einem Baum gibt es zwischen zwei beliebigen Knoten immer nur einen Weg [Rah17a].

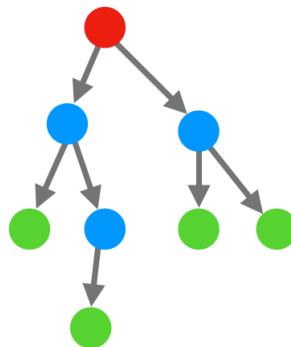


Abbildung 1.1: gewurzelter Baum

Bäume sind Grundlage unter anderen des hierarchischen Datenbankmodells. Dieses Datenbankmodell hat jedoch den Nachteil, dass bedingt durch die Kreisfreiheit eines Baumes keine n:n Beziehungen modelliert werden können. Das Netzwerkdatenbankmodell versucht dieses Problem zu lösen.



### 1.1.3 Property Graphen

Propertygraphen erweitern das Modell des einfachen Graphen. Property Graphen sind gerichtete Graphen, die sich durch ihre den Kanten und Knoten zugewiesenen Eigenschaften (Properties) auszeichnen. Gespeichert werden diese Eigenschaften als Key-Value-Paare. Das Hinzufügen von Attributen an einen Knoten oder eine Kante soll zusammengehörige Daten schneller abrufbar machen [Ang12]. Label ermöglichen die Unterteilung von Knoten und Kanten in verschiedene Knoten- und Kantentypen. Attribute, Label und die Richtung der Kanten erlauben eine sehr detaillierte Modellierung von realen Sachverhalten. Somit sind Property Graphen von sehr großer Bedeutung für die Modellierung von Graphdatenbanken.

Abbildung 1.1.3 zeigt einen Property Graphen. Die Knoten sind den drei Labels Person, Unternehmen und Stadt zugeordnet. Die gerichteten Kanten stellen die Beziehungsverhältnisse zwischen den einzelnen Knoten her und können durch Attribute, wie beispielsweise der Information über die Dauer der bisherigen Beziehung, genauer definiert werden.

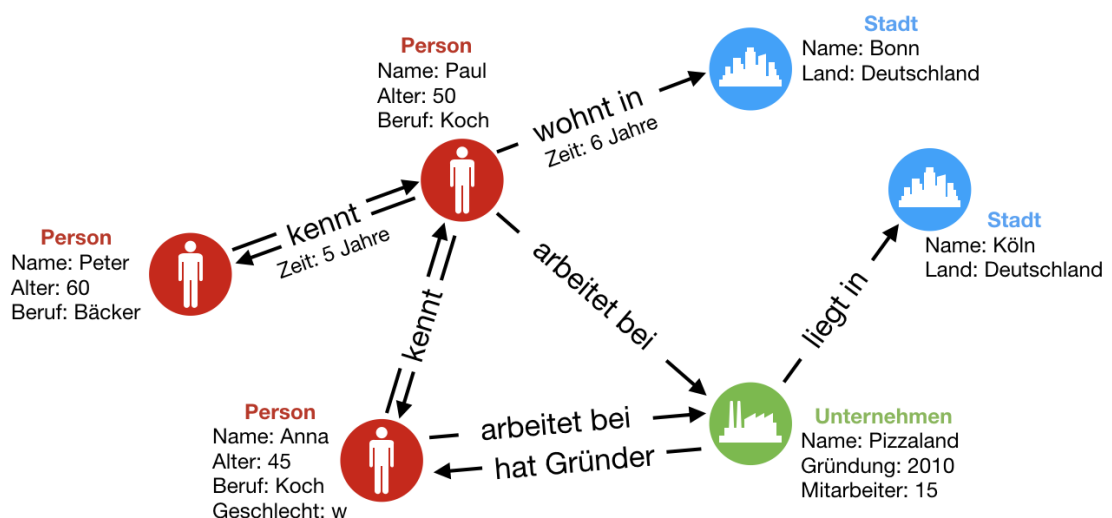


Abbildung 1.2: Property Graph

Der Vorteil von Property Graphen ist, dass diese eine sehr detaillierte Modellierung der Daten ermöglichen. Nachteilig ist jedoch, dass eine komplexere Datenstruktur zu einer komplizierteren Realisierung führt. Derzeit sind Propertygraphen das am häufigsten verwendete Datenmodell für Graphdatenbanken.

### 1.1.4 Hypergraphen

Hypergraphen stellen eine Generalisierung von Graphen dar und ermöglichen die Modellierung komplexer Beziehungen [AG18]. Im Vergleich zum normalen Graphen können die Kanten eines Hypergraphen eine beliebige Kardinalität haben. Die Hyperedges in einem Hypergraphen sind somit eine beliebige Menge von Knoten. Im Falle eines gerichteten Hypergraphen verbindet die Hyperkante den Ausgangsknoten direkt mit allen Zielknoten.

Abbildung 1.1.4 zeigt einen Hypergraphen. Die Kante  $e_4$  verbindet in diesem Graphen die Knoten  $v_5$ ,  $v_6$  und  $v_7$  miteinander.

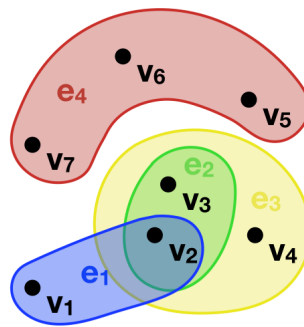


Abbildung 1.3: Hypergraph

Hypergraphen ermöglichen die direkte Darstellung rekursiver Beziehungen [Ior10]. Da die Hypergraphen eine flexiblere Struktur als das einfache Graphen-Modell bietet, werden diese oft zur Modellierung in Graphdatenbanken verwendet [Goe15].

### 1.1.5 Verschachtelte Graphen

In einem verschachtelten Graphen kann ein Knoten wiederum ein Graph sein. Diese Knoten werden als Hypernodes bezeichnet. Das Konzept von Hypernodes erlaubt eine beliebig große Komplexität des Modells. Das Hypernode Modell nutzt gerichtete Kanten. Jeder Knoten des verschachtelten Graphen bekommt ein eindeutiges Label. Um die Knoten in verschiedene Kategorien zu unterteilen werden Tags vergeben [PL94].

Abbildung 1.1.5 stellt einen Teil des in 1.1.3 dargestellten Propertygraphen als Hypernodes dar. Alle Knoten haben ein eindeutiges Label und es wurden zwei Tags vergeben - Unternehmen und Person. Durch gerichtete Kanten werden die Beziehungen zwischen den einzelnen Hypernodes hergestellt. Die Attribute, die im Pro-

Hypergraphen als Key-Value Paare gespeichert werden, werden im verschachtelten Graphen als Knoten einer Hypernode gespeichert.

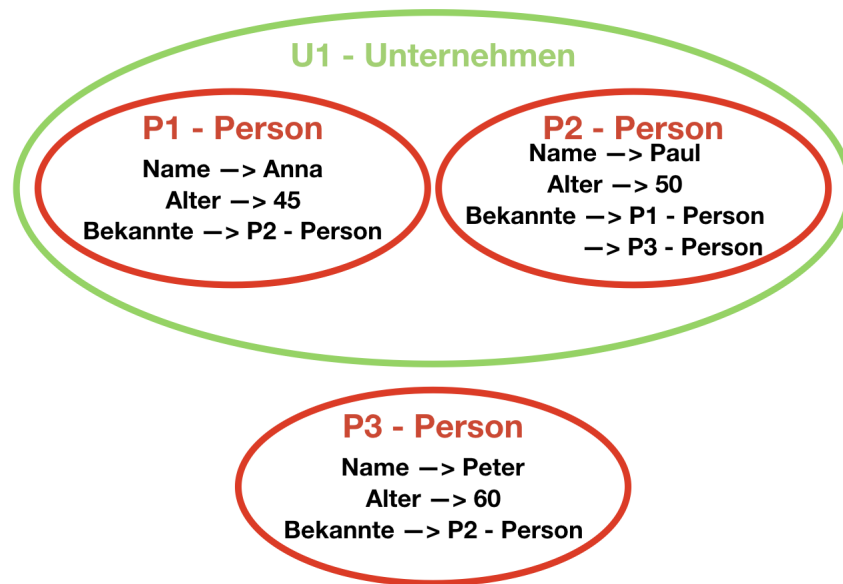


Abbildung 1.4: Hypernode

Hypergraphen lassen sich durch Hypernodes darstellen, indem man die im Hypergraph durch eine Hyperedge verbundenen Knoten in einer Hypernode modelliert. Umgekehrt lassen sich Hypernodes jedoch nicht unbedingt in einem Hypergraphen realisieren [PL94].

## 2 Graph-Datenbanken und -Frameworks - Ausgewählte Systeme

### 2.1 PostgreSQL

#### 2.1.1 Visitenkarte des Systems

- Allgemein
  - Name: PostgreSQL, umgangssprachlich Postgres
  - Kategorie / Modell: PostgreSQL ist ein Relationales Datenbank System
  - Version: 11.1
  - Historie: PostgreSQL ist aus dem POSTGRES Projekt der University of California at Berkeley entstanden, welches unter der Leitung von Professor Michael Stonebraker im Jahre 1986 began.
  - Hersteller: PostgreSQL Global Development Group
  - Lizenz: Open-Source
  - Referenzen / Quellenangaben: [Fro18], [Pos18], [Groa], [Eis03]
- Architektur
  - Programmiersprache: C
  - Systemarchitektur: Objektrelationales Datenbankmanagementsystem
  - Betriebsart: Stand Alone, Cluster Betrieb für Replikation der Datenbank
  - Application Programming Interface (API): u.A. libpq, psycopg, psqLODBC, pq, pgtnlg, Npgsql, node-postgres

- Datenmodell
  - Standardsprache: PL/pgSQL
  - Sichten (Views): Ja
  - Externe Dateien (BLOBs): Ja
  - Schlüssel: Ja
  - Semantische unterschiedliche Beziehungen: Ja
  - Constraints: Ja
- Indexe
  - Sekundärindexe: Ja
  - Gespeicherte Prozeduren: Ja
  - Triggermechanismen: Ja, Prozeduren , die als Trigger aufgerufen werden
  - Versionierung: Ja, Versionierung mit Hilfe von Transaktions-ID(XID)
- Abfragemethoden
  - Kommunikation: SQL über Transmission Control Protocol (TCP)/Internet Protocol (IP)
  - Create, Read, Update, Delete (CRUD)-Operationen: Ja
  - Ad-hoc-Anfragen: Ja
- Konsistenz
  - Atomicity, Consistency, Isolation, Durability (ACID), besonders Multiversion Concurrency Control Modell (MVCC)
  - Transaktionen: Ja
  - Nebenläufigkeit (Synchronisation): Ja
- Administration
  - Werkzeuge: pgAdmin, dataGrip, diverse Erweiterungen
  - Massendatenimport: Ja
  - Datensicherung: Ja

# 3 Graph-Datenbanken im praktischen Einsatz: OLTP

## 3.1 PostgreSQL: OLTP

Die Implementierung von OLTP Anwendungsfällen ist eine klassische Aufgabe für relationale Datenbanken. Da statt einer Graphdatenbank eine relationale Datenbank verwendet wurde, ist die Implementierung des OLTP Anwendungsfalls (Gästebuch) uninteressant. Interessant ist jedoch die Umsetzung der Traversierung von Graphen in relationalen Datenbanken. Für die Graphtraversierung wurden eigene Scripte geschrieben, die in diesem Kapitel vorgestellt werden. Da Graphdatenbanken für das Traversieren von Graphen entwickelt worden sind, sollten diese bei der Traversierung einen Performancegewinn gegenüber objektrelationalen Datenbanken haben. Ziel ist es mit Hilfe einer objektrelationalen Datenbank eine mit den Graphdatenbanken vergleichbar performante Abfrage eines Graphen zu implementieren. Für die Graphtraversierung sind die folgenden 5 Methoden vorgesehen.

Graphtraversierung mit Hilfe von:

- Rekursiven Common Table Expression (CTE)
- Verschachteltem SELECT Statement
- Rekursiven INNER JOIN
- Selbstgeschriebenen Stored Procedure
- Dynamisch generiertem SQL

### 3.1.1 Installation von Postgres

PostgreSQL kann unter Ubuntu über die Paketverwaltung Advanced Package Tool (APT) installiert werden. Weiterhin wird eine Installation über die Red Hat Package Manager (RPM)-Paketverwaltung angeboten. Im Rahmen dieser Arbeit wird PostgreSQL Version 11 verwendet. Ein Parallelbetrieb verschiedener PostgreSQL Versionen ist möglich. Nach der Installation von PostgreSQL muss zunächst der Befehl *initdb* ausgeführt werden. Über *initdb* wird ein PostgreSQL-Cluster angelegt.

Als Parameter kann ein Directory-Pfad angegeben werden. In diesem Pfad wird der PostgreSQL-Cluster von *initdb* angelegt. Gemäß der Vorgaben dieser Arbeit wurde das PostgreSQL-Cluster unter */data/team22/postgresql/11/main* installiert.

### 3.1.2 CSV-Import

Beim Import von (CSV)-Dateien wird zwischen Import vom Clientsystem und Import vom Serversystem unterschieden. Für den Import vom Client wird das `psql`-Statement `\copy` verwendet (siehe SQL Script A.1). `\copy` liest Informationen aus einer Datei, die vom `psql`-Client aus erreichbar sein muss. [Pos18]

### 3.1.3 Datenbankschema

Für die Performancemessung sind 5 Graphen vorgesehen. Ein Graph besteht aus einer *profiles* Tabelle und einer *relation* Tabelle. Die beiden Tabelle werden mit Hilfe der Spalte *ID* aus der jeweiligen *profiles* Tabelle verknüpft. Die Spalten *src* und *dst* aus der *relation* Tabelle sind Fremdschlüssel, sie verweisen auf die Spalte *ID* in der *profiles* Tabelle. *ID* hingegen ist in der *profiles* Tabelle ein Primärschlüssel (siehe A.3).

### 3.1.4 Erstellen von Fremdschlüsseln, Indexen und Partitionen

Um die *profile* Tabelle und die *relation* Tabelle zu verknüpfen, wurde zwischen den beiden Tabellen ein Fremdschlüssel erstellt. Bei der Erstellung wurde die *profile* Tabelle mit einem Zähler versehen. Hierbei wurde der `postgres` Befehl `serial` verwendet, der einen Zähler für jede Zeile der Tabelle erstellt und bei 1 startet. Damit die Tabellen *relation* und *profile* mit Hilfe eines Fremdschlüssels verknüpft werden können, muss der Zähler innerhalb der *profile* Tabelle jedoch bei 0 starten. Der Grund dafür ist, dass innerhalb der *relation* Tabelle die *src* und die *dst* Spalte bei 0 anfangen - somit auf ein Profil verweisen, was die *ID* 0 hat. Hierfür wurde für die *relation* Tabelle ein SQL Script geschrieben, was die Daten zuerst in eine temporäre Tabelle schreibt, von jeder Zeile innerhalb der Spalte *ID* 1 subtrahiert und anschließend die Werte aus der temporären Tabelle in die endgültige *profile* Tabelle schreibt (siehe hierzu auch beispielhaft das Script für die Tabelle *facebook-profiles* (siehe A.4). Für die einzelnen *relation* Tabellen wurden ebenfalls Indexe erstellt, damit die Performance verbessert wird. Die Messung der Performance wurde durchgeführt auf den *relation* Tabellen ohne Index, auf den *relation* Tabellen mit Index und auf den partitionierten Tabellen mit Index. Partitionierung bezeichnet das Aufteilen einer großen Tabelle in mehrere kleine Tabellen [Groc]. Die Motivation hinter der Erstellung von Indexen und der Partitionierung von Tabellen besteht darin, einen Performancegewinn bei der Graphtraversierung zu erzielen. Dies wird genauer im 4. Kapitel diskutiert.

### 3.1.5 Ausführungsplan

Zur Erklärung der Scripts wurde in diesem Kapitel der Ausführungsplan verwendet. Postgres erstellt für jede Query einen Ausführungsplan, bestehend aus n Knoten, der mit Hilfe des Kommandos EXPLAIN angezeigt werden kann. Eine Zeile im Ausführungsplan sieht folgendermaßen aus:

Listing 3.1: Zeile im Ausführungsplan

```
-> Seq Scan on relation_wiki_vote relation_wiki_vote_1 (cost
    =10000000000.00..10000001649.62 rows=100762 width=8) (actual time=0.002..4.758
    rows=100762 loops=1)
```

Diese Zeile entspricht einem Knoten und läßt sich folgendermaßen: Zuerst wird die Operation angegeben. In diesem Fall ein Sequential Scan <sup>1</sup>. Anschließend wird die Tabelle angegeben, diese wird jedoch nicht bei jeder Operation genannt. Innerhalb der ersten Klammern wird die geschätzte Zeit zum Starten und zum Ausführen des Knoten angegeben, die geschätzte Anzahl an Zeilen, die dieser Knoten zurückliefert und die geschätzte durchschnittliche Breite in bytes der zurückgegebenen Zeilen. Innerhalb der zweiten Klammern wird die tatsächliche Zeit angegeben [Grob].

### 3.1.6 Graphtraversierung mit Hilfe von Standard SQL

Bei der Graphtraversierung mit Hilfe von Standard SQL wird der Befehl WITH RECURSIVE und UNION verwendet. Im Anhang befindet sich ein SQL Script, was die Tabelle relation\_facebook mit Hilfe des WITH RECURSIVE Befehl bis zur Rekursionsstufe 5 traversiert (siehe A.20), sowie ein Script, welches den Standard SQL Source Code dynamisch generiert (siehe A.19). Der WITH Befehl erstellt eine temporäre Tabelle, die nur für die angegebene Query existiert. Diese werden oft auch als CTE bezeichnet. Der SQL Befehl RECURSIVE sorgt dafür, dass die Abfrage sich mit der Ergebnismenge wieder selber aufruft. Die Struktur einer RECURSIVE Query sieht folgendermaßen aus: Zuerst wird der **nicht rekursive Teil** der Query angegeben, anschließend wird der **rekursive Teil** beschrieben:

Listing 3.2: Rekursiver und nicht rekursiver Teil

```
WITH RECURSIVE graphtraverse(src, dst, lvl) AS(
    SELECT src ,dst, 1 as lvl FROM public.relation_facebook WHERE src =765
    UNION
    SELECT p1.src,p1.dst,p.lvl+1 as lvl FROM graphtraverse p, relation_facebook p1 WHERE p1.src IN
        ( p.dst ) and lvl<5
) SELECT DISTINCT(dst) FROM graphtraverse order by dst;
```

Die Rekursion operiert hierzu auf 2 temporären Tabellen . Der Working Tabelle und der Intermediate Tabelle. Ein Rekursionsschritt sieht folgendermaßen aus: Die Ergebnisse innerhalb einer Rekursionsstufe werden in die Working Tabelle geschrieben, es wird mit Hilfe des UNION Operators überprüft ob Duplikate vorhanden sind. Die Überprüfung erfolgt bezogen auf eine Rekursionsstufe. Duplikate werden

<sup>1</sup>Der Sequential Scan liest alle Datenblöcke sequenziell [Fro18, S.211]



gelöscht, die Ergebnismenge wird in die Intermediate Tabelle geschrieben, die Ergebnisse aus der Intermediate Tabelle werden in die Working Tabelle kopiert, die Intermediate Tabelle wird gelöscht [Grod]. Die Abbruchbedingung für die Rekursion wird innerhalb der Funktion definiert. Ob die Abbruchbedingung erreicht ist, wird im rekursiven Teil der Funktion **überprüft**:

Listing 3.3: Überprüfen der Abbruchbedingung

```
SELECT p1.src,p1.dst,p.lvl+1 as lvl FROM graphtraverse p, relation$_$facebook p1
WHERE p1.src IN ( p.dst ) and lvl<5
```

Der Ausführungsplan für die Tabelle relation\_ facebook für den WITH RECURSIVE Operator befindet sich im Anhang (siehe A.21)). Das Verknüpfen der zurückgegebenen Ergebnismenge von der graphtraverse Funktion mit der relation\_ facebook Tabelle erfolgt in einem Nested Loop Join.<sup>2</sup> Die Gleichheitsüberprüfung (src = p.dst) wird mit Hilfe eines Index Scans<sup>3</sup> gemacht, die Überprüfung ob das Level erreicht worden ist erfolgt mit Hilfe eines Scans über die Working Table. :

Listing 3.4: Überprüfung der WHERE Bedingung

```
-> Nested Loop (cost=0.29..1434.82 rows=2111 width=12) (actual time=0.018..2.174
rows=8173 loops=5)
-> WorkTable Scan on graphtraverse p (cost=0.00..5.17 rows=77 width=8) (
actual time=0.016..0.065 rows=729 loops=5)
Filter: (lvl < 5)
Rows Removed by Filter: 482
-> Index Scan using indexsrc on relation_facebook p1 (cost=0.29..18.23 rows
=27 width=8) (actual time=0.001..0.002 rows=11 loops=3645)
Index Cond: (src = p.dst)
```

Der RECURSIVE Operator und der UNION Operator finden sich in der Zeile Recursive Union wieder. Dieser Teil der Ausführung dauert am längsten (16,071 milli Sekunden).

Listing 3.5: Überprüfung der WHERE Bedingung

```
-> Recursive Union (cost=0.29..14814.87 rows=21133 width=12) (actual time
=0.014..16.085 rows=6056 loops=1)
```

Der Aufruf der graphtraverse Funktion:

Listing 3.6: Aufruf der graphtraverse Funktion

```
SELECT DISTINCT(dst) FROM graphtraverse order by dst
```

findet sich in der letzten Zeile des Ausführungsplan:

Listing 3.7: Aufruf der graphtraverse Funktion im Ausführungsplan

```
-> CTE Scan on graphtraverse (cost=0.00..422.66 rows=21133 width=4) (actual time
=0.015..16.921 rows=6056 loops=1)
```

<sup>2</sup>Ein Nested Loop Join besteht aus 2 Tabellen über die iteriert wird. Die erste Tabelle (Outer Table) wird einmal durchlaufen. Die zweite Tabelle (Inner Table) wird mehrfach durchlaufen. Der Vergleich wird über eine Indexsuche gebildet [Fro18, Seite 213]

<sup>3</sup>Bei einem Index-Scan ist die Index-Zugriffsmethode dafür verantwortlich, die Tuple Identifier (TID)s aller Tupel, von denen ihr mitgeteilt wurde, dass sie mit den Scan-Schlüsseln übereinstimmen, wiederherzustellen.

Die DISTINCT Funktion wird in ein Hash Aggregate übersetzt:

Listing 3.8: Aufruf der DISTINCT Funktion

```
-> HashAggregate (cost=475.49..477.49 rows=200 width=4) (actual time=17.694..17.716
      rows=321 loops=1)
```

### 3.1.7 Graphtraversierung mit Hilfe von verschachteltem SELECT Statement

Bei der Graphtraversierung mit Hilfe von verschachteltem SQL wird ein selbsterstelltes verschachteltest SELECT Statement verwendet. Ein Beispielstatement befindet sich im Anhang (siehe A.16). Auf der obersten Rekursionsstufe wird der Startknoten des Graphen mitgegeben (in diesem Beispiel ist der Startknoten = 1). Das Ergebnis dieser Abfrage wird als Eingabe für die nächst tiefere Rekursionsstufe verwendet. In der WHERE Klausel wird für die Spalte src der IN Operator verwendet. Der IN Operator erlaubt es, mehrere Werte innerhalb der WHERE Klausel anzugeben. Das DISTINCT in der SELECT Klausel sorgt dafür, dass Duplikate in der Ergebnismenge der momentanen Rekursionsstufe entfernt werden. Die Funktionsweise von DISTINCT ist in der folgenden Grafik nochmal dargestellt:

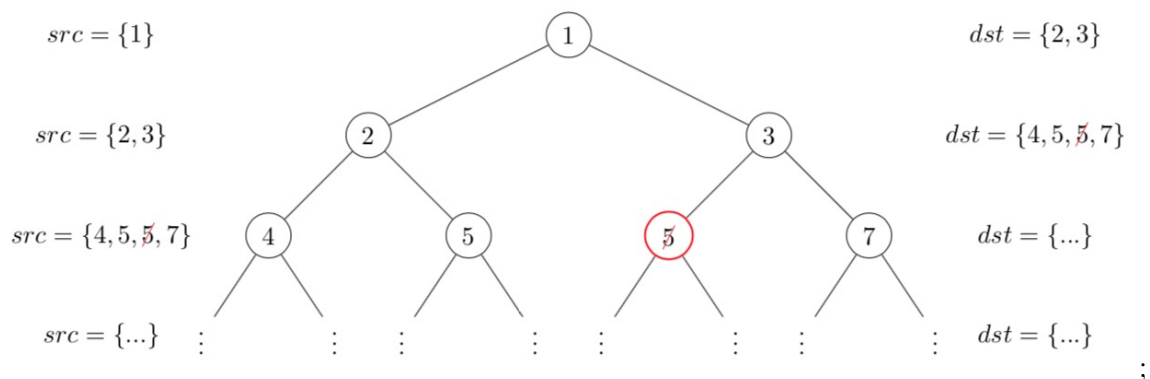


Abbildung 3.1: Löschen von Duplikaten in einer Rekursionsstufe

Hierbei liegt der Knoten 5 so, dass er in der 2. Rekursionsstufe 2 Mal in der Auswahl auftaucht. DISTINCT entfernt das Duplikat. Die Ergebnismenge, entfernt um die Duplikate, wird als Input für die nächste Rekursionsstufe verwendet. Die Ausgabe des verschachtelten SELECT Statement sind die Nachbarn der Knoten, der angegebenen Rekursionstiefe. Wird zum Beispiel ein verschachteltes SELECT Statement der Tiefe 3 erstellt, so gibt dieses Statement alle Nachbarn 3. Grades ausgehend vom Startknoten an. Der Nachteil bei dieser Methode ist, dass Kreise in einem Graph nicht erkannt werden. Die Duplikatüberprüfung erfolgt nicht über mehrere Rekursionsstufen hinweg, sondern immer nur zwischen zwei Rekursionsstufen. Der Ausführungsplan für ein verschachteltes SELECT der Tiefe 5 befindet sich im Anhang (siehe A.22). Für die DISTINCT Überprüfung wird auf jeder Rekursionsebene

ein HashAggregate herangezogen. Die Zugehörigkeitsüberprüfung zu IN Klausel in der WHERE Bedingung:

Listing 3.9: IN Klausel

```
SELECT DISTINCT(dst) FROM relation_facebook WHERE src IN ()
```

erfolgt auf allen Rekursionsstufen, ausgenommen der ersten Rekursionsstufe, mit Hilfe eines Hash Join. Bevor der Hash Join durchgeführt wird, wird zuerst der Hash gebildet und anschließend ein Sequential Scan durchgeführt:

Listing 3.10: Aufruf der DISTINCT Funktion

```
-> Hash Join (cost=2713.40..4389.64 rows=88234 width=4) (actual time=11.821..17.797
      rows=1709 loops=1)
-> Seq Scan on relation_wiki_vote relation_wiki_vote_1 (cost
      =10000000000.00..10000001649.62 rows=100762 width=8) (actual time=0.004..4.822
      rows=100762 loops=1)
-> Hash (cost=60.87..60.87 rows=19 width=4) (actual time=0.022..0.022 rows=4
      loops=1)
```

Auf der ersten Rekursionsstufe erfolgt die Überprüfung der IN Klausel mit Hilfe eines Nested Join.

Bei diesem Ausführungsplan wird der Index Scan nur auf der ersten Rekursionsstufe verwendet. Auf den tieferen Rekursionsstufen wird immer ein Sequential Scan vollzogen. Der Grund dafür ist, dass nur auf der ersten Rekursionsstufe auf einer Tabelle operiert wird auf der ein Index erstellt worden ist. In den tieferen Rekursionsstufen werden mit Hilfe der IN Klausel Zwischentabellen erstellt, auf denen kein Index existiert. Auf diesen Tabelle operiert das verschachtelte SELECT Statement jedoch die meiste Zeit, wesshalb die meiste Zeit ein sequential Scan verwendet wird.

### 3.1.8 Graphtraversierung mit Hilfe von rekursiven INNER JOIN

Bei der Graphtraversierung mit Hilfe von rekursiven INNER JOIN soll der Graph traversiert werden, indem die Relationentabelle immer wieder mit sich selber verknüpft wird. Die Ausgabe ist, ähnlich wie bei der Graphtraversierung mit Hilfe von verschachteltem SELECT Statement, die Nachbarn der Knoten, die sich auf der mitgegebenen Rekursionsstiefe befinden. Ein Beispielstatement für den rekursiven INNER JOIN ist im Anhang gegeben (siehe A.17). Der Ausführungsplan für die relation\_facebook Tabelle befindet sich im Anhang (A.23). Fast alle Verknüpfungen werden mit Hilfe eines Hash Join vollzogen. Bei dem Ausführungsplan fällt der Merge Join auf, der sehr viel Zeit in Anspruch nimmt:

Listing 3.11: Merge JOIN

```
-> Merge Join (cost=89342.23..356052.39 rows=17747686 width=4) (actual time
      =112.509..1178.121 rows=8863706 loops=1)
```

### 3.1.9 Graphtraversierung mit Hilfe von selbstgeschriebenen Stored Procedure

Bei der Graphtraversierung mit Hilfe von selbstgeschriebenen Stored Procedure wird der Graph mit Hilfe eines selbst erstellten Stored Procedure, das sich selber bis zu einer mit gegebenen Rekursionstiefe wieder aufruft, traversiert (siehe A.18). Die Abbruchbedingung wird dem Stored Procedure in Form einer Rekursionsstiefe mitgegeben. In jeder Rekursionsstufe erstellt das Script 2 temporäre Tabelle. Eine temporäre Tabelle <sup>4</sup> wird auf Basis eines Eingabeparameter (Datenstruktur Array) erstellt. Diese temporäre Tabelle besitzt eine Spalte. Diese Tabelle stellt die Spalte src der aktuellen Rekursionsstufe dar. Sie wird im IN Operator der WHERE Klausel verwendet um die 2. temporäre Tabelle zu erstellen. Die 2. temporäre Tabelle beinhaltet die Spalte dst der aktuellen Rekursionsstufe. Die 2. temporäre Tabelle wird als Aufrufparameter für die nächst tiefere Rekursionsstufe mitgegeben. In der nächst tieferen Rekursionsstufe dient die 2. temporäre Tabelle als die Tabelle, die alle src Spalten der aktuellen Rekursionsstufe beinhaltet. Die 2. temporäre Tabelle wird mit Hilfe von dynamischen SQL <sup>5</sup> erstellt.

---

<sup>4</sup>Zuerst wurde das Script mit Hilfe einer standard Tabelle erstellt. Dadurch war das Stored Procedure jedoch um den Faktor 7 langsamer. Es ist die Vermutung, dass durch die Anlage als temporäre Tabelle, die Tabelle im Shared Memory angelegt wird. Hierdurch wird der Performancegewinn erzielt [Fro18, S.26])

<sup>5</sup>"Dynamisches SQL wird gerne verwendet, da zur Zeit der Erstellung des Programms häufig nicht alle Parameter bekannt sind. Die Vervollständigung wird auf den Ausführungszeitpunkt verlagert. Darüber hinaus ist es möglich, Programme zu erstellen, die in gewisser Weise generisch oder allgemeingültig sind." [Fro18, S.316 - 317]

# 4 Graph-Datenbanken im praktischen Einsatz: OLAP

## 4.1 PostgreSQL: OLAP

Im folgenden Abschnitt werden die in Kapitel 3 vorgestellten SQLs auf ihre Leistungsfähigkeit untersucht. Für die Messung wurden vier verschiedene Stored Procedures angelegt:

- `innerJoinGenerator`
- `recursivesearch`
- `selectCascadingGenerator`
- `selectUnionGenerator`

Der Aufruf der Statements funktioniert gleich, es werden die Rekursionstiefe, der Startknoten und die Tabelle, auf welcher die Stored Procedure ausgeführt wird, übergeben.

Die Funktionen `innerJoinGenerator`, `selectCascadingGenerator` und `selectWithUnionSourceCodeGenerator` generieren die entsprechenden Statements und führen diese aus. Die Funktion `innerJoinGenerator` erzeugt ein Select Statement in der die Abfrage, wie in Abschnitt 3.1.8 beschrieben, erzeugt und ausgeführt wird. Entsprechend erzeugt `selectCascadingGenerator` eine verschachtelte Select-Abfrage. Die Funktion ist in Listing A.19 abgebildet. Die Funktion `selectUnionGenerator` erzeugt eine Abfrage wie in Abschnitt 3.1.6 beschrieben. Bei der Funktion `recursivesearch` handelt es sich um die Funktion wie sie in Abschnitt 3.1.9 beschrieben.

### 4.1.1 Indexe und Partionierte Tabellen

Um eine bessere Aussage über die Performance von PostgreSQL zu bekommen wurden zusätzlich zu den "Default" –Relationstabellen für jeden Datensatz zwei weitere Tabellen angelegt.

Die erste Tabelle hat das Suffix ”\_with\_index”. Im Listing ist beispielhaft das Create-Skript für die Relationen aus dem Datensatz ”youtube” dargestellt:

Listing 4.1: Tabelle mit Index anlegen

```
CREATE TABLE IF NOT EXISTS relation_youtube_with_index(
src INTEGER REFERENCES profiles_youtube(ID),
dst INTEGER REFERENCES profiles_youtube(ID),
type VARCHAR(50),
date DATE
);
CREATE INDEX yt_dst ON relation_youtube_with_index (dst);
CREATE INDEX yt_src ON relation_youtube_with_index (src);
```

Im Unterschied zur ”public\_youtube”-Tabelle wurde diese Tabelle noch um zwei Indices erweitert. Dabei wurde ein Index auf die src-Spalte angelegt, der andere auf die dst-Spalte.

Neben den den ”\_with\_index”-Tabellen wurden noch die \_partitioned-Tabellen angelegt. Diese verwenden die gleichen Indices wie die ”\_with\_index”-Tabellen, jedoch sind sie zusätzlich in 4 Partiten aufgeteilt:

Listing 4.2: Partitionierte Tabelle mit Indices anlegen

```
CREATE TABLE IF NOT EXISTS relation_youtube_partitioned(
src INTEGER REFERENCES profiles_youtube(ID),
dst INTEGER REFERENCES profiles_youtube(ID),
type VARCHAR(50),
date DATE
)PARTITION BY RANGE(src);
CREATE INDEX yt_part_src ON relation_youtube_partitioned (src);
CREATE INDEX yt_part_dst ON relation_youtube_partitioned (dst);
CREATE TABLE relation_youtube_partitioned_0 PARTITION OF
relation_youtube_partitioned
FOR VALUES FROM (0) TO (800000);
CREATE TABLE relation_youtube_partitioned_1 PARTITION OF
relation_youtube_partitioned
FOR VALUES FROM (800001) TO (1600000);
CREATE TABLE relation_youtube_partitioned_2 PARTITION OF
relation_youtube_partitioned
FOR VALUES FROM (1600001) TO (2400000);
CREATE TABLE relation_youtube_partitioned_3 PARTITION OF
relation_youtube_partitioned
FOR VALUES FROM (2400001) TO (3200000);
```

Da die Datensätze unterschiedlich groß sind wurden die Partitionsgrößen entsprechend angepasst. Jeder Datensatz wurde auf 4 Partiten verteilt. Damit ergibt sich für den Datensatz youtube eine Partitionsgröße von 800.000.

### 4.1.2 Benchmark

Mit der Standardinstallation von PostgreSQL wird auch pgbench mitinstalliert. Bei pgbench handelt es sich um ein einfaches Tool zur Durchführung von Benchmark-Tests. Bei einem Benchmark-Test wird eine Menge von SQL-Statements beliebig oft wiederholt, dabei können auch mehrere parallele Sessions geöffnet werden. Beim

durchführen des Tests berechnet pgbench die durchschnittliche Latenz aller Requests, sowie die Transaktionen pro Sekunde.

### Verwendung von pgbench

pgbench wird über die Kommandozeile gestartet. Dabei können eine Reihe von Parametern übergeben werden, mit denen das Verhalten von pgbench gesteuert werden kann.

- -c clients  
Über das Flag -c wird die Anzahl der Clients bzw. die Anzahl der gleichzeitigen Datenbankverbindungen festgelegt. Wenn hier nichts angegeben ist wird nur ein Client verwendet.
- -t transactions  
Über das Flag -t wird festgelegt wieviele Transaktionen jeder Client durchführt. Die Anzahl aller Transaktionen ergibt sich durch das Produkt von Clients und Transactions.
- -h hostname  
Der Hostname des Datenbankservers.
- -p Port  
Der Port auf dem die Datenbank hört.
- -U login  
Der Nutzernamen mit dem sich pgbench anmeldet. Hier kann kein Passwort angegeben werden. Das Datenbankpasswort kann in die Systemvariable PGPASSWORD geschrieben werden. Diese Variable wird von pgbench dann ausgewertet.
- -d Database  
Die Datenbank gegen die sich pgbench verbindet
- -f File  
Über -f kann jeweils eine Datei mit mehreren SQL-Statements übergeben werden. Eine Transaktion entspricht dabei der Abarbeitung aller SQL-Befehle innerhalb der Datei. Wird keine Datei angegeben führt pgbench ein Default-Benchmarking aus.

Für die Benchmarktests wurden zehn Clients parallel gestartet werden, die jeweils fünf Transaktionen durchführen sollten. Aus dem Mittelwert dieser 50 Transaktionen wurden bestimmt pgbench Latenz und Transaktionen pro Sekunde.

Listing 4.3: pgbench Statment

```
/usr/lib/postgresql/11/bin/pgbench \  
-h 10.20.110.43 -p 5413 \  
-U postgres -d team22 \  

```

```
-c 10 -t 5 -f Select_innerjoinsourcecodegenerator_1.sql
```

Pgbench kann kein Passwort übergeben werden. Als Workaround kann das Passwort in die Umgebungsvariable PGPASSWORD gespeichert werden. Die Variable wird dann von pgbench ausgewertet.

## Testaufbau

Für die Benchmarktests wurde auf dem Master-Node der Ordner pgbench angelegt. Im Ordner pgbench sind eine Reihe von Unterordnern angelegt sowie das Skript pgbench.sh. Dieses Skript erzeugt das pgbench-Statment. Anschließend ruft das Skript sequentiell alle pgbench.sh-Skripte auf die sich in den Unterordnern befinden auf.

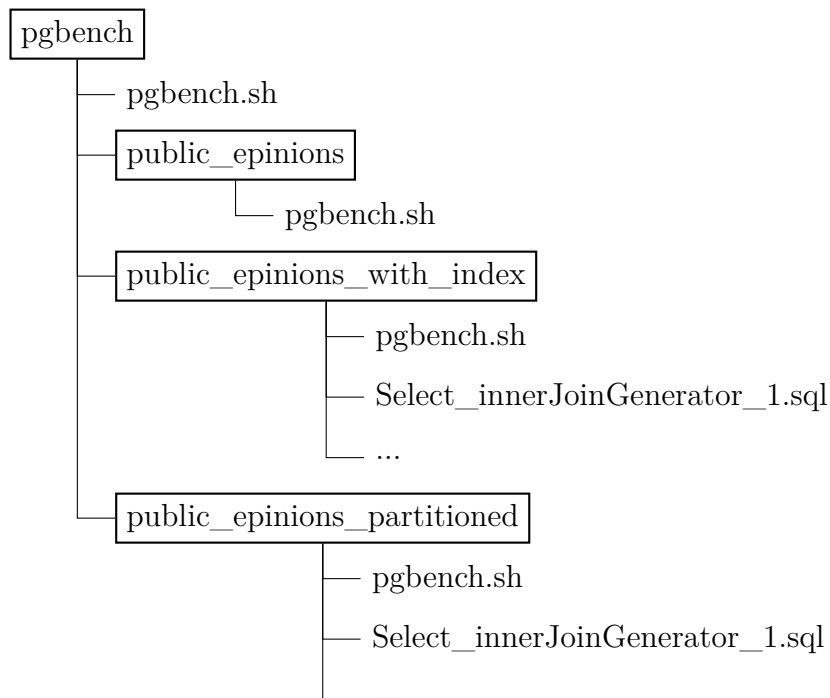


Abbildung 4.1: Ordnerstruktur Benchmarktest

Jedes pgbench.sh-Skript startet sequenziell einen pgbench-Benchmarktest mit jeder SQL-Datei im selben Ordner. Jeder Ordner steht dabei für eine der relations-Tabellen. Für jede Kombination aus Rekursionstiefe und SQL-Statement gibt es eine SQL-Datei, welche genau ein Statement enthält.

Nach jedem Durchlauf von pgbench wird folgende Ausgabe erzeugt:

Listing 4.4: Ausgabe pgbench

```
pgghost: 10.20.110.43 pgport: 5413 nclients: 10 nxacts: 5 dbName: team22
transaction type: Select_innerjoinsourcecodegenerator_1.sql
scaling factor: 1
```



```
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 5
number of transactions actually processed: 50/50
latency average = 9.899 ms
tps = 1010.157069 (including connections establishing)
tps = 1104.908743 (excluding connections establishing)
```

Der Output von pgbench wird mit Hilfe der pgbench.sh Skripte in eine Logdatei geschrieben. Die Logdatei wird innerhalb jedes Ordners unterhalb von pgbench angelegt. Für jeden Testlauf wird eine neue Logdatei angelegt. Damit die Logdateien nicht überschrieben werden, wird die Logdatei mit einem Zeitstempel angelegt. So steht zum Beispiel in der Datei unter dem Pfad `/pgbench/public_epinions_with_index/pgbench_2019_01_12_15_26.log` die Ausgabe des Benchmarktests für die Tabelle `public_epinions_with_index` vom 12.1.2019.

In den folgenden drei Abschnitten sind die durchschnittlichen Latenzen aller Tabellen dargestellt. Die Unterteilung erfolgt hierbei nach Tabellen ohne Indices, Tabellen mit Indices und Partionierte Tabellen mit Indices.

### Antwortzeiten ohne Indices

In diesem Abschnitt sind die Laufzeiten der verschiedenen SQLs auf den Tabellen ohne Indices dargestellt.

#### relation\_epinions

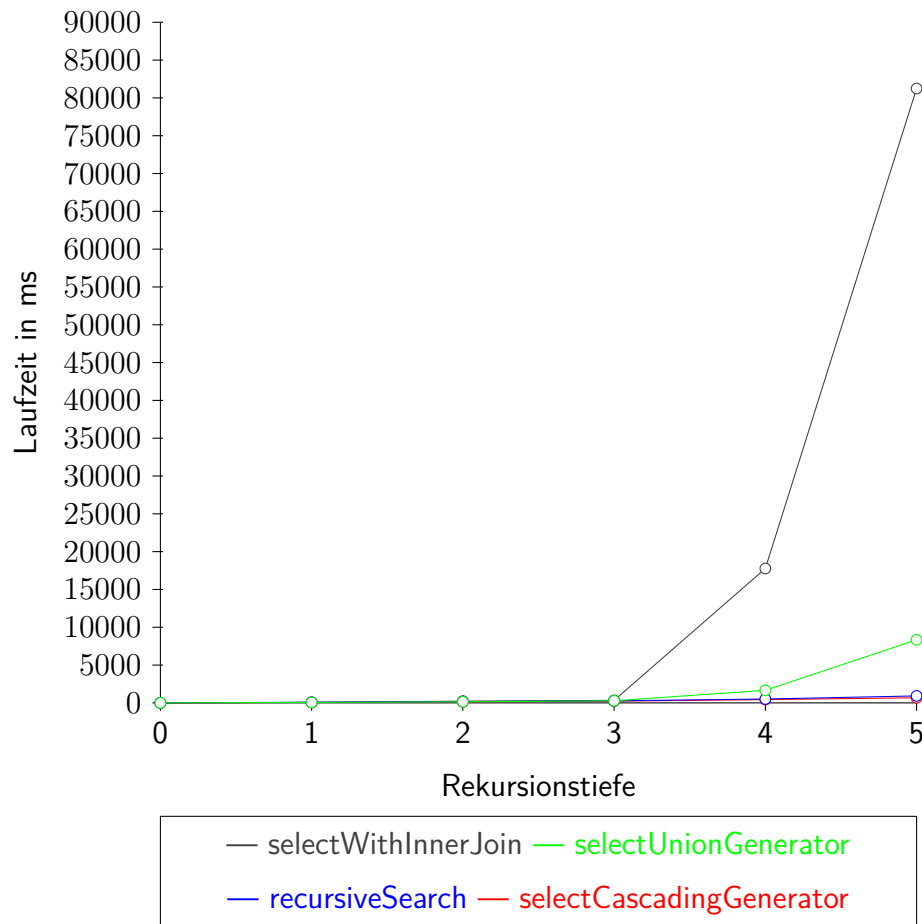


Abbildung 4.2: relation\_epinions

	Laufzeit in MS			
Rerkusions-tiefe	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	63.636	89.858	59.275	61.065
2	139.225	168.822	143.367	231.850
3	229.997	273.452	277.766	333.011
4	432.297	513.038	1664.168	17772.456
5	677.420	921.228	8335.513	81241.284

Tabelle 4.1: Laufzeit der SQLs für Tabelle relation\_epions

Bei der relation\_epinions Tabelle zeigen sich ab der 3. Rekursionstufe deutliche Unterschiede in den Laufzeiten der einzelnen Statements. In der Rekursionsstufe 5 schneidet das Innerjoin-Statement mit 81 Sekunden am schlechtesten ab. Das verschachtelte Select ist mit 677 Millisekunden am schnellsten und benötigt dabei nur 0.8% der Laufzeit des Innerjoin Statements.

### relation\_livejournal

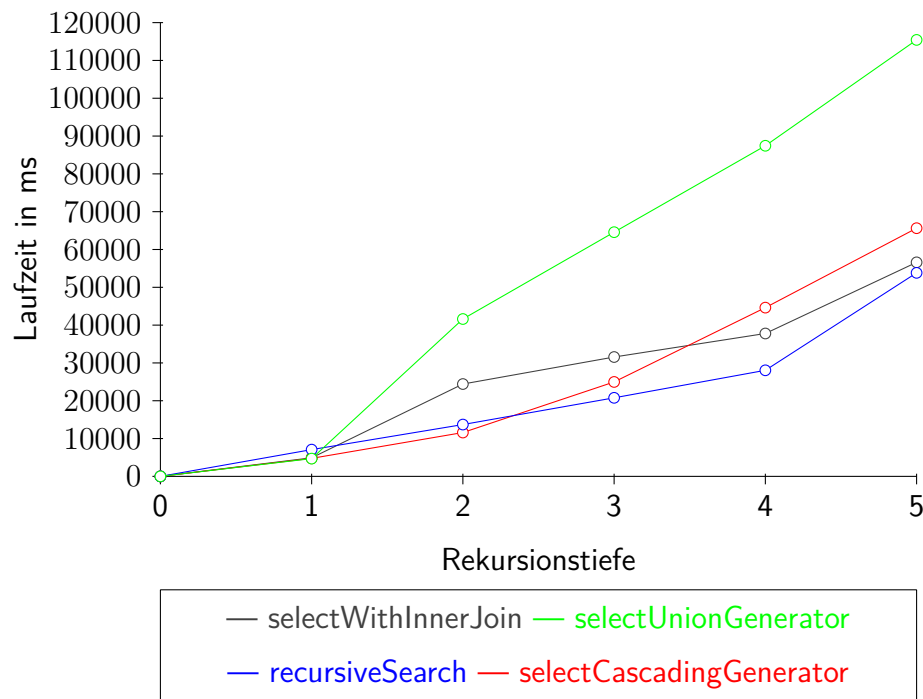


Abbildung 4.3: relation\_livejournal

Rekursions- tiefe	Laufzeit in MS			
	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	4761.733	7075.399	4695.466	4909.862
2	11589.219	13702.381	41616.216	24417.787
3	24971	20773.831	64577.873	31563.339
4	44625.935	28048.866	87434.075	37785.200
5	65632.972	53825.401	115432.071	56601.185

Tabelle 4.2: Laufzeit der SQLs für Tabelle relation\_livejournal

Bei der relation\_livejournal Tabelle sind die Laufzeiten am längsten. Hier schneidet das Standard-SQL am schlechtesten ab. Die Rekursive Stored Procedure ist hier am schnellsten. Jedoch sind die Laufzeiten in der fünften Rekursionsstufe mit im besten

Fall 53 Sekunden und im schlechtesten Fall 115 Sekunden sehr lange. Selbst in der ersten Rekursionstufe benötigt eine Abfrage mindestens 4,6 Sekunden.

### relation\_facebook

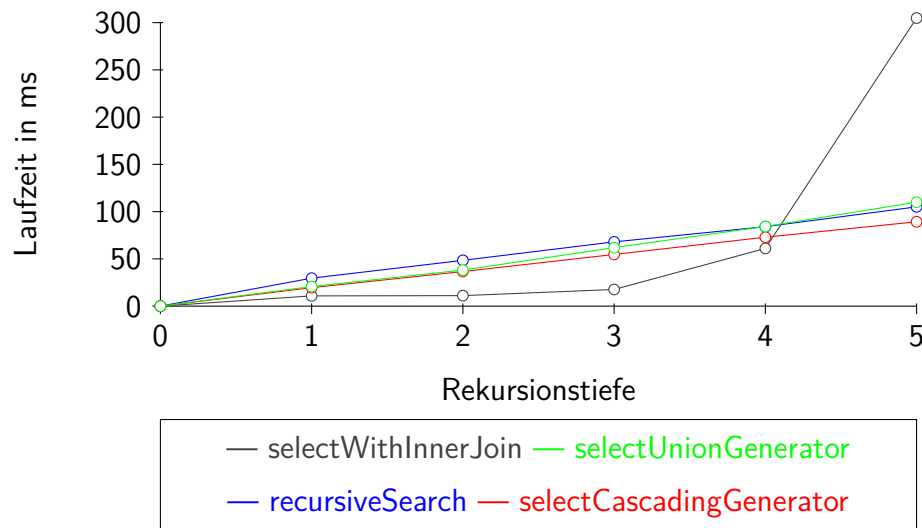


Abbildung 4.4: relation\_facebook

	Laufzeit in MS			
Rerkusions-tiefe	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	19.528	29.595	20.716	10.825
2	36.613	48.442	38.137	11.119
3	54.681	67.989	61.904	17.613
4	72.934	84.084	84.295	61.089
5	89.262	105.049	110.000	304.821

Tabelle 4.3: Laufzeit der SQLs für Tabelle relation\_facebook

Bei den facebook-Daten ist das Laufzeitverhalten des verschachtelten Selects, der rekursiven Funktion und dem Standard SQL gleich. Die Unterschiede in der Latenz sind im Verhältnis zur Latenz gering. Einen Sonderfall bildet das innerJoin hier ist ab der dritten Rekursionsstufe ein exponentieller Anstieg der Laufzeit zu beobachten. Ab der ersten bis zur einschließlich vierten Rekursionstufe ist das innerJoin jedoch am Performantesten.

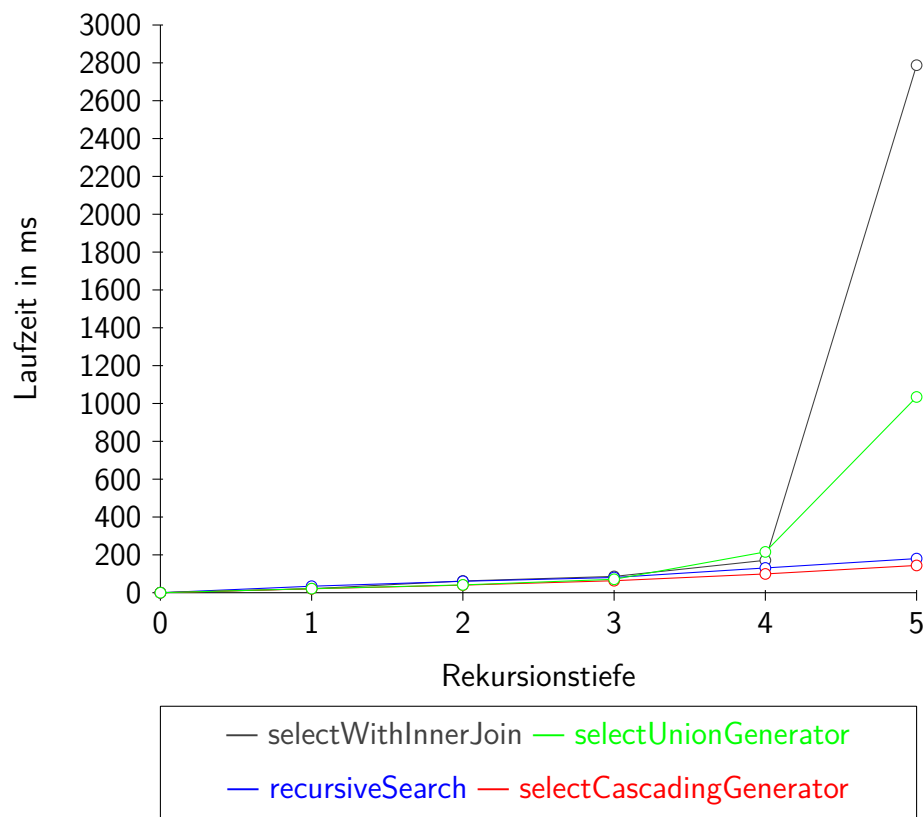
**relation\_wiki\_vote**

Abbildung 4.5: relation\_wiki\_vote

	Laufzeit in MS			
Rerkusions-tiefe	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	20.774	34.365	21.490	21.182
2	40.090	60.394	41.234	62.066
3	63.405	80.454	70.961	86.128
4	99.210	130.866	215.687	171.283
5	144.089	180.301	1034.360	2787.659

Tabelle 4.4: Laufzeit der SQLs für Tabelle relation\_wiki\_vote

Bei den Wikipedia-Daten zeigt sich ein ähnliches Laufzeitverhalten wie bei den Epionions-Daten. InnerJoin und Standard-SQL verschlechtern sich von der vierten auf die fünfte Iterationsstufe massiv. Das verschachtelte Select weist hier über alle Rekursionsstufen die geringste Latenz auf.

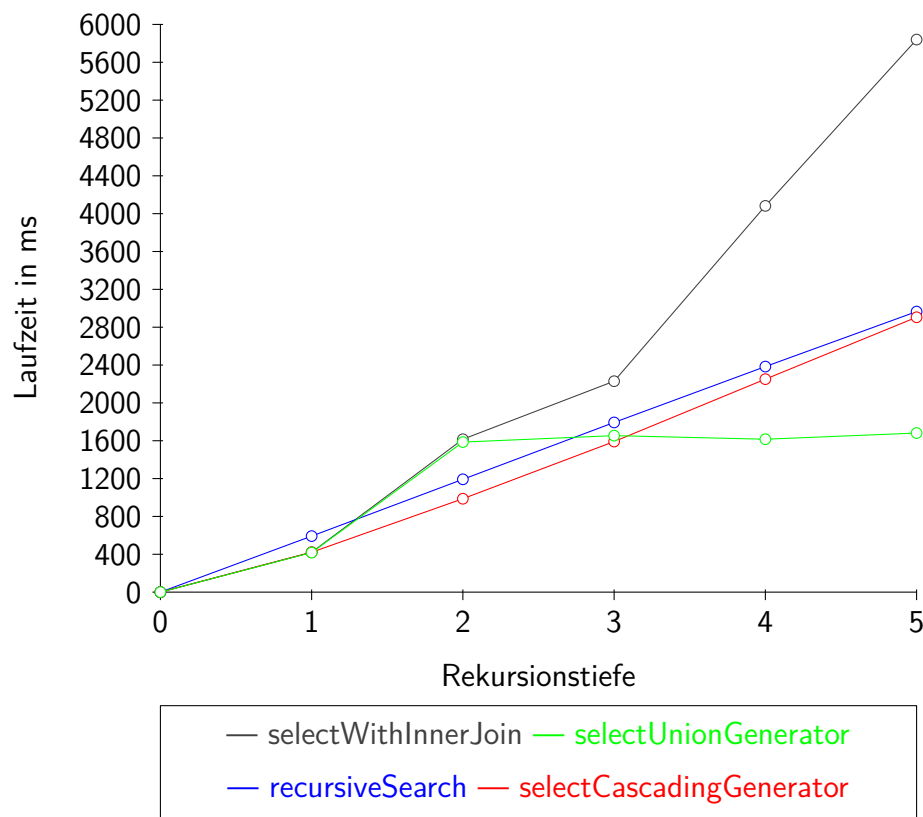
**relation\_youtube**

Abbildung 4.6: relation\_youtube

	Laufzeit in MS			
Rerkusions-tiefe	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	421.464	592.063	418.105	421.124
2	986.995	1192.278	1585.836	1615.843
3	1591.391	1793.281	1654.011	2229.214
4	2250.832	2383.885	1615.986	4082.179
5	2905.167	2964.950	1681.231	5839.987

Tabelle 4.5: Laufzeit der SQLs für Tabelle relation\_youtube

Bei den Youtube-Daten zeigen die Verschiedenen SQL-Statements ein sehr unterschiedliches Verhalten. Beim Standard-SQL gibt es einen deutlichen Anstieg zwischen von der ersten auf die zweite Rekursionstiefe. Für die weiteren Rekursionstiefen bleibt die Laufzeit dann ungefähr konstant. Das verschachtelte Select und die rekursive Funktion sind in ihrem Laufzeitverhalten sehr ähnlich. Beide steigen linear. Das InnerJoin zweigt bei steigender Rekursionstiefe das schlechteste Laufzeitverhalten.

## Antwortzeiten mit Indices

Im Folgenden sind die Laufzeiten der SQL-Staments auf den Tabellen mit den Indices dargestellt.

### relation\_epinions\_with\_index

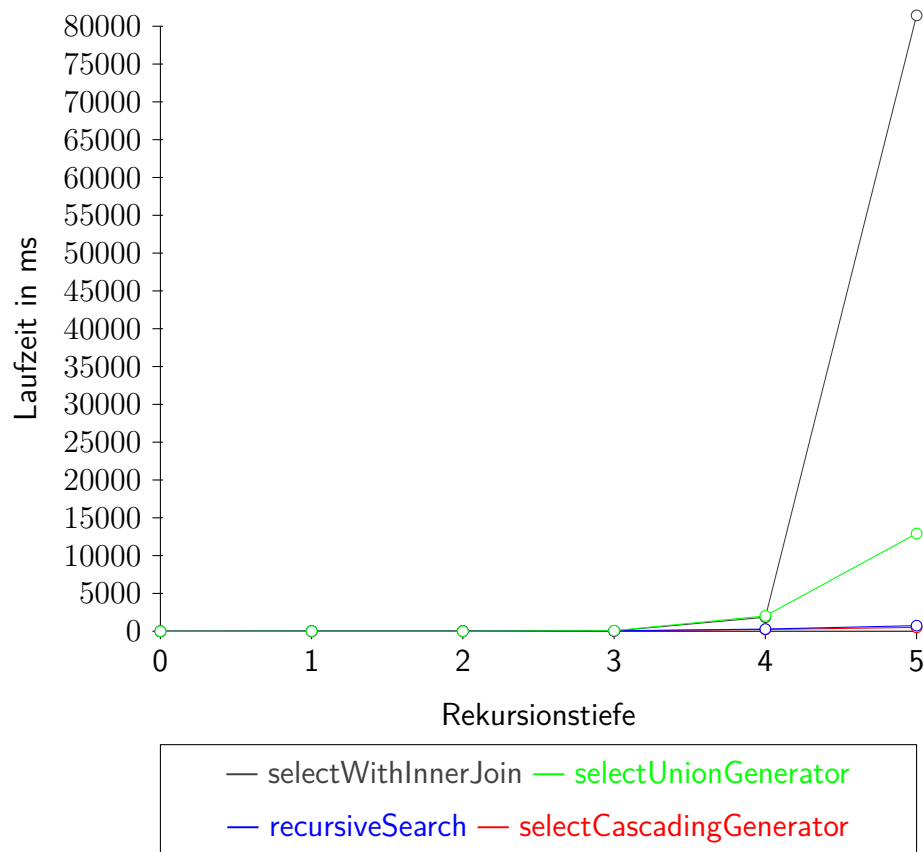


Abbildung 4.7: relation\_epinions\_with\_index

	Laufzeit in MS			
Rerkusions-tiefe	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	8.231	18.999	8.885	10.596
2	9.290	23.549	10.304	10.739
3	36.244	54.953	78.303	41.171
4	235.146	287.319	2024.069	1898.726
5	504.793	737.300	12920.747	81436.561

Tabelle 4.6: Laufzeit der SQLs für Tabelle relation\_epinions\_with\_index

Bei den Epinions-Daten wirkt sich das Einführen der Indices gering aus. Beim inner-Join ist praktisch kein Unterschied zu messbar. Beim StandardSQL führte es sogar zu einer deutlichen Verschlechterung in der fünften Rekursionstufe. Beim verschachtelten Select und bei der rekursiven Funktion gab es eine Verbesserung, diese viel jedoch verhältnismäßig gering aus. Im besten Fall verbesserte sich die fünfte Rekursionstufe von 677 Millisekunden auf 505 Millisekunden.

### relation\_livejournal\_with\_index

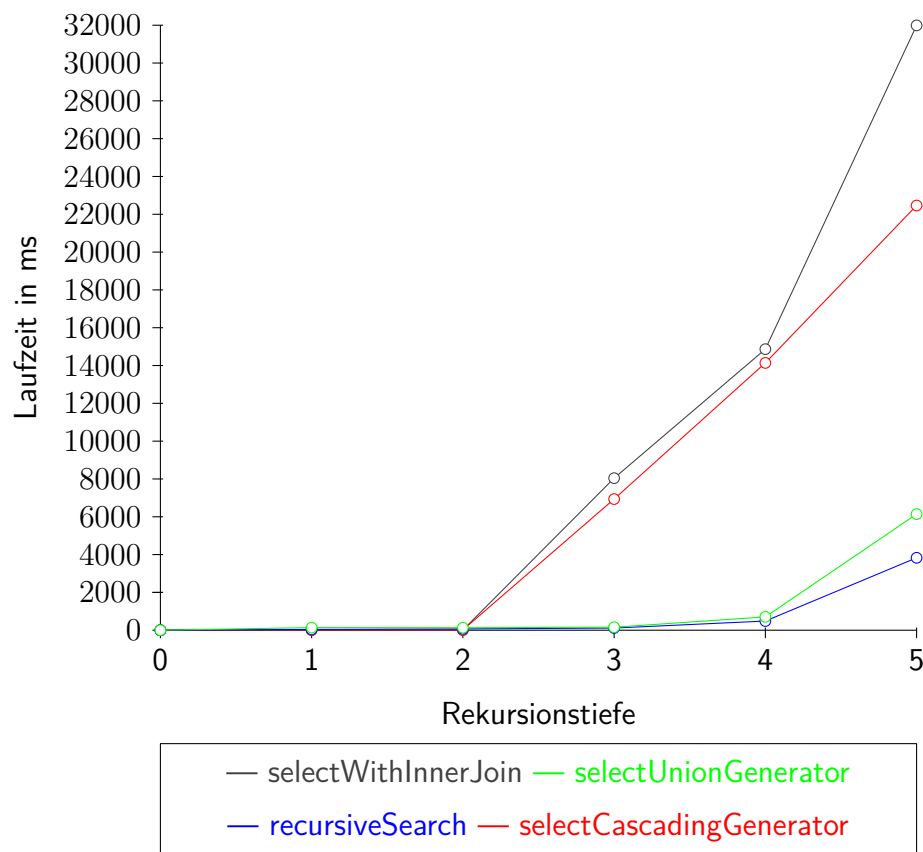


Abbildung 4.8: relation\_livejournal\_with\_index

	Laufzeit in MS			
Rerkusions-tiefe	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	8.672	42.010	139.294	9.165
2	15.240	69.129	131.623	11.862
3	6936.935	114.466	157.707	8042.794
4	14137.544	484.914	705.293	14865.814
5	22464.363	3830.355	6140.824	31990.436

Tabelle 4.7: Laufzeit der SQLs für Tabelle relation\_livejournal\_with\_index



Die Verwendung der Indices führt zu einer erheblich besseren Latenz. Die Laufzeit in der ersten Rekursionsstufe verbessert sich von von mindestens 4,7 Sekunden auf im besten Fall 8.7 Millisekunden. Auch in der fünften Rekursionsstufe verkürzt sich die Laufzeit deutlich.

#### relation\_facebook\_with\_index

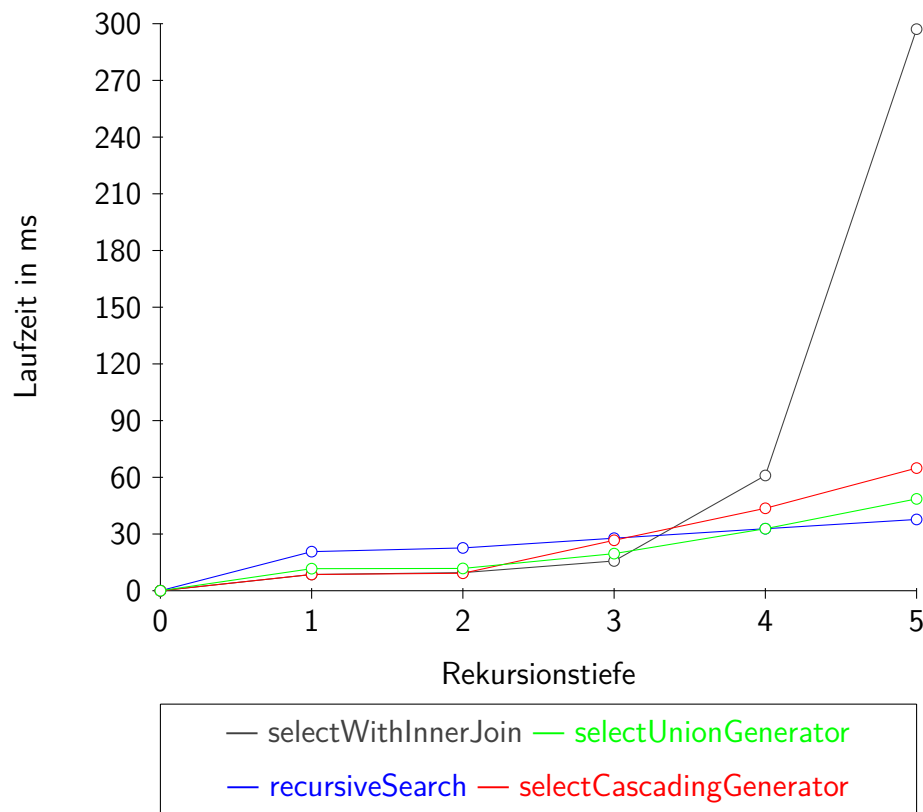


Abbildung 4.9: relation\_facebook\_with\_index

Rekursions- tiefe	Laufzeit in MS			
	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	8.594	20.674	11.672	8.585
2	9.239	22.639	11.787	9.565
3	26.683	27.788	19.625	15.738
4	43.610	32.804	32.817	60.976
5	64.882	37.706	48.580	297.115

Tabelle 4.8: Laufzeit der SQLs für Tabelle relation\_facebook\_with\_index

Bei den Facebook Daten lässt sich durch die Indices in fast allen Fällen eine Verbesserung der Latenz erreichen. Ausnahme ist hier der InnerJoin, hier ist keine wirkliche Verbesserung erkennbar. In der fünften Rekursionsstufe liegt die Latenz im besten Fall bei 38 Millisekunden mit der rekursiven Funktion. Ohne Indices liegt dieser Wert bei 89 Millisekunden und wird durch das verschachtelte SQL erreicht.

#### relation\_wiki\_vote\_with\_index

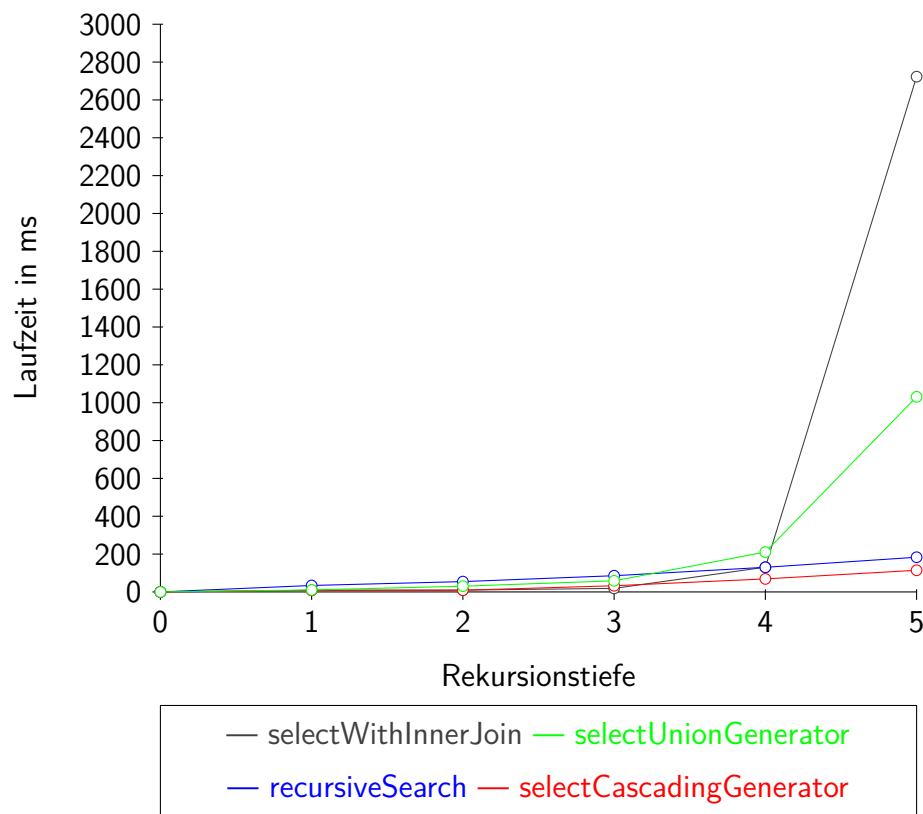


Abbildung 4.10: relation\_wiki\_vote

Rekursions- tiefe	Laufzeit in MS			
	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	8.716	34.144	10.474	8.481
2	8.868	54.617	30.212	9.903
3	32.682	85.721	58.762	18.657
4	68.856	130.150	210.655	130.842
5	114.462	183.301	1031.234	2722.908

Tabelle 4.9: Laufzeit der SQLs für Tabelle relation\_wiki\_vote\_with\_index

Das Einführen der Indices hat eine vergleichsweise geringe Wirkung auf die Laufzeiten der verschiedenen SQLs. Zwar gibt es relativ gesehen deutliche Verbesserungen in den ersten beiden Rekursionsstufen. In der fünften Rekursionsstufe verbessert sich jedoch nur das verschachtelte Select um knapp 21% gegenüber der Tabelle ohne Indices. Für die anderen Statements ergaben sich hier keine Verbesserungen.

#### relation\_youtube\_with\_index

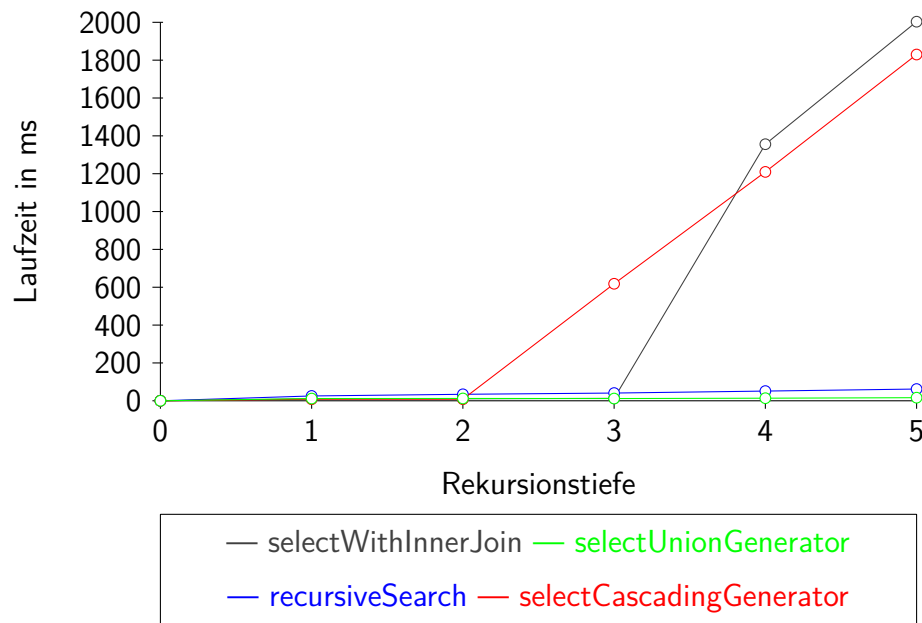


Abbildung 4.11: relation\_youtube\_with\_index

	Laufzeit in MS			
Rerkusions-tiefe	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	8.370	25.508	11.801	10.213
2	8.528	34.273	12.265	10.587
3	618.336	40.852	11.832	12.858
4	1209.513	51.285	13.644	1355.866
5	1829.757	62.037	16.362	2003.275

Tabelle 4.10: Laufzeit der SQLs für Tabelle relation\_youtube\_with\_index

Die Verwendung von Indices zeigt sich eine deutliche Verbesserung der Laufzeiten. Alle Statements weisen eine deutlich geringere Latenz auf. Die größte Veränderung zeigt hierbei das Standard-SQL Statement. Hier verringert sich die Latenz von 1,6 Sekunden auf 16 Millisekunden. Auch die Laufzeit der rekursiven Funktion verbessert sich von knapp 3 Sekunden auf 62 Millisekunden deutlich.

### Antwortzeiten mit partitionierten Tabellen und Indices

Im Folgenden sind die Laufzeiten der SQL-Statements auf den partitionierten Tabellen dargestellt.

#### relation\_facebook\_partitioned

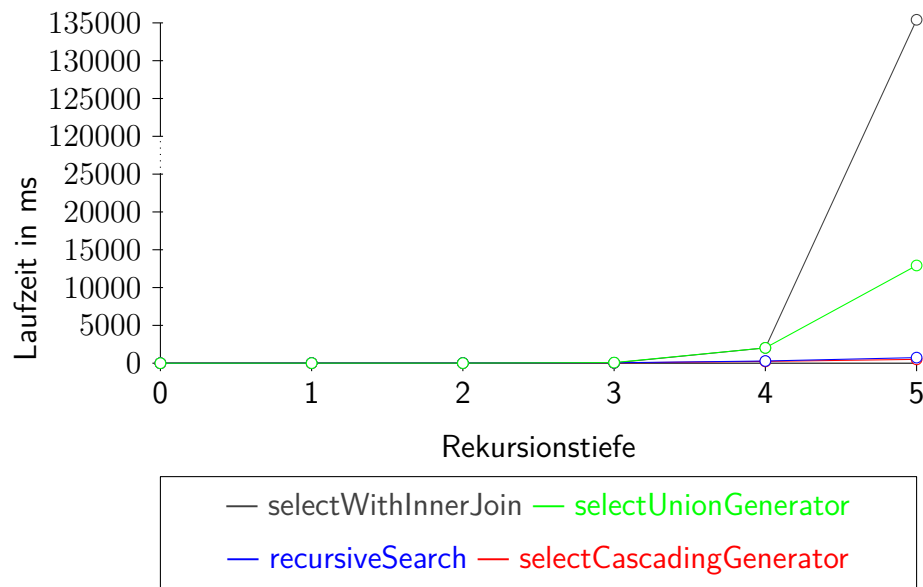


Abbildung 4.12: relation\_epinions\_partitioned

	Laufzeit in MS			
Rerkusions-tiefe	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	8.231	18.999	8.855	8.846
2	9.290	23.549	10.304	9.924
3	36.244	54.953	78.303	43.439
4	235.146	287.319	2024.069	2021.028
5	504.793	737.300	12920.747	136242.608

Tabelle 4.11: Laufzeit der SQLs für Tabelle relation\_epinions\_partitioned

Durch die Partitionierung der Tabelle ergeben sich keine Verbesserungen in der Latenz. Das gilt für alle Statements. Die Laufzeit beim InnerJoin verschlechtert sich sogar deutlich gegenüber der Tabelle bei der nur Indices verwendet wurden.

#### relation\_livejournal\_partitioned

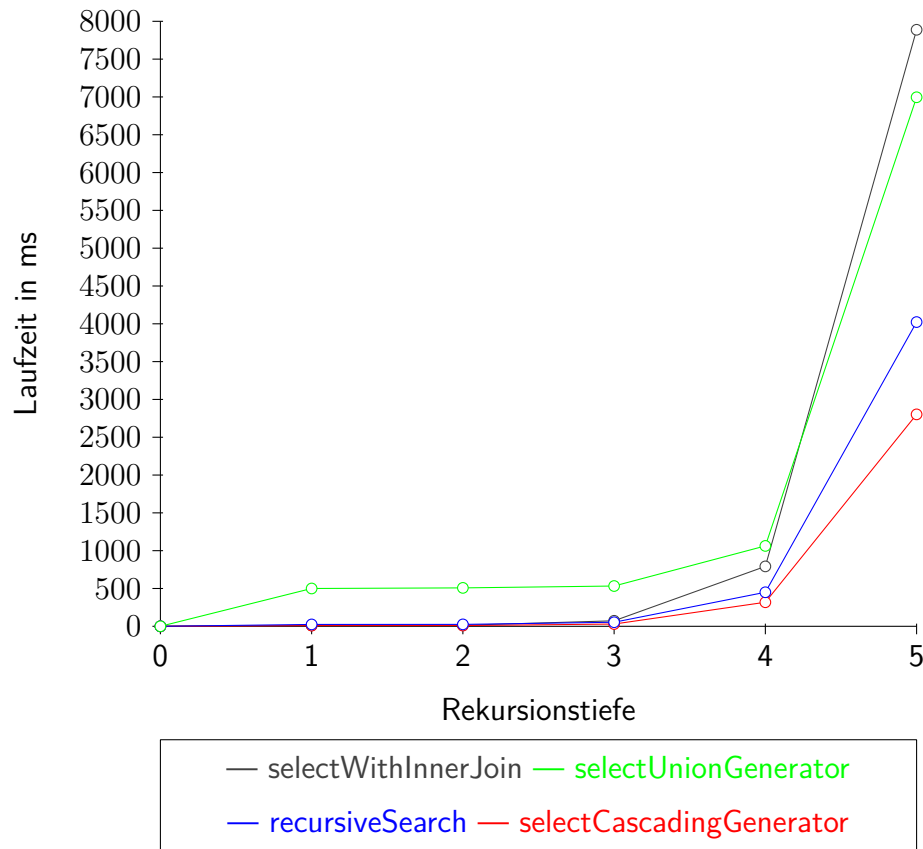


Abbildung 4.13: relation\_livejournal\_partitioned

Rekursions- tiefe	Laufzeit in MS			
	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	8.9415	24.721	500.063	11.849
2	9.875	25.354	507.950	13.616
3	29.855	53.544	532.557	72.842
4	317.921	449.968	1062.957	792.107
5	2802.370	4022.937	6995.607	7888.135

Tabelle 4.12: Laufzeit der SQLs für Tabelle relation\_livejournal\_partitioned

Durch die Partitionierung ergeben sich weitere Verbesserungen in der Performance. So verringert sich die Laufzeit in der fünften Rekursionsstufe um nochmal eine Sekunde. Auffällig ist auch, dass in der partitionierten Tabelle das verschachtelte Select die geringste Latenz aufweist. In den beiden anderen livejournal-Tabellen war die rekursive Stored Procedure am schnellsten. Auffällig ist auch, dass die Partitionierung nur beim verschachtelten Select und beim InnerJoin-Select zu einer Verbesserung geführt hat. Bei den beiden anderen Statements hat sich keine Verbesserung der Latenz ergeben.

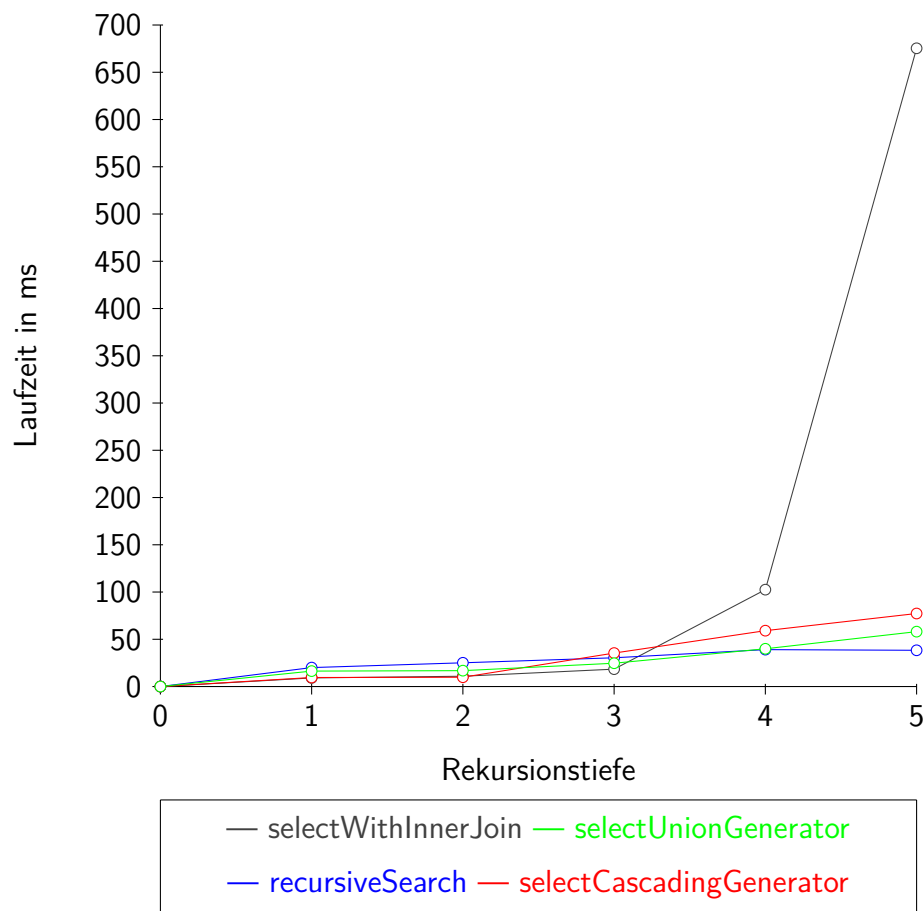
**relation\_epinions\_partitioned**

Abbildung 4.14: relation\_facebook\_partitioned

Rekursions- tiefe	Laufzeit in MS			
	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	9.550	20.057	16.339	9.097
2	9.838	25.119	16.807	10.898
3	35.418	30.430	24.669	18.393
4	59.112	39.050	39.953	102.515
5	77.279	38.354	58.086	675.363

Tabelle 4.13: Laufzeit der SQLs für Tabelle relation\_facebook\_partitioned

Durch die Partitionierung der Tabellen ergibt sich keine weitere Verbesserung gegenüber der Tabelle mit den Indices. Der InnerJoin weist mit einer Latenz von 675 Millisekunden sogar einen deutlich schlechteren Wert auf.

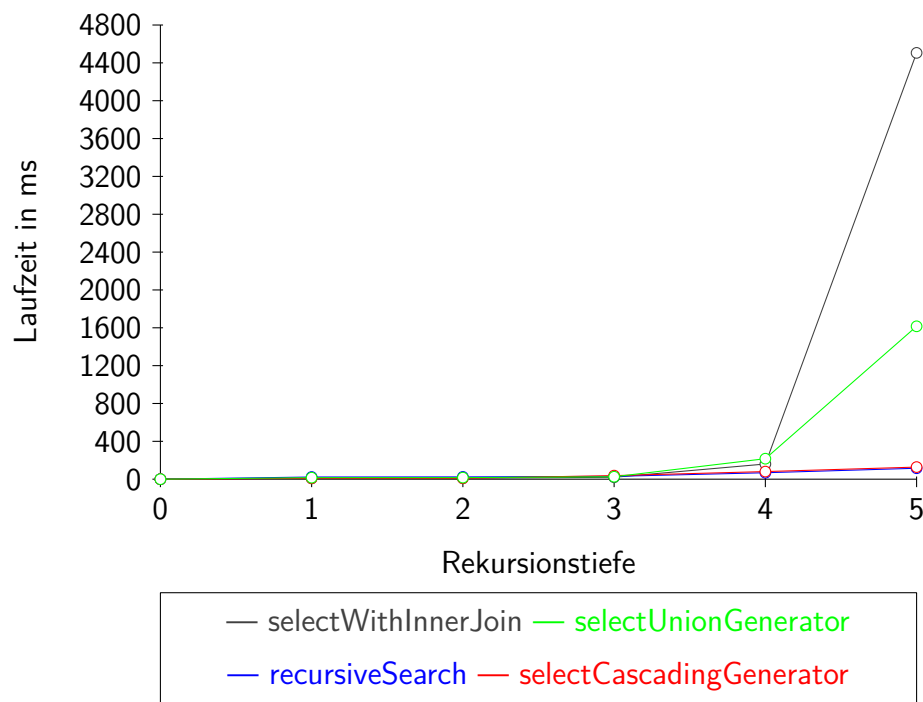
**relation\_wiki\_vote\_partitioned**

Abbildung 4.15: relation\_wiki\_vote\_partitioned

Rekursions- tiefe	Laufzeit in MS			
	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	9.169	23.059	16.152	8.882
2	9.538	24.221	16.649	9.694
3	37.563	29.832	25.806	20.019
4	80.343	67.963	216.439	160.368
5	127.839	115.465	1615.686	4505.545

Tabelle 4.14: Laufzeit der SQLs für Tabelle relation\_wiki\_vote\_partitioned

Durch die Partitionierung der relation\_wiki\_vote Tabelle verschlechtern sich in der fünften Rekursionsstufe alle Statements bis auf die Rekursive Suche. Insgesamt ergibt sich aber keine Verbesserung, da die Rekursive Suche auf der partitionierten Tabelle gleich schnell ist wie das verschachtelte Select auf der nicht partitionierten Tabelle mit den Indices.

**relation\_youtube\_partitioned**

Abbildung 4.16: relation\_youtube\_partitioned

Rekursions- tiefe	Laufzeit in MS			
	selectCascading Generator	recursiveSearch	selectUnion Generator	selectInner JoinGenerator
1	8.505	22.322	492.834	10.423
2	9.090	24.235	490.843	9.162
3	9.629	26.731	500.531	9.909
4	11.286	29.409	510.964	11.589
5	13.694	33.586	497.172	21.320

Tabelle 4.15: Laufzeit der SQLs für Tabelle relation\_youtube\_partitioned

Durch die Partitionierung ergeben sich weitere Verbesserungen in der Laufzeit. Das verschachtelte Select sowie der InnerJoin verbessern sich deutlich. Lediglich das Standard-SQL verschlechtert sich deutlich. Jeoch bleibt die Latenz über alle Rekursionss mit ungefähr 500 Millisekunden Stufen ungefähr konstant. Mit 13.6 Millisekunden ist das verschachtelte Select in der fünften Rekursionstufe am schnellsten.



### 4.1.3 Interpretation der Ergebnisse

Es ist auffallend, dass die Laufzeit der Statements sehr stark an der Anzahl der Eingabeknoten abhängig ist und die Größe der Tabellen eher nachrangig ist. So ist die `relations_youtube` Tabelle deutlich größer als die `relation_epinions` Tabelle, jedoch ist die Latenz der Statements in der fünften Rekursion für die `relations_youtube` Tabelle deutlich geringer als für die `relation_epinions` Tabelle.

Bei der `relation_epinions` Tabelle werden in der vierten Iteration 29184 Zeilen zurückgegeben, die dann in der fünften Iteration als Input benutzt werden. Bei der `relations_youtube` Tabelle sind es nur 760. Entsprechend sind die Unterschiede in der Latenz in der fünften Rekursionsstufe. Für die Youtube-Daten werden im besten Fall nur 14 Millisekunden benötigt, bei den Epinions-Daten sind es 505 Millisekunden.

Weiterhin zeigt sich, dass das `InnerJoin`-Statement in fast allen Fällen in der fünften Rekursionsstufe die schlechteste Wahl darstellt. Besonders bei den Wikipedia und Epinions-Daten wird dies sehr deutlich.

# **5 Fazit**

## **5.1 Fazit**

# A Anhang

## A.1 Graph-Datenbanken - Grundlegende technologische Aspekte

## A.2 Graph-Datenbanken und -Frameworks - Ausgewählte Systeme

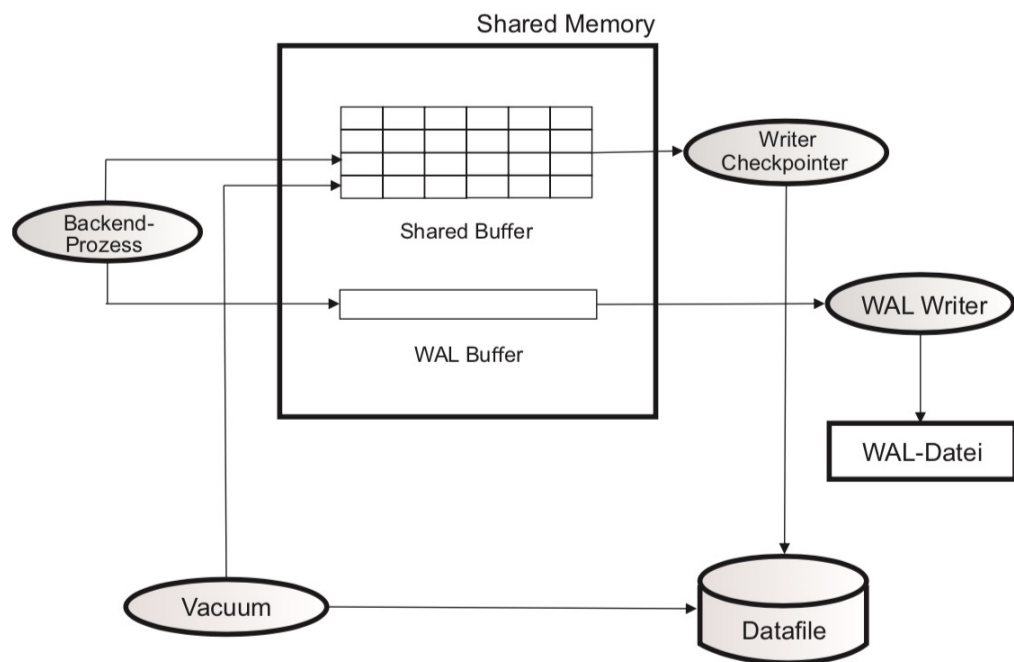


Abbildung A.1: Postgres Architektur

## A.3 Graph-Datenbanken im praktischen Einsatz: OLTP

Listing A.1: CSV Input

```
\copy Beitraege
FROM './data/Beitraege.csv' DELIMITER ',' CSV HEADER;
```

Listing A.2: Anlegen der Tabelle facebook-profiles

```
create TABLE IF NOT EXISTS profiles_facebook(  
    ID INTEGER PRIMARY KEY,  
    first VARCHAR(50),  
    last VARCHAR(50),  
    gender GENDER,  
    birth DATE,  
    country VARCHAR(50)  
);
```

Listing A.3: Anlegen der Tabelle facebook-relation

```
CREATE TABLE IF NOT EXISTS relation_facebook(  
    src INTEGER REFERENCES profiles_facebook(ID),  
    dst INTEGER REFERENCES profiles_facebook(ID),  
    type VARCHAR(50),  
    date DATE  
);
```

Listing A.4: Hinzufügen von Fremdschlüsseln

```
\copy profiles_facebook_tmp(first,last,gender,birth,country) FROM '/data/WS2018/  
facebook-profiles' DELIMITER ',' CSV HEADER;  
INSERT INTO profiles_facebook (ID, first, last, gender, birth, country)  
SELECT ID-1, first, last, gender, birth, country from profiles_facebook_tmp;
```

Listing A.5: Erstellen von partitionierten Tabellen mit Index facebook

```
CREATE TABLE IF NOT EXISTS relation_facebook_partitioned(  
    src INTEGER REFERENCES profiles_facebook(ID),  
    dst INTEGER REFERENCES profiles_facebook(ID),  
    type VARCHAR(50),  
    date DATE  
)PARTITION BY RANGE(src);  
  
CREATE INDEX fb_part_src ON relation_facebook_partitioned (src);  
CREATE INDEX fb_part_dst ON relation_facebook_partitioned (dst);  
  
CREATE TABLE relation_facebook_partitioned_0 PARTITION OF  
    relation_facebook_partitioned  
FOR VALUES FROM (0) TO (23000);  
CREATE TABLE relation_facebook_partitioned_1 PARTITION OF  
    relation_facebook_partitioned  
FOR VALUES FROM (23000) TO (46000);  
CREATE TABLE relation_facebook_partitioned_2 PARTITION OF  
    relation_facebook_partitioned  
FOR VALUES FROM (46000) TO (69000);  
CREATE TABLE relation_facebook_partitioned_3 PARTITION OF  
    relation_facebook_partitioned  
FOR VALUES FROM (69000) TO (92000);
```

Listing A.6: Erstellen von Indexen auf relation Tabelle facebook

```
CREATE INDEX fb_dst ON relation_facebook_with_index (dst);  
CREATE INDEX fb_src ON relation_facebook_with_index (src);
```

Listing A.7: Erstellen von partitionierten Tabellen mit Index youtube

```
CREATE TABLE IF NOT EXISTS relation_youtube_partitioned(  
    src INTEGER REFERENCES profiles_youtube(ID),  
    dst INTEGER REFERENCES profiles_youtube(ID),  
    type VARCHAR(50),  
    date DATE  
)PARTITION BY RANGE(src);  
  
CREATE INDEX yt_part_src ON relation_youtube_partitioned (src);
```

```
CREATE INDEX yt_part_dst ON relation_youtube_partitioned (dst);

CREATE TABLE relation_youtube_partitioned_0 PARTITION OF
  relation_youtube_partitioned
FOR VALUES FROM (0) TO (800000);
CREATE TABLE relation_youtube_partitioned_1 PARTITION OF
  relation_youtube_partitioned
FOR VALUES FROM (800001) TO (1600000);
CREATE TABLE relation_youtube_partitioned_2 PARTITION OF
  relation_youtube_partitioned
FOR VALUES FROM (1600001) TO (2400000);
CREATE TABLE relation_youtube_partitioned_3 PARTITION OF
  relation_youtube_partitioned
FOR VALUES FROM (2400001) TO (3200000);
```

Listing A.8: Erstellen von Indexen auf relation Tabelle youtube

```
CREATE INDEX yt_dst ON relation_youtube_with_index (dst);
CREATE INDEX yt_src ON relation_youtube_with_index (src);
```

Listing A.9: Erstellen von partitionierten Tabellen mit Index livejournal

```
CREATE TABLE IF NOT EXISTS relation_livejournal_partitioned(
  src INTEGER REFERENCES profiles_livejournal(ID),
  dst INTEGER REFERENCES profiles_livejournal(ID),
  type VARCHAR(50),
  date DATE
)PARTITION BY RANGE(src);

CREATE INDEX lj_part_src ON relation_livejournal_partitioned (src);
CREATE INDEX lj_part_dst ON relation_livejournal_partitioned (dst);

CREATE TABLE relation_livejournal_partitioned_0 PARTITION OF
  relation_livejournal_partitioned
FOR VALUES FROM (0) TO (10000000);
CREATE TABLE relation_livejournal_partitioned_1 PARTITION OF
  relation_livejournal_partitioned
FOR VALUES FROM (10000000) TO (20000000);
CREATE TABLE relation_livejournal_partitioned_2 PARTITION OF
  relation_livejournal_partitioned
FOR VALUES FROM (20000000) TO (30000000);
CREATE TABLE relation_livejournal_partitioned_3 PARTITION OF
  relation_livejournal_partitioned
FOR VALUES FROM (30000000) TO (40000000);
```

Listing A.10: Erstellen von Indexen auf relation Tabelle livejournal

```
CREATE INDEX lj_src ON relation_livejournal_with_index (src);
CREATE INDEX lj_dst ON relation_livejournal_with_index (dst);
```

Listing A.11: Erstellen von partitionierten Tabellen mit Index epinion

```
CREATE TABLE IF NOT EXISTS relation_epinions_partitioned(
  src INTEGER REFERENCES profiles_epinions(ID),
  dst INTEGER REFERENCES profiles_epinions(ID),
  type VARCHAR(50),
  date DATE
)PARTITION BY RANGE(src);

CREATE INDEX ep_part_src ON relation_epinions_partitioned (src);
CREATE INDEX ep_part_dst ON relation_epinions_partitioned (dst);

CREATE TABLE relation_epinions_partitioned_0 PARTITION OF
  relation_epinions_partitioned
FOR VALUES FROM (0) TO (102000);
CREATE TABLE relation_epinions_partitioned_1 PARTITION OF
  relation_epinions_partitioned
```

```
FOR VALUES FROM (102000) TO (204000);
CREATE TABLE relation_epinions_partitioned_2 PARTITION OF
    relation_epinions_partitioned
FOR VALUES FROM (204000) TO (306000);
CREATE TABLE relation_epinions_partitioned_3 PARTITION OF
    relation_epinions_partitioned
FOR VALUES FROM (306000) TO (408000);
```

Listing A.12: Erstellen von Indexen auf relation Tabelle epinion

```
CREATE INDEX ep_dst ON relation_epinions_with_index (dst);
CREATE INDEX ep_src ON relation_epinions_with_index (src);
```

Listing A.13: Erstellen von partitionierten Tabellen mit wikivote epinion

```
CREATE TABLE IF NOT EXISTS relation_wiki_vote_partitioned(
    src INTEGER REFERENCES profiles_wiki_vote(ID),
    dst INTEGER REFERENCES profiles_wiki_vote(ID),
    type VARCHAR(50),
    date DATE
)PARTITION BY RANGE(src);

CREATE INDEX ww_part_src ON relation_wiki_vote_partitioned (src);
CREATE INDEX ww_part_dst ON relation_wiki_vote_partitioned (dst);

CREATE TABLE relation_wiki_vote_partitioned_0 PARTITION OF
    relation_wiki_vote_partitioned
FOR VALUES FROM (0) TO (30000);
CREATE TABLE relation_wiki_vote_partitioned_1 PARTITION OF
    relation_wiki_vote_partitioned
FOR VALUES FROM (30000) TO (60000);
CREATE TABLE relation_wiki_vote_partitioned_2 PARTITION OF
    relation_wiki_vote_partitioned
FOR VALUES FROM (60000) TO (90000);
CREATE TABLE relation_wiki_vote_partitioned_3 PARTITION OF
    relation_wiki_vote_partitioned
FOR VALUES FROM (90000) TO (120000);
```

Listing A.14: Erstellen von Indexen auf relation Tabelle wikivote

```
CREATE INDEX ww_src ON relation_wiki_vote_with_index (src);
CREATE INDEX ww_dst ON relation_wiki_vote_with_index (dst);
```

Listing A.15: Erstellen der Indexe für die relation Tabelle facebook

```
CREATE INDEX fb_dst ON relation_facebook_with_index (dst);
CREATE INDEX fb_src ON relation_facebook_with_index (src);
```

Listing A.16: Verschachteltes SELECT Statement

```
SELECT DISTINCT(dst) FROM team22.relation_facebook WHERE src IN(
    SELECT DISTINCT(dst) FROM team22.relation_facebook WHERE src IN(
        SELECT DISTINCT(dst)FROM team22.relation_facebook WHERE src IN(1)
    )
)
```

Listing A.17: Rekursiver JOIN

```
SELECT DISTINCT(rf3.dst)
FROM public.relation_facebook rf1,
public.relation_facebook rf2,
public.relation_facebook rf3
WHERE rf2.src = rf1.dst
AND rf3.src = rf2.dst
AND rf1.src = 765;
```

Listing A.18: Selbstgeschriebenes Stored Procedure

```

CREATE OR REPLACE FUNCTION recursivesearch(tInput integer[], iRecursionDepth
integer, sTable text) RETURNS SETOF integer AS $$
Declare
intermDst_ integer[];
iCount integer;
BEGIN
--iRecursionDepth = iRecursionDepth + 1;
CREATE TEMPORARY TABLE intermDst AS SELECT * FROM unnest(tInput);
EXECUTE 'CREATE TEMPORARY TABLE intermDst1 AS SELECT DISTINCT(dst) FROM ' || sTable
|| ' WHERE src IN (SELECT * FROM intermDst)';
-- Does not return from function!
return query SELECT * FROM intermDst1;
-- Does not return from function!
intermDst_ := ARRAY(SELECT * FROM intermDst1);
raise notice 'timestamp: %', clock_timestamp();
SELECT count(*) INTO iCount FROM intermDst;
raise notice 'Count Table: %', iCount;
DROP TABLE intermDst;
DROP TABLE intermDst1;
-- As recursion depth is 5
if iRecursionDepth > 1 THEN
return query SELECT * FROM recursivesearch(intermdst_, iRecursionDepth - 1, sTable)
;
ELSE
RETURN;
END IF;
END;
$$ LANGUAGE plpgsql;

```

Listing A.19: SQL Standard Generisch

```

CREATE OR REPLACE FUNCTION selectWithUnionSourceCodeGenerator_withDepth(sTable text
, startingNode integer, depth integer ) RETURNS SETOF integer AS $$
Declare
intermDst_ integer[];
tStatement text;
tSelectStatement text;
tWithStatement text;
tUnionStatement text;
tWithStatementClose text;
BEGIN
tWithStatement := 'WITH RECURSIVE graphtraverse(src, dst, lvl) AS(';
tSelectStatement := 'SELECT src ,dst, 1 as lvl FROM ' || sTable || ' WHERE src =' ||
startingNode;
tUnionStatement := ' UNION SELECT p1.src,p1.dst,p.lvl+1 as lvl FROM graphtraverse p
, ' || sTable || ' p1 WHERE p1.src IN ( p.dst ) and lvl<' || depth;
tWithStatementClose := ')' SELECT DISTINCT(dst) FROM graphtraverse';
tStatement := tWithStatement || tSelectStatement || tUnionStatement ||
tWithStatementClose;
raise notice 'Execute String %', tStatement;
return query EXECUTE tStatement;
END;
$$ LANGUAGE plpgsql;

```

Listing A.20: SQL Standard

```

WITH RECURSIVE graphtraverse(src, dst, lvl) AS(
SELECT src ,dst, 1 as lvl FROM public.relation_facebook WHERE src =765
UNION
SELECT p1.src,p1.dst,p.lvl+1 as lvl FROM graphtraverse p, relation_facebook p1
WHERE p1.src IN ( p.dst ) and lvl<5
) SELECT DISTINCT(dst) FROM graphtraverse order by dst;

```

Listing A.21: Ausführungsplan Standard SQL

```

Sort (cost=15300.01..15300.51 rows=200 width=4) (actual time=17.613..17.622 rows
=321 loops=1)

```

```

Sort Key: graphtraverse.dst
Sort Method: quicksort Memory: 40kB
CTE graphtraverse
-> Recursive Union (cost=0.29..14814.87 rows=21133 width=12) (actual time
=0.016..15.903 rows=6056 loops=1)
-> Index Scan using indexsrc on relation_facebook (cost=0.29..44.00 rows=23
width=12) (actual time=0.015..0.020 rows=27 loops=1)
Index Cond: (src = 765)
-> Nested Loop (cost=0.29..1434.82 rows=2111 width=12) (actual time
=0.019..2.130 rows=8173 loops=5)
-> WorkTable Scan on graphtraverse p (cost=0.00..5.17 rows=77 width=8) (
actual time=0.017..0.069 rows=729 loops=5)
Filter: (lvl < 5)
Rows Removed by Filter: 482
-> Index Scan using indexsrc on relation_facebook p1 (cost=0.29..18.23 rows
=27 width=8) (actual time=0.001..0.002 rows=11 loops=3645)
Index Cond: (src = p.dst)
-> HashAggregate (cost=475.49..477.49 rows=200 width=4) (actual time
=17.548..17.571 rows=321 loops=1)
Group Key: graphtraverse.dst
-> CTE Scan on graphtraverse (cost=0.00..422.66 rows=21133 width=4) (actual
time=0.017..16.771 rows=6056 loops=1)
Planning Time: 0.105 ms
Execution Time: 17.813 ms

```

### Listing A.22: Ausführungsplan verschachteltes SELECT

```

HashAggregate (cost=6623.97..6659.95 rows=3598 width=4) (actual time
=27.019..27.049 rows=317 loops=1)
Group Key: relation_facebook.dst
-> Hash Join (cost=4727.16..6403.39 rows=88234 width=4) (actual time
=20.593..26.743 rows=2411 loops=1)
Hash Cond: (relation_facebook.src = relation_facebook_1.dst)
-> Seq Scan on relation_facebook (cost=0.00..1444.34 rows=88234 width=8) (
actual time=0.008..3.889 rows=88234 loops=1)
-> Hash (cost=4682.18..4682.18 rows=3598 width=4) (actual time=18.021..18.021
rows=195 loops=1)
Buckets: 4096 Batches: 1 Memory Usage: 39kB
-> HashAggregate (cost=4610.22..4646.20 rows=3598 width=4) (actual time
=17.976..18.000 rows=195 loops=1)
Group Key: relation_facebook_1.dst
-> Hash Join (cost=2713.40..4389.64 rows=88234 width=4) (actual time
=11.821..17.797 rows=1709 loops=1)
Hash Cond: (relation_facebook_1.src = relation_facebook_2.dst)
-> Seq Scan on relation_facebook relation_facebook_1 (cost
=0.00..1444.34 rows=88234 width=8) (actual time
=0.002..3.916 rows=88234 loops=1)
-> Hash (cost=2668.43..2668.43 rows=3598 width=4) (actual
time=9.177..9.177 rows=144 loops=1)
Buckets: 4096 Batches: 1 Memory Usage: 38kB
-> HashAggregate (cost=2596.47..2632.45 rows=3598 width
=4) (actual time=9.140..9.161 rows=144 loops=1)
Group Key: relation_facebook_2.dst
-> Hash Join (cost=876.96..2553.22 rows=17301 width
=4) (actual time=2.942..9.020 rows=1280 loops=1)
Hash Cond: (relation_facebook_2.src =
relation_facebook_3.dst)
-> Seq Scan on relation_facebook
relation_facebook_2 (cost=0.00..1444.34 rows
=88234 width=8) (actual time=0.002..4.163 rows
=88234 loops=1)
-> Hash (cost=869.07..869.07 rows=631 width=4) (
actual time=0.276..0.276 rows=88 loops=1)
Buckets: 1024 Batches: 1 Memory Usage: 12kB
-> HashAggregate (cost=856.45..862.76 rows
=631 width=4) (actual time=0.259..0.268
rows=88 loops=1)
Group Key: relation_facebook_3.dst

```



```

-> Nested Loop (cost=44.35..854.88
rows=631 width=4) (actual time
=0.022..0.199 rows=629 loops=1)
-> HashAggregate (cost
=44.05..44.28 rows=23 width=4)
(actual time=0.018..0.021 rows
=27 loops=1)
Group Key: relation_facebook_4.dst
-> Index Scan using indexsrc on
relation_facebook
relation_facebook_4 (cost
=0.29..44.00 rows=23 width=4) (
actual time=0.009..0.012 rows
=27 loops=1)
Index Cond: (src = 765)
-> Index Scan using indexsrc on
relation_facebook relation_facebook_3 (
cost=0.29..34.96 rows=27 width=8) (actual
time=0.002..0.005 rows=23 loops=27)
Index Cond: (src = relation_facebook_4.dst)

Planning Time: 0.170 ms
Execution Time: 27.113 ms

```

### Listing A.23: Ausführungsplan INNER JOIN

```

HashAggregate (cost=172324.88..172360.86 rows=3598 width=4) (actual time
=156.786..156.823 rows=317 loops=1)
Group Key: rf5.dst
Buffers: shared hit=2654
-> Merge Join (cost=38989.73..153806.87 rows=7407207 width=4) (actual time
=33.728..105.213 rows=572149 loops=1)
Merge Cond: (rf5.src = rf4.dst)
Buffers: shared hit=2654
-> Index Scan using indexsrc on relation_facebook rf5 (cost=0.29..3539.96 rows
=88234 width=8) (actual time=0.004..3.466 rows=33654 loops=1)
Buffers: shared hit=331
-> Sort (cost=38972.52..39749.91 rows=310956 width=4) (actual time=30.171..50.861
rows=579169 loops=1)
Sort Key: rf4.dst
Sort Method: quicksort Memory: 6709kB
Buffers: shared hit=2323
-> Merge Join (cost=2353.66..10603.46 rows=310956 width=4) (actual time
=9.093..21.946 rows=77587 loops=1)
Merge Cond: (rf3.dst = rf4.src)
Buffers: shared hit=2323
-> Sort (cost=2333.95..2366.58 rows=13054 width=4) (actual time=3.176..3.593 rows
=8177 loops=1)
Sort Key: rf3.dst
Sort Method: quicksort Memory: 576kB
Buffers: shared hit=1993
-> Nested Loop (cost=0.88..1441.56 rows=13054 width=4) (actual time=0.012..2.289
rows=8177 loops=1)
Buffers: shared hit=1993
-> Nested Loop (cost=0.58..854.36 rows=548 width=4) (actual time=0.009..0.156
rows=629 loops=1)
Buffers: shared hit=89
-> Index Scan using indexsrc on relation_facebook rf1 (cost=0.29..44.00 rows=23
width=4) (actual time=0.004..0.008 rows=27 loops=1)
Index Cond: (src = 765)
Buffers: shared hit=3
-> Index Scan using indexsrc on relation_facebook rf2 (cost=0.29..34.96 rows=27
width=8) (actual time=0.001..0.003 rows=23 loops=27)
Index Cond: (src = rf1.dst)
Buffers: shared hit=86
-> Index Scan using indexsrc on relation_facebook rf3 (cost=0.29..0.80 rows=27
width=8) (actual time=0.001..0.002 rows=13 loops=629)
Index Cond: (src = rf2.dst)
Buffers: shared hit=1904
-> Materialize (cost=0.29..3760.54 rows=88234 width=8) (actual time=0.004..8.301
rows=109531 loops=1)

```

```
Buffers: shared hit=330
-> Index Scan using indexsrc on relation_facebook rf4 (cost=0.29..3539.96 rows
    =88234 width=8) (actual time=0.003..3.863 rows=33653 loops=1)
Buffers: shared hit=330
Planning Time: 0.627 ms
Execution Time: 157.021 ms
```

## A.4 Graph-Datenbanken im praktischen Einsatz: OLAP

# Abbildungsverzeichnis

1.1	gewurzelter Baum . . . . .	3
1.2	Property Graph . . . . .	4
1.3	Hypergraph . . . . .	5
1.4	Hypernode . . . . .	6
3.1	Löschen von Duplikaten in einer Rekursionsstufe . . . . .	13
4.1	Ordnerstruktur Benchmarktest . . . . .	19
4.2	relation_epinions . . . . .	21
4.3	relation_livejournal . . . . .	22
4.4	relation_facebook . . . . .	23
4.5	relation_wiki_vote . . . . .	24
4.6	relation_youtube . . . . .	25
4.7	relation_epinions_with_index . . . . .	26
4.8	relation_livejournal_with_index . . . . .	27
4.9	relation_facebook_with_index . . . . .	28
4.10	relation_wiki_vote . . . . .	29
4.11	relation_youtube_with_index . . . . .	30
4.12	relation_epinions_partitioned . . . . .	31
4.13	relation_livejournal_partitioned . . . . .	32
4.14	relation_facebook_partitioned . . . . .	33
4.15	relation_wiki_vote_partitioned . . . . .	34
4.16	relation_youtube_partitioned . . . . .	35
A.1	Postgres Architektur . . . . .	38

# Tabellenverzeichnis

0.1	Aufgabenverteilung . . . . .	III
4.1	Laufzeit der SQLs für Tabelle relation_epions . . . . .	21
4.2	Laufzeit der SQLs für Tabelle relation_livejournal . . . . .	22
4.3	Laufzeit der SQLs für Tabelle relation_facebook . . . . .	23
4.4	Laufzeit der SQLs für Tabelle relation_wiki_vote . . . . .	24
4.5	Laufzeit der SQLs für Tabelle relation_youtube . . . . .	25
4.6	Laufzeit der SQLs für Tabelle relation_epinions_with_index . . . . .	26
4.7	Laufzeit der SQLs für Tabelle relation_livejournal_with_index . . . . .	27
4.8	Laufzeit der SQLs für Tabelle relation_facebook_with_index . . . . .	28
4.9	Laufzeit der SQLs für Tabelle relation_wiki_vote_with_index . . . . .	29
4.10	Laufzeit der SQLs für Tabelle relation_youtube_with_index . . . . .	30
4.11	Laufzeit der SQLs für Tabelle relation_epinions_partitioned . . . . .	31
4.12	Laufzeit der SQLs für Tabelle relation_livejournal_partitioned . . . . .	32
4.13	Laufzeit der SQLs für Tabelle relation_facebook_partitioned . . . . .	33
4.14	Laufzeit der SQLs für Tabelle relation_wiki_vote_partitioned . . . . .	34
4.15	Laufzeit der SQLs für Tabelle relation_youtube_partitioned . . . . .	35

# Listings

3.1	Zeile im Ausführungsplan . . . . .	11
3.2	Rekursiver und nicht rekursiver Teil . . . . .	11
3.3	Überprüfen der Abbruchbedingung . . . . .	12
3.4	Überprüfung der WHERE Bedingung . . . . .	12
3.5	Überprüfung der WHERE Bedingung . . . . .	12
3.6	Aufruf der graphtraverse Funktion . . . . .	12
3.7	Aufruf der graphtraverse Funktion im Ausführungsplan . . . . .	12
3.8	Aufruf der DISTINCT Funktion . . . . .	13
3.9	IN Klausel . . . . .	14
3.10	Aufruf der DISTINCT Funktion . . . . .	14
3.11	Merge JOIN . . . . .	14
4.1	Tabelle mit Index anlegen . . . . .	17
4.2	Partitionierte Tabelle mit Indices anlegen . . . . .	17
4.3	pgbench Statment . . . . .	18
4.4	Ausgabe pgbench . . . . .	19
A.1	CSV Input . . . . .	38
A.2	Anlegen der Tabelle facebook-profiles . . . . .	39
A.3	Anlegen der Tabelle facebook-relation . . . . .	39
A.4	Hinzufügen von Fremdschlüsseln . . . . .	39
A.5	Erstellen von partitionierten Tabellen mit Index facebook . . . . .	39
A.6	Erstellen von Indexen auf relation Tabelle facebook . . . . .	39
A.7	Erstellen von partitionierten Tabellen mit Index youtube . . . . .	39
A.8	Erstellen von Indexen auf relation Tabelle youtube . . . . .	40
A.9	Erstellen von partitionierten Tabellen mit Index livejournal . . . . .	40
A.10	Erstellen von Indexen auf relation Tabelle livejournal . . . . .	40
A.11	Erstellen von partitionierten Tabellen mit Index epinion . . . . .	40
A.12	Erstellen von Indexen auf relation Tabelle epinion . . . . .	41
A.13	Erstellen von partitionierten Tabellen mit wikivote epinion . . . . .	41
A.14	Erstellen von Indexen auf relation Tabelle wikivote . . . . .	41
A.15	Erstellen der Indexe für die relation Tabelle facebook . . . . .	41
A.16	Verschachteltes SELECT Statement . . . . .	41
A.17	Rekursiver JOIN . . . . .	41
A.18	Selbstgeschriebenes Stored Procedure . . . . .	42
A.19	SQL Standard Generisch . . . . .	42
A.20	SQL Standard . . . . .	42
A.21	Ausführungsplan Standard SQL . . . . .	42

A.22 Ausführungsplan verschachteltes SELECT . . . . .	43
A.23 Ausführungsplan INNER JOIN . . . . .	44

# Abkürzungsverzeichnis

<b>ACID</b>	Atomicity, Consistency, Isolation, Durability
<b>API</b>	Application Programming Interface
<b>APT</b>	Advanced Package Tool
<b>CRUD</b>	Create, Read, Update, Delete
<b>CTE</b>	Common Table Expression
<b>IP</b>	Internet Protocol
<b>TCP</b>	Transmission Control Protocol
<b>TID</b>	Tuple Identifier
<b>MVCC</b>	Multiversion Concurrency Control Modell
<b>NoSQL</b>	Not only SQL
<b>OLTP</b>	Online Transaction Processing
<b>RPM</b>	Red Hat Package Manager
<b>SQL</b>	Structured Query Language

# Literaturverzeichnis

- [AG08] ANGLES, Renzo ; GUTIERREZ, Claudio: Survey of graph database models. In: *ACM Computing Surveys (CSUR)* 40 (2008), Nr. 1, S. 1
- [AG18] In: ANGLES, Renzo ; GUTIERREZ, Claudio: *An Introduction to Graph Data Mangement*. Springer International Publishing, 2018
- [Ang12] ANGLES, Renzo: A comparison of current graph database models. In: *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on IEEE*, 2012, S. 171–177
- [APPDSLP13] ANGLES, Renzo ; PRAT-PÉREZ, Arnau ; DOMINGUEZ-SAL, David ; LARRIBA-PEY, Josep-Lluis: Benchmarking database systems for social network applications. In: *First International Workshop on Graph Data Management Experiences and Systems* ACM, 2013, S. 15
- [AU95] AHO, Alfred V. ; ULLMAN, Jeffrey D.: *Foundations of computer science*. USA : Computer Science Press, 1995 <http://infolab.stanford.edu/~ullman/focs.html>
- [Bec18] BECKER, Peter: *Graphentheorie*. <http://www2.inf.h-brs.de/~pbecke2m/graphentheorie/einfuehrung.pdf>. Version: 2018
- [Cod81] CODD, Edgar F.: Data models in database management. In: *ACM Sigmod Record* 11 (1981), Nr. 2, S. 112–114
- [Eis03] EISENTRAUT, Peter: *PostgreSQL Das Offizielle Handbuch*. mitp-Verlag GmbH/Bonn, 2003
- [Fel03] FELSNER, Stefan: *Geometric graphs and arrangements: some chapters from combinatorial geometry*. Springer Science & Business Media, 2003
- [Fro18] FROEHLICH, Lutz: *PostgreSQL 10*. Carl Hanser Verlag München, 2018
- [GFLP18] GEORGE FLETCHER, Jan H. ; LARRIBA-PEY, Josep L.: *Graph Data Management - Fundamental Issues and Recent Developments*. Springer International Publishing, 2018
- [Goe15] GOEBEL, Christopher: NoSQL - FlockDB. (2015)



- [Groc] GROUP, The PostgreSQL Global D.: *PostgreSQL 11.1 Documentation*. <https://www.postgresql.org/files/documentation/pdf/11/postgresql-11-A4.pdf>
- [Grob] GROUP, The PostgreSQL Global D.: *PostgreSQL 11.1 Documentation*. <https://www.postgresql.org/docs/9.4/using-explain.html>
- [Groc] GROUP, The PostgreSQL Global D.: *PostgreSQL 11.1 Documentation*. [url=https://www.postgresql.org/docs/10/ddl-partitioning.html](https://www.postgresql.org/docs/10/ddl-partitioning.html)
- [Grod] GROUP, The PostgreSQL Global D.: *PostgreSQL 11.1 Documentation*. <https://www.postgresql.org/docs/11/queries-with.html>
- [Groe] GROUP, The PostgreSQL Global D.: *PostgreSQL 8.4 Documentation*. <https://www.postgresql.org/docs/8.4/datatype.html>
- [Gru17] GRUCIA, Jelena: *PostgreSQL and GraphQL*. <https://blog.cloudboost.io/postgresql-and-graphql-2da30c6cde26>. Version: 2017
- [Ior10] IORDANOV, Borislav: HyperGraphDB: a generalized graph database. In: *International conference on web-age information management* Springer, 2010, S. 25–36
- [KHA<sup>+</sup>16] KUCUK, Ahmet ; HAMDI, Shah M. ; AYDIN, Berkay ; SCHUH, Michael A. ; ANGRYK, Rafal A.: Pg-Trajectory: A PostgreSQL/PostGIS based data model for spatiotemporal trajectories. In: *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom)* IEEE, 2016, S. 81–88
- [KK15] KNAUER, U. ; KNAUER, K.: *Diskrete und algebraische Strukturen - kurz gefasst*. Springer Berlin Heidelberg, 2015
- [KN12] KRUMKE, S.O. ; NOLTEMEIER, H.: *Graphentheoretische Konzepte und Algorithmen*. Vieweg+Teubner Verlag, 2012 (Leitfäden der Informatik). – ISBN 9783834822642
- [KS96] KORTH, A. Silberschatz; H. F. ; SUDARSHAN, S.: Data Models. In: *ACM Computing Surveys* (1996)
- [Kud15] KUDRASS, Thomas: *Taschenbuc Datenbanken*. Fachbuchverlag Leipzig im Carl Hanser Verlag, 2015
- [PL94] POULOVASSILIS, Alexandra ; LEVENE, Mark: A nested-graph model for the representation and manipulation of complex objects. In: *ACM Transactions on Information Systems (TOIS)* 12 (1994), Nr. 1, S. 35–68

- [Pos18] POSTGRESQL GLOBAL DEVELOPMENT GROUP: *PostgreSQL 11.1 Documentation*. <https://www.postgresql.org/docs/11>. Version: 2018
- [Rah17a] RAHMAN, Md. S.: *Basic Graph Theory*. Springer International Publishing, 2017
- [Rah17b] RAHMAN, Saidur: *Basic Graph Theory*. 2017
- [Red12] REDMOND, Eric: *Sieben Wochen, sieben Datenbanken*. O'Reilly Verlag, 2012
- [Sas18] SASAKI, Bryce M.: *Graph Databases for Beginners: The Basics of Data Modeling*. <https://neo4j.com/blog/data-modeling-basics/>. Version: 2018
- [SF05] STEFAN FELSNER, Gesine K. Christine Puhl P. Christine Puhl: *Graphentheorie*. <http://page.math.tu-berlin.de/~felsner/Lehre/GrTh05/Graphentheorie.pdf>. Version: 2005
- [TT16] In: THORSTEN THEOBALD, Sadik I.: *Graphen*. Springer Fachmedien Wiesbaden, 2016
- [VMZ<sup>+</sup>10] VICKNAIR, Chad ; MACIAS, Michael ; ZHAO, Zhendong ; NAN, Xiaofei ; CHEN, Yixin ; WILKINS, Dawn: A comparison of a graph database and a relational database: a data provenance perspective. In: *Proceedings of the 48th annual Southeast regional conference ACM*, 2010, S. 42
- [ZSHZ18] ZHANG, Hongliang ; SONG, Lingyang ; HAN, Zhu ; ZHANG, Yingjun: *Hypergraph Theory in Wireless Communication Networks*. Springer, 2018

# Eidesstattliche Erklärung

Ich versichere an Eides Statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt habe. Alle Stellen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Ich versichere außerdem, dass ich keine andere als die angegebene Literatur verwendet habe. Diese Versicherung bezieht sich auch auf alle in der Arbeit enthaltenen Zeichnungen, Skizzen, bildlichen Darstellungen und dergleichen.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

*Sankt Augustin,*  
*den 19. Januar 2019*  
Ort, Datum

---

Jennifer Wittling, Rolf  
Kimmelman, Jan  
Löffelsender